Chapter 10 Character Sets, Collations, Unicode

**Table** **of** **Contents**

[10.1 Character Sets Collations General](#_bookmark2)andin [1974](#_bookmark2)

[10.2 Character Sets Collations MySQL](#_bookmark3)andin [1975](#_bookmark3)

[10.2.1 Character Set Repertoire](#_bookmark4) [1977](#_bookmark4)

[10.2.2 UTF-8 Metadata](#_bookmark5)for [1979](#_bookmark5)

[10.3 Specifying Character Sets Collations](#_bookmark6)and [1980](#_bookmark6)

[10.3.1 Collation Naming Conventions](#_bookmark7) [1980](#_bookmark7)

[10.3.2 Server Character Set Collation](#_bookmark8)and [1981](#_bookmark8)

[10.3.3 Database Character Set Collation](#_bookmark9)and [1982](#_bookmark9)

[10.3.4 Table Character Set Collation](#_bookmark10)and [1983](#_bookmark10)

[10.3.5 Column Character Set Collation](#_bookmark11)and [1984](#_bookmark11)

[10.3.6 Character String Literal Character Set Collation](#_bookmark12)and [1985](#_bookmark12)

[10.3.7 The National Character Set](#_bookmark13) [1987](#_bookmark13)

[10.3.8 Character Set Introducers](#_bookmark14) [1987](#_bookmark14)

[10.3.9 Examples Character Set Collation Assignment](#_bookmark15)ofand [1989](#_bookmark15)

[10.3.10 Compatibility Other DBMSs](#_bookmark16)with [1990](#_bookmark16)

[10.4 Connection Character Sets Collations](#_bookmark17)and [1990](#_bookmark17)

[10.5 Configuring Application Character Set Collation](#_bookmark18)and [1996](#_bookmark18)

[10.6 Error Message Character Set](#_bookmark19) [1998](#_bookmark19)

[10.7 Column Character Set Conversion](#_bookmark20) [1999](#_bookmark20)

[10.8 Collation Issues](#_bookmark21) [2000](#_bookmark21)

[10.8.1 Using COLLATE SQL Statements](#_bookmark22)in [2000](#_bookmark22)

[10.8.2 COLLATE Clause Precedence](#_bookmark23) [2000](#_bookmark23)

[10.8.3 Character Set Collation Compatibility](#_bookmark24)and [2001](#_bookmark24)

[10.8.4 Collation Coercibility Expressions](#_bookmark25)in [2001](#_bookmark25)

[10.8.5 The Collation Compared \_ Collations](#_bookmark26)binarytobin [2002](#_bookmark26)

[10.8.6 Examples Effect Collation](#_bookmark27)oftheof [2005](#_bookmark27)

[10.8.7 Using Collation INFORMATION\_SCHEMA Searches](#_bookmark28)in [2006](#_bookmark28)

[10.9 Unicode Support](#_bookmark29) [2008](#_bookmark29)

[10.9.1 The utf8mb4 Character Set (4-Byte UTF-8 Unicode Encoding)](#_bookmark30) [2010](#_bookmark30)

[10.9.2 The utf8mb3 Character Set (3-Byte UTF-8 Unicode Encoding)](#_bookmark31) [2010](#_bookmark31)

[10.9.3 The utf8 Character Set (Alias utf8mb3)](#_bookmark32)for [2011](#_bookmark32)

[10.9.4 The ucs2 Character Set (UCS-2 Unicode Encoding)](#_bookmark33) [2012](#_bookmark33)

[10.9.5 The utf16 Character Set (UTF-16 Unicode Encoding)](#_bookmark34) [2012](#_bookmark34)

[10.9.6 The utf16le Character Set (UTF-16LE Unicode Encoding)](#_bookmark35) [2013](#_bookmark35)

[10.9.7 The utf32 Character Set (UTF-32 Unicode Encoding)](#_bookmark36) [2013](#_bookmark36)

[10.9.8 Converting Between 3-Byte 4-Byte Unicode Character Sets](#_bookmark37)and [2013](#_bookmark37)

[10.10 Supported Character Sets Collations](#_bookmark38)and [2015](#_bookmark38)

[10.10.1 Unicode Character Sets](#_bookmark39) [2016](#_bookmark39)

[10.10.2 West European Character Sets](#_bookmark40) [2023](#_bookmark40)

[10.10.3 Central European Character Sets](#_bookmark41) [2025](#_bookmark41)

[10.10.4 South European Middle East Character Sets](#_bookmark42)and [2026](#_bookmark42)

[10.10.5 Baltic Character Sets](#_bookmark43) [2027](#_bookmark43)

[10.10.6 Cyrillic Character Sets](#_bookmark44) [2027](#_bookmark44)

[10.10.7 Asian Character Sets](#_bookmark45) [2027](#_bookmark45)

[10.10.8 The Binary Character Set](#_bookmark46) [2032](#_bookmark46)

[10.11 Restrictions Character Sets](#_bookmark47)on [2033](#_bookmark47)

[10.12 Setting Error Message Language](#_bookmark48)the [2033](#_bookmark48)

[10.13 Adding Character Set](#_bookmark49)a [2034](#_bookmark49)

[10.13.1 Character Definition Arrays](#_bookmark50) [2036](#_bookmark50)

[10.13.2 String Collating Support Complex Character Sets](#_bookmark51)for [2037](#_bookmark51)

[10.13.3 Multi-Byte Character Support Complex Character Sets](#_bookmark52)for [2037](#_bookmark52)

[10.14 Adding Collation Character Set](#_bookmark53)atoa [2037](#_bookmark53)

[10.14.1 Collation Implementation Types](#_bookmark54) [2038](#_bookmark54)

[10.14.2 Choosing Collation ID](#_bookmark55)a [2041](#_bookmark55)

[10.14.3 Adding Simple Collation 8-Bit Character Set](#_bookmark56)atoan [2042](#_bookmark56)

[10.14.4 Adding UCA Collation Unicode Character Set](#_bookmark57)atoa [2043](#_bookmark57)

[10.15 Character Set Configuration](#_bookmark58) [2049](#_bookmark58)

[10.16 MySQL Server Locale Support](#_bookmark59) [2050](#_bookmark59)

MySQL includes character set support that enables you to store data using a variety of character sets and perform comparisons according to a variety of collations. The default MySQL server character set and collation are utf8mb4 and utf8mb4\_0900\_ai\_ci, but you can specify character sets at the server, database, table, column, and string literal levels.

This chapter discusses the following topics:

• What are character sets and collations?

• The multiple-level default system for character set assignment.

• Syntax for specifying character sets and collations.

• Affected functions and operations.

• Unicode support.

• The character sets and collations that are available, with notes.

• Selecting the language for error messages.

• Selecting the locale for day and month names.

Character set issues affect not only data storage, but also communication between client programs and the MySQL server. If you want the client program to communicate with the server using a character set different from the default, you'll need to indicate which one. For example, to use the utf8mb4 Unicode character set, issue this statement after connecting to the server:

SET NAMES 'utf8mb4';

For more information about configuring character sets for application use and character set-related issues in client/server communication, see [Section 10.5, “Configuring Application Character Set and](#_bookmark18) [Collation”](#_bookmark18) , and [Section 10.4, “Connection Character Sets Collations”](#_bookmark17)and.

**10.1** **Character** **Sets** **and** **Collations** **in** **General**

A *character* *set* is a set of symbols and encodings. A *collation* is a set of rules for comparing characters in a character set. Let's make the distinction clear with an example of an imaginary character set.

Suppose that we have an alphabet with four letters: A, B, a, b. We give each letter a number: A = 0, B = 1, a = 2, b = 3. The letter A is a symbol, the number 0 is the *encoding* for A, and the combination of all four letters and their encodings is a *character* *set*.

Suppose that we want to compare two string values, A and B. The simplest way to do this is to look at the encodings: 0 for A and 1 for B. Because 0 is less than 1, we say A is less than B. What we've just done is apply a collation to our character set. The collation is a set of rules (only one rule in this case): “compare the encodings.” We call this simplest of all possible collations a *binary* collation.

But what if we want to say that the lowercase and uppercase letters are equivalent? Then we would have at least two rules: (1) treat the lowercase letters a and b as equivalent to A and B; (2) then compare the encodings. We call this a *case-insensitive* collation. It is a little more complex than a binary collation.

In real life, most character sets have many characters: not just A and B but whole alphabets, sometimes multiple alphabets or eastern writing systems with thousands of characters, along with many special symbols and punctuation marks. Also in real life, most collations have many rules, not

just for whether to distinguish lettercase, but also for whether to distinguish accents (an “accent” is a mark attached to a character as in German Ö), and for multiple-character mappings (such as the rule

that Ö = OE in one of the two German collations).

MySQL can do these things for you:

• Store strings using a variety of character sets.

• Compare strings using a variety of collations.

• Mix strings with different character sets or collations in the same server, the same database, or even the same table.

• Enable specification of character set and collation at any level.

To use these features effectively, you must know what character sets and collations are available, how to change the defaults, and how they affect the behavior of string operators and functions.

**10.2** **Character** **Sets** **and** **Collations** **in** **MySQL**

MySQL Server supports multiple character sets, including several Unicode character sets. To display the available character sets, use the INFORMATION\_SCHEMA CHARACTER\_SETS table or the SHOW CHARACTER SET statement. A partial listing follows. For more complete information, see [Section 10.10, “Supported Character Sets and Collations”](#_bookmark38) .

mysql> **SHOW** **CHARACTER** **SET;**

+----------+---------------------------------+---------------------+--------+

| Charset | Description | Default collation | Maxlen |

+----------+---------------------------------+---------------------+--------+

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| | big5  | binary  ...  | latin1  ...  | ucs2  ...  | utf8mb3  | utf8mb4  ... | |  |  |  |  |  | | Big5 Traditional Chinese  Binary pseudo charset  cp1252 West European  UCS-2 Unicode  UTF-8 Unicode  UTF-8 Unicode | | big5\_chinese\_ci  | binary  | latin1\_swedish\_ci  | ucs2\_general\_ci  | utf8mb3\_general\_ci  | utf8mb4\_0900\_ai\_ci | |  |  |  |  |  | | 2  1  1  2  3  4 | |  |  |  |  |  | |

By default, the SHOW CHARACTER SET statement displays all available character sets. It takes an optional LIKE or WHERE clause that indicates which character set names to match. The following example shows some of the Unicode character sets (those based on Unicode Transformation Format):

mysql> **SHOW** **CHARACTER** **SET** **LIKE** **'utf%';**

+---------+------------------+--------------------+--------+

| Charset | Description | Default collation | Maxlen |

+---------+------------------+--------------------+--------+

|  |  |  |
| --- | --- | --- |
| | utf16 | UTF-16 Unicode | utf16\_general\_ci | | 4 | | |
| | utf16le | UTF-16LE Unicode | utf16le\_general\_ci | | 4 | | |
| | utf32 | UTF-32 Unicode | utf32\_general\_ci | | 4 | | |
| | utf8mb3 | UTF-8 Unicode | utf8mb3\_general\_ci | | 3 | | |
| | utf8mb4 | UTF-8 Unicode | utf8mb4\_0900\_ai\_ci | | 4 | | |

+---------+------------------+--------------------+--------+

A given character set always has at least one collation, and most character sets have several. To list the display collations for a character set, use the INFORMATION\_SCHEMA COLLATIONS table or the SHOW COLLATION statement.

By default, the SHOW COLLATION statement displays all available collations. It takes an optional LIKE or WHERE clause that indicates which collation names to display. For example, to see the collations for the default character set, utf8mb4, use this statement:

mysql> **SHOW** **COLLATION** **WHERE** **Charset** **=** **'utf8mb4';**

+----------------------------+---------+-----+---------+----------+---------+---------------+

| Collation | Charset | Id | Default | Compiled | Sortlen | Pad\_attribute |

+----------------------------+---------+-----+---------+----------+---------+---------------+

| utf8mb4\_0900\_ai\_ci | utf8mb4 | 255 | Yes | Yes | 0 | NO PAD |

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes  Yes | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | 0  0  1  1  8  0  0  8  8  0  0  0  0  0  0  8  8  0  0  0  0  0  0  1  8  0  0  8  0  0  8  0  0  0  24  8  0  0  8  0  0  0  0  8  0  0  8  8  8  0  0  0  0  8  0  0  8  8  0  0  8  8  0  0  8  0  0  8  8  8  8  0 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |

NO

NO

NO

PAD

PAD

NO

NO

PAD

PAD

NO

NO

NO

NO

NO

NO

PAD

PAD

NO

NO

NO

NO

NO

NO

PAD

PAD

NO

NO

PAD

NO

NO

PAD

NO

NO

NO

NO

PAD

NO

NO

PAD

NO

NO

NO

NO

PAD

NO

NO

PAD

PAD

PAD

NO

NO

NO

NO

PAD

NO

NO

PAD

PAD

NO

NO

PAD

PAD

NO

NO

PAD

NO

NO

PAD

PAD

PAD

PAD

NO

PAD

PAD

PAD

SPACE

SPACE

PAD

PAD

SPACE

SPACE

PAD

PAD

PAD

PAD

PAD

PAD

SPACE

SPACE

PAD

PAD

PAD

PAD

PAD

PAD

SPACE

SPACE

PAD

PAD

SPACE

PAD

PAD

SPACE

PAD

PAD

PAD

PAD

SPACE

PAD

PAD

SPACE

PAD

PAD

PAD

PAD

SPACE

PAD

PAD

SPACE

SPACE

SPACE

PAD

PAD

PAD

PAD

SPACE

PAD

PAD

SPACE

SPACE

PAD

PAD

SPACE

SPACE

PAD

PAD

SPACE

PAD

PAD

SPACE

SPACE

SPACE

SPACE

PAD

|  |  |  |
| --- | --- | --- |
| | utf8mb4\_0900\_as\_ci | | utf8mb4 | | 305 | |
| utf8mb4\_0900\_as\_cs | utf8mb4 | 278 |
| utf8mb4\_0900\_bin | utf8mb4 | 309 |
| | utf8mb4 bin | | utf8mb4 | | 46 | |
| | utf8mb4\_croatian\_ci | | utf8mb4 | | 245 | |
| | utf8mb4\_cs\_0900\_ai\_ci | | utf8mb4 | | 266 | |
| utf8mb4\_cs\_0900\_as\_cs | utf8mb4 | 289 |
| utf8mb4\_czech\_ci | utf8mb4 | 234 |
| | utf8mb4 danish ci | | utf8mb4 | | 235 | |
| | utf8mb4\_da\_0900\_ai\_ci | | utf8mb4 | | 267 | |
| | utf8mb4\_da\_0900\_as\_cs | | utf8mb4 | | 290 | |
| utf8mb4\_de\_pb\_0900\_ai\_ci | utf8mb4 | 256 |
| utf8mb4\_de\_pb\_0900\_as\_cs | utf8mb4 | 279 |
| | utf8mb4 eo 0900 ai ci | | utf8mb4 | | 273 | |
| | utf8mb4\_eo\_0900\_as\_cs | | utf8mb4 | | 296 | |
| | utf8mb4\_esperanto\_ci | | utf8mb4 | | 241 | |
| utf8mb4\_estonian\_ci | utf8mb4 | 230 |
| utf8mb4\_es\_0900\_ai\_ci | utf8mb4 | 263 |
| | utf8mb4 es 0900 as cs | | utf8mb4 | | 286 | |
| | utf8mb4\_es\_trad\_0900\_ai\_ci | | utf8mb4 | | 270 | |
| | utf8mb4\_es\_trad\_0900\_as\_cs | | utf8mb4 | | 293 | |
| utf8mb4\_et\_0900\_ai\_ci | utf8mb4 | 262 |
| utf8mb4\_et\_0900\_as\_cs | utf8mb4 | 285 |
| | utf8mb4 general ci | | utf8mb4 | | 45 | |
| | utf8mb4\_german2\_ci | | utf8mb4 | | 244 | |
| | utf8mb4\_hr\_0900\_ai\_ci | | utf8mb4 | | 275 | |
| utf8mb4\_hr\_0900\_as\_cs | utf8mb4 | 298 |
| utf8mb4\_hungarian\_ci | utf8mb4 | 242 |
| | utf8mb4 hu 0900 ai ci | | utf8mb4 | | 274 | |
| | utf8mb4\_hu\_0900\_as\_cs | | utf8mb4 | | 297 | |
| | utf8mb4\_icelandic\_ci | | utf8mb4 | | 225 | |
| utf8mb4\_is\_0900\_ai\_ci | utf8mb4 | 257 |
| utf8mb4\_is\_0900\_as\_cs | utf8mb4 | 280 |
| | utf8mb4 ja 0900 as cs | | utf8mb4 | | 303 | |
| | utf8mb4\_ja\_0900\_as\_cs\_ks | | utf8mb4 | | 304 | |
| | utf8mb4\_latvian\_ci | | utf8mb4 | | 226 | |
| utf8mb4\_la\_0900\_ai\_ci | utf8mb4 | 271 |
| utf8mb4\_la\_0900\_as\_cs | utf8mb4 | 294 |
| | utf8mb4 lithuanian ci | | utf8mb4 | | 236 | |
| | utf8mb4\_lt\_0900\_ai\_ci | | utf8mb4 | | 268 | |
| | utf8mb4\_lt\_0900\_as\_cs | | utf8mb4 | | 291 | |
| utf8mb4\_lv\_0900\_ai\_ci | utf8mb4 | 258 |
| utf8mb4\_lv\_0900\_as\_cs | utf8mb4 | 281 |
| | utf8mb4 persian ci | | utf8mb4 | | 240 | |
| | utf8mb4\_pl\_0900\_ai\_ci | | utf8mb4 | | 261 | |
| | utf8mb4\_pl\_0900\_as\_cs | | utf8mb4 | | 284 | |
| utf8mb4\_polish\_ci | utf8mb4 | 229 |
| utf8mb4\_romanian\_ci | utf8mb4 | 227 |
| | utf8mb4 roman ci | | utf8mb4 | | 239 | |
| | utf8mb4\_ro\_0900\_ai\_ci | | utf8mb4 | | 259 | |
| | utf8mb4\_ro\_0900\_as\_cs | | utf8mb4 | | 282 | |
| utf8mb4\_ru\_0900\_ai\_ci | utf8mb4 | 306 |
| utf8mb4\_ru\_0900\_as\_cs | utf8mb4 | 307 |
| | utf8mb4 sinhala ci | | utf8mb4 | | 243 | |
| | utf8mb4\_sk\_0900\_ai\_ci | | utf8mb4 | | 269 | |
| | utf8mb4\_sk\_0900\_as\_cs | | utf8mb4 | | 292 | |
| utf8mb4\_slovak\_ci | utf8mb4 | 237 |
| utf8mb4\_slovenian\_ci | utf8mb4 | 228 |
| | utf8mb4 sl 0900 ai ci | | utf8mb4 | | 260 | |
| | utf8mb4\_sl\_0900\_as\_cs | | utf8mb4 | | 283 | |
| | utf8mb4\_spanish2\_ci | | utf8mb4 | | 238 | |
| utf8mb4\_spanish\_ci | utf8mb4 | 231 |
| utf8mb4\_sv\_0900\_ai\_ci | utf8mb4 | 264 |
| | utf8mb4 sv 0900 as cs | | utf8mb4 | | 287 | |
| | utf8mb4\_swedish\_ci | | utf8mb4 | | 232 | |
| | utf8mb4\_tr\_0900\_ai\_ci | | utf8mb4 | | 265 | |
| utf8mb4\_tr\_0900\_as\_cs | utf8mb4 | 288 |
| utf8mb4\_turkish\_ci | utf8mb4 | 233 |
| | utf8mb4 unicode 520 ci | | utf8mb4 | | 246 | |
| | utf8mb4\_unicode\_ci | | utf8mb4 | | 224 | |
| | utf8mb4\_vietnamese\_ci | | utf8mb4 | | 247 | |
| | utf8mb4 vi 0900 ai ci | | utf8mb4 | | 277 | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | utf8mb4\_vi\_0900\_as\_cs | | | utf8mb4 | | | 300 | | | | | Yes | | | 0 | | | NO PAD | | |
| | | utf8mb4\_zh\_0900\_as\_cs | | | utf8mb4 | | | 308 | | | | | Yes | | | 0 | | | NO PAD | | |

+----------------------------+---------+-----+---------+----------+---------+---------------+

For more information about those collations, see [Section 10.10.1, “Unicode Character Sets”](#_bookmark39) . Collations have these general characteristics:

• Two different character sets cannot have the same collation.

• Each character set has a *default* *collation*. For example, the default collations for utf8mb4 and latin1 are utf8mb4\_0900\_ai\_ci and latin1\_swedish\_ci, respectively. The INFORMATION\_SCHEMA CHARACTER\_SETS table and the SHOW CHARACTER SET statement indicate the default collation for each character set. The INFORMATION\_SCHEMA COLLATIONS table and the SHOW COLLATION statement have a column that indicates for each collation whether it is the default for its character set (Yes if so, empty if not).

• Collation names start with the name of the character set with which they are associated, generally followed by one or more suffixes indicating other collation characteristics. For additional information about naming conventions, see [Section 10.3.1, “Collation Naming Conventions”](#_bookmark7) .

When a character set has multiple collations, it might not be clear which collation is most suitable for a given application. To avoid choosing an inappropriate collation, perform some comparisons with representative data values to make sure that a given collation sorts values the way you expect.

**10.2.1** **Character** **Set** **Repertoire**

The *repertoire* of a character set is the collection of characters in the set.

String expressions have a repertoire attribute, which can have two values:

• ASCII: The expression can contain only ASCII characters; that is, characters in the Unicode range U +0000 to U+007F.

• UNICODE: The expression can contain characters in the Unicode range U+0000 to U+10FFFF. This includes characters in the Basic Multilingual Plane (BMP) range (U+0000 to U+FFFF) and supplementary characters outside the BMP range (U+10000 to U+10FFFF).

The ASCII range is a subset of UNICODE range, so a string with ASCII repertoire can be converted safely without loss of information to the character set of any string with UNICODE repertoire. It can also be converted safely to any character set that is a superset of the ascii character set. (All MySQL character sets are supersets of ascii with the exception of swe7, which reuses some punctuation characters for Swedish accented characters.)

The use of repertoire enables character set conversion in expressions for many cases where MySQL would otherwise return an “illegal mix of collations” error when the rules for collation coercibility are insufficient to resolve ambiguities. (For information about coercibility, see [Section 10.8.4, “Collation](#_bookmark25) [Coercibility in Expressions”](#_bookmark25) .)

The following discussion provides examples of expressions and their repertoires, and describes how the use of repertoire changes string expression evaluation:

• The repertoire for a string constant depends on string content and may differ from the repertoire of the string character set. Consider these statements:

SET NAMES utf8mb4; SELECT 'abc';

SELECT \_utf8mb4'def';

Although the character set is utf8mb4 in each of the preceding cases, the strings do not actually contain any characters outside the ASCII range, so their repertoire is ASCII rather than UNICODE.

• A column having the ascii character set has ASCII repertoire because of its character set. In the following table, c1 has ASCII repertoire:

CREATE TABLE t1 (c1 CHAR(1) CHARACTER SET ascii);

The following example illustrates how repertoire enables a result to be determined in a case where an error occurs without repertoire:

CREATE TABLE t1 (

c1 CHAR(1) CHARACTER SET latin1,

c2 CHAR(1) CHARACTER SET ascii

);

INSERT INTO t1 VALUES ('a','b');

SELECT CONCAT(c1,c2) FROM t1;

Without repertoire, this error occurs:

ERROR 1267 (HY000): Illegal mix of collations (latin1\_swedish\_ci,IMPLICIT)

and (ascii\_general\_ci,IMPLICIT) for operation 'concat'

Using repertoire, subset to superset (ascii to latin1) conversion can occur and a result is returned:

+---------------+

| CONCAT(c1,c2) |

+---------------+

| ab |

+---------------+

• Functions with one string argument inherit the repertoire of their argument. The result of UPPER(\_utf8mb4'abc') has ASCII repertoire because its argument has ASCII repertoire. (Despite the \_utf8mb4 introducer, the string 'abc' contains no characters outside the ASCII range.)

• For functions that return a string but do not have string arguments and use character\_set\_connection as the result character set, the result repertoire is ASCII if character\_set\_connection is ascii, and UNICODE otherwise:

FORMAT(*numeric\_column*, 4);

Use of repertoire changes how MySQL evaluates the following example:

SET NAMES ascii;

CREATE TABLE t1 (a INT, b VARCHAR(10) CHARACTER SET latin1);

INSERT INTO t1 VALUES (1,'b');

SELECT CONCAT(FORMAT(a, 4), b) FROM t1;

Without repertoire, this error occurs:

ERROR 1267 (HY000): Illegal mix of collations (ascii\_general\_ci,COERCIBLE)

and (latin1\_swedish\_ci,IMPLICIT) for operation 'concat'

With repertoire, a result is returned:

+-------------------------+

| CONCAT(FORMAT(a, 4), b) |

+-------------------------+

| 1 .0000b |

+-------------------------+

• Functions with two or more string arguments use the “widest” argument repertoire for the result repertoire, where UNICODE is wider than ASCII. Consider the following CONCAT() calls:

CONCAT(\_ucs2 X'0041', \_ucs2 X'0042')

CONCAT(\_ucs2 X'0041', \_ucs2 X'00C2')

For the first call, the repertoire is ASCII because both arguments are within the ASCII range. For the second call, the repertoire is UNICODE because the second argument is outside the ASCII range.

• The repertoire for function return values is determined based on the repertoire of only those arguments that affect the result's character set and collation.

IF(column1 < column2, 'smaller', 'greater')

The result repertoire is ASCII because the two string arguments (the second argument and the third argument) both have ASCII repertoire. The first argument does not matter for the result repertoire, even if the expression uses string values.

**10.2.2** **UTF-8** **for** **Metadata**

*Metadata* is “the data about the data.” Anything that *describes* the database—as opposed to being the *contents* of the database— is metadata. Thus column names, database names, user names, version names, and most of the string results from SHOW are metadata. This is also true of the contents of tables in INFORMATION\_SCHEMA because those tables by definition contain information about database objects.

Representation of metadata must satisfy these requirements:

• All metadata must be in the same character set. Otherwise, neither the SHOW statements nor SELECT statements for tables in INFORMATION\_SCHEMA would work properly because different rows in the same column of the results of these operations would be in different character sets.

• Metadata must include all characters in all languages. Otherwise, users would not be able to name columns and tables using their own languages.

To satisfy both requirements, MySQL stores metadata in a Unicode character set, namely UTF-8. This does not cause any disruption if you never use accented or non-Latin characters. But if you do, you should be aware that metadata is in UTF-8.

The metadata requirements mean that the return values of the USER(), CURRENT\_USER(), SESSION\_USER(), SYSTEM\_USER(), DATABASE(), and VERSION() functions have the UTF-8 character set by default.

The server sets the character\_set\_system system variable to the name of the metadata character set:

mysql> **SHOW** **VARIABLES** **LIKE** **'character\_set\_system';**

+----------------------+---------+

| Variable\_name | Value |

+----------------------+---------+

| character\_set\_system | utf8mb3 |

+----------------------+---------+

Storage of metadata using Unicode does *not* mean that the server returns headers of columns and the results of DESCRIBE functions in the character\_set\_system character set by default. When you use SELECT column1 FROM t, the name column1 itself is returned from the server to the client in the character set determined by the value of the character\_set\_results system variable, which has a default value of utf8mb4. If you want the server to pass metadata results back in a different character set, use the SET NAMES statement to force the server to perform character set conversion. SET NAMES sets the character\_set\_results and other related system variables. (See [Section 10.4, “Connection Character Sets and Collations”](#_bookmark17) .) Alternatively, a client program can perform the conversion after receiving the result from the server. It is more efficient for the client to perform the conversion, but this option is not always available for all clients.

If character\_set\_results is set to NULL, no conversion is performed and the server returns metadata using its original character set (the set indicated by character\_set\_system).

Error messages returned from the server to the client are converted to the client character set automatically, as with metadata.

If you are using (for example) the USER() function for comparison or assignment within a single statement, don't worry. MySQL performs some automatic conversion for you.

SELECT \* FROM t1 WHERE USER() = latin1\_column;

This works because the contents of latin1\_column are automatically converted to UTF-8 before the comparison.

INSERT INTO t1 (latin1\_column) SELECT USER();

This works because the contents of USER() are automatically converted to latin1 before the assignment.

Although automatic conversion is not in the SQL standard, the standard does say that every character set is (in terms of supported characters) a “subset” of Unicode. Because it is a well-known principle that “what applies to a superset can apply to a subset,” we believe that a collation for Unicode can apply for comparisons with non-Unicode strings. For more information about coercion of strings, see [Section 10.8.4, “Collation Coercibility in Expressions”](#_bookmark25) .

**10.3** **Specifying** **Character** **Sets** **and** **Collations**

There are default settings for character sets and collations at four levels: server, database, table, and column. The description in the following sections may appear complex, but it has been found in practice that multiple-level defaulting leads to natural and obvious results.

CHARACTER SET is used in clauses that specify a character set. CHARSET can be used as a synonym for CHARACTER SET.

Character set issues affect not only data storage, but also communication between client programs and the MySQL server. If you want the client program to communicate with the server using a character set different from the default, you'll need to indicate which one. For example, to use the utf8mb4 Unicode character set, issue this statement after connecting to the server:

SET NAMES 'utf8mb4';

For more information about character set-related issues in client/server communication, see [Section 10.4, “Connection Character Sets and Collations”](#_bookmark17) .

**10.3.1** **Collation** **Naming** **Conventions**

MySQL collation names follow these conventions:

• A collation name starts with the name of the character set with which it is associated, generally followed by one or more suffixes indicating other collation characteristics. For example, utf8mb4\_0900\_ai\_ci and latin1\_swedish\_ci are collations for the utf8mb4 and latin1 character sets, respectively. The binary character set has a single collation, also named binary, with no suffixes.

• A language-specific collation includes a locale code or language name. For example, utf8mb4\_tr\_0900\_ai\_ci and utf8mb4\_hu\_0900\_ai\_ci sort characters for the utf8mb4 character set using the rules of Turkish and Hungarian, respectively. utf8mb4\_turkish\_ci and utf8mb4\_hungarian\_ci are similar but based on a less recent version of the Unicode Collation Algorithm.

• Collation suffixes indicate whether a collation is case-sensitive, accent-sensitive, or kana-sensitive (or some combination thereof), or binary. The following table shows the suffixes used to indicate these characteristics.

**Table** **10.1** **Collation** **Suffix** **Meanings**

|  |  |
| --- | --- |
| **Suffix** | **Meaning** |
| \_ai | Accent-insensitive |

|  |  |
| --- | --- |
| **Suffix** | **Meaning** |
| \_as | Accent-sensitive |
| \_ci | Case-insensitive |
| \_cs | Case-sensitive |
| \_ks | Kana-sensitive |
| \_bin | Binary |

For nonbinary collation names that do not specify accent sensitivity, it is determined by case sensitivity. If a collation name does not contain \_ai or \_as, \_ci in the name implies \_ai and \_cs in the name implies \_as. For example, latin1\_general\_ci is explicitly case-insensitive and implicitly accent-insensitive, latin1\_general\_cs is explicitly case-sensitive and implicitly accent- sensitive, and utf8mb4\_0900\_ai\_ci is explicitly case-insensitive and accent-insensitive.

For Japanese collations, the \_ks suffix indicates that a collation is kana-sensitive; that is, it distinguishes Katakana characters from Hiragana characters. Japanese collations without the \_ks suffix are not kana-sensitive and treat Katakana and Hiragana characters equal for sorting.

For the binary collation of the binary character set, comparisons are based on numeric byte values. For the \_bin collation of a nonbinary character set, comparisons are based on numeric character code values, which differ from byte values for multibyte characters. For information about the differences between the binary collation of the binary character set and the \_bin collations of nonbinary character sets, see [Section 10.8.5, “The binary Collation Compared to \_bin Collations”](#_bookmark26) .

• Collation names for Unicode character sets may include a version number to indicate the version of the Unicode Collation Algorithm (UCA) on which the collation is based. UCA-based collations without a version number in the name use the version-4.0.0 UCA weight keys. For example:

• utf8mb4\_0900\_ai\_ci is based on UCA 9.0.0 weight keys ([http://www.unicode.org/Public/](http://www.unicode.org/Public/UCA/9.0.0/allkeys.txt) [UCA/9.0.0/allkeys.txt](http://www.unicode.org/Public/UCA/9.0.0/allkeys.txt)).

• utf8mb4\_unicode\_520\_ci is based on UCA 5.2.0 weight keys ([http://www.unicode.org/Public/](http://www.unicode.org/Public/UCA/5.2.0/allkeys.txt) [UCA/5.2.0/allkeys.txt](http://www.unicode.org/Public/UCA/5.2.0/allkeys.txt)).

• utf8mb4\_unicode\_ci (with no version named) is based on UCA 4.0.0 weight keys ([http://](http://www.unicode.org/Public/UCA/4.0.0/allkeys-4.0.0.txt) [www.unicode.org/Public/UCA/4.0.0/allkeys-4.0.0.txt](http://www.unicode.org/Public/UCA/4.0.0/allkeys-4.0.0.txt)).

• For Unicode character sets, the *xxx*\_general\_mysql500\_ci collations preserve the pre-5.1.24 ordering of the original *xxx*\_general\_ci collations and permit upgrades for tables created before MySQL 5.1.24 (Bug #27877).

**10.3.2** **Server** **Character** **Set** **and** **Collation**

MySQL Server has a server character set and a server collation. By default, these are utf8mb4 and utf8mb4\_0900\_ai\_ci, but they can be set explicitly at server startup on the command line or in an option file and changed at runtime.

Initially, the server character set and collation depend on the options that you use when you start mysqld. You can use --character-set-server for the character set. Along with it, you can add --collation-server for the collation. If you don't specify a character set, that is the same as saying --character-set-server=utf8mb4. If you specify only a character set (for example, utf8mb4) but not a collation, that is the same as saying --character-set-server=utf8mb4 --collation-server=utf8mb4\_0900\_ai\_ci because utf8mb4\_0900\_ai\_ci is the default collation for utf8mb4. Therefore, the following three commands all have the same effect:

mysqld

mysqld --character-set-server=utf8mb4

mysqld --character-set-server=utf8mb4 \

--collation-server=utf8mb4\_0900\_ai\_ci

One way to change the settings is by recompiling. To change the default server character set and collation when building from sources, use the DEFAULT\_CHARSET and DEFAULT\_COLLATION options for CMake. For example:

cmake . -DDEFAULT\_CHARSET=latin1

Or:

cmake . -DDEFAULT\_CHARSET=latin1 \

-DDEFAULT\_COLLATION=latin1\_german1\_ci

Both mysqld and CMake verify that the character set/collation combination is valid. If not, each program displays an error message and terminates.

The server character set and collation are used as default values if the database character set and collation are not specified in CREATE DATABASE statements. They have no other purpose.

The current server character set and collation can be determined from the values of the character\_set\_server and collation\_server system variables. These variables can be changed at runtime.

**10.3.3** **Database** **Character** **Set** **and** **Collation**

Every database has a database character set and a database collation. The CREATE DATABASE and ALTER DATABASE statements have optional clauses for specifying the database character set and collation:

CREATE DATABASE *db\_name*

[[DEFAULT] CHARACTER SET *charset\_name*]

[[DEFAULT] COLLATE *collation\_name*]

ALTER DATABASE *db\_name*

[[DEFAULT] CHARACTER SET *charset\_name*]

[[DEFAULT] COLLATE *collation\_name*]

The keyword SCHEMA can be used instead of DATABASE.

The CHARACTER SET and COLLATE clauses make it possible to create databases with different character sets and collations on the same MySQL server.

Database options are stored in the data dictionary and can be examined by checking the Information Schema SCHEMATA table.

Example:

CREATE DATABASE *db\_name* CHARACTER SET latin1 COLLATE latin1\_swedish\_ci;

MySQL chooses the database character set and database collation in the following manner:

• If both CHARACTER SET *charset\_name* and COLLATE *collation\_name* are specified, character set *charset\_name* and collation *collation\_name* are used.

• If CHARACTER SET *charset\_name* is specified without COLLATE, character set *charset\_name* and its default collation are used. To see the default collation for each character set, use the SHOW CHARACTER SET statement or query the INFORMATION\_SCHEMA CHARACTER\_SETS table.

• If COLLATE *collation\_name* is specified without CHARACTER SET, the character set associated with *collation\_name* and collation *collation\_name* are used.

• Otherwise (neither CHARACTER SET nor COLLATE is specified), the server character set and server collation are used.

The character set and collation for the default database can be determined from the values of the character\_set\_database and collation\_database system variables. The server sets these variables whenever the default database changes. If there is no default database, the variables have the same value as the corresponding server-level system variables, character\_set\_server and collation\_server.

To see the default character set and collation for a given database, use these statements:

USE *db\_name*;

SELECT @@character\_set\_database, @@collation\_database;

Alternatively, to display the values without changing the default database:

SELECT DEFAULT\_CHARACTER\_SET\_NAME, DEFAULT\_COLLATION\_NAME

FROM INFORMATION\_SCHEMA.SCHEMATA WHERE SCHEMA\_NAME = '*db\_name* ';

The database character set and collation affect these aspects of server operation:

• For CREATE TABLE statements, the database character set and collation are used as default values for table definitions if the table character set and collation are not specified. To override this, provide explicit CHARACTER SET and COLLATE table options.

• For LOAD DATA statements that include no CHARACTER SET clause, the server uses the character set indicated by the character\_set\_database system variable to interpret the information in the file. To override this, provide an explicit CHARACTER SET clause.

• For stored routines (procedures and functions), the database character set and collation in effect at routine creation time are used as the character set and collation of character data parameters for which the declaration includes no CHARACTER SET or a COLLATE attribute. To override this, provide CHARACTER SET and COLLATE explicitly.

**10.3.4** **Table** **Character** **Set** **and** **Collation**

Every table has a table character set and a table collation. The CREATE TABLE and ALTER TABLE statements have optional clauses for specifying the table character set and collation:

CREATE TABLE *tbl\_name* (*column\_list*)

[[DEFAULT] CHARACTER SET *charset\_name*]

[COLLATE *collation\_name*]]

ALTER TABLE *tbl\_name*

[[DEFAULT] CHARACTER SET *charset\_name*]

[COLLATE *collation\_name*]

Example:

CREATE TABLE t1 ( ... )

CHARACTER SET latin1 COLLATE latin1\_danish\_ci;

MySQL chooses the table character set and collation in the following manner:

• If both CHARACTER SET *charset\_name* and COLLATE *collation\_name* are specified, character set *charset\_name* and collation *collation\_name* are used.

• If CHARACTER SET *charset\_name* is specified without COLLATE, character set *charset\_name* and its default collation are used. To see the default collation for each character set, use the SHOW CHARACTER SET statement or query the INFORMATION\_SCHEMA CHARACTER\_SETS table.

• If COLLATE *collation\_name* is specified without CHARACTER SET, the character set associated with *collation\_name* and collation *collation\_name* are used.

• Otherwise (neither CHARACTER SET nor COLLATE is specified), the database character set and collation are used.

The table character set and collation are used as default values for column definitions if the column character set and collation are not specified in individual column definitions. The table character set and collation are MySQL extensions; there are no such things in standard SQL.

**10.3.5** **Column** **Character** **Set** **and** **Collation**

Every “character” column (that is, a column of type [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), a [TEXT](#_bookmark61) type, or any synonym) has a column character set and a column collation. Column definition syntax for CREATE TABLE and ALTER TABLE has optional clauses for specifying the column character set and collation:

*col\_name* {CHAR | VARCHAR | TEXT} (*col\_length*)

[CHARACTER SET *charset\_name*]

[COLLATE *collation\_name*]

These clauses can also be used for [ENUM](#_bookmark62) and [SET](#_bookmark63) columns:

*col\_name* {ENUM | SET} (*val\_list*)

[CHARACTER SET *charset\_name*]

[COLLATE *collation\_name*]

Examples:

CREATE TABLE t1

(

col1 VARCHAR(5)

CHARACTER SET latin1

COLLATE latin1\_german1\_ci

);

ALTER TABLE t1 MODIFY

col1 VARCHAR(5)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci;

MySQL chooses the column character set and collation in the following manner:

• If both CHARACTER SET *charset\_name* and COLLATE *collation\_name* are specified, character set *charset\_name* and collation *collation\_name* are used.

CREATE TABLE t1

(

col1 CHAR(10) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci

) CHARACTER SET latin1 COLLATE latin1\_bin;

The character set and collation are specified for the column, so they are used. The column has character set utf8mb4 and collation utf8mb4\_unicode\_ci.

• If CHARACTER SET *charset\_name* is specified without COLLATE, character set *charset\_name* and its default collation are used.

CREATE TABLE t1

(

col1 CHAR(10) CHARACTER SET utf8mb4

) CHARACTER SET latin1 COLLATE latin1\_bin;

The character set is specified for the column, but the collation is not. The column has character set utf8mb4 and the default collation for utf8mb4, which is utf8mb4\_0900\_ai\_ci. To see the default collation for each character set, use the SHOW CHARACTER SET statement or query the INFORMATION\_SCHEMA CHARACTER\_SETS table.

• If COLLATE *collation\_name* is specified without CHARACTER SET, the character set associated with *collation\_name* and collation *collation\_name* are used.

CREATE TABLE t1

(

col1 CHAR(10) COLLATE utf8mb4\_polish\_ci

) CHARACTER SET latin1 COLLATE latin1\_bin;

The collation is specified for the column, but the character set is not. The column has collation utf8mb4\_polish\_ci and the character set is the one associated with the collation, which is

utf8mb4.

• Otherwise (neither CHARACTER SET nor COLLATE is specified), the table character set and collation are used.

CREATE TABLE t1

(

col1 CHAR(10)

) CHARACTER SET latin1 COLLATE latin1\_bin;

Neither the character set nor collation is specified for the column, so the table defaults are used. The column has character set latin1 and collation latin1\_bin.

The CHARACTER SET and COLLATE clauses are standard SQL.

If you use ALTER TABLE to convert a column from one character set to another, MySQL attempts to map the data values, but if the character sets are incompatible, there may be data loss.

**10.3.6** **Character** **String** **Literal** **Character** **Set** **and** **Collation**

Every character string literal has a character set and a collation.

For the simple statement SELECT '*string*', the string has the connection default character set and collation defined by the character\_set\_connection and collation\_connection system variables.

A character string literal may have an optional character set introducer and COLLATE clause, to designate it as a string that uses a particular character set and collation:

[\_*charset\_name*]'*string*' [COLLATE *collation\_name*]

The \_*charset\_name* expression is formally called an *introducer*. It tells the parser, “the string that follows uses character set *charset\_name*.” An introducer does not change the string to the introducer character set like CONVERT() would do. It does not change the string value, although padding may occur. The introducer is just a signal. See [Section 10.3.8, “Character Set Introducers”](#_bookmark14) .

Examples:

SELECT 'abc';

SELECT \_latin1'abc';

SELECT \_binary'abc';

SELECT \_utf8mb4'abc' COLLATE utf8mb4\_danish\_ci;

Character set introducers and the COLLATE clause are implemented according to standard SQL specifications.

MySQL determines the character set and collation of a character string literal in the following manner:

• If both *\_charset\_name* and COLLATE *collation\_name* are specified, character set *charset\_name* and collation *collation\_name* are used. *collation\_name* must be a permitted collation for *charset\_name*.

• If *\_charset\_name* is specified but COLLATE is not specified, character set *charset\_name* and its default collation are used. To see the default collation for each character set, use the SHOW CHARACTER SET statement or query the INFORMATION\_SCHEMA CHARACTER\_SETS table.

• If *\_charset\_name* is not specified but COLLATE *collation\_name* is specified, the connection default character set given by the character\_set\_connection system variable and collation

*collation\_name* are used. *collation\_name* must be a permitted collation for the connection default character set.

• Otherwise (neither *\_charset\_name* nor COLLATE *collation\_name* is specified), the connection default character set and collation given by the character\_set\_connection and

collation\_connection system variables are used.

Examples:

• A nonbinary string with latin1 character set and latin1\_german1\_ci collation: SELECT \_latin1'Müller' COLLATE latin1\_german1\_ci;

• A nonbinary string with utf8mb4 character set and its default collation (that is, utf8mb4\_0900\_ai\_ci):

SELECT \_utf8mb4'Müller';

• A binary string with binary character set and its default collation (that is, binary): SELECT \_binary'Müller';

• A nonbinary string with the connection default character set and utf8mb4\_0900\_ai\_ci collation (fails if the connection character set is not utf8mb4):

SELECT 'Müller' COLLATE utf8mb4\_0900\_ai\_ci;

• A string with the connection default character set and collation: SELECT 'Müller';

An introducer indicates the character set for the following string, but does not change how the parser performs escape processing within the string. Escapes are always interpreted by the parser according to the character set given by character\_set\_connection.

The following examples show that escape processing occurs using character\_set\_connection even in the presence of an introducer. The examples use SET NAMES (which changes character\_set\_connection, as discussed in [Section 10.4, “Connection Character Sets and](#_bookmark17) [Collations”](#_bookmark17)), and display the resulting strings using the HEX() function so that the exact string contents can be seen.

Example 1:

mysql> **SET** **NAMES** **latin1;**

mysql> **SELECT** **HEX('à\n'),** **HEX(\_sjis'à\n');**

+------------+-----------------+

| HEX('à\n') | HEX(\_sjis'à\n') |

+------------+-----------------+

|  |  |  |  |
| --- | --- | --- | --- |
| | E00A | | | E00A | | |

+------------+-----------------+

Here, à (hexadecimal value E0) is followed by \n, the escape sequence for newline. The escape sequence is interpreted using the character\_set\_connection value of latin1 to produce a literal newline (hexadecimal value 0A). This happens even for the second string. That is, the \_sjis introducer does not affect the parser's escape processing.

Example 2:

mysql> **SET** **NAMES** **sjis;**

mysql> **SELECT** **HEX('à\n'),** **HEX(\_latin1'à\n');**

+------------+-------------------+

| HEX('à\n') | HEX(\_latin1'à\n') |

+------------+-------------------+

|  |  |  |  |
| --- | --- | --- | --- |
| | E05C6E | | | E05C6E | | |

+------------+-------------------+

Here, character\_set\_connection is sjis, a character set in which the sequence of à followed by \ (hexadecimal values 05 and 5C) is a valid multibyte character. Hence, the first two bytes of the string are interpreted as a single sjis character, and the \ is not interpreted as an escape character. The following n (hexadecimal value 6E) is not interpreted as part of an escape sequence. This is true even for the second string; the \_latin1 introducer does not affect escape processing.

**10.3.7** **The** **National** **Character** **Set**

Standard SQL defines [NCHAR](#_bookmark60) or [NATIONAL CHAR](#_bookmark60) as a way to indicate that a [CHAR](#_bookmark60) column should use some predefined character set. MySQL uses utf8 as this predefined character set. For example, these data type declarations are equivalent:

CHAR(10) CHARACTER SET utf8

NATIONAL CHARACTER(10)

NCHAR(10)

As are these:

VARCHAR(10) CHARACTER SET utf8

NATIONAL VARCHAR(10)

NVARCHAR(10)

NCHAR VARCHAR(10)

NATIONAL CHARACTER VARYING(10)

NATIONAL CHAR VARYING(10)

You can use N'*literal*' (or n'*literal*') to create a string in the national character set. These statements are equivalent:

SELECT N'some text';

SELECT n'some text';

SELECT \_utf8'some text';

MySQL 8.0 interprets the national character set as utf8mb3, which is now deprecated. Thus, using NATIONAL CHARACTER or one of its synonyms to define the character set for a database, table, or column raises a warning similar to this one:

NATIONAL/NCHAR/NVARCHAR implies the character set UTF8MB3, which will be replaced by UTF8MB4 in a future release. Please consider using CHAR(x) CHARACTER SET UTF8MB4 in order to be unambiguous.

**10.3.8** **Character** **Set** **Introducers**

A character string literal, hexadecimal literal, or bit-value literal may have an optional character set introducer and COLLATE clause, to designate it as a string that uses a particular character set and collation:

[\_*charset\_name*] *literal* [COLLATE *collation\_name*]

The \_*charset\_name* expression is formally called an *introducer*. It tells the parser, “the string that follows uses character set *charset\_name*.” An introducer does not change the string to the introducer character set like CONVERT() would do. It does not change the string value, although padding may occur. The introducer is just a signal.

For character string literals, space between the introducer and the string is permitted but optional.

For character set literals, an introducer indicates the character set for the following string, but does not change how the parser performs escape processing within the string. Escapes are always interpreted by the parser according to the character set given by character\_set\_connection. For additional discussion and examples, see [Section 10.3.6, “Character String Literal Character Set and Collation”](#_bookmark12) .

Examples:

SELECT 'abc';

SELECT \_latin1'abc';

SELECT \_binary'abc';

SELECT \_utf8mb4'abc' COLLATE utf8mb4\_danish\_ci;

SELECT \_latin1 X'4D7953514C';

SELECT \_utf8mb4 0x4D7953514C COLLATE utf8mb4\_danish\_ci;

SELECT \_latin1 b'1000001';

SELECT \_utf8mb4 0b1000001 COLLATE utf8mb4\_danish\_ci;

Character set introducers and the COLLATE clause are implemented according to standard SQL specifications.

Character string literals can be designated as binary strings by using the \_binary introducer. Hexadecimal literals and bit-value literals are binary strings by default, so \_binary is permitted, but normally unnecessary. \_binary may be useful to preserve a hexadecimal or bit literal as a binary string in contexts for which the literal is otherwise treated as a number. For example, bit operations permit numeric or binary string arguments in MySQL 8.0 and higher, but treat hexadecimal and bit literals as numbers by default. To explicitly specify binary string context for such literals, use a \_binary introducer for at least one of the arguments:

mysql> **SET** **@v1** **=** **X'000D'** **|** **X'0BC0';**

mysql> **SET** **@v2** **=** **\_binary** **X'000D'** **|** **X'0BC0';**

mysql> **SELECT** **HEX(@v1),** **HEX(@v2);**

+----------+----------+

| HEX(@v1) | HEX(@v2) |

+----------+----------+

|  |  |  |  |
| --- | --- | --- | --- |
| | BCD | | | 0BCD | | |

+----------+----------+

The displayed result appears similar for both bit operations, but the result without \_binary is a BIGINT value, whereas the result with \_binary is a binary string. Due to the difference in result types, the displayed values differ: High-order 0 digits are not displayed for the numeric result.

MySQL determines the character set and collation of a character string literal, hexadecimal literal, or bit-value literal in the following manner:

• If both *\_charset\_name* and COLLATE *collation\_name* are specified, character set *charset\_name* and collation *collation\_name* are used. *collation\_name* must be a permitted collation for *charset\_name*.

• If *\_charset\_name* is specified but COLLATE is not specified, character set *charset\_name* and its default collation are used. To see the default collation for each character set, use the SHOW CHARACTER SET statement or query the INFORMATION\_SCHEMA CHARACTER\_SETS table.

• If *\_charset\_name* is not specified but COLLATE *collation\_name* is specified:

• For a character string literal, the connection default character set given by the character\_set\_connection system variable and collation *collation\_name* are used. *collation\_name* must be a permitted collation for the connection default character set.

• For a hexadecimal literal or bit-value literal, the only permitted collation is binary because these types of literals are binary strings by default.

• Otherwise (neither *\_charset\_name* nor COLLATE *collation\_name* is specified):

• For a character string literal, the connection default character set and collation given by the character\_set\_connection and collation\_connection system variables are used.

• For a hexadecimal literal or bit-value literal, the character set and collation are binary.

Examples:

• Nonbinary strings with latin1 character set and latin1\_german1\_ci collation:

SELECT \_latin1'Müller' COLLATE latin1\_german1\_ci;

SELECT \_latin1 X'0A0D' COLLATE latin1\_german1\_ci;

SELECT \_latin1 b'0110' COLLATE latin1\_german1\_ci;

• Nonbinary strings with utf8mb4 character set and its default collation (that is, utf8mb4\_0900\_ai\_ci):

SELECT \_utf8mb4'Müller';

SELECT \_utf8mb4 X'0A0D';

SELECT \_utf8mb4 b'0110';

• Binary strings with binary character set and its default collation (that is, binary):

SELECT \_binary'Müller';

SELECT X'0A0D';

SELECT b'0110';

The hexadecimal literal and bit-value literal need no introducer because they are binary strings by default.

• A nonbinary string with the connection default character set and utf8mb4\_0900\_ai\_ci collation (fails if the connection character set is not utf8mb4):

SELECT 'Müller' COLLATE utf8mb4\_0900\_ai\_ci;

This construction (COLLATE only) does not work for hexadecimal literals or bit literals because their character set is binary no matter the connection character set, and binary is not compatible with the utf8mb4\_0900\_ai\_ci collation. The only permitted COLLATE clause in the absence of an introducer is COLLATE binary.

• A string with the connection default character set and collation:

SELECT 'Müller';

**10.3.9** **Examples** **of** **Character** **Set** **and** **Collation** **Assignment**

The following examples show how MySQL determines default character set and collation values.

**Example** **1:** **Table** **and** **Column** **Definition**

CREATE TABLE t1

(

c1 CHAR(10) CHARACTER SET latin1 COLLATE latin1\_german1\_ci

) DEFAULT CHARACTER SET latin2 COLLATE latin2\_bin;

Here we have a column with a latin1 character set and a latin1\_german1\_ci collation. The definition is explicit, so that is straightforward. Notice that there is no problem with storing a latin1 column in a latin2 table.

**Example** **2:** **Table** **and** **Column** **Definition**

CREATE TABLE t1

(

c1 CHAR(10) CHARACTER SET latin1

) DEFAULT CHARACTER SET latin1 COLLATE latin1\_danish\_ci;

This time we have a column with a latin1 character set and a default collation. Although it might seem natural, the default collation is not taken from the table level. Instead, because the default collation for latin1 is always latin1\_swedish\_ci, column c1 has a collation of latin1\_swedish\_ci (not latin1\_danish\_ci).

**Example** **3:** **Table** **and** **Column** **Definition**

CREATE TABLE t1

(

c1 CHAR(10)

) DEFAULT CHARACTER SET latin1 COLLATE latin1\_danish\_ci;

We have a column with a default character set and a default collation. In this circumstance, MySQL checks the table level to determine the column character set and collation. Consequently, the character set for column c1 is latin1 and its collation is latin1\_danish\_ci.

**Example** **4:** **Database,** **Table,** **and** **Column** **Definition**

CREATE DATABASE d1

DEFAULT CHARACTER SET latin2 COLLATE latin2\_czech\_cs;

USE d1;

CREATE TABLE t1

(

c1 CHAR(10)

);

We create a column without specifying its character set and collation. We're also not specifying a character set and a collation at the table level. In this circumstance, MySQL checks the database level to determine the table settings, which thereafter become the column settings.) Consequently, the character set for column c1 is latin2 and its collation is latin2\_czech\_cs.

**10.3.10** **Compatibility** **with** **Other** **DBMSs**

For MaxDB compatibility these two statements are the same:

CREATE TABLE t1 (f1 CHAR(*N*) UNICODE);

CREATE TABLE t1 (f1 CHAR(*N*) CHARACTER SET ucs2);

Both the UNICODE attribute and the ucs2 character set are deprecated in MySQL 8.0.28.

**10.4** **Connection** **Character** **Sets** **and** **Collations**

A “connection” is what a client program makes when it connects to the server, to begin a session within which it interacts with the server. The client sends SQL statements, such as queries, over the session connection. The server sends responses, such as result sets or error messages, over the connection back to the client.

• [Connection Character Set and Collation System Variables](#_bookmark64)

• [Impermissible Client Character Sets](#_bookmark65)

• [Client Program Connection Character Set Configuration](#_bookmark66)

• [SQL Statements for Connection Character Set Configuration](#_bookmark67)

• [Connection Character Set Error Handling](#_bookmark68)

**Connection** **Character** **Set** **and** **Collation** **System** **Variables**

Several character set and collation system variables relate to a client's interaction with the server. Some of these have been mentioned in earlier sections:

• The character\_set\_server and collation\_server system variables indicate the server character set and collation. See [Section 10.3.2, “Server Character Set and Collation”](#_bookmark8) .

• The character\_set\_database and collation\_database system variables indicate the character set and collation of the default database. See [Section 10.3.3, “Database Character Set and](#_bookmark9) [Collation”](#_bookmark9) .

Additional character set and collation system variables are involved in handling traffic for the connection between a client and the server. Every client has session-specific connection-related character set and collation system variables. These session system variable values are initialized at connect time, but can be changed within the session.



Several questions about character set and collation handling for client connections can be answered in terms of system variables:

• What character set are statements in when they leave the client?

The server takes the character\_set\_client system variable to be the character set in which statements are sent by the client.

**Note**

Some character sets cannot be used as the client character set. See [Impermissible Client Character Sets](#_bookmark65).

• What character set should the server translate statements to after receiving them?

To determine this, the server uses the character\_set\_connection and

collation\_connection system variables:

• The server converts statements sent by the client from character\_set\_client to character\_set\_connection. Exception: For string literals that have an introducer such as \_utf8mb4 or \_latin2, the introducer determines the character set. See [Section 10.3.8,](#_bookmark14) [“Character Set Introducers”](#_bookmark14) .

• collation\_connection is important for comparisons of literal strings. For comparisons of strings with column values, collation\_connection does not matter because columns have their own collation, which has a higher collation precedence (see [Section 10.8.4, “Collation](#_bookmark25) [Coercibility in Expressions”](#_bookmark25)).

• What character set should the server translate query results to before shipping them back to the client?

The character\_set\_results system variable indicates the character set in which the server returns query results to the client. This includes result data such as column values, result metadata such as column names, and error messages.

To tell the server to perform no conversion of result sets or error messages, set character\_set\_results to NULL or binary:

SET character\_set\_results = NULL;

SET character\_set\_results = binary;

For more information about character sets and error messages, see [Section 10.6, “Error Message](#_bookmark19) [Character Set”](#_bookmark19) .

To see the values of the character set and collation system variables that apply to the current session, use this statement:

SELECT \* FROM performance\_schema.session\_variables

WHERE VARIABLE\_NAME IN (

'character\_set\_client', 'character\_set\_connection',

'character\_set\_results', 'collation\_connection'

) ORDER BY VARIABLE\_NAME;

The following simpler statements also display the connection variables, but include other related variables as well. They can be useful to see *all* character set and collation system variables:

SHOW SESSION VARIABLES LIKE 'character\\_set\\_%';

SHOW SESSION VARIABLES LIKE 'collation\\_%';

Clients can fine-tune the settings for these variables, or depend on the defaults (in which case, you can

skip the rest of this section). If you do not use the defaults, you must change the character settings *for* *each* *connection* *to* *the* *server*.

**Impermissible** **Client** **Character** **Sets**

The character\_set\_client system variable cannot be set to certain character sets:

ucs2

utf16

utf16le

utf32

Attempting to use any of those character sets as the client character set produces an error:

mysql> **SET** **character\_set\_client** **=** **'ucs2';**

ERROR 1231 (42000): Variable 'character\_set\_client'

can't be set to the value of 'ucs2'

The same error occurs if any of those character sets are used in the following contexts, all of which result in an attempt to set character\_set\_client to the named character set:

• The --default-character-set=*charset\_name* command option used by MySQL client programs such as mysql and mysqladmin.

• The SET NAMES '*charset\_name*' statement.

• The SET CHARACTER SET '*charset\_name*' statement.

**Client** **Program** **Connection** **Character** **Set** **Configuration**

When a client connects to the server, it indicates which character set it wants to use for communication with the server. (Actually, the client indicates the default collation for that character set, from

which the server can determine the character set.) The server uses this information to set the character\_set\_client, character\_set\_results, character\_set\_connection system variables to the character set, and collation\_connection to the character set default collation. In effect, the server performs the equivalent of a SET NAMES operation.

If the server does not support the requested character set or collation, it falls back to using the server character set and collation to configure the connection. For additional detail about this fallback behavior, see [Connection Character Set Error Handling](#_bookmark68).

The mysql, mysqladmin, mysqlcheck, mysqlimport, and mysqlshow client programs determine the default character set to use as follows:

• In the absence of other information, each client uses the compiled-in default character set, usually utf8mb4.

• Each client can autodetect which character set to use based on the operating system setting, such as the value of the LANG or LC\_ALL locale environment variable on Unix systems or the code page setting on Windows systems. For systems on which the locale is available from the OS, the client uses it to set the default character set rather than using the compiled-in default. For example, setting LANG to ru\_RU.KOI8-R causes the koi8r character set to be used. Thus, users can configure the locale in their environment for use by MySQL clients.

The OS character set is mapped to the closest MySQL character set if there is no exact match. If the client does not support the matching character set, it uses the compiled-in default. For example, utf8 and utf-8 map to utf8mb4, and ucs2 is not supported as a connection character set, so it maps to the compiled-in default.

C applications can use character set autodetection based on the OS setting by invoking [mysql\_options()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-options.html) as follows before connecting to the server:

mysql\_options(mysql,

MYSQL\_SET\_CHARSET\_NAME,

MYSQL\_AUTODETECT\_CHARSET\_NAME);

• Each client supports a --default-character-set option, which enables users to specify the character set explicitly to override whatever default the client otherwise determines.



**Note**

Some character sets cannot be used as the client character set. Attempting to use them with --default-character-set produces an error. See [Impermissible Client Character Sets](#_bookmark65).

With the mysql client, to use a character set different from the default, you could explicitly execute a SET NAMES statement every time you connect to the server (see [Client Program Connection Character](#_bookmark66) [Set Configuration](#_bookmark66)). To accomplish the same result more easily, specify the character set in your option file. For example, the following option file setting changes the three connection-related character set system variables set to koi8r each time you invoke mysql:

[mysql]

default-character-set=koi8r

If you are using the mysql client with auto-reconnect enabled (which is not recommended), it is preferable to use the charset command rather than SET NAMES. For example:

mysql> **charset** **koi8r**

Charset changed

The charset command issues a SET NAMES statement, and also changes the default character set that mysql uses when it reconnects after the connection has dropped.

When configuration client programs, you must also consider the environment within which they execute. See [Section 10.5, “Configuring Application Character Set and Collation”](#_bookmark18) .

**SQL** **Statements** **for** **Connection** **Character** **Set** **Configuration**

After a connection has been established, clients can change the character set and collation system variables for the current session. These variables can be changed individually using SET statements, but two more convenient statements affect the connection-related character set system variables as a group:

• SET NAMES '*charset\_name*' [COLLATE '*collation\_name*']

SET NAMES indicates what character set the client uses to send SQL statements to the server. Thus, SET NAMES 'cp1251' tells the server, “future incoming messages from this client are in character set cp1251.” It also specifies the character set that the server should use for sending results back to the client. (For example, it indicates what character set to use for column values if you use a SELECT statement that produces a result set.)

A SET NAMES '*charset\_name*' statement is equivalent to these three statements:

SET character\_set\_client = *charset\_name*;

SET character\_set\_results = *charset\_name*;

SET character\_set\_connection = *charset\_name*;

Setting character\_set\_connection to *charset\_name* also implicitly sets collation\_connection to the default collation for *charset\_name*. It is unnecessary to set that collation explicitly. To specify a particular collation to use for collation\_connection, add a COLLATE clause:

SET NAMES '*charset\_name* ' COLLATE '*collation\_name* '

• SET CHARACTER SET '*charset\_name*'

SET CHARACTER SET is similar to SET NAMES but sets character\_set\_connection and collation\_connection to character\_set\_database and collation\_database (which, as mentioned previously, indicate the character set and collation of the default database).

A SET CHARACTER SET *charset\_name* statement is equivalent to these three statements:



SET character\_set\_client = *charset\_name*;

SET character\_set\_results = *charset\_name*;

SET collation\_connection = @@collation\_database;

Setting collation\_connection also implicitly sets character\_set\_connection to the character set associated with the collation (equivalent to executing SET character\_set\_connection = @@character\_set\_database). It is unnecessary to set character\_set\_connection explicitly.

**Note**

Some character sets cannot be used as the client character set. Attempting to use them with SET NAMES or SET CHARACTER SET produces an error. See [Impermissible Client Character Sets](#_bookmark65).

Example: Suppose that column1 is defined as CHAR(5) CHARACTER SET latin2. If you do not say SET NAMES or SET CHARACTER SET, then for SELECT column1 FROM t, the server sends back all the values for column1 using the character set that the client specified when it connected. On the other hand, if you say SET NAMES 'latin1' or SET CHARACTER SET 'latin1' before issuing the SELECT statement, the server converts the latin2 values to latin1just before sending results back. Conversion may be lossy for characters that are not in both character sets.

**Connection** **Character** **Set** **Error** **Handling**

Attempts to use an inappropriate connection character set or collation can produce an error, or cause the server to fall back to its default character set and collation for a given connection. This section describes problems that can occur when configuring the connection character set. These problems can occur when establishing a connection or when changing the character set within an established connection.

• [Connect-Time Error Handling](#_bookmark69)

• [Runtime Error Handling](#_bookmark70)

**Connect-Time** **Error** **Handling**

Some character sets cannot be used as the client character set; see [Impermissible Client Character](#_bookmark65) [Sets](#_bookmark65). If you specify a character set that is valid but not permitted as a client character set, the server returns an error:

$> **mysql** **--default-character-set=ucs2**

ERROR 1231 (42000): Variable 'character\_set\_client' can't be set to

the value of 'ucs2'

If you specify a character set that the client does not recognize, it produces an error:

$> **mysql** **--default-character-set=bogus**

mysql: Character set 'bogus' is not a compiled character set and is

not specified in the '/usr/local/mysql/share/charsets/Index .xml' file

ERROR 2019 (HY000): Can't initialize character set bogus

(path: /usr/local/mysql/share/charsets/)

If you specify a character set that the client recognizes but the server does not, the server falls back to its default character set and collation. Suppose that the server is configured to use latin1 and latin1\_swedish\_ci as its defaults, and that it does not recognize gb18030 as a valid character set. A client that specifies --default-character-set=gb18030 is able to connect to the server, but the resulting character set is not what the client wants:

mysql> **SHOW** **SESSION** **VARIABLES** **LIKE** **'character\\_set\\_%';**

+--------------------------+--------+

| Variable\_name | Value |

+--------------------------+--------+

| character\_set\_client | latin1 |

| character\_set\_connection | latin1 |

...

| character\_set\_results | latin1 |

...

+--------------------------+--------+

mysql> **SHOW** **SESSION** **VARIABLES** **LIKE** **'collation\_connection';**

+----------------------+-------------------+

| Variable\_name | Value |

+----------------------+-------------------+

| collation\_connection | latin1\_swedish\_ci |

+----------------------+-------------------+

You can see that the connection system variables have been set to reflect a character set and collation of latin1 and latin1\_swedish\_ci. This occurs because the server cannot satisfy the client character set request and falls back to its defaults.

In this case, the client cannot use the character set that it wants because the server does not support it. The client must either be willing to use a different character set, or connect to a different server that supports the desired character set.

The same problem occurs in a more subtle context: When the client tells the server to use a character set that the server recognizes, but the default collation for that character set on the client side is not known on the server side. This occurs, for example, when a MySQL 8.0 client wants to connect to a MySQL 5.7 server using utf8mb4 as the client character set. A client that specifies --default- character-set=utf8mb4 is able to connect to the server. However, as in the previous example, the server falls back to its default character set and collation, not what the client requested:

mysql> **SHOW** **SESSION** **VARIABLES** **LIKE** **'character\\_set\\_%';**

+--------------------------+--------+

|  |  |
| --- | --- |
| | Variable\_name | | Value | |

+--------------------------+--------+

| character\_set\_client | latin1 |

| character\_set\_connection | latin1 |

...

| character\_set\_results | latin1 |

...

+--------------------------+--------+

mysql> **SHOW** **SESSION** **VARIABLES** **LIKE** **'collation\_connection';**

+----------------------+-------------------+

| Variable\_name | Value |

+----------------------+-------------------+

| collation\_connection | latin1\_swedish\_ci |

+----------------------+-------------------+

Why does this occur? After all, utf8mb4 is known to the 8.0 client and the 5.7 server, so both of them recognize it. To understand this behavior, it is necessary to understand that when the client tells the server which character set it wants to use, it really tells the server the default collation for that character set. Therefore, the aforementioned behavior occurs due to a combination of factors:

• The default collation for utf8mb4 differs between MySQL 5.7 and 8.0 (utf8mb4\_general\_ci for 5.7, utf8mb4\_0900\_ai\_ci for 8.0).

• When the 8.0 client requests a character set of utf8mb4, what it sends to the server is the default 8.0 utf8mb4 collation; that is, the utf8mb4\_0900\_ai\_ci.

• utf8mb4\_0900\_ai\_ci is implemented only as of MySQL 8.0, so the 5.7 server does not recognize it.

• Because the 5.7 server does not recognize utf8mb4\_0900\_ai\_ci, it cannot satisfy the client character set request, and falls back to its default character set and collation (latin1 and latin1\_swedish\_ci).

In this case, the client can still use utf8mb4 by issuing a SET NAMES 'utf8mb4' statement

after connecting. The resulting collation is the 5.7 default utf8mb4 collation; that is, utf8mb4\_general\_ci. If the client additionally wants a collation of utf8mb4\_0900\_ai\_ci, it cannot achieve that because the server does not recognize that collation. The client must either be willing to use a different utf8mb4 collation, or connect to a server from MySQL 8.0 or higher.



**Runtime** **Error** **Handling**

Within an established connection, the client can request a change of connection character set and collation with SET NAMES or SET CHARACTER SET.

Some character sets cannot be used as the client character set; see [Impermissible Client Character](#_bookmark65) [Sets](#_bookmark65). If you specify a character set that is valid but not permitted as a client character set, the server returns an error:

mysql> **SET** **NAMES** **'ucs2';**

ERROR 1231 (42000): Variable 'character\_set\_client' can't be set to

the value of 'ucs2'

If the server does not recognize the character set (or the collation), it produces an error:

mysql> **SET** **NAMES** **'bogus';**

ERROR 1115 (42000): Unknown character set: 'bogus'

mysql> **SET** **NAMES** **'utf8mb4'** **COLLATE** **'bogus';**

ERROR 1273 (HY000): Unknown collation: 'bogus'

**Tip**

A client that wants to verify whether its requested character set was honored by the server can execute the following statement after connecting and checking that the result is the expected character set:

SELECT @@character\_set\_client;

**10.5** **Configuring** **Application** **Character** **Set** **and** **Collation**

For applications that store data using the default MySQL character set and collation (utf8mb4, utf8mb4\_0900\_ai\_ci), no special configuration should be needed. If applications require data storage using a different character set or collation, you can configure character set information several ways:

• Specify character settings per database. For example, applications that use one database might use the default of utf8mb4, whereas applications that use another database might use sjis.

• Specify character settings at server startup. This causes the server to use the given settings for all applications that do not make other arrangements.

• Specify character settings at configuration time, if you build MySQL from source. This causes the server to use the given settings as the defaults for all applications, without having to specify them at server startup.

When different applications require different character settings, the per-database technique provides a good deal of flexibility. If most or all applications use the same character set, specifying character settings at server startup or configuration time may be most convenient.

For the per-database or server-startup techniques, the settings control the character set for data storage. Applications must also tell the server which character set to use for client/server communications, as described in the following instructions.

The examples shown here assume use of the latin1 character set and latin1\_swedish\_ci collation in particular contexts as an alternative to the defaults of utf8mb4 and utf8mb4\_0900\_ai\_ci.

• **Specify** **character** **settings** **per** **database.** To create a database such that its tables use a given default character set and collation for data storage, use a CREATE DATABASE statement like this:

CREATE DATABASE mydb



CHARACTER SET latin1

COLLATE latin1\_swedish\_ci;

Tables created in the database use latin1 and latin1\_swedish\_ci by default for any character columns.

Applications that use the database should also configure their connection to the server each time they connect. This can be done by executing a SET NAMES 'latin1' statement after connecting. The statement can be used regardless of connection method (the mysql client, PHP scripts, and so forth).

In some cases, it may be possible to configure the connection to use the desired character set some other way. For example, to connect using mysql, you can specify the --default-character- set=latin1 command-line option to achieve the same effect as SET NAMES 'latin1'.

For more information about configuring client connections, see [Section 10.4, “Connection Character](#_bookmark17) [Sets and Collations”](#_bookmark17) .

**Note**

If you use ALTER DATABASE to change the database default character set or collation, existing stored routines in the database that use those defaults must be dropped and recreated so that they use the new defaults. (In a stored routine, variables with character data types use the database defaults if the character set or collation are not specified explicitly. See Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements” .)

• **Specify** **character** **settings** **at** **server** **startup.** To select a character set and collation at server startup, use the --character-set-server and --collation-server options. For example, to specify the options in an option file, include these lines:

[mysqld]

character-set-server=latin1

collation-server=latin1\_swedish\_ci

These settings apply server-wide and apply as the defaults for databases created by any application, and for tables created in those databases.

It is still necessary for applications to configure their connection using SET NAMES or equivalent after they connect, as described previously. You might be tempted to start the server with the --init\_connect="SET NAMES 'latin1'" option to cause SET NAMES to be executed automatically for each client that connects. However, this may yield inconsistent results because the init\_connect value is not executed for users who have the CONNECTION\_ADMIN privilege (or the deprecated SUPER privilege).

• **Specify** **character** **settings** **at** **MySQL** **configuration** **time.** To select a character set and collation if you configure and build MySQL from source, use the DEFAULT\_CHARSET and DEFAULT\_COLLATION CMake options:

cmake . -DDEFAULT\_CHARSET=latin1 \

-DDEFAULT\_COLLATION=latin1\_swedish\_ci

The resulting server uses latin1 and latin1\_swedish\_ci as the default for databases and tables and for client connections. It is unnecessary to use --character-set-server and --collation-server to specify those defaults at server startup. It is also unnecessary for applications to configure their connection using SET NAMES or equivalent after they connect to the server.

Regardless of how you configure the MySQL character set for application use, you must also consider the environment within which those applications execute. For example, if you intend to send statements using UTF-8 text taken from a file that you create in an editor, you should edit the file with the locale of your environment set to UTF-8 so that the file encoding is correct and so that the operating system

handles it correctly. If you use the mysql client from within a terminal window, the window must be configured to use UTF-8 or characters may not display properly. For a script that executes in a Web environment, the script must handle character encoding properly for its interaction with the MySQL server, and it must generate pages that correctly indicate the encoding so that browsers know how to display the content of the pages. For example, you can include this <meta> tag within your <head> element:

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

**10.6** **Error** **Message** **Character** **Set**

This section describes how the MySQL server uses character sets for constructing error messages. For information about the language of error messages (rather than the character set), see [Section 10.12,](#_bookmark48) [“Setting the Error Message Language”](#_bookmark48) . For general information about configuring error logging, see Section 5.4.2, “The Error Log” .

• [Character Set for Error Message Construction](#_bookmark71)

• [Character Set for Error Message Disposition](#_bookmark72)

**Character** **Set** **for** **Error** **Message** **Construction**

The server constructs error messages as follows:

• The message template uses UTF-8 (utf8mb3).

• Parameters in the message template are replaced with values that apply to a specific error occurrence:

• Identifiers such as table or column names use UTF-8 internally so they are copied as is.

• Character (nonbinary) string values are converted from their character set to UTF-8.

• Binary string values are copied as is for bytes in the range 0x20 to 0x7E, and using \x hexadecimal encoding for bytes outside that range. For example, if a duplicate-key error occurs for an attempt to insert 0x41CF9F into a [VARBINARY](#_bookmark73) unique column, the resulting error message uses UTF-8 with some bytes hexadecimal encoded:

Duplicate entry 'A\xCF\x9F' for key 1

**Character** **Set** **for** **Error** **Message** **Disposition**

An error message, once constructed, can be written by the server to the error log or sent to clients:

• If the server writes the error message to the error log, it writes it in UTF-8, as constructed, without conversion to another character set.

• If the server sends the error message to a client program, the server converts it from

UTF-8 to the character set specified by the character\_set\_results system variable. If character\_set\_results has a value of NULL or binary, no conversion occurs. No conversion occurs if the variable value is utf8mb3 or utf8mb4, either, because those character sets have a repertoire that includes all UTF-8 characters used in message construction.

If characters cannot be represented in character\_set\_results, some encoding may occur during the conversion. The encoding uses Unicode code point values:

• Characters in the Basic Multilingual Plane (BMP) range ( 0x0000 to 0xFFFF) are written using \*nnnn* notation.

• Characters outside the BMP range ( 0x10000 to 0x10FFFF) are written using \+*nnnnnn* notation.

Clients can set character\_set\_results to control the character set in which they receive error messages. The variable can be set directly, or indirectly by means such as SET NAMES. For more information about character\_set\_results, see [Section 10.4, “Connection Character Sets and](#_bookmark17) [Collations”](#_bookmark17) .

**10.7** **Column** **Character** **Set** **Conversion**

To convert a binary or nonbinary string column to use a particular character set, use ALTER TABLE. For successful conversion to occur, one of the following conditions must apply:

• If the column has a binary data type ([BINARY](#_bookmark73), [VARBINARY](#_bookmark73), [BLOB](#_bookmark61)), all the values that it contains must be encoded using a single character set (the character set you're converting the column to). If you use a binary column to store information in multiple character sets, MySQL has no way to know which values use which character set and cannot convert the data properly.

• If the column has a nonbinary data type ([CHAR](#_bookmark60), [VARCHAR](#_bookmark60), [TEXT](#_bookmark61)), its contents should be encoded in the column character set, not some other character set. If the contents are encoded in a different character set, you can convert the column to use a binary data type first, and then to a nonbinary column with the desired character set.

Suppose that a table t has a binary column named col1 defined as VARBINARY(50). Assuming that the information in the column is encoded using a single character set, you can convert it to a nonbinary column that has that character set. For example, if col1 contains binary data representing characters in the greek character set, you can convert it as follows:

ALTER TABLE t MODIFY col1 VARCHAR(50) CHARACTER SET greek;

If your original column has a type of BINARY(50), you could convert it to CHAR(50), but the resulting values are padded with 0x00 bytes at the end, which may be undesirable. To remove these bytes, use the TRIM() function:

UPDATE t SET col1 = TRIM(TRAILING 0x00 FROM col1);

Suppose that table t has a nonbinary column named col1 defined as CHAR(50) CHARACTER SET latin1 but you want to convert it to use utf8mb4 so that you can store values from many languages. The following statement accomplishes this:

ALTER TABLE t MODIFY col1 CHAR(50) CHARACTER SET utf8mb4;

Conversion may be lossy if the column contains characters that are not in both character sets.

A special case occurs if you have old tables from before MySQL 4.1 where a nonbinary column contains values that actually are encoded in a character set different from the server's default character set. For example, an application might have stored sjis values in a column, even though MySQL's default character set was different. It is possible to convert the column to use the proper character set but an additional step is required. Suppose that the server's default character set was latin1 and col1 is defined as CHAR(50) but its contents are sjis values. The first step is to convert the column to a binary data type, which removes the existing character set information without performing any character conversion:

ALTER TABLE t MODIFY col1 BLOB;

The next step is to convert the column to a nonbinary data type with the proper character set:

ALTER TABLE t MODIFY col1 CHAR(50) CHARACTER SET sjis;

This procedure requires that the table not have been modified already with statements such as INSERT or UPDATE after an upgrade to MySQL 4.1 or higher. In that case, MySQL would store new values

in the column using latin1, and the column would contain a mix of sjis and latin1 values and cannot be converted properly.

If you specified attributes when creating a column initially, you should also specify them when altering the table with ALTER TABLE. For example, if you specified NOT NULL and an explicit DEFAULT value, you should also provide them in the ALTER TABLE statement. Otherwise, the resulting column definition does not include those attributes.

To convert all character columns in a table, the ALTER TABLE ... CONVERT TO CHARACTER SET *charset* statement may be useful. See Section 13.1.9, “ALTER TABLE Statement” .

**10.8** **Collation** **Issues**

The following sections discuss various aspects of character set collations.

**10.8.1** **Using** **COLLATE** **in** **SQL** **Statements**

With the COLLATE clause, you can override whatever the default collation is for a comparison. COLLATE may be used in various parts of SQL statements. Here are some examples:

• With ORDER BY:

SELECT k

FROM t1

ORDER BY k COLLATE latin1\_german2\_ci;

• With AS:

SELECT k COLLATE latin1\_german2\_ci AS k1

FROM t1

ORDER BY k1;

• With GROUP BY:

SELECT k

FROM t1

GROUP BY k COLLATE latin1\_german2\_ci;

• With aggregate functions:

SELECT MAX(k COLLATE latin1\_german2\_ci)

FROM t1;

• With DISTINCT:

SELECT DISTINCT k COLLATE latin1\_german2\_ci

FROM t1;

• With WHERE:

SELECT \*

FROM t1

WHERE \_latin1 'Müller' COLLATE latin1\_german2\_ci = k;

SELECT \*

FROM t1

WHERE k LIKE \_latin1 'Müller' COLLATE latin1\_german2\_ci;

• With HAVING:

SELECT k

FROM t1

GROUP BY k

HAVING k = \_latin1 'Müller' COLLATE latin1\_german2\_ci;

**10.8.2** **COLLATE** **Clause** **Precedence**

The COLLATE clause has high precedence (higher than [||](#_bookmark74)), so the following two expressions are equivalent:

x || y COLLATE z

x || (y COLLATE z)

**10.8.3** **Character** **Set** **and** **Collation** **Compatibility**

Each character set has one or more collations, but each collation is associated with one and only one character set. Therefore, the following statement causes an error message because the latin2\_bin collation is not legal with the latin1 character set:

mysql> **SELECT** **\_latin1** **'x'** **COLLATE** **latin2\_bin;**

ERROR 1253 (42000): COLLATION 'latin2\_bin' is not valid

for CHARACTER SET 'latin1'

**10.8.4** **Collation** **Coercibility** **in** **Expressions**

In the great majority of statements, it is obvious what collation MySQL uses to resolve a comparison operation. For example, in the following cases, it should be clear that the collation is the collation of column x:

SELECT x FROM T ORDER BY x;

SELECT x FROM T WHERE x = x;

SELECT DISTINCT x FROM T;

However, with multiple operands, there can be ambiguity. For example, this statement performs a comparison between the column x and the string literal 'Y':

SELECT x FROM T WHERE x = 'Y';

If x and 'Y' have the same collation, there is no ambiguity about the collation to use for the comparison. But if they have different collations, should the comparison use the collation of x, or of 'Y'? Both x and 'Y' have collations, so which collation takes precedence?

A mix of collations may also occur in contexts other than comparison. For example, a multiple- argument concatenation operation such as CONCAT(x,'Y') combines its arguments to produce a single string. What collation should the result have?

To resolve questions like these, MySQL checks whether the collation of one item can be coerced to the collation of the other. MySQL assigns coercibility values as follows:

• An explicit COLLATE clause has a coercibility of 0 (not coercible at all).

• The concatenation of two strings with different collations has a coercibility of 1.

• The collation of a column or a stored routine parameter or local variable has a coercibility of 2.

• A “system constant” (the string returned by functions such as USER() or VERSION()) has a coercibility of 3.

• The collation of a literal has a coercibility of 4.

• The collation of a numeric or temporal value has a coercibility of 5.

• NULL or an expression that is derived from NULL has a coercibility of 6. MySQL uses coercibility values with the following rules to resolve ambiguities:

• Use the collation with the lowest coercibility value.

• If both sides have the same coercibility, then:

• If both sides are Unicode, or both sides are not Unicode, it is an error.

• If one of the sides has a Unicode character set, and another side has a non-Unicode character set, the side with Unicode character set wins, and automatic character set conversion is applied to the non-Unicode side. For example, the following statement does not return an error:

SELECT CONCAT(utf8mb4\_column, latin1\_column) FROM t1;

It returns a result that has a character set of utf8mb4 and the same collation as utf8mb4\_column. Values of latin1\_column are automatically converted to utf8mb4 before concatenating.

• For an operation with operands from the same character set but that mix a \_bin collation and a \_ci or \_cs collation, the \_bin collation is used. This is similar to how operations that mix nonbinary and binary strings evaluate the operands as binary strings, applied to collations rather than data types.

Although automatic conversion is not in the SQL standard, the standard does say that every character set is (in terms of supported characters) a “subset” of Unicode. Because it is a well-known principle that “what applies to a superset can apply to a subset,” we believe that a collation for Unicode can apply for comparisons with non-Unicode strings. More generally, MySQL uses the concept of character set repertoire, which can sometimes be used to determine subset relationships among character sets and enable conversion of operands in operations that would otherwise produce an error. See

[Section 10.2.1, “Character Set Repertoire”](#_bookmark4) .

The following table illustrates some applications of the preceding rules.

|  |  |
| --- | --- |
| **Comparison** | **Collation** **Used** |
| column1 = 'A' | Use collation of column1 |
| column1 = 'A' COLLATE x | Use collation of 'A' COLLATE x |
| column1 COLLATE x = 'A' COLLATE y | Error |

To determine the coercibility of a string expression, use the COERCIBILITY() function (see Section 12.16, “Information Functions”):

mysql> **SELECT** **COERCIBILITY(\_utf8mb4'A'** **COLLATE** **utf8mb4\_bin);**

-> 0

mysql> **SELECT**

-> 3

mysql> **SELECT**

-> 4

mysql> **SELECT**

-> 5

mysql> **SELECT**

-> 6

**COERCIBILITY(VERSION());** **COERCIBILITY('A');** **COERCIBILITY(1000);**

**COERCIBILITY(NULL);**

For implicit conversion of a numeric or temporal value to a string, such as occurs for the argument 1 in the expression CONCAT(1, 'abc'), the result is a character (nonbinary) string that has a character set and collation determined by the character\_set\_connection and collation\_connection system variables. See [Section 12.3, “Type Conversion in Expression Evaluation”](#_bookmark75) .

**10.8.5** **The** **binary** **Collation** **Compared** **to** **\_bin** **Collations**

This section describes how the binary collation for binary strings compares to \_bin collations for nonbinary strings.

Binary strings (as stored using the [BINARY](#_bookmark73), [VARBINARY](#_bookmark73), and [BLOB](#_bookmark61) data types) have a character set and collation named binary. Binary strings are sequences of bytes and the numeric values of those bytes determine comparison and sort order. See [Section 10.10.8, “The Binary Character Set”](#_bookmark46) .

Nonbinary strings (as stored using the [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), and [TEXT](#_bookmark61) data types) have a character set and collation other than binary. A given nonbinary character set can have several collations, each of which defines a particular comparison and sort order for the characters in the set. For most character sets, one of these is the binary collation, indicated by a \_bin suffix in the collation name. For example, the binary collations for latin1 and big5 are named latin1\_bin and big5\_bin, respectively.

utf8mb4 is an exception that has two binary collations, utf8mb4\_bin and utf8mb4\_0900\_bin; see [Section 10.10.1, “Unicode Character Sets”](#_bookmark39) .

The binary collation differs from \_bin collations in several respects, discussed in the following sections:

• [The Unit for Comparison and Sorting](#_bookmark76)

• [Character Set Conversion](#_bookmark77)

• [Lettercase Conversion](#_bookmark78)

• [Trailing Space Handling in Comparisons](#_bookmark79)

• [Trailing Space Handling for Inserts and Retrievals](#_bookmark80)

**The** **Unit** **for** **Comparison** **and** **Sorting**

Binary strings are sequences of bytes. For the binary collation, comparison and sorting are based on numeric byte values. Nonbinary strings are sequences of characters, which might be multibyte. Collations for nonbinary strings define an ordering of the character values for comparison and sorting. For \_bin collations, this ordering is based on numeric character code values, which is similar to ordering for binary strings except that character code values might be multibyte.

**Character** **Set** **Conversion**

A nonbinary string has a character set and is automatically converted to another character set in many cases, even when the string has a \_bin collation:

• When assigning column values to another column that has a different character set:

UPDATE t1 SET utf8mb4\_bin\_column=latin1\_column;

INSERT INTO t1 (latin1\_column) SELECT utf8mb4\_bin\_column FROM t2;

• When assigning column values for INSERT or UPDATE using a string literal:

SET NAMES latin1;

INSERT INTO t1 (utf8mb4\_bin\_column) VALUES ('string-in-latin1');

• When sending results from the server to a client:

SET NAMES latin1;

SELECT utf8mb4\_bin\_column FROM t2;

For binary string columns, no conversion occurs. For cases similar to those preceding, the string value is copied byte-wise.

**Lettercase** **Conversion**

Collations for nonbinary character sets provide information about lettercase of characters, so characters in a nonbinary string can be converted from one lettercase to another, even for \_bin collations that ignore lettercase for ordering:

mysql> **SET** **NAMES** **utf8mb4** **COLLATE** **utf8mb4\_bin;**

mysql> **SELECT** **LOWER('aA'),** **UPPER('zZ');**

+-------------+-------------+

| LOWER('aA') | UPPER('zZ') |

+-------------+-------------+

| aa | ZZ |

+-------------+-------------+

The concept of lettercase does not apply to bytes in a binary string. To perform lettercase conversion, the string must first be converted to a nonbinary string using a character set appropriate for the data stored in the string:



mysql> **SET** **NAMES** **binary;**

mysql> **SELECT** **LOWER('aA'),** **LOWER(CONVERT('aA'** **USING** **utf8mb4));**

+-------------+------------------------------------+

| LOWER('aA') | LOWER(CONVERT('aA' USING utf8mb4)) |

+-------------+------------------------------------+

| aA | aa |

+-------------+------------------------------------+

**Trailing** **Space** **Handling** **in** **Comparisons**

MySQL collations have a pad attribute, which has a value of PAD SPACE or NO PAD:

• Most MySQL collations have a pad attribute of PAD SPACE.

• The Unicode collations based on UCA 9.0.0 and higher have a pad attribute of NO PAD; see [Section 10.10.1, “Unicode Character Sets”](#_bookmark39) .

For nonbinary strings (CHAR, VARCHAR, and TEXT values), the string collation pad attribute determines treatment in comparisons of trailing spaces at the end of strings:

• For PAD SPACE collations, trailing spaces are insignificant in comparisons; strings are compared without regard to trailing spaces.

• NO PAD collations treat trailing spaces as significant in comparisons, like any other character.

The differing behaviors can be demonstrated using the two utf8mb4 binary collations, one of which is PAD SPACE, the other of which is NO PAD. The example also shows how to use the INFORMATION\_SCHEMA COLLATIONS table to determine the pad attribute for collations.

mysql> **SELECT** **COLLATION\_NAME,** **PAD\_ATTRIBUTE**

**FROM** **INFORMATION\_SCHEMA.COLLATIONS**

**WHERE** **COLLATION\_NAME** **LIKE** **'utf8mb4%bin';**

+------------------+---------------+

| COLLATION\_NAME | PAD\_ATTRIBUTE |

+------------------+---------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | utf8mb4\_bin  utf8mb4\_0900\_bin | |  | | PAD SPACE  NO PAD | |  | |

+------------------+---------------+

mysql> **SET** **NAMES** **utf8mb4** **COLLATE** **utf8mb4\_bin;**

mysql> **SELECT** **'a** **'** **=** **'a';**

+------------+

| 'a ' = 'a' |

+------------+

| 1 |

+------------+

mysql> **SET** **NAMES** **utf8mb4** **COLLATE** **utf8mb4\_0900\_bin;**

mysql> **SELECT** **'a** **'** **=** **'a';**

+------------+

| 'a ' = 'a' |

+------------+

| 0 |

+------------+

**Note**

“Comparison” in this context does not include the LIKE pattern-matching operator, for which trailing spaces are significant, regardless of collation.

For binary strings (BINARY, VARBINARY, and BLOB values), all bytes are significant in comparisons, including trailing spaces:

mysql> **SET** **NAMES** **binary;**

mysql> **SELECT** **'a** **'** **=** **'a';**

+------------+

| 'a ' = 'a' |

+------------+

| 0 |

**);**

mysql> **INSERT** **INTO** **t1** **VALUES** **('x','x');**

mysql> **INSERT** **INTO** **t1** **VALUES** **('x** **','x** **');**

mysql> **SELECT** **a,** **b,** **HEX(a),** **HEX(b)** **FROM** **t1;**

+------+------------------------+--------+----------------------+

| a | b | HEX(a) | HEX(b) |

+------+------------------------+--------+----------------------+

| x

| x

|

|

|

|

|

|

|

|

78

78

78000000000000000000

78200000000000000000

0x78000000000000000000

0x78200000000000000000

+------+------------------------+--------+----------------------+

+------------+

**Trailing** **Space** **Handling** **for** **Inserts** **and** **Retrievals**

CHAR(*N*) columns store nonbinary strings *N*characters long. For inserts, values shorter than *N* characters are extended with spaces. For retrievals, trailing spaces are removed.

BINARY(*N*) columns store binary strings *N* bytes long. For inserts, values shorter than *N* bytes are extended with 0x00 bytes. For retrievals, nothing is removed; a value of the declared length is always returned.

mysql> **CREATE** **TABLE** **t1** **(**

**a** **CHAR(10)** **CHARACTER** **SET** **utf8mb4** **COLLATE** **utf8mb4\_bin,**

**b** **BINARY(10)**

**10.8.6** **Examples** **of** **the** **Effect** **of** **Collation**

**Example** **1:** **Sorting** **German** **Umlauts**

Suppose that column X in table T has these latin1 column values:

Muffler

Müller

MX Systems

MySQL

Suppose also that the column values are retrieved using the following statement:

SELECT X FROM T ORDER BY X COLLATE

*collation\_name*;

The following table shows the resulting order of the values if we use ORDER BY with different collations.

|  |  |  |
| --- | --- | --- |
| **latin1\_swedish\_ci** | **latin1\_german1\_ci** | **latin1\_german2\_ci** |
| Muffler | Muffler | Müller |
| MX Systems | Müller | Muffler |
| Müller | MX Systems | MX Systems |
| MySQL | MySQL | MySQL |

The character that causes the different sort orders in this example is ü (German “U-umlaut”).

• The first column shows the result of the SELECT using the Swedish/Finnish collating rule, which says that U-umlaut sorts with Y.

• The second column shows the result of the SELECT using the German DIN-1 rule, which says that U- umlaut sorts with U.

• The third column shows the result of the SELECT using the German DIN-2 rule, which says that U- umlaut sorts with UE.

**Example** **2:** **Searching** **for** **German** **Umlauts**

Suppose that you have three tables that differ only by the character set and collation used:

mysql> **SET** **NAMES** **utf8mb4;**

mysql> **CREATE** **TABLE** **german1** **(**

**c** **CHAR(10)**

**)** **CHARACTER** **SET** **latin1** **COLLATE** **latin1\_german1\_ci;**

mysql> **CREATE** **TABLE** **german2** **(**

**c** **CHAR(10)**

**)** **CHARACTER** **SET** **latin1** **COLLATE** **latin1\_german2\_ci;**

mysql> **CREATE** **TABLE** **germanutf8** **(**

**c** **CHAR(10)**

**)** **CHARACTER** **SET** **utf8mb4** **COLLATE** **utf8mb4\_unicode\_ci;**

Each table contains two records:

mysql> **INSERT** **INTO** **german1** **VALUES** **('Bar'),** **('Bär');**

mysql> **INSERT** **INTO** **german2** **VALUES** **('Bar'),** **('Bär');**

mysql> **INSERT** **INTO** **germanutf8** **VALUES** **('Bar'),** **('Bär');**

Two of the above collations have an A = Ä equality, and one has no such equality (latin1\_german2\_ci). For that reason, comparisons yield the results shown here:

mysql> **SELECT** **\*** **FROM** **german1** **WHERE** **c** **=** **'Bär';**

+------+

| c |

+------+

| Bar |

| Bär |

+------+

mysql> **SELECT** **\*** **FROM** **german2** **WHERE** **c** **=** **'Bär';**

+------+

| c |

+------+

| Bär |

+------+

mysql> **SELECT** **\*** **FROM** **germanutf8** **WHERE** **c** **=** **'Bär';**

+------+

| c |

+------+

| Bar |

| Bär |

+------+

This is not a bug but rather a consequence of the sorting properties of latin1\_german1\_ci and

utf8mb4\_unicode\_ci (the sorting shown is done according to the German DIN 5007 standard).

**10.8.7** **Using** **Collation** **in** **INFORMATION\_SCHEMA** **Searches**

String columns in INFORMATION\_SCHEMA tables have a collation of utf8mb3\_general\_ci, which is case-insensitive. However, for values that correspond to objects that are represented in the file system, such as databases and tables, searches in INFORMATION\_SCHEMA string columns can be case- sensitive or case-insensitive, depending on the characteristics of the underlying file system and the lower\_case\_table\_names system variable setting. For example, searches may be case-sensitive if the file system is case-sensitive. This section describes this behavior and how to modify it if necessary.

Suppose that a query searches the SCHEMATA.SCHEMA\_NAME column for the test database. On Linux, file systems are case-sensitive, so comparisons of SCHEMATA.SCHEMA\_NAME with 'test' match, but comparisons with 'TEST' do not:

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **=** **'test';**

+-------------+

| SCHEMA\_NAME |

+-------------+

| test |

+-------------+

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **=** **'TEST';**



Empty set (0 .00 sec)

These results occur with the lower\_case\_table\_names system variable set to 0. A lower\_case\_table\_names setting of 1 or 2 causes the second query to return the same (nonempty) result as the first query.

**Note**

It is prohibited to start the server with a lower\_case\_table\_names setting that is different from the setting used when the server was initialized.

On Windows or macOS, file systems are not case-sensitive, so comparisons match both 'test' and 'TEST':

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **=** **'test';**

+-------------+

| SCHEMA\_NAME |

+-------------+

| test |

+-------------+

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **=** **'TEST';**

+-------------+

| SCHEMA\_NAME |

+-------------+

| TEST |

+-------------+

The value of lower\_case\_table\_names makes no difference in this context.

The preceding behavior occurs because the utf8mb3\_general\_ci collation is not used for INFORMATION\_SCHEMA queries when searching for values that correspond to objects represented in the file system.

If the result of a string operation on an INFORMATION\_SCHEMA column differs from expectations, a workaround is to use an explicit COLLATE clause to force a suitable collation (see [Section 10.8.1,](#_bookmark22) [“Using COLLATE in SQL Statements”](#_bookmark22)). For example, to perform a case-insensitive search, use COLLATE with the INFORMATION\_SCHEMA column name:

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **COLLATE** **utf8mb3\_general\_ci** **=** **'test';**

+-------------+

| SCHEMA\_NAME |

+-------------+

| test |

+-------------+

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **COLLATE** **utf8mb3\_general\_ci** **=** **'TEST';**

+-------------+

| SCHEMA\_NAME |

+-------------+

| test |

+-------------+

You can also use the UPPER() or LOWER() function:

WHERE UPPER(SCHEMA\_NAME) = 'TEST'

WHERE LOWER(SCHEMA\_NAME) = 'test'

Although a case-insensitive comparison can be performed even on platforms with case-sensitive file systems, as just shown, it is not necessarily always the right thing to do. On such platforms, it is possible to have multiple objects with names that differ only in lettercase. For example, tables named city, CITY, and City can all exist simultaneously. Consider whether a search should match all

such names or just one and write queries accordingly. The first of the following comparisons (with utf8mb3\_bin) is case-sensitive; the others are not:

WHERE TABLE\_NAME COLLATE utf8mb3\_bin = 'City'

WHERE TABLE\_NAME COLLATE utf8mb3\_general\_ci = 'city'

WHERE UPPER(TABLE\_NAME) = 'CITY'

WHERE LOWER(TABLE\_NAME) = 'city'

Searches in INFORMATION\_SCHEMA string columns for values that refer to INFORMATION\_SCHEMA itself do use the utf8mb3\_general\_ci collation because INFORMATION\_SCHEMA is

a “virtual” database not represented in the file system. For example, comparisons with

SCHEMATA.SCHEMA\_NAME match 'information\_schema' or 'INFORMATION\_SCHEMA' regardless of platform:

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **=** **'information\_schema';**

+--------------------+

| SCHEMA\_NAME |

+--------------------+

| information\_schema |

+--------------------+

mysql> **SELECT** **SCHEMA\_NAME** **FROM** **INFORMATION\_SCHEMA.SCHEMATA**

**WHERE** **SCHEMA\_NAME** **=** **'INFORMATION\_SCHEMA';**

+--------------------+

| SCHEMA\_NAME |

+--------------------+

| information\_schema |

+--------------------+

**10.9** **Unicode** **Support**

The Unicode Standard includes characters from the Basic Multilingual Plane (BMP) and supplementary characters that lie outside the BMP. This section describes support for Unicode in MySQL. For information about the Unicode Standard itself, visit the [Unicode Consortium website](http://www.unicode.org/).

BMP characters have these characteristics:

• Their code point values are between 0 and 65535 (or U+0000 and U+FFFF).

• They can be encoded in a variable-length encoding using 8, 16, or 24 bits (1 to 3 bytes).

• They can be encoded in a fixed-length encoding using 16 bits (2 bytes).

• They are sufficient for almost all characters in major languages. Supplementary characters lie outside the BMP:

• Their code point values are between U+10000 and U+10FFFF).

• Unicode support for supplementary characters requires character sets that have a range outside BMP characters and therefore take more space than BMP characters (up to 4 bytes per character).

The UTF-8 (Unicode Transformation Format with 8-bit units) method for encoding Unicode data is implemented according to RFC 3629, which describes encoding sequences that take from one to four bytes. The idea of UTF-8 is that various Unicode characters are encoded using byte sequences of different lengths:

• Basic Latin letters, digits, and punctuation signs use one byte.

• Most European and Middle East script letters fit into a 2-byte sequence: extended Latin letters (with tilde, macron, acute, grave and other accents), Cyrillic, Greek, Armenian, Hebrew, Arabic, Syriac, and others.

• Korean, Chinese, and Japanese ideographs use 3-byte or 4-byte sequences.



MySQL supports these Unicode character sets:

• utf8mb4: A UTF-8 encoding of the Unicode character set using one to four bytes per character.

• utf8mb3: A UTF-8 encoding of the Unicode character set using one to three bytes per character. This character set is deprecated in MySQL 8.0, and you should use utfmb4 instead.

• utf8: An alias for utf8mb3. In MySQL 8.0, this alias is deprecated; use utf8mb4 instead. utf8 is expected in a future release to become an alias for utf8mb4.

• ucs2: The UCS-2 encoding of the Unicode character set using two bytes per character. Deprecated in MySQL 8.0.28; you should expect support for this character set to be removed in a future release.

• utf16: The UTF-16 encoding for the Unicode character set using two or four bytes per character. Like ucs2 but with an extension for supplementary characters.

• utf16le: The UTF-16LE encoding for the Unicode character set. Like utf16 but little-endian rather than big-endian.

• utf32: The UTF-32 encoding for the Unicode character set using four bytes per character.

**Note**

The utf8mb3 character set is deprecated and you should expect it to be removed in a future MySQL release. Please use utf8mb4 instead. utf8 is currently an alias for utf8mb3, but it is now deprecated as such, and utf8 is expected subsequently to become a reference to utf8mb4. Beginning with MySQL 8.0.28, utf8mb3 is also displayed in place of utf8 in columns of Information Schema tables, and in the output of SQL SHOW statements.

In addition, in MySQL 8.0.30, all collations using the utf8\_ prefix are renamed using the prefix utf8mb3\_.

To avoid ambiguity about the meaning of utf8, consider specifying utf8mb4 explicitly for character set references.

[Table 10.2, “Unicode Character Set General Characteristics”](#_bookmark81) , summarizes the general characteristics of Unicode character sets supported by MySQL.

**Table** **10.2** **Unicode** **Character** **Set** **General** **Characteristics**

|  |  |  |
| --- | --- | --- |
| **Character** **Set** | **Supported** **Characters** | **Required** **Storage** **Per**  **Character** |
| utf8mb3, utf8 (deprecated) | BMP only | 1, 2, or 3 bytes |
| ucs2 | BMP only | 2 bytes |
| utf8mb4 | BMP and supplementary | 1, 2, 3, or 4 bytes |
| utf16 | BMP and supplementary | 2 or 4 bytes |
| utf16le | BMP and supplementary | 2 or 4 bytes |
| utf32 | BMP and supplementary | 4 bytes |

Characters outside the BMP compare as REPLACEMENT CHARACTER and convert to '?' when converted to a Unicode character set that supports only BMP characters (utf8mb3 or ucs2).

If you use character sets that support supplementary characters and thus are “wider” than the BMP- only utf8mb3 and ucs2 character sets, there are potential incompatibility issues for your applications; see [Section 10.9.8, “Converting Between 3-Byte and 4-Byte Unicode Character Sets”](#_bookmark37) . That section also describes how to convert tables from the (3-byte) utf8mb3 to the (4-byte) utf8mb4, and what constraints may apply in doing so.

A similar set of collations is available for most Unicode character sets. For example, each has a Danish collation, the names of which are utf8mb4\_danish\_ci, utf8mb3\_danish\_ci (deprecated), utf8\_danish\_ci (deprecated), ucs2\_danish\_ci, utf16\_danish\_ci, and utf32\_danish\_ci. The exception is utf16le, which has only two collations. For information about Unicode collations and their differentiating properties, including collation properties for supplementary characters, see [Section 10.10.1, “Unicode Character Sets”](#_bookmark39) .

The MySQL implementation of UCS-2, UTF-16, and UTF-32 stores characters in big-endian byte order and does not use a byte order mark (BOM) at the beginning of values. Other database systems might use little-endian byte order or a BOM. In such cases, conversion of values needs to be performed when transferring data between those systems and MySQL. The implementation of UTF-16LE is little-endian.

MySQL uses no BOM for UTF-8 values.

Client applications that communicate with the server using Unicode should set the client character set accordingly (for example, by issuing a SET NAMES 'utf8mb4' statement). Some character sets cannot be used as the client character set. Attempting to use them with SET NAMES or SET CHARACTER SET produces an error. See [Impermissible Client Character Sets](#_bookmark65).

The following sections provide additional detail on the Unicode character sets in MySQL.

**10.9.1** **The** **utf8mb4** **Character** **Set** **(4-Byte** **UTF-8** **Unicode** **Encoding)**

The utfmb4 character set has these characteristics:

• Supports BMP and supplementary characters.

• Requires a maximum of four bytes per multibyte character.

utf8mb4 contrasts with the utf8mb3 character set, which supports only BMP characters and uses a maximum of three bytes per character:

• For a BMP character, utf8mb4 and utf8mb3 have identical storage characteristics: same code values, same encoding, same length.

• For a supplementary character, utf8mb4 requires four bytes to store it, whereas utf8mb3 cannot store the character at all. When converting utf8mb3 columns to utf8mb4, you need not worry about converting supplementary characters because there are none.

utf8mb4 is a superset of utf8mb3, so for an operation such as the following concatenation, the result has character set utf8mb4 and the collation of utf8mb4\_col:

SELECT CONCAT(utf8mb3\_col, utf8mb4\_col);

Similarly, the following comparison in the WHERE clause works according to the collation of utf8mb4\_col:

SELECT \* FROM utf8mb3\_tbl, utf8mb4\_tbl

WHERE utf8mb3\_tbl.utf8mb3\_col = utf8mb4\_tbl.utf8mb4\_col;

For information about data type storage as it relates to multibyte character sets, see [String Type](#_bookmark82) [Storage Requirements](#_bookmark82).

**10.9.2** **The** **utf8mb3** **Character** **Set** **(3-Byte** **UTF-8** **Unicode** **Encoding)**

The utf8mb3 character set has these characteristics:

• Supports BMP characters only (no support for supplementary characters)

• Requires a maximum of three bytes per multibyte character.



Applications that use UTF-8 data but require supplementary character support should use utf8mb4 rather than utf8mb3 (see [Section 10.9.1, “The utf8mb4 Character Set (4-Byte UTF-8 Unicode](#_bookmark30) [Encoding)”](#_bookmark30)).

Exactly the same set of characters is available in utf8mb3 and ucs2. That is, they have the same repertoire.

**Note**

Historically, MySQL has used utf8 as an alias for utf8mb3; beginning with MySQL 8.0.28, utf8mb3 is used exclusively in the output of SHOW statements and in Information Schema tables when this character set is meant.

At some point in the future utf8 is expected to become a reference to utf8mb4. To avoid ambiguity about the meaning of utf8, consider specifying utf8mb4 explicitly for character set references instead of utf8.

You should also be aware that the utf8mb3 character set is deprecated and you should expect it to be removed in a future MySQL release. Please use utf8mb4 instead.

utf8mb3 can be used in CHARACTER SET clauses, and utf8mb3\_*collation\_substring* in COLLATE clauses, where *collation\_substring* is bin, czech\_ci, danish\_ci, esperanto\_ci, estonian\_ci, and so forth. For example:

CREATE TABLE t (s1 CHAR(1)) CHARACTER SET utf8mb3;

SELECT \* FROM t WHERE s1 COLLATE utf8mb3\_general\_ci = 'x';

DECLARE x VARCHAR(5) CHARACTER SET utf8mb3 COLLATE utf8mb3\_danish\_ci;

SELECT CAST('a' AS CHAR CHARACTER SET utf8mb4) COLLATE utf8mb4\_czech\_ci;

Prior to MySQL 8.0.29, instances of utf8mb3 in statements were converted to utf8. In MySQL 8.0.30 and later, the reverse is true, so that in statements such as SHOW CREATE TABLE or SELECT CHARACTER\_SET\_NAME FROM INFORMATION\_SCHEMA.COLUMNS or SELECT COLLATION\_NAME FROM INFORMATION\_SCHEMA.COLUMNS, users see the character set or collation name prefixed with utf8mb3 or utf8mb3\_.

utf8mb3 is also valid (but deprecated) in contexts other than CHARACTER SET clauses. For example:

mysqld --character-set-server=utf8mb3

SET NAMES 'utf8mb3'; /\* and other SET statements that have similar effect \*/

SELECT \_utf8mb3 'a';

For information about data type storage as it relates to multibyte character sets, see [String Type](#_bookmark82) [Storage Requirements](#_bookmark82).

**10.9.3** **The** **utf8** **Character** **Set** **(Alias** **for** **utf8mb3)**

utf8 has been used by MySQL is an alias for the utf8mb3 character set, but this usage is being phased out; as of MySQL 8.0.28, SHOW statements and columns of Information Schema tables display utf8mb3 instead. For more information, see [Section 10.9.2, “The utf8mb3 Character Set (3-Byte](#_bookmark31) [UTF-8 Unicode Encoding)”](#_bookmark31) .

**Note**

The utf8mb3 character set is deprecated and you should expect it to be removed in a future MySQL release. Please use utf8mb4 instead. utf8 is currently an alias for utf8mb3, but it is now deprecated as such, and utf8 is expected subsequently to become a reference to utf8mb4. Beginning with MySQL 8.0.28, utf8mb3 is also displayed in place of utf8 in columns of Information Schema tables, and in the output of SQL SHOW statements.



 To avoid ambiguity about the meaning of utf8, consider specifying utf8mb4

explicitly for character set references.

**10.9.4** **The** **ucs2** **Character** **Set** **(UCS-2** **Unicode** **Encoding)**

**Note**

The ucs2 character set is deprecated in MySQL 8.0.28; expect it to be removed in a future MySQL release. Please use utf8mb4 instead.

In UCS-2, every character is represented by a 2-byte Unicode code with the most significant byte first. For example: LATIN CAPITAL LETTER A has the code 0x0041 and it is stored as a 2-byte sequence: 0x00 0x41. CYRILLIC SMALL LETTER YERU (Unicode 0x044B) is stored as a 2- byte sequence: 0x04 0x4B. For Unicode characters and their codes, please refer to the [Unicode](http://www.unicode.org/) [Consortium website](http://www.unicode.org/).

The ucs2 character set has these characteristics:

• Supports BMP characters only (no support for supplementary characters)

• Uses a fixed-length 16-bit encoding and requires two bytes per character.

**10.9.5** **The** **utf16** **Character** **Set** **(UTF-16** **Unicode** **Encoding)**

The utf16 character set is the ucs2 character set with an extension that enables encoding of supplementary characters:

• For a BMP character, utf16 and ucs2 have identical storage characteristics: same code values, same encoding, same length.

• For a supplementary character, utf16 has a special sequence for representing the character using

32 bits. This is called the “surrogate” mechanism: For a number greater than 0xffff, take 10 bits and add them to 0xd800 and put them in the first 16-bit word, take 10 more bits and add them to 0xdc00 and put them in the next 16-bit word. Consequently, all supplementary characters require 32 bits, where the first 16 bits are a number between 0xd800 and 0xdbff, and the last 16 bits are a number between 0xdc00 and 0xdfff. Examples are in Section [15.5 Surrogates Area](http://www.unicode.org/versions/Unicode4.0.0/ch15.pdf) of the Unicode 4.0 document.

Because utf16 supports surrogates and ucs2 does not, there is a validity check that applies only in utf16: You cannot insert a top surrogate without a bottom surrogate, or vice versa. For example:

INSERT INTO t (ucs2\_column) VALUES (0xd800); /\* legal \*/

INSERT INTO t (utf16\_column)VALUES (0xd800); /\* illegal \*/

There is no validity check for characters that are technically valid but are not true Unicode (that is, characters that Unicode considers to be “unassigned code points” or “private use” characters or even “illegals” like 0xffff). For example, since U+F8FF is the Apple Logo, this is legal:

INSERT INTO t (utf16\_column)VALUES (0xf8ff); /\* legal \*/

Such characters cannot be expected to mean the same thing to everyone.

Because MySQL must allow for the worst case (that one character requires four bytes) the maximum length of a utf16 column or index is only half of the maximum length for a ucs2 column or index. For example, the maximum length of a MEMORY table index key is 3072 bytes, so these statements create tables with the longest permitted indexes for ucs2 and utf16 columns:

CREATE TABLE tf (s1 VARCHAR(1536) CHARACTER SET ucs2) ENGINE=MEMORY;

CREATE INDEX i ON tf (s1);

CREATE TABLE tg (s1 VARCHAR(768) CHARACTER SET utf16) ENGINE=MEMORY;



CREATE INDEX i ON tg (s1);

**10.9.6** **The** **utf16le** **Character** **Set** **(UTF-16LE** **Unicode** **Encoding)**

This is the same as utf16 but is little-endian rather than big-endian.

**10.9.7** **The** **utf32** **Character** **Set** **(UTF-32** **Unicode** **Encoding)**

The utf32 character set is fixed length (like ucs2 and unlike utf16). utf32 uses 32 bits for every character, unlike ucs2 (which uses 16 bits for every character), and unlike utf16 (which uses 16 bits for some characters and 32 bits for others).

utf32 takes twice as much space as ucs2 and more space than utf16, but utf32 has the same advantage as ucs2 that it is predictable for storage: The required number of bytes for utf32 equals the number of characters times 4. Also, unlike utf16, there are no tricks for encoding in utf32, so the stored value equals the code value.

To demonstrate how the latter advantage is useful, here is an example that shows how to determine a utf8mb4 value given the utf32 code value:

/\* Assume code value = 100cc LINEAR B WHEELED CHARIOT \*/

CREATE TABLE tmp (utf32\_col CHAR(1) CHARACTER SET utf32,

utf8mb4\_col CHAR(1) CHARACTER SET utf8mb4);

INSERT INTO tmp VALUES (0x000100cc,NULL);

UPDATE tmp SET utf8mb4\_col = utf32\_col;

SELECT HEX(utf32\_col),HEX(utf8mb4\_col) FROM tmp;

MySQL is very forgiving about additions of unassigned Unicode characters or private-use-area characters. There is in fact only one validity check for utf32: No code value may be greater than 0x10ffff. For example, this is illegal:

INSERT INTO t (utf32\_column) VALUES (0x110000); /\* illegal \*/

**10.9.8** **Converting** **Between** **3-Byte** **and** **4-Byte** **Unicode** **Character** **Sets**

This section describes issues that you may face when converting character data between the utf8mb3 and utf8mb4 character sets.

**Note**

This discussion focuses primarily on converting between utf8mb3 and utf8mb4, but similar principles apply to converting between the ucs2 character set and character sets such as utf16 or utf32.

The utf8mb3 and utf8mb4 character sets differ as follows:

• utf8mb3 supports only characters in the Basic Multilingual Plane (BMP). utf8mb4 additionally supports supplementary characters that lie outside the BMP.

• utf8mb3 uses a maximum of three bytes per character. utf8mb4 uses a maximum of four bytes per character.

**Note**

This discussion refers to the utf8mb3 and utf8mb4 character set names to be explicit about referring to 3-byte and 4-byte UTF-8 character set data.

One advantage of converting from utf8mb3 to utf8mb4 is that this enables applications to use supplementary characters. One tradeoff is that this may increase data storage space requirements.

In terms of table content, conversion from utf8mb3 to utf8mb4 presents no problems:

• For a BMP character, utf8mb4 and utf8mb3 have identical storage characteristics: same code values, same encoding, same length.

• For a supplementary character, utf8mb4 requires four bytes to store it, whereas utf8mb3 cannot store the character at all. When converting utf8mb3 columns to utf8mb4, you need not worry about converting supplementary characters because there are none.

In terms of table structure, these are the primary potential incompatibilities:

• For the variable-length character data types ([VARCHAR](#_bookmark60) and the [TEXT](#_bookmark61) types), the maximum permitted length in characters is less for utf8mb4 columns than for utf8mb3 columns.

• For all character data types ([CHAR](#_bookmark60), [VARCHAR](#_bookmark60), and the [TEXT](#_bookmark61) types), the maximum number of characters that can be indexed is less for utf8mb4 columns than for utf8mb3 columns.

Consequently, to convert tables from utf8mb3 to utf8mb4, it may be necessary to change some column or index definitions.

Tables can be converted from utf8mb3 to utf8mb4 by using ALTER TABLE. Suppose that a table has this definition:

|  |  |
| --- | --- |
| CREATE TABLE t1 (  col1 CHAR(10) CHARACTER SET utf8mb3  col2 CHAR(10) CHARACTER SET utf8mb3  ) CHARACTER SET utf8mb3; | COLLATE utf8mb3\_unicode\_ci NOT NULL,  COLLATE utf8mb3\_bin NOT NULL |

The following statement converts t1 to use utf8mb4:

|  |  |
| --- | --- |
| ALTER TABLE t1  DEFAULT CHARACTER SET utf8mb4,  MODIFY col1 CHAR(10)  CHARACTER SET utf8mb4 COLLATE  MODIFY col2 CHAR(10)  CHARACTER SET utf8mb4 COLLATE | utf8mb4\_unicode\_ci NOT NULL,  utf8mb4\_bin NOT NULL; |

The catch when converting from utf8mb3 to utf8mb4 is that the maximum length of a column or index key is unchanged in terms of *bytes*. Therefore, it is smaller in terms of *characters* because the maximum length of a character is four bytes instead of three. For the [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), and [TEXT](#_bookmark61) data types, watch for these issues when converting your MySQL tables:

• Check all definitions of utf8mb3 columns and make sure they do not exceed the maximum length for the storage engine.

• Check all indexes on utf8mb3 columns and make sure they do not exceed the maximum length for the storage engine. Sometimes the maximum can change due to storage engine enhancements.

If the preceding conditions apply, you must either reduce the defined length of columns or indexes, or continue to use utf8mb3 rather than utf8mb4.

Here are some examples where structural changes may be needed:

• A [TINYTEXT](#_bookmark61) column can hold up to 255 bytes, so it can hold up to 85 3-byte or 63 4-byte characters. Suppose that you have a [TINYTEXT](#_bookmark61) column that uses utf8mb3 but must be able to contain more than 63 characters. You cannot convert it to utf8mb4 unless you also change the data type to a longer type such as [TEXT](#_bookmark61).

Similarly, a very long [VARCHAR](#_bookmark60) column may need to be changed to one of the longer [TEXT](#_bookmark61) types if you want to convert it from utf8mb3 to utf8mb4.

• InnoDB has a maximum index length of 767 bytes for tables that use COMPACT or REDUNDANT row format, so for utf8mb3 or utf8mb4 columns, you can index a maximum of 255 or 191 characters, respectively. If you currently have utf8mb3 columns with indexes longer than 191 characters, you must index a smaller number of characters.



In an InnoDB table that uses COMPACT or REDUNDANT row format, these column and index definitions are legal:

col1 VARCHAR(500) CHARACTER SET utf8mb3, INDEX (col1(255))

To use utf8mb4 instead, the index must be smaller:

col1 VARCHAR(500) CHARACTER SET utf8mb4, INDEX (col1(191))

**Note**

For InnoDB tables that use COMPRESSED or DYNAMIC row format, index key prefixes longer than 767 bytes (up to 3072 bytes) are permitted. Tables created with these row formats enable you to index a maximum of 1024 or 768 characters for utf8mb3 or utf8mb4 columns, respectively. For related information, see Section 15.22, “InnoDB Limits” , and DYNAMIC Row Format.

The preceding types of changes are most likely to be required only if you have very long columns or indexes. Otherwise, you should be able to convert your tables from utf8mb3 to utf8mb4 without problems, using ALTER TABLE as described previously.

The following items summarize other potential incompatibilities:

• SET NAMES 'utf8mb4' causes use of the 4-byte character set for connection character sets. As long as no 4-byte characters are sent from the server, there should be no problems. Otherwise, applications that expect to receive a maximum of three bytes per character may have problems. Conversely, applications that expect to send 4-byte characters must ensure that the server understands them.

• For replication, if character sets that support supplementary characters are to be used on the source, all replicas must understand them as well.

Also, keep in mind the general principle that if a table has different definitions on the source and replica, this can lead to unexpected results. For example, the differences in maximum index key length make it risky to use utf8mb3 on the source and utf8mb4 on the replica.

If you have converted to utf8mb4, utf16, utf16le, or utf32, and then decide to convert back to utf8mb3 or ucs2 (for example, to downgrade to an older version of MySQL), these considerations apply:

• utf8mb3 and ucs2 data should present no problems.

• The server must be recent enough to recognize definitions referring to the character set from which you are converting.

• For object definitions that refer to the utf8mb4 character set, you can dump them with mysqldump prior to downgrading, edit the dump file to change instances of utf8mb4 to utf8, and reload the file in the older server, as long as there are no 4-byte characters in the data. The older server sees utf8 in the dump file object definitions and create new objects that use the (3-byte) utf8 character set.

**10.10** **Supported** **Character** **Sets** **and** **Collations**

This section indicates which character sets MySQL supports. There is one subsection for each group of related character sets. For each character set, the permissible collations are listed.

To list the available character sets and their default collations, use the SHOW CHARACTER SET statement or query the INFORMATION\_SCHEMA CHARACTER\_SETS table. For example:

mysql> **SHOW** **CHARACTER** **SET;**

+----------+---------------------------------+---------------------+--------+

| Charset | Description | Default collation | Maxlen |

+----------+---------------------------------+---------------------+--------+

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| | armscii8 | | ARMSCII-8 Armenian |  | armscii8\_general\_ci | | |  | 1 | |
| ascii | US ASCII | | ascii\_general\_ci | |  | 1 |
| big5 | Big5 Traditional Chinese | | big5\_chinese\_ci | |  | 2 |
| | binary | | Binary pseudo charset | | | binary |  |  | 1 | |
| | cp1250 | | Windows Central European | | | cp1250\_general\_ci | | | | 1 | |
| | cp1251 | | Windows Cyrillic |  | cp1251\_general\_ci | | |  | 1 | |
| cp1256 | Windows Arabic | | cp1256\_general\_ci | |  | 1 |
| cp1257 | Windows Baltic | | cp1257\_general\_ci | |  | 1 |
| | cp850 | | DOS West European |  | cp850 general ci | | |  | 1 | |
| | cp852 | | DOS Central European | | | cp852\_general\_ci | | | | 1 | |
| | cp866 | | DOS Russian |  | cp866\_general\_ci | | |  | 1 | |
| cp932 | SJIS for Windows Japanese | | cp932\_japanese\_ci | |  | 2 |
| dec8 | DEC West European | | dec8\_swedish\_ci | |  | 1 |
| | eucjpms | | UJIS for Windows Japanese | | | eucjpms japanese ci | | |  | 3 | |
| | euckr | | EUC-KR Korean | | | euckr\_korean\_ci | | | | 2 | |
| | gb18030 | | China National Standard GB18030 | | | gb18030\_chinese\_ci | | |  | 4 | |
| gb2312 | GB2312 Simplified Chinese | | gb2312\_chinese\_ci | |  | 2 |
| gbk | GBK Simplified Chinese | | gbk\_chinese\_ci | |  | 2 |
| | geostd8 | | GEOSTD8 Georgian |  | geostd8 general ci | | |  | 1 | |
| | greek | | ISO 8859-7 Greek | | | greek\_general\_ci | | | | 1 | |
| | hebrew | | ISO 8859-8 Hebrew |  | hebrew\_general\_ci | | |  | 1 | |
| hp8 | HP West European | | hp8\_english\_ci | |  | 1 |
| keybcs2 | DOS Kamenicky Czech-Slovak | | keybcs2\_general\_ci | |  | 1 |
| | koi8r | | KOI8-R Relcom Russian | | | koi8r general ci | | |  | 1 | |
| | koi8u | | KOI8-U Ukrainian | | | koi8u\_general\_ci | | | | 1 | |
| | latin1 | | cp1252 West European |  | latin1\_swedish\_ci | | |  | 1 | |
| latin2 | ISO 8859-2 Central European | | latin2\_general\_ci | |  | 1 |
| latin5 | ISO 8859-9 Turkish | | latin5\_turkish\_ci | |  | 1 |
| | latin7 | | ISO 8859-13 Baltic |  | latin7 general ci | | |  | 1 | |
| | macce | | Mac Central European | | | macce\_general\_ci | | | | 1 | |
| | macroman | | Mac West European |  | macroman\_general\_ci | | |  | 1 | |
| sjis | Shift-JIS Japanese | | sjis\_japanese\_ci | |  | 2 |
| swe7 | 7bit Swedish | | swe7\_swedish\_ci | |  | 1 |
| | tis620 | | TIS620 Thai |  | tis620 thai ci | | |  | 1 | |
| | ucs2 | | UCS-2 Unicode | | | ucs2\_general\_ci | | | | 2 | |
| | ujis | | EUC-JP Japanese |  | ujis\_japanese\_ci | | |  | 3 | |
| utf16 | UTF-16 Unicode | | utf16\_general\_ci | |  | 4 |
| utf16le | UTF-16LE Unicode | | utf16le\_general\_ci | |  | 4 |
| | utf32 | | UTF-32 Unicode |  | utf32 general ci | | |  | 4 | |
| | utf8mb3 | | UTF-8 Unicode | | | utf8mb3\_general\_ci | | | | 3 | |
| | utf8mb4 | | UTF-8 Unicode | | | utf8mb4 0900 ai ci | | | | 4 | |

+----------+---------------------------------+---------------------+--------+

|

In cases where a character set has multiple collations, it might not be clear which collation is most suitable for a given application. To avoid choosing the wrong collation, it can be helpful to perform some comparisons with representative data values to make sure that a given collation sorts values the way you expect.

**10.10.1** **Unicode** **Character** **Sets**

This section describes the collations available for Unicode character sets and their differentiating properties. For general information about Unicode, see [Section 10.9, “Unicode Support”](#_bookmark29) .

MySQL supports multiple Unicode character sets:

• utf8mb4: A UTF-8 encoding of the Unicode character set using one to four bytes per character.

• utf8mb3: A UTF-8 encoding of the Unicode character set using one to three bytes per character. This character set is deprecated in MySQL 8.0, and you should use utfmb4 instead.

• utf8: An alias for utf8mb3. In MySQL 8.0, this alias is deprecated; use utf8mb4 instead. utf8 is expected in a future release to become an alias for utf8mb4.

• ucs2: The UCS-2 encoding of the Unicode character set using two bytes per character. Deprecated in MySQL 8.0.28; you should expect support for this character set to be removed in a future release.

• utf16: The UTF-16 encoding for the Unicode character set using two or four bytes per character. Like ucs2 but with an extension for supplementary characters.



• utf16le: The UTF-16LE encoding for the Unicode character set. Like utf16 but little-endian rather than big-endian.

• utf32: The UTF-32 encoding for the Unicode character set using four bytes per character.

**Note**

The utf8mb3 character set is deprecated and you should expect it to be removed in a future MySQL release. Please use utf8mb4 instead. utf8 is currently an alias for utf8mb3, but it is now deprecated as such, and utf8 is expected subsequently to become a reference to utf8mb4. Beginning with MySQL 8.0.28, utf8mb3 is also displayed in place of utf8 in columns of Information Schema tables, and in the output of SQL SHOW statements.

To avoid ambiguity about the meaning of utf8, consider specifying utf8mb4 explicitly for character set references.

utf8mb4, utf16, utf16le, and utf32 support Basic Multilingual Plane (BMP) characters and supplementary characters that lie outside the BMP. utf8mb3 and ucs2 support only BMP characters.

Most Unicode character sets have a general collation (indicated by \_general in the name or by the absence of a language specifier), a binary collation (indicated by \_bin in the name), and several language-specific collations (indicated by language specifiers). For example, for utf8mb4, utf8mb4\_general\_ci and utf8mb4\_bin are its general and binary collations, and utf8mb4\_danish\_ci is one of its language-specific collations.

Most character sets have a single binary collation. utf8mb4 is an exception that has two: utf8mb4\_bin and (as of MySQL 8.0.17) utf8mb4\_0900\_bin. These two binary collations have the same sort order but are distinguished by their pad attribute and collating weight characteristics. See [Collation Pad Attributes](#_bookmark83), and [Character Collating Weights](#_bookmark84).

Collation support for utf16le is limited. The only collations available are utf16le\_general\_ci and utf16le\_bin. These are similar to utf16\_general\_ci and utf16\_bin.

• [Unicode Collation Algorithm (UCA) Versions](#_bookmark85)

• [Collation Pad Attributes](#_bookmark83)

• [Language-Specific Collations](#_bookmark86)

• [\_general\_ci Versus \_unicode\_ci Collations](#_bookmark87)

• [Character Collating Weights](#_bookmark84)

• [Miscellaneous Information](#_bookmark88)

**Unicode** **Collation** **Algorithm** **(UCA)** **Versions**

MySQL implements the *xxx*\_unicode\_ci collations according to the Unicode Collation Algorithm (UCA) described at <http://www.unicode.org/reports/tr10/>. The collation uses the version-4.0.0 UCA weight keys: <http://www.unicode.org/Public/UCA/4.0.0/allkeys-4.0.0.txt>. The *xxx*\_unicode\_ci collations have only partial support for the Unicode Collation Algorithm. Some characters are not supported, and combining marks are not fully supported. This affects languages such as Vietnamese, Yoruba, and Navajo. A combined character is considered different from the same character written with a single unicode character in string comparisons, and the two characters are considered to have a different length (for example, as returned by the CHAR\_LENGTH() function or in result set metadata).

Unicode collations based on UCA versions higher than 4.0.0 include the version in the collation name. Examples:

• utf8mb4\_unicode\_520\_ci is based on UCA 5.2.0 weight keys ([http://www.unicode.org/Public/](http://www.unicode.org/Public/UCA/5.2.0/allkeys.txt) [UCA/5.2.0/allkeys.txt](http://www.unicode.org/Public/UCA/5.2.0/allkeys.txt)),

• utf8mb4\_0900\_ai\_ci is based on UCA 9.0.0 weight keys ([http://www.unicode.org/Public/](http://www.unicode.org/Public/UCA/9.0.0/allkeys.txt) [UCA/9.0.0/allkeys.txt](http://www.unicode.org/Public/UCA/9.0.0/allkeys.txt)).

The LOWER() and UPPER() functions perform case folding according to the collation of their argument. A character that has uppercase and lowercase versions only in a Unicode version higher than 4.0.0 is converted by these functions only if the argument collation uses a high enough UCA version.

**Collation** **Pad** **Attributes**

Collations based on UCA 9.0.0 and higher are faster than collations based on UCA versions prior to 9.0.0. They also have a pad attribute of NO PAD, in contrast to PAD SPACE as used in collations based on UCA versions prior to 9.0.0. For comparison of nonbinary strings, NO PAD collations treat spaces at the end of strings like any other character (see [Trailing Space Handling in Comparisons](#_bookmark79)).

To determine the pad attribute for a collation, use the INFORMATION\_SCHEMA COLLATIONS table, which has a PAD\_ATTRIBUTE column. For example:

mysql> **SELECT** **COLLATION\_NAME,** **PAD\_ATTRIBUTE**

**FROM** **INFORMATION\_SCHEMA.COLLATIONS**

**WHERE** **CHARACTER\_SET\_NAME** **=** **'utf8mb4';**

+----------------------------+---------------+

| COLLATION\_NAME | PAD\_ATTRIBUTE |

+----------------------------+---------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| utf8mb4\_general\_ci  utf8mb4\_bin  utf8mb4\_unicode\_ci  utf8mb4\_icelandic\_ci  ...  utf8mb4\_0900\_ai\_ci  utf8mb4\_de\_pb\_0900\_ai\_ci  utf8mb4\_is\_0900\_ai\_ci  ...  utf8mb4\_ja\_0900\_as\_cs  utf8mb4\_ja\_0900\_as\_cs\_ks  utf8mb4\_0900\_as\_ci  utf8mb4\_ru\_0900\_ai\_ci  utf8mb4\_ru\_0900\_as\_cs  utf8mb4\_zh\_0900\_as\_cs  utf8mb4\_0900\_bin | |  |  |  |  |  |  |  |  |  |  |  |  |  | | PAD  PAD  PAD  PAD  NO  NO  NO  NO  NO  NO  NO  NO  NO  NO | SPACE  SPACE  SPACE  SPACE  PAD  PAD  PAD  PAD  PAD  PAD  PAD  PAD  PAD  PAD | |  |  |  |  |  |  |  |  |  |  |  |  |  | |

+----------------------------+---------------+

|

|

|

|

|

|

|

|

|

|

|

|

|

|

Comparison of nonbinary string values (CHAR, VARCHAR, and TEXT) that have a NO PAD collation differ from other collations with respect to trailing spaces. For example, 'a' and 'a ' compare as different strings, not the same string. This can be seen using the binary collations for utf8mb4. The pad attribute for utf8mb4\_bin is PAD SPACE, whereas for utf8mb4\_0900\_bin it is NO PAD. Consequently, operations involving utf8mb4\_0900\_bin do not add trailing spaces, and comparisons involving strings with trailing spaces may differ for the two collations:

mysql> **CREATE** **TABLE** **t1** **(c** **CHAR(10)** **COLLATE** **utf8mb4\_bin);**

Query OK, 0 rows affected (0.03 sec)

mysql> **INSERT** **INTO** **t1** **VALUES('a');**

Query OK, 1 row affected (0.01 sec)

mysql> **SELECT** **\*** **FROM** **t1** **WHERE** **c** **=** **'a** **';**

+------+

| c |

+------+

| a |

+------+

1 row in set (0.00 sec)

mysql> **ALTER** **TABLE** **t1** **MODIFY** **c** **CHAR(10)** **COLLATE** **utf8mb4\_0900\_bin;**

Query OK, 0 rows affected (0.02 sec)

Records: 0 Duplicates: 0 Warnings: [0](#_bookmark89)

mysql> **SELECT** **\*** **FROM** **t1** **WHERE** **c** **=** **'a** **';**

Empty set (0.00 sec)

**Language-Specific** **Collations**

MySQL implements language-specific Unicode collations if the ordering based only on the Unicode Collation Algorithm (UCA) does not work well for a language. Language-specific collations are UCA- based, with additional language tailoring rules. Examples of such rules appear later in this section. For questions about particular language orderings, <unicode.org> provides Common Locale Data Repository (CLDR) collation charts at <http://www.unicode.org/cldr/charts/30/collation/index.html>.

For example, the nonlanguage-specific utf8mb4\_0900\_ai\_ci and language-specific utf8mb4\_*LOCALE*\_0900\_ai\_ci Unicode collations each have these characteristics:

• The collation is based on UCA 9.0.0 and CLDR v30, is accent-insensitive and case-insensitive. These characteristics are indicated by \_0900, \_ai, and \_ci in the collation name. Exception: utf8mb4\_la\_0900\_ai\_ci is not based on CLDR because Classical Latin is not defined in CLDR.

• The collation works for all characters in the range [U+0, U+10FFFF].

• If the collation is not language specific, it sorts all characters, including supplementary characters, in default order (described following). If the collation is language specific, it sorts characters of the language correctly according to language-specific rules, and characters not in the language in default order.

• By default, the collation sorts characters having a code point listed in the DUCET table (Default Unicode Collation Element Table) according to the weight value assigned in the table. The collation sorts characters not having a code point listed in the DUCET table using their implicit weight value, which is constructed according to the UCA.

• For non-language-specific collations, characters in contraction sequences are treated as separate characters. For language-specific collations, contractions might change character sorting order.

A collation name that includes a locale code or language name shown in the following table is a language-specific collation. Unicode character sets may include collations for one or more of these languages.

**Table** **10.3** **Unicode** **Collation** **Language** **Specifiers**

|  |  |
| --- | --- |
| **Language** | **Language** **Specifier** |
| Bosnian | bs |
| Bulgarian | bg |
| Chinese | zh |
| Classical Latin | la or roman |
| Croatian | hr or croatian |
| Czech | cs or czech |
| Danish | da or danish |
| Esperanto | eo or esperanto |
| Estonian | et or estonian |
| Galician | gl |
| German phone book order | de\_pb or german2 |
| Hungarian | hu or hungarian |
| Icelandic | is or icelandic |
| Japanese | ja |
| Latvian | lv or latvian |
| Lithuanian | lt or lithuanian |

|  |  |
| --- | --- |
| **Language** | **Language** **Specifier** |
| Mongolian | mn |
| Norwegian / Bokmål | nb |
| Norwegian / Nynorsk | nn |
| Persian | persian |
| Polish | pl or polish |
| Romanian | ro or romanian |
| Russian | ru |
| Serbian | sr |
| Sinhala | sinhala |
| Slovak | sk or slovak |
| Slovenian | sl or slovenian |
| Modern Spanish | es or spanish |
| Traditional Spanish | es\_trad or spanish2 |
| Swedish | sv or swedish |
| Turkish | tr or turkish |
| Vietnamese | vi or vietnamese |

MySQL 8.0.30 and later provides the Bulgarian collations utf8mb4\_bg\_0900\_ai\_ci and utf8mb4\_bg\_0900\_as\_cs.

Croatian collations are tailored for these Croatian letters: Č , Ć , Dž , Đ , Lj, Nj, Š , Ž .

MySQL 8.0.30 and later provides the utf8mb4\_sr\_latn\_0900\_ai\_ci and utf8mb4\_sr\_latn\_0900\_as\_cs collations for Serbian and the utf8mb4\_bs\_0900\_ai\_ci and utf8mb4\_bs\_0900\_as\_cs collations for Bosnian, when these languages are written with the Latin alphabet.

Beginning with MySQL 8.0.30, MySQL provides collations for both major varieties of Norwegian: for Bokmål, you can use utf8mb4\_nb\_0900\_ai\_ci and utf8mb4\_nb\_0900\_as\_cs; for Nynorsk, MySQL now provides utf8mb4\_nn\_0900\_ai\_ci and utf8mb4\_nn\_0900\_as\_cs.

For Japanese, the utf8mb4 character set includes utf8mb4\_ja\_0900\_as\_cs and utf8mb4\_ja\_0900\_as\_cs\_ks collations. Both collations are accent-sensitive and case-sensitive. utf8mb4\_ja\_0900\_as\_cs\_ks is also kana-sensitive and distinguishes Katakana characters from Hiragana characters, whereas utf8mb4\_ja\_0900\_as\_cs treats Katakana and Hiragana characters as equal for sorting. Applications that require a Japanese collation but not kana sensitivity may use utf8mb4\_ja\_0900\_as\_cs for better sort performance. utf8mb4\_ja\_0900\_as\_cs uses three weight levels for sorting; utf8mb4\_ja\_0900\_as\_cs\_ks uses four.

For Classical Latin collations that are accent-insensitive, I and J compare as equal, and U and V compare as equal. I and J, and U and V compare as equal on the base letter level. In other words, J is regarded as an accented I, and U is regarded as an accented V.

MySQL 8.0.30 and later provides collations for the Mongolian language when written with Cyrillic characters, utf8mb4\_mn\_cyrl\_0900\_ai\_ci and utf8mb4\_mn\_cyrl\_0900\_as\_cs.

Spanish collations are available for modern and traditional Spanish. For both, ñ (n-tilde) is a separate letter between n and o. In addition, for traditional Spanish, ch is a separate letter between c and d, and ll is a separate letter between l and m.

Traditional Spanish collations may also be used for Asturian and Galician. Beginning with MySQL 8.0.30, MySQL also provides utf8mb4\_gl\_0900\_ai\_ci and utf8mb4\_gl\_0900\_as\_cs

collations for Galician. (These are the same collations as utf8mb4\_es\_0900\_ai\_ci and utf8mb4\_es\_0900\_as\_cs, respectively.)

Swedish collations include Swedish rules. For example, in Swedish, the following relationship holds, which is not something expected by a German or French speaker:

Ü = Y < Ö

**\_general\_ci** **Versus** **\_unicode\_ci** **Collations**

For any Unicode character set, operations performed using the *xxx*\_general\_ci collation are faster than those for the *xxx*\_unicode\_ci collation. For example, comparisons for the utf8mb4\_general\_ci collation are faster, but slightly less correct, than comparisons for utf8mb4\_unicode\_ci. The reason is that utf8mb4\_unicode\_ci supports mappings such as expansions; that is, when one character compares as equal to combinations of other characters. For example, ß is equal to ss in German and some other languages. utf8mb4\_unicode\_ci also supports contractions and ignorable characters. utf8mb4\_general\_ci is a legacy collation that does not support expansions, contractions, or ignorable characters. It can make only one-to-one comparisons between characters.

To further illustrate, the following equalities hold in both utf8mb4\_general\_ci and utf8mb4\_unicode\_ci (for the effect of this in comparisons or searches, see [Section 10.8.6,](#_bookmark27) [“Examples of the Effect of Collation”](#_bookmark27)):

Ä = A

Ö = O

Ü = U

A difference between the collations is that this is true for utf8mb4\_general\_ci:

ß = s

Whereas this is true for utf8mb4\_unicode\_ci, which supports the German DIN-1 ordering (also known as dictionary order):

ß = ss

MySQL implements language-specific Unicode collations if the ordering with utf8mb4\_unicode\_ci does not work well for a language. For example, utf8mb4\_unicode\_ci works fine for German dictionary order and French, so there is no need to create special utf8mb4 collations.

utf8mb4\_general\_ci also is satisfactory for both German and French, except that ß is equal to s, and not to ss. If this is acceptable for your application, you should use utf8mb4\_general\_ci because it is faster. If this is not acceptable (for example, if you require German dictionary order), use utf8mb4\_unicode\_ci because it is more accurate.

If you require German DIN-2 (phone book) ordering, use the utf8mb4\_german2\_ci collation, which compares the following sets of characters equal:

Ä = Æ = AE

Ö = Œ = OE

Ü = UE

ß = ss

utf8mb4\_german2\_ci is similar to latin1\_german2\_ci, but the latter does not compare Æ equal to AE or Œ equal to OE. There is no utf8mb4\_german\_ci corresponding to latin1\_german\_ci for German dictionary order because utf8mb4\_general\_ci suffices.

**Character** **Collating** **Weights**

A character's collating weight is determined as follows:

• For all Unicode collations except the \_bin (binary) collations, MySQL performs a table lookup to find a character's collating weight.

• For \_bin collations except utf8mb4\_0900\_bin, the weight is based on the code point, possibly with leading zero bytes added.

• For utf8mb4\_0900\_bin, the weight is the utf8mb4 encoding bytes. The sort order is the same as for utf8mb4\_bin, but much faster.

Collating weights can be displayed using the WEIGHT\_STRING() function. (See Section 12.8, “String Functions and Operators” .) If a collation uses a weight lookup table, but a character is not in the table (for example, because it is a “new” character), collating weight determination becomes more complex:

• For BMP characters in general collations (*xxx*\_general\_ci), the weight is the code point.

• For BMP characters in UCA collations (for example, *xxx*\_unicode\_ci and language-specific collations), the following algorithm applies:

if (code >= 0x3400 && code <= 0x4DB5)

base= 0xFB80; /\* CJK Ideograph Extension \*/

else if (code >= 0x4E00 && code <= 0x9FA5)

base= 0xFB40; /\* CJK Ideograph \*/

else

base= 0xFBC0; /\* All other characters \*/

aaaa= base + (code >> 15);

bbbb= (code & 0x7FFF) | 0x8000;

The result is a sequence of two collating elements, aaaa followed by bbbb. For example:

mysql> **SELECT** **HEX(WEIGHT\_STRING(\_ucs2** **0x04CF** **COLLATE** **ucs2\_unicode\_ci));**

+----------------------------------------------------------+

| HEX(WEIGHT\_STRING(\_ucs2 0x04CF COLLATE ucs2\_unicode\_ci)) |

+----------------------------------------------------------+

| FBC084CF |

+----------------------------------------------------------+

Thus, U+04cf CYRILLIC SMALL LETTER PALOCHKA (#) is, with all UCA 4.0.0 collations, greater than U+04c0 CYRILLIC LETTER PALOCHKA ( Ӏ ). With UCA 5.2.0 collations, all palochkas sort together.

• For supplementary characters in general collations, the weight is the weight for 0xfffd REPLACEMENT CHARACTER. For supplementary characters in UCA 4.0.0 collations, their collating weight is 0xfffd. That is, to MySQL, all supplementary characters are equal to each other, and greater than almost all BMP characters.

An example with Deseret characters and COUNT(DISTINCT):

CREATE TABLE t (s1 VARCHAR(5) CHARACTER SET utf32 COLLATE utf32\_unicode\_ci);

INSERT INTO t VALUES (0xfffd); /\* REPLACEMENT CHARACTER \*/

INSERT INTO t VALUES (0x010412); /\* DESERET CAPITAL LETTER BEE \*/

INSERT INTO t VALUES (0x010413); /\* DESERET CAPITAL LETTER TEE \*/

SELECT COUNT(DISTINCT s1) FROM t;

The result is 2 because in the MySQL *xxx*\_unicode\_ci collations, the replacement character has a weight of 0x0dc6, whereas Deseret Bee and Deseret Tee both have a weight of 0xfffd. (Were the utf32\_general\_ci collation used instead, the result is 1 because all three characters have a weight of 0xfffd in that collation.)

An example with cuneiform characters and WEIGHT\_STRING():

/\*

The four characters in the INSERT string are

00000041 # LATIN CAPITAL LETTER A

0001218F # CUNEIFORM SIGN KAB

000121A7 # CUNEIFORM SIGN KISH

00000042 # LATIN CAPITAL LETTER B

\*/

CREATE TABLE t (s1 CHAR(4) CHARACTER SET utf32 COLLATE utf32\_unicode\_ci);

INSERT INTO t VALUES (0x000000410001218f000121a700000042);

SELECT HEX(WEIGHT\_STRING(s1)) FROM t;

The result is:

0E33 FFFD FFFD 0E4A

0E33 and 0E4A are primary weights as in [UCA 4.0.0](ftp://www.unicode.org/Public/UCA/4.0.0/allkeys-4.0.0.txt). FFFD is the weight for KAB and also for KISH.

The rule that all supplementary characters are equal to each other is nonoptimal but is not expected to cause trouble. These characters are very rare, so it is very rare that a multi-character string consists entirely of supplementary characters. In Japan, since the supplementary characters are obscure Kanji ideographs, the typical user does not care what order they are in, anyway. If you really want rows sorted by the MySQL rule and secondarily by code point value, it is easy:

ORDER BY s1 COLLATE utf32\_unicode\_ci, s1 COLLATE utf32\_bin

• For supplementary characters based on UCA versions higher than 4.0.0 (for example, *xxx*\_unicode\_520\_ci), supplementary characters do not necessarily all have the same collating weight. Some have explicit weights from the UCA allkeys.txt file. Others have weights calculated from this algorithm:

aaaa= base + (code >> 15);

bbbb= (code & 0x7FFF) | 0x8000;

There is a difference between “ordering by the character's code value” and “ordering by the character's binary representation,” a difference that appears only with utf16\_bin, because of surrogates.

Suppose that utf16\_bin (the binary collation for utf16) was a binary comparison “byte by byte” rather than “character by character.” If that were so, the order of characters in utf16\_bin would differ from the order in utf8mb4\_bin. For example, the following chart shows two rare characters. The first character is in the range E000-FFFF, so it is greater than a surrogate but less than a supplementary. The second character is a supplementary.

Code point Character utf8mb4 utf16

0FF9D HALFWIDTH KATAKANA LETTER N EF BE 9D FF [9D](#_bookmark90)

10384 UGARITIC LETTER DELTA F0 90 8E 84 D8 00 DF [84](#_bookmark91)

The two characters in the chart are in order by code point value because 0xff9d < 0x10384. And they are in order by utf8mb4 value because 0xef < 0xf0. But they are not in order by utf16 value, if we use byte-by-byte comparison, because 0xff > 0xd8.

So MySQL's utf16\_bin collation is not “byte by byte.” It is “by code point.” When MySQL sees a supplementary-character encoding in utf16, it converts to the character's code-point value, and then compares. Therefore, utf8mb4\_bin and utf16\_bin are the same ordering. This is consistent with the SQL:2008 standard requirement for a UCS\_BASIC collation: “UCS\_BASIC is a collation in which the ordering is determined entirely by the Unicode scalar values of the characters in the strings being sorted. It is applicable to the UCS character repertoire. Since every character repertoire is a subset of the UCS repertoire, the UCS\_BASIC collation is potentially applicable to every character set. NOTE 11: The Unicode scalar value of a character is its code point treated as an unsigned integer.”

If the character set is ucs2, comparison is byte-by-byte, but ucs2 strings should not contain surrogates, anyway.

**Miscellaneous** **Information**

The *xxx*\_general\_mysql500\_ci collations preserve the pre-5.1.24 ordering of the original *xxx*\_general\_ci collations and permit upgrades for tables created before MySQL 5.1.24 (Bug #27877).

**10.10.2** **West** **European** **Character** **Sets**

Western European character sets cover most West European languages, such as French, Spanish, Catalan, Basque, Portuguese, Italian, Albanian, Dutch, German, Danish, Swedish, Norwegian, Finnish, Faroese, Icelandic, Irish, Scottish, and English.

• ascii (US ASCII) collations:

• ascii\_bin

• ascii\_general\_ci (default)

• cp850 (DOS West European) collations:

• cp850\_bin

• cp850\_general\_ci (default)

• dec8 (DEC Western European) collations:

• dec8\_bin

• dec8\_swedish\_ci (default)

The dec character set is deprecated in MySQL 8.0.28; expect support for it to be removed in a subsequent MySQL release.

• hp8 (HP Western European) collations:

• hp8\_bin

• hp8\_english\_ci (default)

The hp8 character set is deprecated in MySQL 8.0.28; expect support for it to be removed in a subsequent MySQL release.

• latin1 (cp1252 West European) collations:

• latin1\_bin

• latin1\_danish\_ci

• latin1\_general\_ci

• latin1\_general\_cs

• latin1\_german1\_ci

• latin1\_german2\_ci

• latin1\_spanish\_ci

• latin1\_swedish\_ci (default)

MySQL's latin1 is the same as the Windows cp1252 character set. This means it is the same as the official ISO 8859-1 or IANA (Internet Assigned Numbers Authority) latin1, except that IANA latin1 treats the code points between 0x80 and 0x9f as “undefined,” whereas cp1252, and therefore MySQL's latin1, assign characters for those positions. For example, 0x80 is the Euro sign. For the “undefined” entries in cp1252, MySQL translates 0x81 to Unicode 0x0081, 0x8d to 0x008d, 0x8f to 0x008f, 0x90 to 0x0090, and 0x9d to 0x009d.

The latin1\_swedish\_ci collation is the default that probably is used by the majority of MySQL customers. Although it is frequently said that it is based on the Swedish/Finnish collation rules, there are Swedes and Finns who disagree with this statement.

The latin1\_german1\_ci and latin1\_german2\_ci collations are based on the DIN-1 and DIN-2 standards, where DIN stands for *Deutsches* *Institut* *für* *Normung* (the German equivalent of ANSI). DIN-1 is called the “dictionary collation” and DIN-2 is called the “phone book collation.” For an example of the effect this has in comparisons or when doing searches, see [Section 10.8.6,](#_bookmark27) [“Examples of the Effect of Collation”](#_bookmark27) .

• latin1\_german1\_ci (dictionary) rules:

Ä = A

Ö = O

Ü = U

ß = s

• latin1\_german2\_ci (phone-book) rules:

Ä = AE

Ö = OE

Ü = UE

ß = ss

In the latin1\_spanish\_ci collation, ñ (n-tilde) is a separate letter between n and o.

• macroman (Mac West European) collations:

• macroman\_bin

• macroman\_general\_ci (default)

macroroman is deprecated in MySQL 8.0.28; expect support for it to be removed in a subsequent MySQL release.

• swe7 (7bit Swedish) collations:

• swe7\_bin

• swe7\_swedish\_ci (default)

**10.10.3** **Central** **European** **Character** **Sets**

MySQL provides some support for character sets used in the Czech Republic, Slovakia, Hungary, Romania, Slovenia, Croatia, Poland, and Serbia (Latin).

• cp1250 (Windows Central European) collations:

• cp1250\_bin

• cp1250\_croatian\_ci

• cp1250\_czech\_cs

• cp1250\_general\_ci (default)

• cp1250\_polish\_ci

• cp852 (DOS Central European) collations:

• cp852\_bin

• cp852\_general\_ci (default)

• keybcs2 (DOS Kamenicky Czech-Slovak) collations:

• keybcs2\_bin

• keybcs2\_general\_ci (default)

• latin2 (ISO 8859-2 Central European) collations:

• latin2\_bin

• latin2\_croatian\_ci

• latin2\_czech\_cs

• latin2\_general\_ci (default)

• latin2\_hungarian\_ci

• macce (Mac Central European) collations:

• macce\_bin

• macce\_general\_ci (default)

macce is deprecated in MySQL 8.0.28; expect support for it to be removed in a subsequent MySQL release.

**10.10.4** **South** **European** **and** **Middle** **East** **Character** **Sets**

South European and Middle Eastern character sets supported by MySQL include Armenian, Arabic, Georgian, Greek, Hebrew, and Turkish.

• armscii8 (ARMSCII-8 Armenian) collations:

• armscii8\_bin

• armscii8\_general\_ci (default)

• cp1256 (Windows Arabic) collations:

• cp1256\_bin

• cp1256\_general\_ci (default)

• geostd8 (GEOSTD8 Georgian) collations:

• geostd8\_bin

• geostd8\_general\_ci (default)

• greek (ISO 8859-7 Greek) collations:

• greek\_bin

• greek\_general\_ci (default)

• hebrew (ISO 8859-8 Hebrew) collations:

• hebrew\_bin

• hebrew\_general\_ci (default)

• latin5 (ISO 8859-9 Turkish) collations:

• latin5\_bin

• latin5\_turkish\_ci (default)

**10.10.5** **Baltic** **Character** **Sets**

The Baltic character sets cover Estonian, Latvian, and Lithuanian languages.

• cp1257 (Windows Baltic) collations:

• cp1257\_bin

• cp1257\_general\_ci (default)

• cp1257\_lithuanian\_ci

• latin7 (ISO 8859-13 Baltic) collations:

• latin7\_bin

• latin7\_estonian\_cs

• latin7\_general\_ci (default)

• latin7\_general\_cs

**10.10.6** **Cyrillic** **Character** **Sets**

The Cyrillic character sets and collations are for use with Belarusian, Bulgarian, Russian, Ukrainian, and Serbian (Cyrillic) languages.

• cp1251 (Windows Cyrillic) collations:

• cp1251\_bin

• cp1251\_bulgarian\_ci

• cp1251\_general\_ci (default)

• cp1251\_general\_cs

• cp1251\_ukrainian\_ci

• cp866 (DOS Russian) collations:

• cp866\_bin

• cp866\_general\_ci (default)

• koi8r (KOI8-R Relcom Russian) collations:

• koi8r\_bin

• koi8r\_general\_ci (default)

• koi8u (KOI8-U Ukrainian) collations:

• koi8u\_bin

• koi8u\_general\_ci (default)

**10.10.7** **Asian** **Character** **Sets**

The Asian character sets that we support include Chinese, Japanese, Korean, and Thai. These can be complicated. For example, the Chinese sets must allow for thousands of different characters. See [Section 10.10.7.1, “The cp932 Character Set”](#_bookmark92) , for additional information about the cp932 and sjis

character sets. See [Section 10.10.7.2, “The gb18030 Character Set”](#_bookmark93) , for additional information about character set support for the Chinese National Standard GB 18030.

For answers to some common questions and problems relating support for Asian character sets in MySQL, see Section A.11, “MySQL 8.0 FAQ: MySQL Chinese, Japanese, and Korean Character Sets” .

• big5 (Big5 Traditional Chinese) collations:

• big5\_bin

• big5\_chinese\_ci (default)

• [cp932](#_bookmark92) (SJIS for Windows Japanese) collations:

• cp932\_bin

• cp932\_japanese\_ci (default)

• eucjpms (UJIS for Windows Japanese) collations:

• eucjpms\_bin

• eucjpms\_japanese\_ci (default)

• euckr (EUC-KR Korean) collations:

• euckr\_bin

• euckr\_korean\_ci (default)

• gb2312 (GB2312 Simplified Chinese) collations:

• gb2312\_bin

• gb2312\_chinese\_ci (default)

• gbk (GBK Simplified Chinese) collations:

• gbk\_bin

• gbk\_chinese\_ci (default)

• [gb18030](#_bookmark93) (China National Standard GB18030) collations:

• gb18030\_bin

• gb18030\_chinese\_ci (default)

• gb18030\_unicode\_520\_ci

• sjis (Shift-JIS Japanese) collations:

• sjis\_bin

• sjis\_japanese\_ci (default)

• tis620 (TIS620 Thai) collations:

• tis620\_bin

• tis620\_thai\_ci (default)

• ujis (EUC-JP Japanese) collations:

• ujis\_bin

• ujis\_japanese\_ci (default)

The big5\_chinese\_ci collation sorts on number of strokes.

**10.10.7.1** **The** **cp932** **Character** **Set**

**Why** **is** **cp932** **needed?**

In MySQL, the sjis character set corresponds to the Shift\_JIS character set defined by IANA, which supports JIS X0201 and JIS X0208 characters. (See [http://www.iana.org/assignments/character-](http://www.iana.org/assignments/character-sets) [sets](http://www.iana.org/assignments/character-sets).)

However, the meaning of “SHIFT JIS” as a descriptive term has become very vague and it often includes the extensions to Shift\_JIS that are defined by various vendors.

For example, “SHIFT JIS” used in Japanese Windows environments is a Microsoft extension of Shift\_JIS and its exact name is Microsoft Windows Codepage : 932 or cp932. In addition to the characters supported by Shift\_JIS, cp932 supports extension characters such as NEC special characters, NEC selected— IBM extended characters, and IBM selected characters.

Many Japanese users have experienced problems using these extension characters. These problems stem from the following factors:

• MySQL automatically converts character sets.

• Character sets are converted using Unicode (ucs2).

• The sjis character set does not support the conversion of these extension characters.

• There are several conversion rules from so-called “SHIFT JIS” to Unicode, and some characters are converted to Unicode differently depending on the conversion rule. MySQL supports only one of these rules (described later).

The MySQL cp932 character set is designed to solve these problems.

Because MySQL supports character set conversion, it is important to separate IANA Shift\_JIS and cp932 into two different character sets because they provide different conversion rules.

**How** **does** **cp932** **differ** **from** **sjis?**

The cp932 character set differs from sjis in the following ways:

• cp932 supports NEC special characters, NEC selected— IBM extended characters, and IBM selected characters.

• Some cp932 characters have two different code points, both of which convert to the same Unicode code point. When converting from Unicode back to cp932, one of the code points must be selected. For this “round trip conversion,” the rule recommended by Microsoft is used. (See [http://](http://support.microsoft.com/kb/170559/EN-US/) [support.microsoft.com/kb/170559/EN-US/](http://support.microsoft.com/kb/170559/EN-US/).)

The conversion rule works like this:

• If the character is in both JIS X 0208 and NEC special characters, use the code point of JIS X 0208.

• If the character is in both NEC special characters and IBM selected characters, use the code point of NEC special characters.

• If the character is in both IBM selected characters and NEC selected— IBM extended characters, use the code point of IBM extended characters.

The table shown at <https://msdn.microsoft.com/en-us/goglobal/cc305152.aspx> provides information about the Unicode values of cp932 characters. For cp932 table entries with characters under which a four-digit number appears, the number represents the corresponding Unicode (ucs2) encoding. For table entries with an underlined two-digit value appears, there is a range of cp932 character values that begin with those two digits. Clicking such a table entry takes you to a page that displays the Unicode value for each of the cp932 characters that begin with those digits.

The following links are of special interest. They correspond to the encodings for the following sets of characters:

• NEC special characters (lead byte 0x87):

<https://msdn.microsoft.com/en-us/goglobal/gg674964>

• NEC selected— IBM extended characters (lead byte 0xED and 0xEE):

<https://msdn.microsoft.com/en-us/goglobal/gg671837>

<https://msdn.microsoft.com/en-us/goglobal/gg671838>

• IBM selected characters (lead byte 0xFA, 0xFB, 0xFC):

<https://msdn.microsoft.com/en-us/goglobal/gg671839>

<https://msdn.microsoft.com/en-us/goglobal/gg671840>

<https://msdn.microsoft.com/en-us/goglobal/gg671841>

• cp932 supports conversion of user-defined characters in combination with eucjpms, and solves the problems with sjis/ujis conversion. For details, please refer to [http://www.sljfaq.org/afaq/](http://www.sljfaq.org/afaq/encodings.html) [encodings.html](http://www.sljfaq.org/afaq/encodings.html).

For some characters, conversion to and from ucs2 is different for sjis and cp932. The following tables illustrate these differences.

Conversion to ucs2:

|  |  |  |
| --- | --- | --- |
| **sjis/cp932** **Value** | **sjis** **->** **ucs2** **Conversion** | **cp932** **->** **ucs2** **Conversion** |
| 5C | 005C | 005C |
| 7E | 007E | 007E |
| 815C | 2015 | 2015 |
| 815F | 005C | FF3C |
| 8160 | 301C | FF5E |
| 8161 | 2016 | 2225 |
| 817C | 2212 | FF0D |
| 8191 | 00A2 | FFE0 |
| 8192 | 00A3 | FFE1 |
| 81CA | 00AC | FFE2 |

Conversion from ucs2:

|  |  |  |
| --- | --- | --- |
| **ucs2** **value** | **ucs2** **->** **sjis** **Conversion** | **ucs2** **->** **cp932** **Conversion** |
| 005C | 815F | 5C |
| 007E | 7E | 7E |

|  |  |  |
| --- | --- | --- |
| **ucs2** **value** | **ucs2** **->** **sjis** **Conversion** | **ucs2** **->** **cp932** **Conversion** |
| 00A2 | 8191 | 3F |
| 00A3 | 8192 | 3F |
| 00AC | 81CA | 3F |
| 2015 | 815C | 815C |
| 2016 | 8161 | 3F |
| 2212 | 817C | 3F |
| 2225 | 3F | 8161 |
| 301C | 8160 | 3F |
| FF0D | 3F | 817C |
| FF3C | 3F | 815F |
| FF5E | 3F | 8160 |
| FFE0 | 3F | 8191 |
| FFE1 | 3F | 8192 |
| FFE2 | 3F | 81CA |

Users of any Japanese character sets should be aware that using --character-set-client- handshake (or --skip-character-set-client-handshake) has an important effect. See Section 5.1.7, “Server Command Options” .

**10.10.7.2** **The** **gb18030** **Character** **Set**

In MySQL, the gb18030 character set corresponds to the “Chinese National Standard GB 18030-2005: Information technology — Chinese coded character set” , which is the official character set of the People's Republic of China (PRC).

**Characteristics** **of** **the** **MySQL** **gb18030** **Character** **Set**

• Supports all code points defined by the GB 18030-2005 standard. Unassigned code points in the ranges (GB+8431A439, GB+90308130) and (GB+E3329A36, GB+EF39EF39) are treated as ' ?' (0x3F). Conversion of unassigned code points return ' ?'.

• Supports UPPER and LOWER conversion for all GB18030 code points. Case folding defined by Unicode is also supported (based on CaseFolding-6.3.0.txt).

• Supports Conversion of data to and from other character sets.

• Supports SQL statements such as SET NAMES.

• Supports comparison between gb18030 strings, and between gb18030 strings and strings of other character sets. There is a conversion if strings have different character sets. Comparisons that include or ignore trailing spaces are also supported.

• The private use area (U+E000, U+F8FF) in Unicode is mapped to gb18030.

• There is no mapping between (U+D800, U+DFFF) and GB18030. Attempted conversion of code points in this range returns ' ?'.

• If an incoming sequence is illegal, an error or warning is returned. If an illegal sequence is used in CONVERT(), an error is returned. Otherwise, a warning is returned.

• For consistency with utf8mb3 and utf8mb4, UPPER is not supported for ligatures.

• Searches for ligatures also match uppercase ligatures when using the gb18030\_unicode\_520\_ci collation.

• If a character has more than one uppercase character, the chosen uppercase character is the one whose lowercase is the character itself.

• The minimum multibyte length is 1 and the maximum is 4. The character set determines the length of a sequence using the first 1 or 2 bytes.

**Supported** **Collations**

• gb18030\_bin: A binary collation.

• gb18030\_chinese\_ci: The default collation, which supports Pinyin. Sorting of non-Chinese characters is based on the order of the original sort key. The original sort key is GB(UPPER(ch)) if UPPER(ch) exists. Otherwise, the original sort key is GB(ch). Chinese characters are sorted according to the Pinyin collation defined in the Unicode Common Locale Data Repository (CLDR 24). Non-Chinese characters are sorted before Chinese characters with the exception of GB+FE39FE39, which is the code point maximum.

• gb18030\_unicode\_520\_ci: A Unicode collation. Use this collation if you need to ensure that ligatures are sorted correctly.

**10.10.8** **The** **Binary** **Character** **Set**

The binary character set is the character set for binary strings, which are sequences of bytes. The binary character set has one collation, also named binary. Comparison and sorting are based on numeric byte values, rather than on numeric character code values (which for multibyte characters differ from numeric byte values). For information about the differences between the binary collation of the binary character set and the \_bin collations of nonbinary character sets, see [Section 10.8.5,](#_bookmark26) [“The binary Collation Compared to \_bin Collations”](#_bookmark26) .

For the binary character set, the concepts of lettercase and accent equivalence do not apply:

• For single-byte characters stored as binary strings, character and byte boundaries are the same, so lettercase and accent differences are significant in comparisons. That is, the binary collation is case-sensitive and accent-sensitive.

mysql> **SET** **NAMES** **'binary';**

mysql> **SELECT** **CHARSET('abc'),** **COLLATION('abc');**

+----------------+------------------+

| CHARSET('abc') | COLLATION('abc') |

+----------------+------------------+

| binary | binary |

+----------------+------------------+

mysql> **SELECT** **'abc'** **=** **'ABC',** **'a'** **=** **'ä';**

+---------------+------------+

| 'abc' = 'ABC' | 'a' = 'ä' |

+---------------+------------+

| 0 | 0 |

+---------------+------------+

• For multibyte characters stored as binary strings, character and byte boundaries differ. Character boundaries are lost, so comparisons that depend on them are not meaningful.

To perform lettercase conversion of a binary string, first convert it to a nonbinary string using a character set appropriate for the data stored in the string:

mysql> **SET** **@str** **=** **BINARY** **'New** **York';**

mysql> **SELECT** **LOWER(@str),** **LOWER(CONVERT(@str** **USING** **utf8mb4));**

+-------------+------------------------------------+

| LOWER(@str) | LOWER(CONVERT(@str USING utf8mb4)) |

+-------------+------------------------------------+

|  |  |  |  |
| --- | --- | --- | --- |
| | New York | | new | york | | |

+-------------+------------------------------------+

To convert a string expression to a binary string, these constructs are equivalent:



BINARY *expr*

CAST(*expr* AS BINARY)

CONVERT(*expr* USING BINARY)

If a value is a character string literal, the \_binary introducer may be used to designate it as a binary string. For example:

\_binary 'a'

The \_binary introducer is permitted for hexadecimal literals and bit-value literals as well, but unnecessary; such literals are binary strings by default.

For more information about introducers, see [Section 10.3.8, “Character Set Introducers”](#_bookmark14) .

**Note**

Within the mysql client, binary strings display using hexadecimal notation, depending on the value of the --binary-as-hex. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client” .

**10.11** **Restrictions** **on** **Character** **Sets**

• Identifiers are stored in mysql database tables (user, db, and so forth) using utf8mb3, but identifiers can contain only characters in the Basic Multilingual Plane (BMP). Supplementary characters are not permitted in identifiers.

• The ucs2, utf16, utf16le, and utf32 character sets have the following restrictions:

• None of them can be used as the client character set. See [Impermissible Client Character Sets](#_bookmark65).

• It is currently not possible to use LOAD DATA to load data files that use these character sets.

• FULLTEXT indexes cannot be created on a column that uses any of these character sets. However, you can perform IN BOOLEAN MODE searches on the column without an index.

• The REGEXP and RLIKE operators work in byte-wise fashion, so they are not multibyte safe and may produce unexpected results with multibyte character sets. In addition, these operators compare characters by their byte values and accented characters may not compare as equal even if a given

collation treats them as equal.

**10.12** **Setting** **the** **Error** **Message** **Language**

By default, mysqld produces error messages in English, but they can be displayed instead in any of several other languages: Czech, Danish, Dutch, Estonian, French, German, Greek, Hungarian, Italian, Japanese, Korean, Norwegian, Norwegian-ny, Polish, Portuguese, Romanian, Russian, Slovak, Spanish, or Swedish. This applies to messages the server writes to the error log and sends to clients.

To select the language in which the server writes error messages, follow the instructions in this section. For information about changing the character set for error messages (rather than the language), see [Section 10.6, “Error Message Character Set”](#_bookmark19) . For general information about configuring error logging, see Section 5.4.2, “The Error Log” .

The server searches for the error message file using these rules:

• It looks for the file in a directory constructed from two system variable values, lc\_messages\_dir and lc\_messages, with the latter converted to a language name. Suppose that you start the server using this command:

mysqld --lc\_messages\_dir=/usr/share/mysql --lc\_messages=fr\_FR

In this case, mysqld maps the locale fr\_FR to the language french and looks for the error file in the /usr/share/mysql/french directory.

By default, the language files are located in the share/mysql/*LANGUAGE* directory under the MySQL base directory.

• If the message file cannot be found in the directory constructed as just described, the server ignores the lc\_messages value and uses only the lc\_messages\_dir value as the location in which to look.

• If the server cannot find the configured message file, it writes a message to the error log to indicate the problem and defaults to built-in English messages.

The lc\_messages\_dir system variable can be set only at server startup and has only a global read-only value at runtime. lc\_messages can be set at server startup and has global and session values that can be modified at runtime. Thus, the error message language can be changed while the server is running, and each client can have its own error message language by setting its session lc\_messages value to the desired locale name. For example, if the server is using the fr\_FR locale for error messages, a client can execute this statement to receive error messages in English:

SET lc\_messages = 'en\_US';

**10.13** **Adding** **a** **Character** **Set**

This section discusses the procedure for adding a character set to MySQL. The proper procedure depends on whether the character set is simple or complex:

• If the character set does not need special string collating routines for sorting and does not need multibyte character support, it is simple.

• If the character set needs either of those features, it is complex.

For example, greek and swe7 are simple character sets, whereas big5 and czech are complex character sets.

To use the following instructions, you must have a MySQL source distribution. In the instructions, *MYSET* represents the name of the character set that you want to add.

1. Add a <charset> element for *MYSET* to the sql/share/charsets/Index.xml file. Use the existing contents in the file as a guide to adding new contents. A partial listing for the latin1 <charset> element follows:

<charset name="latin1">

<family>Western</family>

<description>cp1252 West European</description>

...

<collation name="latin1\_swedish\_ci" id="8" order="Finnish, Swedish">

<flag>primary</flag>

<flag>compiled</flag>

</collation>

<collation name="latin1\_danish\_ci" id="15" order="Danish"/>

...

<collation name="latin1\_bin" id="47" order="Binary">

<flag>binary</flag>

<flag>compiled</flag>

</collation>

...

</charset>

The <charset> element must list all the collations for the character set. These must include at least a binary collation and a default (primary) collation. The default collation is often named using a suffix of general\_ci (general, case-insensitive). It is possible for the binary collation to be the

default collation, but usually they are different. The default collation should have a primary flag. The binary collation should have a binary flag.

You must assign a unique ID number to each collation. The range of IDs from 1024 to 2047 is reserved for user-defined collations. To find the maximum of the currently used collation IDs, use this query:

SELECT MAX(ID) FROM INFORMATION\_SCHEMA.COLLATIONS;

2. This step depends on whether you are adding a simple or complex character set. A simple character set requires only a configuration file, whereas a complex character set requires C source file that defines collation functions, multibyte functions, or both.

For a simple character set, create a configuration file, *MYSET*.xml, that describes the character set properties. Create this file in the sql/share/charsets directory. You can use a copy of latin1.xml as the basis for this file. The syntax for the file is very simple:

• Comments are written as ordinary XML comments (<!-- *text* -->).

• Words within <map> array elements are separated by arbitrary amounts of whitespace.

• Each word within <map> array elements must be a number in hexadecimal format.

• The <map> array element for the <ctype> element has 257 words. The other <map> array elements after that have 256 words. See [Section 10.13.1, “Character Definition Arrays”](#_bookmark50) .

• For each collation listed in the <charset> element for the character set in Index.xml, *MYSET*.xml must contain a <collation> element that defines the character ordering.

For a complex character set, create a C source file that describes the character set properties and defines the support routines necessary to properly perform operations on the character set:

• Create the file ctype-*MYSET*.c in the strings directory. Look at one of the existing ctype- \*.c files (such as ctype-big5.c) to see what needs to be defined. The arrays in your file must have names like ctype\_*MYSET*, to\_lower\_*MYSET*, and so on. These correspond to the arrays for a simple character set. See [Section 10.13.1, “Character Definition Arrays”](#_bookmark50) .

• For each <collation> element listed in the <charset> element for the character set in Index.xml, the ctype-*MYSET*.c file must provide an implementation of the collation.

• If the character set requires string collating functions, see [Section 10.13.2, “String Collating](#_bookmark51) [Support for Complex Character Sets”](#_bookmark51) .

• If the character set requires multibyte character support, see [Section 10.13.3, “Multi-Byte](#_bookmark52) [Character Support for Complex Character Sets”](#_bookmark52) .

3. Modify the configuration information. Use the existing configuration information as a guide to adding information for *MYSYS*. The example here assumes that the character set has default and binary collations, but more lines are needed if *MYSET* has additional collations.

a. Edit mysys/charset-def.c, and “register” the collations for the new character set. Add these lines to the “declaration” section:

#ifdef HAVE\_CHARSET\_*MYSET*

extern CHARSET\_INFO my\_charset\_*MYSET*\_general\_ci;

extern CHARSET\_INFO my\_charset\_*MYSET*\_bin;

#endif

Add these lines to the “registration” section:

#ifdef HAVE\_CHARSET\_*MYSET*

add\_compiled\_collation(&my\_charset\_*MYSET*\_general\_ci);

add\_compiled\_collation(&my\_charset\_*MYSET*\_bin);

#endif

b. If the character set uses ctype-*MYSET*.c, edit strings/CMakeLists.txt and add ctype-*MYSET*.c to the definition of the STRINGS\_SOURCES variable.

c. Edit cmake/character\_sets.cmake:

i. Add *MYSET* to the value of with CHARSETS\_AVAILABLE in alphabetic order.

ii. Add *MYSET* to the value of CHARSETS\_COMPLEX in alphabetic order. This is needed even for simple character sets, so that CMake can recognize -DDEFAULT\_CHARSET=*MYSET*.

4. Reconfigure, recompile, and test.

**10.13.1** **Character** **Definition** **Arrays**

Each simple character set has a configuration file located in the sql/share/charsets directory. For a character set named *MYSYS*, the file is named *MYSET*.xml. It uses <map> array elements to list character set properties. <map> elements appear within these elements:

• <ctype> defines attributes for each character.

• <lower> and <upper> list the lowercase and uppercase characters.

• <unicode> maps 8-bit character values to Unicode values.

• <collation> elements indicate character ordering for comparison and sorting, one element per collation. Binary collations need no <map> element because the character codes themselves provide the ordering.

For a complex character set as implemented in a ctype-*MYSET*.c file in the strings directory, there are corresponding arrays: ctype\_*MYSET*[], to\_lower\_*MYSET*[], and so forth. Not every complex character set has all of the arrays. See also the existing ctype-\*.c files for examples. See the CHARSET\_INFO.txt file in the strings directory for additional information.

Most of the arrays are indexed by character value and have 256 elements. The <ctype> array is indexed by character value + 1 and has 257 elements. This is a legacy convention for handling EOF.

<ctype> array elements are bit values. Each element describes the attributes of a single character in the character set. Each attribute is associated with a bitmask, as defined in include/m\_ctype.h:

|  |  |  |  |
| --- | --- | --- | --- |
| #define | \_MY\_U | 01 | /\* Upper case \*/ |
| #define | \_MY\_L | 02 | /\* Lower case \*/ |
| #define | \_MY\_NMR | 04 | /\* Numeral (digit) \*/ |
| #define | \_MY\_SPC | 010 | /\* Spacing character \*/ |
| #define | \_MY\_PNT | 020 | /\* Punctuation \*/ |
| #define | \_MY\_CTR | 040 | /\* Control character \*/ |
| #define | \_MY\_B | 0100 | /\* Blank \*/ |
| #define | \_MY\_X | 0200 | /\* heXadecimal digit \*/ |

The <ctype> value for a given character should be the union of the applicable bitmask values that describe the character. For example, 'A' is an uppercase character (\_MY\_U) as well as a hexadecimal digit (\_MY\_X), so its ctype value should be defined like this:

ctype['A'+1] = \_MY\_U | \_MY\_X = 01 | 0200 = 0201

The bitmask values in m\_ctype.h are octal values, but the elements of the <ctype> array in *MYSET*.xml should be written as hexadecimal values.

The <lower> and <upper> arrays hold the lowercase and uppercase characters corresponding to each member of the character set. For example:

lower['A'] should contain 'a'

upper['a'] should contain 'A'



Each <collation> array indicates how characters should be ordered for comparison and sorting purposes. MySQL sorts characters based on the values of this information. In some cases, this is the same as the <upper> array, which means that sorting is case-insensitive. For more complicated sorting rules (for complex character sets), see the discussion of string collating in [Section 10.13.2,](#_bookmark51) [“String Collating Support for Complex Character Sets”](#_bookmark51) .

**10.13.2** **String** **Collating** **Support** **for** **Complex** **Character** **Sets**

For a simple character set named *MYSET*, sorting rules are specified in the *MYSET*.xml configuration file using <map> array elements within <collation> elements. If the sorting rules for your language are too complex to be handled with simple arrays, you must define string collating functions in the ctype-*MYSET*.c source file in the strings directory.

The existing character sets provide the best documentation and examples to show how these functions are implemented. Look at the ctype-\*.c files in the strings directory, such as the files for the big5, czech, gbk, sjis, and tis160 character sets. Take a look at the MY\_COLLATION\_HANDLER structures to see how they are used. See also the CHARSET\_INFO.txt file in the strings directory for additional information.

**10.13.3** **Multi-Byte** **Character** **Support** **for** **Complex** **Character** **Sets**

If you want to add support for a new character set named *MYSET* that includes multibyte characters, you must use multibyte character functions in the ctype-*MYSET*.c source file in the strings directory.

The existing character sets provide the best documentation and examples to show how these functions are implemented. Look at the ctype-\*.c files in the strings directory, such as the files for the euc\_kr, gb2312, gbk, sjis, and ujis character sets. Take a look at the MY\_CHARSET\_HANDLER structures to see how they are used. See also the CHARSET\_INFO.txt file in the strings directory for additional information.

**10.14** **Adding** **a** **Collation** **to** **a** **Character** **Set**

**Warning**

User-defined collations are deprecated; you should expect support for them to be removed in a future version of MySQL. As of MySQL 8.0.33, the server issues a warning for any use of COLLATE *user\_defined\_collation* in an SQL statement; a warning is also issued when the server is started with -- collation-server set equal to the name of a user-defined collation.

A collation is a set of rules that defines how to compare and sort character strings. Each collation in MySQL belongs to a single character set. Every character set has at least one collation, and most have two or more collations.

A collation orders characters based on weights. Each character in a character set maps to a weight. Characters with equal weights compare as equal, and characters with unequal weights compare according to the relative magnitude of their weights.

The WEIGHT\_STRING() function can be used to see the weights for the characters in a

string. The value that it returns to indicate weights is a binary string, so it is convenient to use HEX(WEIGHT\_STRING(*str*)) to display the weights in printable form. The following example shows that weights do not differ for lettercase for the letters in 'AaBb' if it is a nonbinary case-insensitive string, but do differ if it is a binary string:

mysql> **SELECT** **HEX(WEIGHT\_STRING('AaBb'** **COLLATE** **latin1\_swedish\_ci));**

+------------------------------------------------------+

| HEX(WEIGHT\_STRING('AaBb' COLLATE latin1\_swedish\_ci)) |

+------------------------------------------------------+

|  |  |
| --- | --- |
| | 41414242 | | |

+------------------------------------------------------+



mysql> **SELECT** **HEX(WEIGHT\_STRING(BINARY** **'AaBb'));**

+-----------------------------------+

| HEX(WEIGHT\_STRING(BINARY 'AaBb')) |

+-----------------------------------+

|  |  |
| --- | --- |
| | 41614262 | | |

+-----------------------------------+

MySQL supports several collation implementations, as discussed in [Section 10.14.1, “Collation](#_bookmark54) [Implementation Types”](#_bookmark54) . Some of these can be added to MySQL without recompiling:

• Simple collations for 8-bit character sets.

• UCA-based collations for Unicode character sets.

• Binary (*xxx*\_bin) collations.

The following sections describe how to add user-defined collations of the first two types to existing character sets. All existing character sets already have a binary collation, so there is no need here to describe how to add one.

**Warning**

Redefining built-in collations is not supported and may result in unexpected server behavior.

Summary of the procedure for adding a new user-defined collation:

1. Choose a collation ID.

2. Add configuration information that names the collation and describes the character-ordering rules.

3. Restart the server.

4. Verify that the server recognizes the collation.

The instructions here cover only user-defined collations that can be added without recompiling MySQL. To add a collation that does require recompiling (as implemented by means of functions in a C source file), use the instructions in [Section 10.13, “Adding a Character Set”](#_bookmark49) . However, instead of adding all the information required for a complete character set, just modify the appropriate files for an existing character set. That is, based on what is already present for the character set's current collations, add data structures, functions, and configuration information for the new collation.

**Note**

If you modify an existing user-defined collation, that may affect the ordering of rows for indexes on columns that use the collation. In this case, rebuild any such indexes to avoid problems such as incorrect query results. See Section 2.10.13, “Rebuilding or Repairing Tables or Indexes” .

**Additional** **Resources**

• Example showing how to add a collation for full-text searches: Section 12.10.7, “Adding a User- Defined Collation for Full-Text Indexing”

• The Unicode Collation Algorithm (UCA) specification: <http://www.unicode.org/reports/tr10/>

• The Locale Data Markup Language (LDML) specification: <http://www.unicode.org/reports/tr35/>

**10.14.1** **Collation** **Implementation** **Types**

MySQL implements several types of collations:

**Simple** **collations** **for** **8-bit** **character** **sets**

This kind of collation is implemented using an array of 256 weights that defines a one-to-one mapping from character codes to weights. latin1\_swedish\_ci is an example. It is a case-insensitive collation, so the uppercase and lowercase versions of a character have the same weights and they compare as equal.

mysql> **SET** **NAMES** **'latin1'** **COLLATE** **'latin1\_swedish\_ci';**

Query OK, 0 rows affected (0.01 sec)

mysql> **SELECT** **HEX(WEIGHT\_STRING('a')),** **HEX(WEIGHT\_STRING('A'));**

+-------------------------+-------------------------+

| HEX(WEIGHT\_STRING('a')) | HEX(WEIGHT\_STRING('A')) |

+-------------------------+-------------------------+

| 41 | 41 |

+-------------------------+-------------------------+

1 row in set (0.01 sec)

mysql> **SELECT** **'a'** **=** **'A';**

+-----------+

| 'a' = 'A' |

+-----------+

| 1 |

+-----------+

1 row in set (0.12 sec)

For implementation instructions, see [Section 10.14.3, “Adding a Simple Collation to an 8-Bit Character](#_bookmark56) [Set”](#_bookmark56) .

**Complex** **collations** **for** **8-bit** **character** **sets**

This kind of collation is implemented using functions in a C source file that define how to order characters, as described in [Section 10.13, “Adding a Character Set”](#_bookmark49) .

**Collations** **for** **non-Unicode** **multibyte** **character** **sets**

For this type of collation, 8-bit (single-byte) and multibyte characters are handled differently. For 8-bit characters, character codes map to weights in case-insensitive fashion. (For example, the single-byte characters 'a' and 'A' both have a weight of 0x41.) For multibyte characters, there are two types of relationship between character codes and weights:

• Weights equal character codes. sjis\_japanese\_ci is an example of this kind of collation. The multibyte character 'ぢ ' has a character code of 0x82C0, and the weight is also 0x82C0.

mysql> **CREATE** **TABLE** **t1**

**(c1** **VARCHAR(2)** **CHARACTER** **SET** **sjis** **COLLATE** **sjis\_japanese\_ci);**

Query OK, 0 rows affected (0.01 sec)

mysql> **INSERT** **INTO** **t1** **VALUES** **('a'),('A'),(0x82C0);**

Query OK, 3 rows affected (0.00 sec)

Records: 3 Duplicates: 0 Warnings: [0](#_bookmark94)

mysql> **SELECT** **c1,** **HEX(c1),** **HEX(WEIGHT\_STRING(c1))** **FROM** **t1;**

+ +---------+------------------------+

| c1 | HEX(c1) | HEX(WEIGHT\_STRING(c1)) |

+------+---------+------------------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | a  A  ぢ | | 61  | 41  | 82C0 | | 41  | 41  | 82C0 | |  |  | |

+------+---------+------------------------+

3 rows in set (0.00 sec)

• Character codes map one-to-one to weights, but a code is not necessarily equal to the weight. gbk\_chinese\_ci is an example of this kind of collation. The multibyte character '膰 ' has a character code of 0x81B0 but a weight of 0xC286.

mysql> **CREATE** **TABLE** **t1**

**(c1** **VARCHAR(2)** **CHARACTER** **SET** **gbk** **COLLATE** **gbk\_chinese\_ci);**

Query OK, 0 rows affected (0.33 sec)

mysql> **INSERT** **INTO** **t1** **VALUES** **('a'),('A'),(0x81B0);**

Query OK, 3 rows affected (0.00 sec)

Records: 3 Duplicates: 0 Warnings: [0](#_bookmark95)

mysql> **SELECT** **c1,** **HEX(c1),** **HEX(WEIGHT\_STRING(c1))** **FROM** **t1;**

+ +---------+------------------------+

| c1 | HEX(c1) | HEX(WEIGHT\_STRING(c1)) |

+------+---------+------------------------+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | a  A  膰 | | 61  | 41  | 81B0 | | 41  | 41  | C286 | |  |  | |

+------+---------+------------------------+

3 rows in set (0.00 sec)

For implementation instructions, see [Section 10.13, “Adding a Character Set”](#_bookmark49) .

**Collations** **for** **Unicode** **multibyte** **character** **sets**

Some of these collations are based on the Unicode Collation Algorithm (UCA), others are not.

Non-UCA collations have a one-to-one mapping from character code to weight. In MySQL, such collations are case-insensitive and accent-insensitive. utf8mb4\_general\_ci is an example: 'a', 'A', 'À', and 'á' each have different character codes but all have a weight of 0x0041 and compare as equal.

mysql> **SET** **NAMES** **'utf8mb4'** **COLLATE** **'utf8mb4\_general\_ci';**

Query OK, 0 rows affected (0.00 sec)

mysql> **CREATE** **TABLE** **t1**

**(c1** **CHAR(1)** **CHARACTER** **SET** **UTF8MB4** **COLLATE** **utf8mb4\_general\_ci);**

Query OK, 0 rows affected (0.01 sec)

mysql> **INSERT** **INTO** **t1** **VALUES** **('a'),('A'),('À'),('á');**

Query OK, 4 rows affected (0.00 sec)

Records: 4 Duplicates: 0 Warnings: [0](#_bookmark96)

mysql> **SELECT** **c1,** **HEX(c1),** **HEX(WEIGHT\_STRING(c1))** **FROM** **t1;**

+ +---------+------------------------+

| c1 | HEX(c1) | HEX(WEIGHT\_STRING(c1)) |

+------+---------+------------------------+

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | a  A  À  á | |  |  |  | | 61  41  C380  C3A1 | |  |  |  | | 0041  0041  0041  0041 | |  |  |  | |

+------+---------+------------------------+

4 rows in set (0.00 sec)

UCA-based collations in MySQL have these properties:

• If a character has weights, each weight uses 2 bytes (16 bits).

• A character may have zero weights (or an empty weight). In this case, the character is ignorable. Example: "U+0000 NULL" does not have a weight and is ignorable.

• A character may have one weight. Example: 'a' has a weight of 0x0E33.

mysql> **SET** **NAMES** **'utf8mb4'** **COLLATE** **'utf8mb4\_unicode\_ci';**

Query OK, 0 rows affected (0.05 sec)

mysql> **SELECT** **HEX('a'),** **HEX(WEIGHT\_STRING('a'));**

+----------+-------------------------+

| HEX('a') | HEX(WEIGHT\_STRING('a')) |

+----------+-------------------------+

| 61 | 0E33 |

+----------+-------------------------+

1 row in set (0.02 sec)

• A character may have many weights. This is an expansion. Example: The German letter 'ß' (SZ ligature, or SHARP S) has a weight of 0x0FEA0FEA.

mysql> **SET** **NAMES** **'utf8mb4'** **COLLATE** **'utf8mb4\_unicode\_ci';**

Query OK, 0 rows affected (0.11 sec)

mysql> **SELECT** **HEX('ß'),** **HEX(WEIGHT\_STRING('ß'));**

+-----------+--------------------------+

| HEX('ß') | HEX(WEIGHT\_STRING('ß')) |

+-----------+--------------------------+

| C39F | 0FEA0FEA |

+-----------+--------------------------+

1 row in set (0.00 sec)

• Many characters may have one weight. This is a contraction. Example: 'ch' is a single letter in Czech and has a weight of 0x0EE2.

mysql> **SET** **NAMES** **'utf8mb4'** **COLLATE** **'utf8mb4\_czech\_ci';**

Query OK, 0 rows affected (0.09 sec)

mysql> **SELECT** **HEX('ch'),** **HEX(WEIGHT\_STRING('ch'));**

+-----------+--------------------------+

| HEX('ch') | HEX(WEIGHT\_STRING('ch')) |

+-----------+--------------------------+

| 6368 | 0EE2 |

+-----------+--------------------------+

1 row in set (0.00 sec)

A many-characters-to-many-weights mapping is also possible (this is contraction with expansion), but is not supported by MySQL.

For implementation instructions, for a non-UCA collation, see [Section 10.13, “Adding a Character Set”](#_bookmark49) . For a UCA collation, see [Section 10.14.4, “Adding a UCA Collation to a Unicode Character Set”](#_bookmark57) .

**Miscellaneous** **collations**

There are also a few collations that do not fall into any of the previous categories.

**10.14.2** **Choosing** **a** **Collation** **ID**

Each collation must have a unique ID. To add a collation, you must choose an ID value that is not currently used. MySQL supports two-byte collation IDs. The range of IDs from 1024 to 2047 is reserved for user-defined collations.

The collation ID that you choose appears in these contexts:

• The ID column of the Information Schema COLLATIONS table.

• The Id column of SHOW COLLATION output.

• The charsetnr member of the MYSQL\_FIELD C API data structure.

• The number member of the MY\_CHARSET\_INFO data structure returned by the [mysql\_get\_character\_set\_info()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-get-character-set-info.html) C API function.

To determine the largest currently used ID, issue the following statement:

mysql> **SELECT** **MAX(ID)** **FROM** **INFORMATION\_SCHEMA.COLLATIONS;**

+---------+

| MAX(ID) |

+---------+

|  |  |
| --- | --- |
| | | 247 | |

+---------+

To display a list of all currently used IDs, issue this statement:

mysql> **SELECT** **ID** **FROM** **INFORMATION\_SCHEMA.COLLATIONS** **ORDER** **BY** **ID;**

+-----+

| ID |



+-----+

| 1 |

| 2 |

| ... |

| 52 |

| 53 |

| 57 |

| 58 |

| ... |

| 98 |

| 99 |

| 128 |

| 129 |

| ... |

| 247 |

+-----+

**Warning**

Before upgrading, you should save the configuration files that you change. If you upgrade in place, the process replaces the modified files.

**10.14.3** **Adding** **a** **Simple** **Collation** **to** **an** **8-Bit** **Character** **Set**

This section describes how to add a simple collation for an 8-bit character set by writing the <collation> elements associated with a <charset> character set description in the MySQL Index.xml file. The procedure described here does not require recompiling MySQL. The example adds a collation named latin1\_test\_ci to the latin1 character set.

1. Choose a collation ID, as shown in [Section 10.14.2, “Choosing a Collation ID”](#_bookmark55) . The following steps use an ID of 1024.

2. Modify the Index.xml and latin1.xml configuration files. These files are located in the directory named by the character\_sets\_dir system variable. You can check the variable value as follows, although the path name might be different on your system:

mysql> **SHOW** **VARIABLES** **LIKE** **'character\_sets\_dir';**

+--------------------+-----------------------------------------+

| Variable\_name | Value |

+--------------------+-----------------------------------------+

| character\_sets\_dir | /user/local/mysql/share/mysql/charsets/ |

+--------------------+-----------------------------------------+

3. Choose a name for the collation and list it in the Index.xml file. Find the <charset> element for the character set to which the collation is being added, and add a <collation> element that indicates the collation name and ID, to associate the name with the ID. For example:

<charset name="latin1">

...

<collation name="latin1\_test\_ci" id="1024"/>

...

</charset>

4. In the latin1.xml configuration file, add a <collation> element that names the collation and that contains a <map> element that defines a character code-to-weight mapping table for character codes 0 to 255. Each value within the <map> element must be a number in hexadecimal format.

<collation name="latin1\_test\_ci">

<map>

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F

20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F

30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F

40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F

50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F

60 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F

50 51 52 53 54 55 56 57 58 59 5A 7B 7C 7D 7E 7F

80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF

B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF

41 41 41 41 5B 5D 5B 43 45 45 45 45 49 49 49 49

44 4E 4F 4F 4F 4F 5C D7 5C 55 55 55 59 59 DE DF

41 41 41 41 5B 5D 5B 43 45 45 45 45 49 49 49 49

44 4E 4F 4F 4F 4F 5C F7 5C 55 55 55 59 59 DE FF

</map>

</collation>

5. Restart the server and use this statement to verify that the collation is present:

mysql> **SHOW** **COLLATION** **WHERE** **Collation** **=** **'latin1\_test\_ci';**

+----------------+---------+------+---------+----------+---------+

| Collation | Charset | Id | Default | Compiled | Sortlen |

+----------------+---------+------+---------+----------+---------+

| latin1\_test\_ci | latin1 | 1024 | | | 1 |

+----------------+---------+------+---------+----------+---------+

**10.14.4** **Adding** **a** **UCA** **Collation** **to** **a** **Unicode** **Character** **Set**

This section describes how to add a UCA collation for a Unicode character set by writing the <collation> element within a <charset> character set description in the MySQL Index.xml file. The procedure described here does not require recompiling MySQL. It uses a subset of the Locale Data Markup Language (LDML) specification, which is available at <http://www.unicode.org/reports/tr35/>. With this method, you need not define the entire collation. Instead, you begin with an existing “base” collation and describe the new collation in terms of how it differs from the base collation. The following table lists the base collations of the Unicode character sets for which UCA collations can be defined. It is not possible to create user-defined UCA collations for utf16le; there is no utf16le\_unicode\_ci collation that would serve as the basis for such collations.

**Table** **10.4** **MySQL** **Character** **Sets** **Available** **for** **User-Defined** **UCA** **Collations**

|  |  |
| --- | --- |
| **Character** **Set** | **Base** **Collation** |
| utf8mb4 | utf8mb4\_unicode\_ci |
| ucs2 | ucs2\_unicode\_ci |
| utf16 | utf16\_unicode\_ci |
| utf32 | utf32\_unicode\_ci |

The following sections show how to add a collation that is defined using LDML syntax, and provide a summary of LDML rules supported in MySQL.

**10.14.4.1** **Defining** **a** **UCA** **Collation** **Using** **LDML** **Syntax**

To add a UCA collation for a Unicode character set without recompiling MySQL, use the following procedure. If you are unfamiliar with the LDML rules used to describe the collation's sort characteristics, see [Section 10.14.4.2, “LDML Syntax Supported in MySQL”](#_bookmark97) .

The example adds a collation named utf8mb4\_phone\_ci to the utf8mb4 character set. The collation is designed for a scenario involving a Web application for which users post their names and phone numbers. Phone numbers can be given in very different formats:

+7-12345-67

+7-12-345-67

+7 12 345 67

+7 (12) 345 67

+71234567

The problem raised by dealing with these kinds of values is that the varying permissible formats make searching for a specific phone number very difficult. The solution is to define a new collation that reorders punctuation characters, making them ignorable.

1. Choose a collation ID, as shown in [Section 10.14.2, “Choosing a Collation ID”](#_bookmark55) . The following steps use an ID of 1029.

2. To modify the Index.xml configuration file. This file is located in the directory named by the character\_sets\_dir system variable. You can check the variable value as follows, although the path name might be different on your system:

mysql> **SHOW** **VARIABLES** **LIKE** **'character\_sets\_dir';**

+--------------------+-----------------------------------------+

| Variable\_name | Value |

+--------------------+-----------------------------------------+

| character\_sets\_dir | /user/local/mysql/share/mysql/charsets/ |

+--------------------+-----------------------------------------+

3. Choose a name for the collation and list it in the Index.xml file. In addition, you'll need to provide the collation ordering rules. Find the <charset> element for the character set to which the collation is being added, and add a <collation> element that indicates the collation name and ID, to associate the name with the ID. Within the <collation> element, provide a <rules> element containing the ordering rules:

<charset name="utf8mb4">

...

<collation name="utf8mb4\_phone\_ci" id="1029">

<rules>

<reset>\u0000</reset>

<i>\u0020</i> <!-- space -->

<i>\u0028</i> <!-- left parenthesis -->

<i>\u0029</i> <!-- right parenthesis -->

<i>\u002B</i> <!-- plus -->

<i>\u002D</i> <!-- hyphen -->

</rules>

</collation>

...

</charset>

4. If you want a similar collation for other Unicode character sets, add other <collation> elements.

For example, to define ucs2\_phone\_ci, add a <collation> element to the <charset name="ucs2"> element. Remember that each collation must have its own unique ID.

5. Restart the server and use this statement to verify that the collation is present:

mysql> **SHOW** **COLLATION** **WHERE** **Collation** **=** **'utf8mb4\_phone\_ci';**

+------------------+---------+------+---------+----------+---------+

| Collation | Charset | Id | Default | Compiled | Sortlen |

+------------------+---------+------+---------+----------+---------+

| utf8mb4\_phone\_ci | utf8mb4 | 1029 | | | 8 |

+------------------+---------+------+---------+----------+---------+

Now test the collation to make sure that it has the desired properties.

Create a table containing some sample phone numbers using the new collation:

mysql> **CREATE** **TABLE** **phonebook** **(**

**name** **VARCHAR(64),**

**phone** **VARCHAR(64)** **CHARACTER** **SET** **utf8mb4** **COLLATE** **utf8mb4\_phone\_ci**

**);**

Query OK, 0 rows affected (0.09 sec)

mysql> **INSERT** **INTO** **phonebook** **VALUES** **('Svoj','+7** **912** **800** **80** **02');**

Query OK, 1 row affected (0.00 sec)

mysql> **INSERT** **INTO** **phonebook** **VALUES** **('Hf','+7** **(912)** **800** **80** **04');**

Query OK, 1 row affected (0.00 sec)

mysql> **INSERT** **INTO** **phonebook** **VALUES** **('Bar','+7-912-800-80-01');**

Query OK, 1 row affected (0.00 sec)

mysql> **INSERT** **INTO** **phonebook** **VALUES** **('Ramil','(7912)** **800** **80** **03');**

Query OK, 1 row affected (0.00 sec)

mysql> **INSERT** **INTO** **phonebook** **VALUES** **('Sanja','+380** **(912)** **8008005');**

Query OK, 1 row affected (0.00 sec)

Run some queries to see whether the ignored punctuation characters are in fact ignored for comparison and sorting:

mysql> **SELECT** **\*** **FROM** **phonebook** **ORDER** **BY** **phone;**

+-------+--------------------+

|

+-------+--------------------+

| +380 (912) 8008005 |

| +7-912-800-80-01 |

| +7 912 800 80 02 |

| (7912) 800 80 03 |

| +7 (912) 800 80 04 |

+-------+--------------------+

5 rows in set (0.00 sec)

mysql> **SELECT** **\*** **FROM** **phonebook** **WHERE** **phone='+7(912)800-80-01';**

+------+------------------+

| name | phone |

+------+------------------+

| Bar | +7-912-800-80-01 |

+------+------------------+

1 row in set (0.00 sec)

mysql> **SELECT** **\*** **FROM** **phonebook** **WHERE** **phone='79128008001';**

+------+------------------+

| name | phone |

+------+------------------+

| Bar | +7-912-800-80-01 |

+------+------------------+

1 row in set (0.00 sec)

mysql> **SELECT** **\*** **FROM** **phonebook** **WHERE** **phone='7** **9** **1** **2** **8** **0** **0** **8** **0** **0** **1';**

+------+------------------+

| name | phone |

+------+------------------+

| Bar | +7-912-800-80-01 |

+------+------------------+

1 row in set (0.00 sec)

| Sanja

| Bar

| Svoj

| Ramil

| Hf

| name | phone

**10.14.4.2** **LDML** **Syntax** **Supported** **in** **MySQL**

This section describes the LDML syntax that MySQL recognizes. This is a subset of the syntax described in the LDML specification available at <http://www.unicode.org/reports/tr35/>, which should be consulted for further information. MySQL recognizes a large enough subset of the syntax that, in many cases, it is possible to download a collation definition from the Unicode Common Locale Data Repository and paste the relevant part (that is, the part between the <rules> and </rules> tags) into the MySQL Index.xml file. The rules described here are all supported except that character sorting occurs only at the primary level. Rules that specify differences at secondary or higher sort levels are recognized (and thus can be included in collation definitions) but are treated as equality at the primary level.

The MySQL server generates diagnostics when it finds problems while parsing the Index.xml file. See [Section 10.14.4.3, “Diagnostics During Index.xml Parsing”](#_bookmark98) .

**Character** **Representation**

Characters named in LDML rules can be written literally or in \u*nnnn* format, where *nnnn* is the hexadecimal Unicode code point value. For example, A and á can be written literally or as \u0041 and \u00E1. Within hexadecimal values, the digits A through F are not case-sensitive; \u00E1 and \u00e1 are equivalent. For UCA 4.0.0 collations, hexadecimal notation can be used only for characters in the Basic Multilingual Plane, not for characters outside the BMP range of 0000 to FFFF. For UCA 5.2.0 collations, hexadecimal notation can be used for any character.

The Index.xml file itself should be written using UTF-8 encoding.

**Syntax** **Rules**

LDML has reset rules and shift rules to specify character ordering. Orderings are given as a set of rules that begin with a reset rule that establishes an anchor point, followed by shift rules that indicate how characters sort relative to the anchor point.

• A <reset> rule does not specify any ordering in and of itself. Instead, it “resets” the ordering for subsequent shift rules to cause them to be taken in relation to a given character. Either of the following rules resets subsequent shift rules to be taken in relation to the letter 'A':

<reset>A</reset>

<reset>\u0041</reset>

• The <p>, <s>, and <t> shift rules define primary, secondary, and tertiary differences of a character from another character:

• Use primary differences to distinguish separate letters.

• Use secondary differences to distinguish accent variations.

• Use tertiary differences to distinguish lettercase variations.

Either of these rules specifies a primary shift rule for the 'G' character:

<p>G</p>

<p>\u0047</p>

• The <i> shift rule indicates that one character sorts identically to another. The following rules cause 'b' to sort the same as 'a':

<reset>a</reset>

<i>b</i>

• Abbreviated shift syntax specifies multiple shift rules using a single pair of tags. The following table shows the correspondence between abbreviated syntax rules and the equivalent nonabbreviated rules.

**Table** **10.5** **Abbreviated** **Shift** **Syntax**

|  |  |
| --- | --- |
| **Abbreviated** **Syntax** | **Nonabbreviated** **Syntax** |
| <pc>xyz</pc> | <p>x</p><p>y</p><p>z</p> |
| <sc>xyz</sc> | <s>x</s><s>y</s><s>z</s> |
| <tc>xyz</tc> | <t>x</t><t>y</t><t>z</t> |
| <ic>xyz</ic> | <i>x</i><i>y</i><i>z</i> |

• An expansion is a reset rule that establishes an anchor point for a multiple-character sequence. MySQL supports expansions 2 to 6 characters long. The following rules put 'z' greater at the primary level than the sequence of three characters 'abc':

<reset>abc</reset>

<p>z</p>

• A contraction is a shift rule that sorts a multiple-character sequence. MySQL supports contractions 2 to 6 characters long. The following rules put the sequence of three characters 'xyz' greater at the primary level than 'a':

<reset>a</reset>

<p>xyz</p>

• Long expansions and long contractions can be used together. These rules put the sequence of three characters 'xyz' greater at the primary level than the sequence of three characters 'abc':

<reset>abc</reset>

<p>xyz</p>

• Normal expansion syntax uses <x> plus <extend> elements to specify an expansion. The following rules put the character 'k' greater at the secondary level than the sequence 'ch'. That is, 'k' behaves as if it expands to a character after 'c' followed by 'h':

<reset>c</reset>

<x><s>k</s><extend>h</extend></x>

This syntax permits long sequences. These rules sort the sequence 'ccs' greater at the tertiary level than the sequence 'cscs':

<reset>cs</reset>

<x><t>ccs</t><extend>cs</extend></x>

The LDML specification describes normal expansion syntax as “tricky.” See that specification for details.

• Previous context syntax uses <x> plus <context> elements to specify that the context before a character affects how it sorts. The following rules put '-' greater at the secondary level than 'a', but only when '-' occurs after 'b':

<reset>a</reset>

<x><context>b</context><s>-</s></x>

• Previous context syntax can include the <extend> element. These rules put 'def' greater at the primary level than 'aghi', but only when 'def' comes after 'abc':

<reset>a</reset>

<x><context>abc</context><p>def</p><extend>ghi</extend></x>

• Reset rules permit a before attribute. Normally, shift rules after a reset rule indicate characters

that sort after the reset character. Shift rules after a reset rule that has the before attribute indicate characters that sort before the reset character. The following rules put the character 'b' immediately before 'a' at the primary level:

<reset before="primary">a</reset>

<p>b</p>

Permissible before attribute values specify the sort level by name or the equivalent numeric value:

<reset before="primary">

<reset before="1">

<reset before="secondary">

<reset before="2">

<reset before="tertiary">

<reset before="3">

• A reset rule can name a logical reset position rather than a literal character:

<first\_tertiary\_ignorable/>

<last\_tertiary\_ignorable/>

<first\_secondary\_ignorable/>

<last\_secondary\_ignorable/>

<first\_primary\_ignorable/>

<last\_primary\_ignorable/>

<first\_variable/>

<last\_variable/>

<first\_non\_ignorable/>

<last\_non\_ignorable/>

<first\_trailing/>

<last\_trailing/>

These rules put 'z' greater at the primary level than nonignorable characters that have a Default Unicode Collation Element Table (DUCET) entry and that are not CJK:

<reset><last\_non\_ignorable/></reset>

<p>z</p>

Logical positions have the code points shown in the following table.

**Table** **10.6** **Logical** **Reset** **Position** **Code** **Points**

|  |  |  |
| --- | --- | --- |
| **Logical** **Position** | **Unicode** **4.0.0** **Code** **Point** | **Unicode** **5.2.0** **Code** **Point** |
| <first\_non\_ignorable/> | U+02D0 | U+02D0 |
| <last\_non\_ignorable/> | U+A48C | U+1342E |
| <first\_primary\_ignorable/  > | U+0332 | U+0332 |
| <last\_primary\_ignorable/  > | U+20EA | U+101FD |
| <first\_secondary\_ignorab  > | l0000 | U+0000 |
| <last\_secondary\_ignorable/  > | U+FE73 | U+FE73 |
| <first\_tertiary\_ignorable/  > | U+0000 | U+0000 |
| <last\_tertiary\_ignorable/  > | U+FE73 | U+FE73 |
| <first\_trailing/> | U+0000 | U+0000 |
| <last\_trailing/> | U+0000 | U+0000 |
| <first\_variable/> | U+0009 | U+0009 |
| <last\_variable/> | U+2183 | U+1D371 |

• The <collation> element permits a shift-after-method attribute that affects character weight calculation for shift rules. The attribute has these permitted values:

• simple: Calculate character weights as for reset rules that do not have a before attribute. This is the default if the attribute is not given.

• expand: Use expansions for shifts after reset rules.

Suppose that '0' and '1' have weights of 0E29 and 0E2A and we want to put all basic Latin letters between '0' and '1':

<reset>0</reset>

<pc>abcdefghijklmnopqrstuvwxyz</pc>

For simple shift mode, weights are calculated as follows:

|  |  |
| --- | --- |
| 'a' has weight  'b' has weight  'c' has weight  ... | 0E29+1  0E29+2  0E29+3 |

However, there are not enough vacant positions to put 26 characters between '0' and '1'. The result is that digits and letters are intermixed.

To solve this, use shift-after-method="expand". Then weights are calculated like this:

'a' has weight [0E29][233D+1]

'b' has weight [0E29][233D+2]

'c' has weight [0E29][233D+3]

...

233D is the UCA 4.0.0 weight for character 0xA48C, which is the last nonignorable character (a sort of the greatest character in the collation, excluding CJK). UCA 5.2.0 is similar but uses 3ACA, for character 0x1342E.

**MySQL-Specific** **LDML** **Extensions**

An extension to LDML rules permits the <collation> element to include an optional version attribute in <collation> tags to indicate the UCA version on which the collation is based. If the version attribute is omitted, its default value is 4.0.0. For example, this specification indicates a collation that is based on UCA 5.2.0:

<collation id="*nnn*" name="utf8mb4\_*xxx*\_ci" version="5.2.0">

...

</collation>

**10.14.4.3** **Diagnostics** **During** **Index.xml** **Parsing**

The MySQL server generates diagnostics when it finds problems while parsing the Index.xml file:

• Unknown tags are written to the error log. For example, the following message results if a collation definition contains a <aaa> tag:

[Warning] Buffered warning: Unknown LDML tag:

'charsets/charset/collation/rules/aaa'

• If collation initialization is not possible, the server reports an “Unknown collation” error, and also generates warnings explaining the problems, such as in the previous example. In other cases, when a collation description is generally correct but contains some unknown tags, the collation is initialized and is available for use. The unknown parts are ignored, but a warning is generated in the error log.

• Problems with collations generate warnings that clients can display with SHOW WARNINGS. Suppose that a reset rule contains an expansion longer than the maximum supported length of 6 characters:

<reset>abcdefghi</reset>

<i>x</i>

An attempt to use the collation produces warnings:

mysql> **SELECT** **\_utf8mb4'test'** **COLLATE** **utf8mb4\_test\_ci;**

ERROR 1273 (HY000): Unknown collation: 'utf8mb4\_test\_ci'

mysql> **SHOW** **WARNINGS;**

+---------+------+----------------------------------------+

| Level | Code | Message |

+---------+------+----------------------------------------+

| Error | 1273 | Unknown collation: 'utf8mb4\_test\_ci' |

| Warning | 1273 | Expansion is too long at 'abcdefghi=x' |

+---------+------+----------------------------------------+

**10.15** **Character** **Set** **Configuration**

The MySQL server has a compiled-in default character set and collation. To change these defaults, use the --character-set-server and --collation-server options when you start the server. See Section 5.1.7, “Server Command Options” . The collation must be a legal collation for the default character set. To determine which collations are available for each character set, use the SHOW COLLATION statement or query the INFORMATION\_SCHEMA COLLATIONS table.

If you try to use a character set that is not compiled into your binary, you might run into the following problems:

• If your program uses an incorrect path to determine where the character sets are stored (which is typically the share/mysql/charsets or share/charsets directory under the MySQL installation

directory), this can be fixed by using the --character-sets-dir option when you run the program. For example, to specify a directory to be used by MySQL client programs, list it in the [client] group of your option file. The examples given here show what the setting might look like for Unix or Windows, respectively:

[client]

character-sets-dir=/usr/local/mysql/share/mysql/charsets

[client]

character-sets-dir="C:/Program Files/MySQL/MySQL Server 8.0/share/charsets"

• If the character set is a complex character set that cannot be loaded dynamically, you must recompile the program with support for the character set.

For Unicode character sets, you can define collations without recompiling by using LDML notation. See [Section 10.14.4, “Adding a UCA Collation to a Unicode Character Set”](#_bookmark57) .

• If the character set is a dynamic character set, but you do not have a configuration file for it, you should install the configuration file for the character set from a new MySQL distribution.

• If your character set index file ( Index.xml) does not contain the name for the character set, your program displays an error message:

Character set '*charset\_name* ' is not a compiled character set and is not

specified in the '/usr/share/mysql/charsets/Index.xml' file

To solve this problem, you should either get a new index file or manually add the name of any missing character sets to the current file.

You can force client programs to use specific character set as follows:

[client]

default-character-set=*charset\_name*

This is normally unnecessary. However, when character\_set\_system differs from character\_set\_server or character\_set\_client, and you input characters manually (as database object identifiers, column values, or both), these may be displayed incorrectly in output from the client or the output itself may be formatted incorrectly. In such cases, starting the mysql client with --default-character-set=*system\_character\_set*—that is, setting the client character set to match the system character set—should fix the problem.

**10.16** **MySQL** **Server** **Locale** **Support**

The locale indicated by the lc\_time\_names system variable controls the language used to display day and month names and abbreviations. This variable affects the output from the DATE\_FORMAT(), DAYNAME(), and MONTHNAME() functions.

lc\_time\_names does not affect the STR\_TO\_DATE() or GET\_FORMAT() function.

The lc\_time\_names value does not affect the result from FORMAT(), but this function takes an optional third parameter that enables a locale to be specified to be used for the result number's decimal point, thousands separator, and grouping between separators. Permissible locale values are the same as the legal values for the lc\_time\_names system variable.

Locale names have language and region subtags listed by IANA ([http://www.iana.org/assignments/](http://www.iana.org/assignments/language-subtag-registry) [language-subtag-registry](http://www.iana.org/assignments/language-subtag-registry)) such as 'ja\_JP' or 'pt\_BR'. The default value is 'en\_US' regardless of your system's locale setting, but you can set the value at server startup, or set the GLOBAL value at runtime if you have privileges sufficient to set global system variables; see Section 5.1.9.1, “System Variable Privileges” . Any client can examine the value of lc\_time\_names or set its SESSION value to affect the locale for its own connection.

mysql> **SET** **NAMES** **'utf8mb4';**

Query OK, 0 rows affected (0.09 sec)

mysql> **SELECT** **@@lc\_time\_names;**

+-----------------+

| @@lc\_time\_names |

+-----------------+

| en\_US |

+-----------------+

1 row in set (0.00 sec)

mysql> **SELECT** **DAYNAME('2020-01-01'),** **MONTHNAME('2020-01-01');**

+-----------------------+-------------------------+

| DAYNAME('2020-01-01') | MONTHNAME('2020-01-01') |

+-----------------------+-------------------------+

| Wednesday | January |

+-----------------------+-------------------------+

1 row in set (0.00 sec)

mysql> **SELECT** **DATE\_FORMAT('2020-01-01','%W** **%a** **%M** **%b');**

+-----------------------------------------+

| DATE\_FORMAT('2020-01-01','%W %a %M %b') |

+-----------------------------------------+

| Wednesday Wed January Jan |

+-----------------------------------------+

1 row in set (0.00 sec)

mysql> **SET** **lc\_time\_names** **=** **'es\_MX';**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT** **@@lc\_time\_names;**

+-----------------+

| @@lc\_time\_names |

+-----------------+

| es\_MX |

+-----------------+

1 row in set (0.00 sec)

mysql> **SELECT** **DAYNAME('2020-01-01'),** **MONTHNAME('2020-01-01');**

+-----------------------+-------------------------+

| DAYNAME('2020-01-01') | MONTHNAME('2020-01-01') |

+-----------------------+-------------------------+

| miércoles | enero |

+-----------------------+-------------------------+

1 row in set (0.00 sec)

mysql> **SELECT** **DATE\_FORMAT('2020-01-01','%W** **%a** **%M** **%b');**

+-----------------------------------------+

| DATE\_FORMAT('2020-01-01','%W %a %M %b') |

+-----------------------------------------+

| miércoles mié enero ene |

+-----------------------------------------+

1 row in set (0.00 sec)

The day or month name for each of the affected functions is converted from utf8mb4 to the character set indicated by the character\_set\_connection system variable.

lc\_time\_names may be set to any of the following locale values. The set of locales supported by MySQL may differ from those supported by your operating system.

|  |  |
| --- | --- |
| **Locale** **Value** | **Meaning** |
| ar\_AE | Arabic - United Arab Emirates |
| ar\_BH | Arabic - Bahrain |
| ar\_DZ | Arabic - Algeria |
| ar\_EG | Arabic - Egypt |
| ar\_IN | Arabic - India |
| ar\_IQ | Arabic - Iraq |
| ar\_JO | Arabic - Jordan |

|  |  |
| --- | --- |
| **Locale** **Value** | **Meaning** |
| ar\_KW | Arabic - Kuwait |
| ar\_LB | Arabic - Lebanon |
| ar\_LY | Arabic - Libya |
| ar\_MA | Arabic - Morocco |
| ar\_OM | Arabic - Oman |
| ar\_QA | Arabic - Qatar |
| ar\_SA | Arabic - Saudi Arabia |
| ar\_SD | Arabic - Sudan |
| ar\_SY | Arabic - Syria |
| ar\_TN | Arabic - Tunisia |
| ar\_YE | Arabic - Yemen |
| be\_BY | Belarusian - Belarus |
| bg\_BG | Bulgarian - Bulgaria |
| ca\_ES | Catalan - Spain |
| cs\_CZ | Czech - Czech Republic |
| da\_DK | Danish - Denmark |
| de\_AT | German - Austria |
| de\_BE | German - Belgium |
| de\_CH | German - Switzerland |
| de\_DE | German - Germany |
| de\_LU | German - Luxembourg |
| el\_GR | Greek - Greece |
| en\_AU | English - Australia |
| en\_CA | English - Canada |
| en\_GB | English - United Kingdom |
| en\_IN | English - India |
| en\_NZ | English - New Zealand |
| en\_PH | English - Philippines |
| en\_US | English - United States |
| en\_ZA | English - South Africa |
| en\_ZW | English - Zimbabwe |
| es\_AR | Spanish - Argentina |
| es\_BO | Spanish - Bolivia |
| es\_CL | Spanish - Chile |
| es\_CO | Spanish - Colombia |
| es\_CR | Spanish - Costa Rica |
| es\_DO | Spanish - Dominican Republic |
| es\_EC | Spanish - Ecuador |
| es\_ES | Spanish - Spain |
| es\_GT | Spanish - Guatemala |
| es\_HN | Spanish - Honduras |

|  |  |
| --- | --- |
| **Locale** **Value** | **Meaning** |
| es\_MX | Spanish - Mexico |
| es\_NI | Spanish - Nicaragua |
| es\_PA | Spanish - Panama |
| es\_PE | Spanish - Peru |
| es\_PR | Spanish - Puerto Rico |
| es\_PY | Spanish - Paraguay |
| es\_SV | Spanish - El Salvador |
| es\_US | Spanish - United States |
| es\_UY | Spanish - Uruguay |
| es\_VE | Spanish - Venezuela |
| et\_EE | Estonian - Estonia |
| eu\_ES | Basque - Spain |
| fi\_FI | Finnish - Finland |
| fo\_FO | Faroese - Faroe Islands |
| fr\_BE | French - Belgium |
| fr\_CA | French - Canada |
| fr\_CH | French - Switzerland |
| fr\_FR | French - France |
| fr\_LU | French - Luxembourg |
| gl\_ES | Galician - Spain |
| gu\_IN | Gujarati - India |
| he\_IL | Hebrew - Israel |
| hi\_IN | Hindi - India |
| hr\_HR | Croatian - Croatia |
| hu\_HU | Hungarian - Hungary |
| id\_ID | Indonesian - Indonesia |
| is\_IS | Icelandic - Iceland |
| it\_CH | Italian - Switzerland |
| it\_IT | Italian - Italy |
| ja\_JP | Japanese - Japan |
| ko\_KR | Korean - Republic of Korea |
| lt\_LT | Lithuanian - Lithuania |
| lv\_LV | Latvian - Latvia |
| mk\_MK | Macedonian - North Macedonia |
| mn\_MN | Mongolia - Mongolian |
| ms\_MY | Malay - Malaysia |
| nb\_NO | Norwegian(Bokmål) - Norway |
| nl\_BE | Dutch - Belgium |
| nl\_NL | Dutch - The Netherlands |
| no\_NO | Norwegian - Norway |
| pl\_PL | Polish - Poland |

|  |  |
| --- | --- |
| **Locale** **Value** | **Meaning** |
| pt\_BR | Portugese - Brazil |
| pt\_PT | Portugese - Portugal |
| rm\_CH | Romansh - Switzerland |
| ro\_RO | Romanian - Romania |
| ru\_RU | Russian - Russia |
| ru\_UA | Russian - Ukraine |
| sk\_SK | Slovak - Slovakia |
| sl\_SI | Slovenian - Slovenia |
| sq\_AL | Albanian - Albania |
| sr\_RS | Serbian - Serbia |
| sv\_FI | Swedish - Finland |
| sv\_SE | Swedish - Sweden |
| ta\_IN | Tamil - India |
| te\_IN | Telugu - India |
| th\_TH | Thai - Thailand |
| tr\_TR | Turkish - Turkey |
| uk\_UA | Ukrainian - Ukraine |
| ur\_PK | Urdu - Pakistan |
| vi\_VN | Vietnamese - Vietnam |
| zh\_CN | Chinese - China |
| zh\_HK | Chinese - Hong Kong |
| zh\_TW | Chinese - Taiwan |

Chapter 11 Data Types

**Table** **of** **Contents**

[11.1 Numeric Data Types](#_bookmark99) [2056](#_bookmark99)

[11.1.1 Numeric Data Type Syntax](#_bookmark100) [2056](#_bookmark100)

[11.1.2 Integer Types (Exact Value) - INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT,](#_bookmark101)

[BIGINT](#_bookmark101) [2060](#_bookmark101)

[11.1.3 Fixed-Point Types (Exact Value) - DECIMAL, NUMERIC](#_bookmark102) [2060](#_bookmark102)

[11.1.4 Floating-Point Types (Approximate Value) - FLOAT, DOUBLE](#_bookmark103) [2061](#_bookmark103)

[11.1.5 Bit-Value Type - BIT](#_bookmark104) [2061](#_bookmark104)

[11.1.6 Numeric Type Attributes](#_bookmark105) [2061](#_bookmark105)

[11.1.7 Out Range Overflow Handling](#_bookmark106)ofand . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . [2063](#_bookmark106)

[11.2 Date Time Data Types](#_bookmark107)and [2064](#_bookmark107)

[11.2.1 Date Time Data Type Syntax](#_bookmark108)and [2065](#_bookmark108)

[11.2.2 The DATE, DATETIME, TIMESTAMP Types](#_bookmark109)and [2067](#_bookmark109)

[11.2.3 The TIME Type](#_bookmark110) [2069](#_bookmark110)

[11.2.4 The YEAR Type](#_bookmark111) [2069](#_bookmark111)

[11.2.5 Automatic Initialization Updating TIMESTAMP DATETIME](#_bookmark112)andforand [2070](#_bookmark112)

[11.2.6 Fractional Seconds Time Values](#_bookmark113)in [2073](#_bookmark113)

[11.2.7 Conversion Between Date Time Types](#_bookmark114)and [2074](#_bookmark114)

[11.2.8 2-Digit Years Dates](#_bookmark115)in [2075](#_bookmark115)

[11.3 String Data Types](#_bookmark116) [2076](#_bookmark116)

[11.3.1 String Data Type Syntax](#_bookmark117) [2076](#_bookmark117)

[11.3.2 The CHAR VARCHAR Types](#_bookmark60)and [2079](#_bookmark60)

[11.3.3 The BINARY VARBINARY Types](#_bookmark73)and [2081](#_bookmark73)

[11.3.4 The BLOB TEXT Types](#_bookmark61)and [2082](#_bookmark61)

[11.3.5 The ENUM Type](#_bookmark62) [2084](#_bookmark62)

[11.3.6 The SET Type](#_bookmark63) [2087](#_bookmark63)

[11.4 Spatial Data Types](#_bookmark118) [2089](#_bookmark118)

[11.4.1 Spatial Data Types](#_bookmark119) [2090](#_bookmark119)

[11.4.2 The OpenGIS Geometry Model](#_bookmark120) [2091](#_bookmark120)

[11.4.3 Supported Spatial Data Formats](#_bookmark121) [2097](#_bookmark121)

[11.4.4 Geometry Well-Formedness Validity](#_bookmark122)and [2100](#_bookmark122)

[11.4.5 Spatial Reference System Support](#_bookmark123) [2100](#_bookmark123)

[11.4.6 Creating Spatial Columns](#_bookmark124) [2102](#_bookmark124)

[11.4.7 Populating Spatial Columns](#_bookmark125) [2102](#_bookmark125)

[11.4.8 Fetching Spatial Data](#_bookmark126) [2103](#_bookmark126)

[11.4.9 Optimizing Spatial Analysis](#_bookmark127) [2103](#_bookmark127)

[11.4.10 Creating Spatial Indexes](#_bookmark128) [2104](#_bookmark128)

[11.4.11 Using Spatial Indexes](#_bookmark129) [2105](#_bookmark129)

[11.5 The JSON Data Type](#_bookmark130) [2106](#_bookmark130)

[11.6 Data Type Default Values](#_bookmark131) [2122](#_bookmark131)

[11.7 Data Type Storage Requirements](#_bookmark132) [2125](#_bookmark132)

[11.8 Choosing Right Type Column](#_bookmark133)thefora [2129](#_bookmark133)

[11.9 Using Data Types Other Database Engines](#_bookmark134)from [2130](#_bookmark134)

MySQL supports SQL data types in several categories: numeric types, date and time types, string

(character and byte) types, spatial types, and the [JSON](#_bookmark130) data type. This chapter provides an overview

and more detailed description of the properties of the types in each category, and a summary of the data type storage requirements. The initial overviews are intentionally brief. Consult the more detailed descriptions for additional information about particular data types, such as the permissible formats in which you can specify values.

Data type descriptions use these conventions:

• For integer types, *M* indicates the maximum display width. For floating-point and fixed-point types, *M* is the total number of digits that can be stored (the precision). For string types, *M* is the maximum length. The maximum permissible value of *M*depends on the data type.

• *D* applies to floating-point and fixed-point types and indicates the number of digits following the decimal point (the scale). The maximum possible value is 30, but should be no greater than *M*−2.

• *fsp* applies to the [TIME](#_bookmark110), [DATETIME](#_bookmark109), and [TIMESTAMP](#_bookmark109) types and represents fractional seconds precision; that is, the number of digits following the decimal point for fractional parts of seconds. The *fsp* value, if given, must be in the range 0 to 6. A value of 0 signifies that there is no fractional part. If omitted, the default precision is 0. (This differs from the standard SQL default of 6, for compatibility with previous MySQL versions.)

• Square brackets ( [ and ]) indicate optional parts of type definitions.

**11.1** **Numeric** **Data** **Types**

MySQL supports all standard SQL numeric data types. These types include the exact numeric data types ([INTEGER](#_bookmark101), [SMALLINT](#_bookmark101), [DECIMAL](#_bookmark102), and [NUMERIC](#_bookmark102)), as well as the approximate numeric data types ([FLOAT](#_bookmark103), [REAL](#_bookmark103), and [DOUBLE PRECISION](#_bookmark103)). The keyword [INT](#_bookmark101) is a synonym for [INTEGER](#_bookmark101), and the keywords [DEC](#_bookmark102) and [FIXED](#_bookmark102) are synonyms for [DECIMAL](#_bookmark102). MySQL treats [DOUBLE](#_bookmark103) as a synonym for [DOUBLE PRECISION](#_bookmark103) (a nonstandard extension). MySQL also treats [REAL](#_bookmark103) as a synonym for [DOUBLE](#_bookmark103) [PRECISION](#_bookmark103) (a nonstandard variation), unless the REAL\_AS\_FLOAT SQL mode is enabled.

The [BIT](#_bookmark104) data type stores bit values and is supported for MyISAM, MEMORY, InnoDB, and NDB tables.

For information about how MySQL handles assignment of out-of-range values to columns and overflow during expression evaluation, see [Section 11.1.7, “Out-of-Range and Overflow Handling”](#_bookmark106) .

For information about storage requirements of the numeric data types, see [Section 11.7, “Data Type](#_bookmark132) [Storage Requirements”](#_bookmark132) .

For descriptions of functions that operate on numeric values, see Section 12.6, “Numeric Functions and Operators” . The data type used for the result of a calculation on numeric operands depends on the types of the operands and the operations performed on them. For more information, see Section 12.6.1, “Arithmetic Operators” .

**11.1.1** **Numeric** **Data** **Type** **Syntax**

For integer data types, *M* indicates the maximum display width. The maximum display width is 255. Display width is unrelated to the range of values a type can store, as described in [Section 11.1.6,](#_bookmark105) [“Numeric Type Attributes”](#_bookmark105) .

For floating-point and fixed-point data types, *M* is the total number of digits that can be stored.

As of MySQL 8.0.17, the display width attribute is deprecated for integer data types; you should expect support for it to be removed in a future version of MySQL.

If you specify ZEROFILL for a numeric column, MySQL automatically adds the UNSIGNED attribute to the column.

As of MySQL 8.0.17, the ZEROFILL attribute is deprecated for numeric data types; you should expect support for it to be removed in a future version of MySQL. Consider using an alternative means of producing the effect of this attribute. For example, applications could use the LPAD() function to zero- pad numbers up to the desired width, or they could store the formatted numbers in [CHAR](#_bookmark60) columns.

Numeric data types that permit the UNSIGNED attribute also permit SIGNED. However, these data types are signed by default, so the SIGNED attribute has no effect.



As of MySQL 8.0.17, the UNSIGNED attribute is deprecated for columns of type [FLOAT](#_bookmark103), [DOUBLE](#_bookmark103), and [DECIMAL](#_bookmark102) (and any synonyms); you should expect support for it to be removed in a future version of MySQL. Consider using a simple CHECK constraint instead for such columns.

SERIAL is an alias for BIGINT UNSIGNED NOT NULL AUTO\_INCREMENT UNIQUE.

SERIAL DEFAULT VALUE in the definition of an integer column is an alias for NOT NULL

AUTO\_INCREMENT UNIQUE.

**Warning**

When you use subtraction between integer values where one is of type UNSIGNED, the result is unsigned unless the NO\_UNSIGNED\_SUBTRACTION SQL mode is enabled. See Section 12.11, “Cast Functions and Operators” .

• [BIT[(*M*)]](#_bookmark104)

A bit-value type. *M* indicates the number of bits per value, from 1 to 64. The default is 1 if *M* is omitted. • [TINYINT[(*M*)] [UNSIGNED] [ZEROFILL]](#_bookmark101)

A very small integer. The signed range is -128 to 127. The unsigned range is 0 to 255.

• [BOOL](#_bookmark101), [BOOLEAN](#_bookmark101)

These types are synonyms for [TINYINT(1)](#_bookmark101). A value of zero is considered false. Nonzero values are considered true:

mysql> **SELECT** **IF(0,** **'true',** **'false');**

+------------------------+

| IF(0, 'true', 'false') |

+------------------------+

| false |

+------------------------+

mysql> **SELECT** **IF(1,** **'true',** **'false');**

+------------------------+

| IF(1, 'true', 'false') |

+------------------------+

| true |

+------------------------+

mysql> **SELECT** **IF(2,** **'true',** **'false');**

+------------------------+

| IF(2, 'true', 'false') |

+------------------------+

| true |

+------------------------+

However, the values TRUE and FALSE are merely aliases for 1 and 0, respectively, as shown here:

mysql> **SELECT** **IF(0** **=** **FALSE,** **'true',** **'false');**

+--------------------------------+

| IF(0 = FALSE, 'true', 'false') |

+--------------------------------+

| true |

+--------------------------------+

mysql> **SELECT** **IF(1** **=** **TRUE,** **'true',** **'false');**

+-------------------------------+

| IF(1 = TRUE, 'true', 'false') |

+-------------------------------+

| true |

+-------------------------------+

mysql> **SELECT** **IF(2** **=** **TRUE,** **'true',** **'false');**

+-------------------------------+

| IF(2 = TRUE, 'true', 'false') |

+-------------------------------+

|  |  |
| --- | --- |
| | false | | |

+-------------------------------+

mysql> **SELECT** **IF(2** **=** **FALSE,** **'true',** **'false');**

+--------------------------------+

| IF(2 = FALSE, 'true', 'false') |

+--------------------------------+

| false |

+--------------------------------+

The last two statements display the results shown because 2 is equal to neither 1 nor 0.

• [SMALLINT[(*M*)] [UNSIGNED] [ZEROFILL]](#_bookmark101)

A small integer. The signed range is -32768 to 32767. The unsigned range is 0 to 65535.

• [MEDIUMINT[(*M*)] [UNSIGNED] [ZEROFILL]](#_bookmark101)

A medium-sized integer. The signed range is -8388608 to 8388607. The unsigned range is 0 to

16777215.

• [INT[(*M*)] [UNSIGNED] [ZEROFILL]](#_bookmark101)

A normal-size integer. The signed range is -2147483648 to 2147483647. The unsigned range is 0 to 4294967295.

• [INTEGER[(*M*)] [UNSIGNED] [ZEROFILL]](#_bookmark101) This type is a synonym for [INT](#_bookmark101).

• [BIGINT[(*M*)] [UNSIGNED] [ZEROFILL]](#_bookmark101)

A large integer. The signed range is -9223372036854775808 to 9223372036854775807. The unsigned range is 0 to 18446744073709551615.

SERIAL is an alias for BIGINT UNSIGNED NOT NULL AUTO\_INCREMENT UNIQUE. Some things you should be aware of with respect to [BIGINT](#_bookmark101) columns:

• All arithmetic is done using signed [BIGINT](#_bookmark101) or [DOUBLE](#_bookmark103) values, so you should not use unsigned big integers larger than 9223372036854775807 (63 bits) except with bit functions! If you do that, some of the last digits in the result may be wrong because of rounding errors when converting a [BIGINT](#_bookmark101) value to a [DOUBLE](#_bookmark103).

MySQL can handle [BIGINT](#_bookmark101) in the following cases:

• When using integers to store large unsigned values in a [BIGINT](#_bookmark101) column.

• In MIN(*col\_name*) or MAX(*col\_name*), where *col\_name* refers to a [BIGINT](#_bookmark101) column.

• When using operators (+, -, \*, and so on) where both operands are integers.

• You can always store an exact integer value in a [BIGINT](#_bookmark101) column by storing it using a string. In this case, MySQL performs a string-to-number conversion that involves no intermediate double- precision representation.

• The -, +, and \* operators use [BIGINT](#_bookmark101) arithmetic when both operands are integer values. This means that if you multiply two big integers (or results from functions that return integers), you may get unexpected results when the result is larger than 9223372036854775807.

• [DECIMAL[(*M*[,*D*])] [UNSIGNED] [ZEROFILL]](#_bookmark102)

A packed “exact” fixed-point number. *M* is the total number of digits (the precision) and *D* is the number of digits after the decimal point (the scale). The decimal point and (for negative numbers) the

- sign are not counted in *M*. If *D* is 0, values have no decimal point or fractional part. The maximum number of digits (*M*) for [DECIMAL](#_bookmark102) is 65. The maximum number of supported decimals (*D*) is 30. If *D* is omitted, the default is 0. If *M* is omitted, the default is 10. (There is also a limit on how long the text of [DECIMAL](#_bookmark102) literals can be; see Section 12.25.3, “Expression Handling” .)

UNSIGNED, if specified, disallows negative values. As of MySQL 8.0.17, the UNSIGNED attribute is deprecated for columns of type [DECIMAL](#_bookmark102) (and any synonyms); you should expect support for it to be removed in a future version of MySQL. Consider using a simple CHECK constraint instead for such columns.

All basic calculations (+, -, \*, /) with [DECIMAL](#_bookmark102) columns are done with a precision of 65 digits.

• [DEC[(*M*[,*D*])] [UNSIGNED] [ZEROFILL]](#_bookmark102), [NUMERIC[(*M*[,*D*])] [UNSIGNED]](#_bookmark102) [[ZEROFILL]](#_bookmark102), [FIXED[(*M* *D* [UNSIGNED] [ZEROFILL]](#_bookmark102)[,])]

These types are synonyms for [DECIMAL](#_bookmark102). The [FIXED](#_bookmark102) synonym is available for compatibility with other database systems.

• [FLOAT[(*M*,*D*)] [UNSIGNED] [ZEROFILL]](#_bookmark103)

A small (single-precision) floating-point number. Permissible values are -3.402823466E+38 to -1.175494351E-38, 0, and 1.175494351E-38 to 3.402823466E+38. These are the theoretical limits, based on the IEEE standard. The actual range might be slightly smaller depending on your hardware or operating system.

*M* is the total number of digits and *D* is the number of digits following the decimal point. If *M*and *D* are omitted, values are stored to the limits permitted by the hardware. A single-precision floating-point number is accurate to approximately 7 decimal places.

FLOAT(*M*,*D*) is a nonstandard MySQL extension. As of MySQL 8.0.17, this syntax is deprecated, and you should expect support for it to be removed in a future version of MySQL.

UNSIGNED, if specified, disallows negative values. As of MySQL 8.0.17, the UNSIGNED attribute is deprecated for columns of type [FLOAT](#_bookmark103) (and any synonyms) and you should expect support for it to be removed in a future version of MySQL. Consider using a simple CHECK constraint instead for such columns.

Using [FLOAT](#_bookmark103) might give you some unexpected problems because all calculations in MySQL are done with double precision. See Section B.3.4.7, “Solving Problems with No Matching Rows” .

• [FLOAT(*p*) [UNSIGNED] [ZEROFILL]](#_bookmark103)

A floating-point number. *p* represents the precision in bits, but MySQL uses this value only to determine whether to use [FLOAT](#_bookmark103) or [DOUBLE](#_bookmark103) for the resulting data type. If *p* is from 0 to 24, the data type becomes [FLOAT](#_bookmark103) with no *M*or *D* values. If *p* is from 25 to 53, the data type becomes [DOUBLE](#_bookmark103) with no *M*or *D* values. The range of the resulting column is the same as for the single-precision [FLOAT](#_bookmark103) or double-precision [DOUBLE](#_bookmark103) data types described earlier in this section.

UNSIGNED, if specified, disallows negative values. As of MySQL 8.0.17, the UNSIGNED attribute is deprecated for columns of type [FLOAT](#_bookmark103) (and any synonyms) and you should expect support for it to be removed in a future version of MySQL. Consider using a simple CHECK constraint instead for such columns.

[FLOAT(*p*)](#_bookmark103) syntax is provided for ODBC compatibility.

• [DOUBLE[(*M*,*D*)] [UNSIGNED] [ZEROFILL]](#_bookmark103)

A normal-size (double-precision) floating-point number. Permissible values are -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308. These are the theoretical limits,

based on the IEEE standard. The actual range might be slightly smaller depending on your hardware or operating system.

*M* is the total number of digits and *D* is the number of digits following the decimal point. If *M*and *D* are omitted, values are stored to the limits permitted by the hardware. A double-precision floating-point number is accurate to approximately 15 decimal places.

DOUBLE(*M*,*D*) is a nonstandard MySQL extension. As of MySQL 8.0.17, this syntax is deprecated and you should expect support for it to be removed in a future version of MySQL.

UNSIGNED, if specified, disallows negative values. As of MySQL 8.0.17, the UNSIGNED attribute is deprecated for columns of type [DOUBLE](#_bookmark103) (and any synonyms) and you should expect support for it to be removed in a future version of MySQL. Consider using a simple CHECK constraint instead for such columns.

• [DOUBLE PRECISION[(*M*,*D*)] [UNSIGNED] [ZEROFILL]](#_bookmark103), [REAL[(*M*,*D*)] [UNSIGNED]](#_bookmark103) [[ZEROFILL]](#_bookmark103)

These types are synonyms for [DOUBLE](#_bookmark103). Exception: If the REAL\_AS\_FLOAT SQL mode is enabled, [REAL](#_bookmark103) is a synonym for [FLOAT](#_bookmark103) rather than [DOUBLE](#_bookmark103).

**11.1.2** **Integer** **Types** **(Exact** **Value)** **-** **INTEGER,** **INT,** **SMALLINT,** **TINYINT,**

**MEDIUMINT,** **BIGINT**

MySQL supports the SQL standard integer types INTEGER (or INT) and SMALLINT. As an extension to the standard, MySQL also supports the integer types TINYINT, MEDIUMINT, and BIGINT. The following table shows the required storage and range for each integer type.

**Table** **11.1** **Required** **Storage** **and** **Range** **for** **Integer** **Types** **Supported** **by** **MySQL**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Storage**  **(Bytes)** | **Minimum** **Value** **Signed** | **Minimum**  **Value**  **Unsigned** | **Maximum** **Value** **Signed** | **Maximum**  **Value**  **Unsigned** |
| TINYINT | 1 | -128 | 0 | 127 | 255 |
| SMALLINT | 2 | -32768 | 0 | 32767 | 65535 |
| MEDIUMINT | 3 | -8388608 | 0 | 8388607 | 16777215 |
| INT | 4 | -2147483648 | 0 | 2147483647 | 4294967295 |
| BIGINT | 8 | -263 | 0 | 263-1 | 264-1 |

**11.1.3** **Fixed-Point** **Types** **(Exact** **Value)** **-** **DECIMAL,** **NUMERIC**

The DECIMAL and NUMERIC types store exact numeric data values. These types are used when it is important to preserve exact precision, for example with monetary data. In MySQL, NUMERIC is implemented as DECIMAL, so the following remarks about DECIMAL apply equally to NUMERIC.

MySQL stores DECIMAL values in binary format. See Section 12.25, “Precision Math” .

In a DECIMAL column declaration, the precision and scale can be (and usually is) specified. For example:

salary DECIMAL(5,2)

In this example, 5 is the precision and 2 is the scale. The precision represents the number of significant digits that are stored for values, and the scale represents the number of digits that can be stored following the decimal point.

Standard SQL requires that DECIMAL(5,2) be able to store any value with five digits and two decimals, so values that can be stored in the salary column range from -999.99 to 999.99.

In standard SQL, the syntax DECIMAL(*M*) is equivalent to DECIMAL(*M*,0). Similarly, the syntax DECIMAL is equivalent to DECIMAL(*M*,0), where the implementation is permitted to decide the value of *M*. MySQL supports both of these variant forms of DECIMAL syntax. The default value of *M* is 10.

If the scale is 0, DECIMAL values contain no decimal point or fractional part.

The maximum number of digits for DECIMAL is 65, but the actual range for a given DECIMAL column can be constrained by the precision or scale for a given column. When such a column is assigned a value with more digits following the decimal point than are permitted by the specified scale, the value is converted to that scale. (The precise behavior is operating system-specific, but generally the effect is truncation to the permissible number of digits.)

**11.1.4** **Floating-Point** **Types** **(Approximate** **Value)** **-** **FLOAT,** **DOUBLE**

The FLOAT and DOUBLE types represent approximate numeric data values. MySQL uses four bytes for single-precision values and eight bytes for double-precision values.

For FLOAT, the SQL standard permits an optional specification of the precision (but not the range of the exponent) in bits following the keyword FLOAT in parentheses, that is, [FLOAT(*p*)](#_bookmark103). MySQL also supports this optional precision specification, but the precision value in [FLOAT(*p*)](#_bookmark103) is used only to determine storage size. A precision from 0 to 23 results in a 4-byte single-precision FLOAT column. A precision from 24 to 53 results in an 8-byte double-precision DOUBLE column.

MySQL permits a nonstandard syntax: FLOAT(*M*,*D*) or REAL(*M*,*D*) or DOUBLE PRECISION(*M*,*D*). Here, (*M*,*D*) means than values can be stored with up to *M*digits in total, of which *D* digits may be after the decimal point. For example, a column defined as FLOAT(7,4) is displayed as -999.9999. MySQL performs rounding when storing values, so if you insert 999.00009 into a FLOAT(7,4) column, the approximate result is 999.0001.

As of MySQL 8.0.17, the nonstandard FLOAT(*M*,*D*) and DOUBLE(*M*,*D*) syntax is deprecated and you should expect support for it to be removed in a future version of MySQL.

Because floating-point values are approximate and not stored as exact values, attempts to treat them as exact in comparisons may lead to problems. They are also subject to platform or implementation dependencies. For more information, see Section B.3.4.8, “Problems with Floating-Point Values” .

For maximum portability, code requiring storage of approximate numeric data values should use FLOAT or DOUBLE PRECISION with no specification of precision or number of digits.

**11.1.5** **Bit-Value** **Type** **-** **BIT**

The BIT data type is used to store bit values. A type of BIT(*M*) enables storage of *M*-bit values. *M*can range from 1 to 64.

To specify bit values, b'*value*' notation can be used. *value* is a binary value written using zeros and ones. For example, b'111' and b'10000000' represent 7 and 128, respectively. See Section 9.1.5,

“Bit-Value Literals” .

If you assign a value to a BIT(*M*) column that is less than *M* bits long, the value is padded on the left with zeros. For example, assigning a value of b'101' to a BIT(6) column is, in effect, the same as assigning b'000101'.

**NDB** **Cluster.** The maximum combined size of all BIT columns used in a given NDB table must not exceed 4096 bits.

**11.1.6** **Numeric** **Type** **Attributes**

MySQL supports an extension for optionally specifying the display width of integer data types in parentheses following the base keyword for the type. For example, [INT(4)](#_bookmark101) specifies an [INT](#_bookmark101) with a display width of four digits. This optional display width may be used by applications to display integer



values having a width less than the width specified for the column by left-padding them with spaces. (That is, this width is present in the metadata returned with result sets. Whether it is used is up to the application.)

The display width does *not* constrain the range of values that can be stored in the column. Nor does it prevent values wider than the column display width from being displayed correctly. For example, a column specified as [SMALLINT(3)](#_bookmark101) has the usual [SMALLINT](#_bookmark101) range of -32768 to 32767, and values outside the range permitted by three digits are displayed in full using more than three digits.

When used in conjunction with the optional (nonstandard) ZEROFILL attribute, the default padding of spaces is replaced with zeros. For example, for a column declared as [INT(4) ZEROFILL](#_bookmark101), a value of 5 is retrieved as 0005.

**Note**

The ZEROFILL attribute is ignored for columns involved in expressions or UNION queries.

If you store values larger than the display width in an integer column that has the ZEROFILL attribute, you may experience problems when MySQL generates temporary tables for some complicated joins. In these cases, MySQL assumes that the data values fit within the column display width.

As of MySQL 8.0.17, the ZEROFILL attribute is deprecated for numeric data types, as is the display width attribute for integer data types. You should expect support for ZEROFILL and display widths for integer data types to be removed in a future version of MySQL. Consider using an alternative means of producing the effect of these attributes. For example, applications can use the LPAD() function to zero-pad numbers up to the desired width, or they can store the formatted numbers in [CHAR](#_bookmark60) columns.

All integer types can have an optional (nonstandard) UNSIGNED attribute. An unsigned type can be used to permit only nonnegative numbers in a column or when you need a larger upper numeric range for the column. For example, if an [INT](#_bookmark101) column is UNSIGNED, the size of the column's range is the same but its endpoints shift up, from -2147483648 and 2147483647 to 0 and 4294967295.

Floating-point and fixed-point types also can be UNSIGNED. As with integer types, this attribute prevents negative values from being stored in the column. Unlike the integer types, the upper range of column values remains the same. As of MySQL 8.0.17, the UNSIGNED attribute is deprecated for columns of type [FLOAT](#_bookmark103), [DOUBLE](#_bookmark103), and [DECIMAL](#_bookmark102) (and any synonyms) and you should expect support for it to be removed in a future version of MySQL. Consider using a simple CHECK constraint instead for such columns.

If you specify ZEROFILL for a numeric column, MySQL automatically adds the UNSIGNED attribute.

Integer or floating-point data types can have the AUTO\_INCREMENT attribute. When you insert a value of NULL into an indexed AUTO\_INCREMENT column, the column is set to the next sequence value. Typically this is *value*+1, where *value* is the largest value for the column currently in the table. (AUTO\_INCREMENT sequences begin with 1.)

Storing 0 into an AUTO\_INCREMENT column has the same effect as storing NULL, unless the NO\_AUTO\_VALUE\_ON\_ZERO SQL mode is enabled.

Inserting NULL to generate AUTO\_INCREMENT values requires that the column be declared NOT NULL. If the column is declared NULL, inserting NULL stores a NULL. When you insert any other value into an AUTO\_INCREMENT column, the column is set to that value and the sequence is reset so that the next automatically generated value follows sequentially from the inserted value.

Negative values for AUTO\_INCREMENT columns are not supported.

CHECK constraints cannot refer to columns that have the AUTO\_INCREMENT attribute, nor can the AUTO\_INCREMENT attribute be added to existing columns that are used in CHECK constraints.

As of MySQL 8.0.17, AUTO\_INCREMENT support is deprecated for [FLOAT](#_bookmark103) and [DOUBLE](#_bookmark103) columns; you should expect it to be removed in a future version of MySQL. Consider removing the AUTO\_INCREMENT attribute from such columns, or convert them to an integer type.

**11.1.7** **Out-of-Range** **and** **Overflow** **Handling**

When MySQL stores a value in a numeric column that is outside the permissible range of the column data type, the result depends on the SQL mode in effect at the time:

• If strict SQL mode is enabled, MySQL rejects the out-of-range value with an error, and the insert fails, in accordance with the SQL standard.

• If no restrictive modes are enabled, MySQL clips the value to the appropriate endpoint of the column data type range and stores the resulting value instead.

When an out-of-range value is assigned to an integer column, MySQL stores the value representing the corresponding endpoint of the column data type range.

When a floating-point or fixed-point column is assigned a value that exceeds the range implied by the specified (or default) precision and scale, MySQL stores the value representing the corresponding endpoint of that range.

Suppose that a table t1 has this definition:

CREATE TABLE t1 (i1 TINYINT, i2 TINYINT UNSIGNED);

With strict SQL mode enabled, an out of range error occurs:

|  |  |
| --- | --- |
| mysql> **SET** **sql\_mode** **=** **'TRADITIONAL';**  mysql> **INSERT** **INTO** **t1** **(i1,** **i2)** **VALUES(256,**  ERROR 1264 (22003): Out of range value for  mysql> **SELECT** **\*** **FROM** **t1;**  Empty set (0.00 sec) | **256);**  column 'i1' at row 1 |

With strict SQL mode not enabled, clipping with warnings occurs:

mysql> **SET** **sql\_mode** **=** **'';**

mysql> **INSERT** **INTO** **t1** **(i1,** **i2)** **VALUES(256,** **256);**

mysql> **SHOW** **WARNINGS;**

+---------+------+---------------------------------------------+

| Level | Code | Message |

+---------+------+---------------------------------------------+

| Warning | 1264 | Out of range value for column 'i1' at row 1 |

| Warning | 1264 | Out of range value for column 'i2' at row 1 |

+---------+------+---------------------------------------------+

mysql> **SELECT** **\*** **FROM** **t1;**

+------+------+

| i1 | i2 |

+------+------+

| 127 | 255 |

+------+------+

When strict SQL mode is not enabled, column-assignment conversions that occur due to clipping are reported as warnings for ALTER TABLE, LOAD DATA, UPDATE, and multiple-row INSERT statements. In strict mode, these statements fail, and some or all the values are not inserted or changed, depending on whether the table is a transactional table and other factors. For details, see Section 5.1.11, “Server

SQL Modes” .

Overflow during numeric expression evaluation results in an error. For example, the largest signed [BIGINT](#_bookmark101) value is 9223372036854775807, so the following expression produces an error:

mysql> **SELECT** **9223372036854775807** **+** **1;**

ERROR 1690 (22003): BIGINT value is out of range in '(9223372036854775807 + 1)'

To enable the operation to succeed in this case, convert the value to unsigned;

mysql> **SELECT** **CAST(9223372036854775807** **AS** **UNSIGNED)** **+** **1;**

+-------------------------------------------+

| CAST(9223372036854775807 AS UNSIGNED) + 1 |

+-------------------------------------------+

| 9223372036854775808 |

+-------------------------------------------+

Whether overflow occurs depends on the range of the operands, so another way to handle the preceding expression is to use exact-value arithmetic because [DECIMAL](#_bookmark102) values have a larger range than integers:

mysql> **SELECT** **9223372036854775807.0** **+** **1;**

+---------------------------+

| 9223372036854775807.0 + 1 |

+---------------------------+

| 9223372036854775808.0 |

+---------------------------+

Subtraction between integer values, where one is of type UNSIGNED, produces an unsigned result by default. If the result would otherwise have been negative, an error results:

mysql> **SET** **sql\_mode** **=** **'';**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT** **CAST(0** **AS** **UNSIGNED)** **-** **1;**

ERROR 1690 (22003): BIGINT UNSIGNED value is out of range in '(cast(0 as unsigned) - 1)'

If the NO\_UNSIGNED\_SUBTRACTION SQL mode is enabled, the result is negative:

mysql> **SET** **sql\_mode** **=** **'NO\_UNSIGNED\_SUBTRACTION';**

mysql> **SELECT** **CAST(0** **AS** **UNSIGNED)** **-** **1;**

+-------------------------+

| CAST(0 AS UNSIGNED) - 1 |

+-------------------------+

| -1 |

+-------------------------+

If the result of such an operation is used to update an UNSIGNED integer column, the result is clipped to the maximum value for the column type, or clipped to 0 if NO\_UNSIGNED\_SUBTRACTION is enabled. If strict SQL mode is enabled, an error occurs and the column remains unchanged.

**11.2** **Date** **and** **Time** **Data** **Types**

The date and time data types for representing temporal values are [DATE](#_bookmark109), [TIME](#_bookmark110), [DATETIME](#_bookmark109), [TIMESTAMP](#_bookmark109), and [YEAR](#_bookmark111). Each temporal type has a range of valid values, as well as a “zero” value that may be used when you specify an invalid value that MySQL cannot represent. The [TIMESTAMP](#_bookmark109) and [DATETIME](#_bookmark109) types have special automatic updating behavior, described in [Section 11.2.5, “Automatic](#_bookmark112) [Initialization and Updating for TIMESTAMP and DATETIME”](#_bookmark112) .

For information about storage requirements of the temporal data types, see [Section 11.7, “Data Type](#_bookmark132) [Storage Requirements”](#_bookmark132) .

For descriptions of functions that operate on temporal values, see Section 12.7, “Date and Time

Functions” .

Keep in mind these general considerations when working with date and time types:

• MySQL retrieves values for a given date or time type in a standard output format, but it attempts to interpret a variety of formats for input values that you supply (for example, when you specify a value to be assigned to or compared to a date or time type). For a description of the permitted formats for date and time types, see Section 9.1.3, “Date and Time Literals” . It is expected that you supply valid values. Unpredictable results may occur if you use values in other formats.

• Although MySQL tries to interpret values in several formats, date parts must always be given in year- month-day order (for example, '98-09-04'), rather than in the month-day-year or day-month-year orders commonly used elsewhere (for example, '09-04-98', '04-09-98'). To convert strings in other orders to year-month-day order, the STR\_TO\_DATE() function may be useful.

• Dates containing 2-digit year values are ambiguous because the century is unknown. MySQL interprets 2-digit year values using these rules:

• Year values in the range 70-99 become 1970-1999.

• Year values in the range 00-69 become 2000-2069. See also [Section 11.2.8, “2-Digit Years in Dates”](#_bookmark115) .

• Conversion of values from one temporal type to another occurs according to the rules in [Section 11.2.7, “Conversion Between Date and Time Types”](#_bookmark114) .

• MySQL automatically converts a date or time value to a number if the value is used in numeric context and vice versa.

• By default, when MySQL encounters a value for a date or time type that is out of range or otherwise invalid for the type, it converts the value to the “zero” value for that type. The exception is that out-of- range [TIME](#_bookmark110) values are clipped to the appropriate endpoint of the [TIME](#_bookmark110) range.

• By setting the SQL mode to the appropriate value, you can specify more exactly what kind of dates you want MySQL to support. (See Section 5.1.11, “Server SQL Modes” .) You can get MySQL to accept certain dates, such as '2009-11-31', by enabling the ALLOW\_INVALID\_DATES SQL mode. This is useful when you want to store a “possibly wrong” value which the user has specified (for example, in a web form) in the database for future processing. Under this mode, MySQL verifies only that the month is in the range from 1 to 12 and that the day is in the range from 1 to 31.

• MySQL permits you to store dates where the day or month and day are zero in a [DATE](#_bookmark109) or [DATETIME](#_bookmark109) column. This is useful for applications that need to store birthdates for which you may not know the exact date. In this case, you simply store the date as '2009-00-00' or '2009-01-00'. However, with dates such as these, you should not expect to get correct results for functions such as DATE\_SUB() or DATE\_ADD() that require complete dates. To disallow zero month or day parts in dates, enable the NO\_ZERO\_IN\_DATE mode.

• MySQL permits you to store a “zero” value of '0000-00-00' as a “dummy date.” In some cases, this is more convenient than using NULL values, and uses less data and index space. To disallow '0000-00-00', enable the NO\_ZERO\_DATE mode.

• “Zero” date or time values used through Connector/ODBC are converted automatically to NULL because ODBC cannot handle such values.

The following table shows the format of the “zero” value for each type. The “zero” values are special, but you can store or refer to them explicitly using the values shown in the table. You can also do

this using the values '0' or 0, which are easier to write. For temporal types that include a date part ([DATE](#_bookmark109), [DATETIME](#_bookmark109), and [TIMESTAMP](#_bookmark109)), use of these values may produce warning or errors. The precise behavior depends on which, if any, of the strict and NO\_ZERO\_DATE SQL modes are enabled; see Section 5. 1. 11, “Server SQL Modes” .

|  |  |
| --- | --- |
| **Data** **Type** | **“Zero”** **Value** |
| [DATE](#_bookmark109) | '0000-00-00' |
| [TIME](#_bookmark110) | '00:00:00' |
| [DATETIME](#_bookmark109) | '0000-00-00 00:00:00' |
| [TIMESTAMP](#_bookmark109) | '0000-00-00 00:00:00' |
| [YEAR](#_bookmark111) | 0000 |

**11.2.1** **Date** **and** **Time** **Data** **Type** **Syntax**

The date and time data types for representing temporal values are [DATE](#_bookmark109), [TIME](#_bookmark110), [DATETIME](#_bookmark109), [TIMESTAMP](#_bookmark109), and [YEAR](#_bookmark111).

For the [DATE](#_bookmark109) and [DATETIME](#_bookmark109) range descriptions, “supported” means that although earlier values might work, there is no guarantee.

MySQL permits fractional seconds for [TIME](#_bookmark110), [DATETIME](#_bookmark109), and [TIMESTAMP](#_bookmark109) values, with up to microseconds (6 digits) precision. To define a column that includes a fractional seconds part, use the syntax *type\_name* (*fsp*), where *type\_name* is [TIME](#_bookmark110), [DATETIME](#_bookmark109), or [TIMESTAMP](#_bookmark109), and *fsp* is the fractional seconds precision. For example:

CREATE TABLE t1 (t TIME(3), dt DATETIME(6), ts TIMESTAMP(0));

The *fsp* value, if given, must be in the range 0 to 6. A value of 0 signifies that there is no fractional part. If omitted, the default precision is 0. (This differs from the standard SQL default of 6, for compatibility with previous MySQL versions.)

Any [TIMESTAMP](#_bookmark109) or [DATETIME](#_bookmark109) column in a table can have automatic initialization and updating properties; see [Section 11.2.5, “Automatic Initialization and Updating for TIMESTAMP and](#_bookmark112) [DATETIME”](#_bookmark112) .

• [DATE](#_bookmark109)

A date. The supported range is '1000-01-01' to '9999-12-31'. MySQL displays [DATE](#_bookmark109) values in '*YYYY-MM-DD*' format, but permits assignment of values to [DATE](#_bookmark109) columns using either strings or numbers.

• [DATETIME[(*fsp*)]](#_bookmark109)

A date and time combination. The supported range is '1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'. MySQL displays [DATETIME](#_bookmark109) values in '*YYYY-MM-DD* *hh:mm:ss* [.*fraction*]' format, but permits assignment of values to [DATETIME](#_bookmark109) columns using either strings or numbers.

An optional *fsp* value in the range from 0 to 6 may be given to specify fractional seconds precision. A value of 0 signifies that there is no fractional part. If omitted, the default precision is 0.

Automatic initialization and updating to the current date and time for [DATETIME](#_bookmark109) columns can be specified using DEFAULT and ON UPDATE column definition clauses, as described in [Section 11.2.5,](#_bookmark112) [“Automatic Initialization and Updating for TIMESTAMP and DATETIME”](#_bookmark112) .

• [TIMESTAMP[(*fsp*)]](#_bookmark109)

A timestamp. The range is '1970-01-01 00:00:01.000000' UTC to '2038-01-19 03:14:07.999999' UTC. [TIMESTAMP](#_bookmark109) values are stored as the number of seconds since the epoch ( '1970-01-01 00:00:00' UTC). A [TIMESTAMP](#_bookmark109) cannot represent the value '1970-01-01 00:00:00' because that is equivalent to 0 seconds from the epoch and the value 0 is reserved for representing '0000-00-00 00:00:00', the “zero” [TIMESTAMP](#_bookmark109) value.

An optional *fsp* value in the range from 0 to 6 may be given to specify fractional seconds precision. A value of 0 signifies that there is no fractional part. If omitted, the default precision is 0.

The way the server handles TIMESTAMP definitions depends on the value of the explicit\_defaults\_for\_timestamp system variable (see Section 5.1.8, “Server System Variables” ).

If explicit\_defaults\_for\_timestamp is enabled, there is no automatic assignment of the DEFAULT CURRENT\_TIMESTAMP or ON UPDATE CURRENT\_TIMESTAMP attributes to any [TIMESTAMP](#_bookmark109) column. They must be included explicitly in the column definition. Also, any [TIMESTAMP](#_bookmark109) not explicitly declared as NOT NULL permits NULL values.

If explicit\_defaults\_for\_timestamp is disabled, the server handles TIMESTAMP as follows:

Unless specified otherwise, the first [TIMESTAMP](#_bookmark109) column in a table is defined to be automatically set to the date and time of the most recent modification if not explicitly assigned a value. This makes



[TIMESTAMP](#_bookmark109) useful for recording the timestamp of an INSERT or UPDATE operation. You can also set any [TIMESTAMP](#_bookmark109) column to the current date and time by assigning it a NULL value, unless it has been defined with the NULL attribute to permit NULL values.

Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP column definition clauses. By default, the first [TIMESTAMP](#_bookmark109) column has these properties, as previously noted. However, any [TIMESTAMP](#_bookmark109) column in a table can be defined to have these properties.

• [TIME[(*fsp*)]](#_bookmark110)

A time. The range is '-838:59:59.000000' to '838:59:59.000000'. MySQL displays [TIME](#_bookmark110) values in '*hh:mm:ss* [.*fraction*]' format, but permits assignment of values to [TIME](#_bookmark110) columns using either strings or numbers.

An optional *fsp* value in the range from 0 to 6 may be given to specify fractional seconds precision. A value of 0 signifies that there is no fractional part. If omitted, the default precision is 0.

• [YEAR[(4)]](#_bookmark111)

A year in 4-digit format. MySQL displays [YEAR](#_bookmark111) values in *YYYY*format, but permits assignment of values to [YEAR](#_bookmark111) columns using either strings or numbers. Values display as 1901 to 2155, or 0000.

For additional information about [YEAR](#_bookmark111) display format and interpretation of input values, see [Section 11.2.4, “The YEAR Type”](#_bookmark111) .

**Note**

As of MySQL 8.0.19, the [YEAR(4)](#_bookmark111) data type with an explicit display width is deprecated; you should expect support for it to be removed in a future version of MySQL. Instead, use [YEAR](#_bookmark111) without a display width, which has the same meaning.

MySQL 8.0 does not support the 2-digit [YEAR(2)](#_bookmark111) data type permitted in older versions of MySQL. For instructions on converting to 4-digit [YEAR](#_bookmark111), see [2-Digit](https://dev.mysql.com/doc/refman/5.7/en/migrating-from-year2.html) [YEAR(2) Limitations and Migrating to 4-Digit YEAR](https://dev.mysql.com/doc/refman/5.7/en/migrating-from-year2.html), in [MySQL 5.7 Reference](https://dev.mysql.com/doc/refman/5.7/en/) [Manual](https://dev.mysql.com/doc/refman/5.7/en/).

The SUM() and AVG() aggregate functions do not work with temporal values. (They convert the values to numbers, losing everything after the first nonnumeric character.) To work around this problem, convert to numeric units, perform the aggregate operation, and convert back to a temporal value. Examples:

SELECT SEC\_TO\_TIME(SUM(TIME\_TO\_SEC(*time\_col*))) FROM *tbl\_name*;

SELECT FROM\_DAYS(SUM(TO\_DAYS(*date\_col*))) FROM *tbl\_name*;

**11.2.2** **The** **DATE,** **DATETIME,** **and** **TIMESTAMP** **Types**

The DATE, DATETIME, and TIMESTAMP types are related. This section describes their characteristics, how they are similar, and how they differ. MySQL recognizes DATE, DATETIME, and TIMESTAMP values in several formats, described in Section 9.1.3, “Date and Time Literals” . For the DATE and DATETIME range descriptions, “supported” means that although earlier values might work, there is no guarantee.

The DATE type is used for values with a date part but no time part. MySQL retrieves and displays DATE values in '*YYYY-MM-DD*' format. The supported range is '1000-01-01' to '9999-12-31'.

The DATETIME type is used for values that contain both date and time parts. MySQL retrieves and displays DATETIME values in '*YYYY-MM-DD* *hh:mm:ss*' format. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.

The TIMESTAMP data type is used for values that contain both date and time parts. TIMESTAMP has a range of '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.

A DATETIME or TIMESTAMP value can include a trailing fractional seconds part in up to microseconds (6 digits) precision. In particular, any fractional part in a value inserted into a DATETIME or TIMESTAMP column is stored rather than discarded. With the fractional part included, the format for these values is '*YYYY-MM-DD* *hh:mm:ss* [.*fraction*]', the range for DATETIME values is '1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999', and the range for TIMESTAMP values is '1970-01-01 00:00:01.000000' to '2038-01-19 03:14:07.999999'. The fractional

part should always be separated from the rest of the time by a decimal point; no other fractional seconds delimiter is recognized. For information about fractional seconds support in MySQL, see [Section 11.2.6, “Fractional Seconds in Time Values”](#_bookmark113) .

The TIMESTAMP and DATETIME data types offer automatic initialization and updating to the current date and time. For more information, see [Section 11.2.5, “Automatic Initialization and Updating for](#_bookmark112) [TIMESTAMP and DATETIME”](#_bookmark112) .

MySQL converts TIMESTAMP values from the current time zone to UTC for storage, and back from UTC to the current time zone for retrieval. (This does not occur for other types such as DATETIME.) By default, the current time zone for each connection is the server's time. The time zone can be set on a per-connection basis. As long as the time zone setting remains constant, you get back the same value you store. If you store a TIMESTAMP value, and then change the time zone and retrieve the value, the retrieved value is different from the value you stored. This occurs because the same time zone was not used for conversion in both directions. The current time zone is available as the value of the time\_zone system variable. For more information, see Section 5.1.15, “MySQL Server Time Zone Support” .

In MySQL 8.0.19 and later, you can specify a time zone offset when inserting a TIMESTAMP or DATETIME value into a table. See Section 9.1.3, “Date and Time Literals” , for more information and examples.

Invalid DATE, DATETIME, or TIMESTAMP values are converted to the “zero” value of the appropriate type ( '0000-00-00' or '0000-00-00 00:00:00'), if the SQL mode permits this conversion. The precise behavior depends on which if any of strict SQL mode and the NO\_ZERO\_DATE SQL mode are enabled; see Section 5.1.11, “Server SQL Modes” .

In MySQL 8.0.22 and later, you can convert TIMESTAMP values to UTC DATETIME values when retrieving them using CAST() with the AT TIME ZONE operator, as shown here:

mysql> **SELECT** **col,**

|  |  |
| --- | --- |
| >  > | **CAST(col** **AT** **TIME** **ZONE** **INTERVAL** **'+00:00'** **AS** **DATETIME)** **AS** **ut**  **FROM** **ts** **ORDER** **BY** **id;** |

+---------------------+---------------------+

| col | ut |

+---------------------+---------------------+

| 2020-01-01 10:10:10 | 2020-01-01 15:10:10 |

| 2019-12-31 23:40:10 | 2020-01-01 04:40:10 |

| 2020-01-01 13:10:10 | 2020-01-01 18:10:10 |

| 2020-01-01 10:10:10 | 2020-01-01 15:10:10 |

| 2020-01-01 04:40:10 | 2020-01-01 09:40:10 |

| 2020-01-01 18:10:10 | 2020-01-01 23:10:10 |

+---------------------+---------------------+

For complete information regarding syntax and additional examples, see the description of the CAST() function.

Be aware of certain properties of date value interpretation in MySQL:

• MySQL permits a “relaxed” format for values specified as strings, in which any punctuation character may be used as the delimiter between date parts or time parts. In some cases, this syntax can be deceiving. For example, a value such as '10:11:12' might look like a time value because of the :, but is interpreted as the year '2010-11-12' if used in date context. The value '10:45:15' is converted to '0000-00-00' because '45' is not a valid month.

The only delimiter recognized between a date and time part and a fractional seconds part is the decimal point.

• The server requires that month and day values be valid, and not merely in the range 1 to 12 and 1 to 31, respectively. With strict mode disabled, invalid dates such as '2004-04-31' are converted to '0000-00-00' and a warning is generated. With strict mode enabled, invalid dates generate an error. To permit such dates, enable ALLOW\_INVALID\_DATES. See Section 5.1.11, “Server SQL Modes” , for more information.

• MySQL does not accept TIMESTAMP values that include a zero in the day or month column or values that are not a valid date. The sole exception to this rule is the special “zero” value '0000-00-00 00:00:00', if the SQL mode permits this value. The precise behavior depends on which if any of strict SQL mode and the NO\_ZERO\_DATE SQL mode are enabled; see Section 5.1.11, “Server SQL Modes” .

• Dates containing 2-digit year values are ambiguous because the century is unknown. MySQL interprets 2-digit year values using these rules:

• Year values in the range 00-69 become 2000-2069.

• Year values in the range 70-99 become 1970-1999. See also [Section 11.2.8, “2-Digit Years in Dates”](#_bookmark115) .

**11.2.3** **The** **TIME** **Type**

MySQL retrieves and displays TIME values in *'hh:mm:ss'* format (or *'hhh:mm:ss'* format for large hours values). TIME values may range from '-838:59:59' to '838:59:59'. The hours part may be so large because the TIME type can be used not only to represent a time of day (which must be less than 24 hours), but also elapsed time or a time interval between two events (which may be much greater than 24 hours, or even negative).

MySQL recognizes TIME values in several formats, some of which can include a trailing fractional seconds part in up to microseconds (6 digits) precision. See Section 9.1.3, “Date and Time Literals” . For information about fractional seconds support in MySQL, see [Section 11.2.6, “Fractional](#_bookmark113) [Seconds in Time Values”](#_bookmark113) . In particular, any fractional part in a value inserted into a TIME column is stored rather than discarded. With the fractional part included, the range for TIME values is '-838:59:59.000000' to '838:59:59.000000'.

Be careful about assigning abbreviated values to a TIME column. MySQL interprets abbreviated TIME values with colons as time of the day. That is, '11:12' means '11:12:00', not '00:11:12'. MySQL interprets abbreviated values without colons using the assumption that the two rightmost digits represent seconds (that is, as elapsed time rather than as time of day). For example, you might think of '1112' and 1112 as meaning '11:12:00' (12 minutes after 11 o'clock), but MySQL interprets them as '00:11:12' (11 minutes, 12 seconds). Similarly, '12' and 12 are interpreted as '00:00:12'.

The only delimiter recognized between a time part and a fractional seconds part is the decimal point.

By default, values that lie outside the TIME range but are otherwise valid are clipped to the

closest endpoint of the range. For example, '-850:00:00' and '850:00:00' are converted to '-838:59:59' and '838:59:59'. Invalid TIME values are converted to '00:00:00'. Note that because '00:00:00' is itself a valid TIME value, there is no way to tell, from a value of '00:00:00' stored in a table, whether the original value was specified as '00:00:00' or whether it was invalid.

For more restrictive treatment of invalid TIME values, enable strict SQL mode to cause errors to occur. See Section 5.1.11, “Server SQL Modes” .

**11.2.4** **The** **YEAR** **Type**



The YEAR type is a 1-byte type used to represent year values. It can be declared as YEAR with an implicit display width of 4 characters, or equivalently as YEAR(4) with an explicit display width.

**Note**

As of MySQL 8.0.19, the [YEAR(4)](#_bookmark111) data type with an explicit display width is deprecated and you should expect support for it to be removed in a future version of MySQL. Instead, use [YEAR](#_bookmark111) without a display width, which has the same meaning.

MySQL 8.0 does not support the 2-digit [YEAR(2)](#_bookmark111) data type permitted in older versions of MySQL. For instructions on converting to 4-digit [YEAR](#_bookmark111), see [2-Digit](https://dev.mysql.com/doc/refman/5.7/en/migrating-from-year2.html) [YEAR(2) Limitations and Migrating to 4-Digit YEAR](https://dev.mysql.com/doc/refman/5.7/en/migrating-from-year2.html), in [MySQL 5.7 Reference](https://dev.mysql.com/doc/refman/5.7/en/) [Manual](https://dev.mysql.com/doc/refman/5.7/en/).

MySQL displays YEAR values in *YYYY*format, with a range of 1901 to 2155, and 0000. YEAR accepts input values in a variety of formats:

• As 4-digit strings in the range '1901' to '2155'.

• As 4-digit numbers in the range 1901 to 2155.

• As 1- or 2-digit strings in the range '0' to '99'. MySQL converts values in the ranges '0' to '69' and '70' to '99' to YEAR values in the ranges 2000 to 2069 and 1970 to 1999.

• As 1- or 2-digit numbers in the range 0 to 99. MySQL converts values in the ranges 1 to 69 and 70 to 99 to YEAR values in the ranges 2001 to 2069 and 1970 to 1999.

The result of inserting a numeric 0 has a display value of 0000 and an internal value of 0000. To insert zero and have it be interpreted as 2000, specify it as a string '0' or '00'.

• As the result of functions that return a value that is acceptable in YEAR context, such as NOW().

If strict SQL mode is not enabled, MySQL converts invalid YEAR values to 0000. In strict SQL mode, attempting to insert an invalid YEAR value produces an error.

See also [Section 11.2.8, “2-Digit Years in Dates”](#_bookmark115) .

**11.2.5** **Automatic** **Initialization** **and** **Updating** **for** **TIMESTAMP** **and** **DATETIME**

[TIMESTAMP](#_bookmark109) and [DATETIME](#_bookmark109) columns can be automatically initialized and updated to the current date and time (that is, the current timestamp).

For any [TIMESTAMP](#_bookmark109) or [DATETIME](#_bookmark109) column in a table, you can assign the current timestamp as the default value, the auto-update value, or both:

• An auto-initialized column is set to the current timestamp for inserted rows that specify no value for the column.

• An auto-updated column is automatically updated to the current timestamp when the value of any other column in the row is changed from its current value. An auto-updated column remains unchanged if all other columns are set to their current values. To prevent an auto-updated column from updating when other columns change, explicitly set it to its current value. To update an auto- updated column even when other columns do not change, explicitly set it to the value it should have (for example, set it to CURRENT\_TIMESTAMP).

In addition, if the explicit\_defaults\_for\_timestamp system variable is disabled, you can initialize or update any [TIMESTAMP](#_bookmark109) (but not DATETIME) column to the current date and time by assigning it a NULL value, unless it has been defined with the NULL attribute to permit NULL values.

To specify automatic properties, use the DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP clauses in column definitions. The order of the clauses does not



matter. If both are present in a column definition, either can occur first. Any of the synonyms for CURRENT\_TIMESTAMP have the same meaning as CURRENT\_TIMESTAMP. These are CURRENT\_TIMESTAMP(), NOW(), LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, and

LOCALTIMESTAMP().

Use of DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP is specific to [TIMESTAMP](#_bookmark109) and [DATETIME](#_bookmark109). The DEFAULT clause also can be used to specify a constant (nonautomatic) default value (for example, DEFAULT 0 or DEFAULT '2000-01-01 00:00:00').

**Note**

The following examples use DEFAULT 0, a default that can produce warnings or errors depending on whether strict SQL mode or the NO\_ZERO\_DATE SQL mode is enabled. Be aware that the TRADITIONAL SQL mode includes strict mode and NO\_ZERO\_DATE. See Section 5.1.11, “Server SQL Modes” .

[TIMESTAMP](#_bookmark109) or [DATETIME](#_bookmark109) column definitions can specify the current timestamp for both the default and auto-update values, for one but not the other, or for neither. Different columns can have different combinations of automatic properties. The following rules describe the possibilities:

• With both DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP, the column has the current timestamp for its default value and is automatically updated to the current timestamp.

CREATE TABLE t1 (

ts TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

dt DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

);

• With a DEFAULT clause but no ON UPDATE CURRENT\_TIMESTAMP clause, the column has the given default value and is not automatically updated to the current timestamp.

The default depends on whether the DEFAULT clause specifies CURRENT\_TIMESTAMP or a constant value. With CURRENT\_TIMESTAMP, the default is the current timestamp.

CREATE TABLE t1 (

ts TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

dt DATETIME DEFAULT CURRENT\_TIMESTAMP

);

With a constant, the default is the given value. In this case, the column has no automatic properties at all.

CREATE TABLE t1 (

ts TIMESTAMP DEFAULT 0,

dt DATETIME DEFAULT 0

);

• With an ON UPDATE CURRENT\_TIMESTAMP clause and a constant DEFAULT clause, the column is automatically updated to the current timestamp and has the given constant default value.

CREATE TABLE t1 (

ts TIMESTAMP DEFAULT 0 ON UPDATE CURRENT\_TIMESTAMP,

dt DATETIME DEFAULT 0 ON UPDATE CURRENT\_TIMESTAMP

);

• With an ON UPDATE CURRENT\_TIMESTAMP clause but no DEFAULT clause, the column is automatically updated to the current timestamp but does not have the current timestamp for its default value.

The default in this case is type dependent. [TIMESTAMP](#_bookmark109) has a default of 0 unless defined with the NULL attribute, in which case the default is NULL.

CREATE TABLE t1 (

ts1 TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP, -- default 0

ts2 TIMESTAMP NULL ON UPDATE CURRENT\_TIMESTAMP -- default NULL

);

[DATETIME](#_bookmark109) has a default of NULL unless defined with the NOT NULL attribute, in which case the default is 0.

CREATE TABLE t1 (

dt1 DATETIME ON UPDATE CURRENT\_TIMESTAMP, -- default NULL

dt2 DATETIME NOT NULL ON UPDATE CURRENT\_TIMESTAMP -- default 0

);

[TIMESTAMP](#_bookmark109) and [DATETIME](#_bookmark109) columns have no automatic properties unless they are specified explicitly, with this exception: If the explicit\_defaults\_for\_timestamp system variable is disabled, the *first* [TIMESTAMP](#_bookmark109) column has both DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP if neither is specified explicitly. To suppress automatic properties for the first [TIMESTAMP](#_bookmark109) column, use one of these strategies:

• Enable the explicit\_defaults\_for\_timestamp system variable. In this case, the DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP clauses that specify automatic initialization and updating are available, but are not assigned to any [TIMESTAMP](#_bookmark109) column unless explicitly included in the column definition.

• Alternatively, if explicit\_defaults\_for\_timestamp is disabled, do either of the following:

• Define the column with a DEFAULT clause that specifies a constant default value.

• Specify the NULL attribute. This also causes the column to permit NULL values, which means that you cannot assign the current timestamp by setting the column to NULL. Assigning NULL sets the column to NULL, not the current timestamp. To assign the current timestamp, set the column to CURRENT\_TIMESTAMP or a synonym such as NOW().

Consider these table definitions:

|  |  |
| --- | --- |
| CREATE TABLE t1 | ( |
| ts1 TIMESTAMP | DEFAULT 0, |
| ts2 TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP  ON UPDATE CURRENT\_TIMESTAMP); |
| CREATE TABLE t2 | ( |
| ts1 TIMESTAMP | NULL, |
| ts2 TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP  ON UPDATE CURRENT\_TIMESTAMP); |
| CREATE TABLE t3 | ( |
| ts1 TIMESTAMP | NULL DEFAULT 0, |
| ts2 TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP  ON UPDATE CURRENT\_TIMESTAMP); |

The tables have these properties:

• In each table definition, the first [TIMESTAMP](#_bookmark109) column has no automatic initialization or updating.

• The tables differ in how the ts1 column handles NULL values. For t1, ts1 is NOT NULL and assigning it a value of NULL sets it to the current timestamp. For t2 and t3, ts1 permits NULL and assigning it a value of NULL sets it to NULL.

• t2 and t3 differ in the default value for ts1. For t2, ts1 is defined to permit NULL, so the default is also NULL in the absence of an explicit DEFAULT clause. For t3, ts1 permits NULL but has an explicit default of 0.

If a [TIMESTAMP](#_bookmark109) or [DATETIME](#_bookmark109) column definition includes an explicit fractional seconds precision value anywhere, the same value must be used throughout the column definition. This is permitted:

CREATE TABLE t1 (

ts TIMESTAMP(6) DEFAULT CURRENT\_TIMESTAMP(6) ON UPDATE CURRENT\_TIMESTAMP(6)

);

This is not permitted:

CREATE TABLE t1 (

ts TIMESTAMP(6) DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP(3)

);

**TIMESTAMP** **Initialization** **and** **the** **NULL** **Attribute**

If the explicit\_defaults\_for\_timestamp system variable is disabled, [TIMESTAMP](#_bookmark109) columns by default are NOT NULL, cannot contain NULL values, and assigning NULL assigns the current timestamp. To permit a [TIMESTAMP](#_bookmark109) column to contain NULL, explicitly declare it with the NULL attribute. In this case, the default value also becomes NULL unless overridden with a DEFAULT clause that specifies a different default value. DEFAULT NULL can be used to explicitly specify NULL as the default value. (For a [TIMESTAMP](#_bookmark109) column not declared with the NULL attribute, DEFAULT NULL is invalid.) If a [TIMESTAMP](#_bookmark109) column permits NULL values, assigning NULL sets it to NULL, not to the current timestamp.

The following table contains several [TIMESTAMP](#_bookmark109) columns that permit NULL values:

CREATE TABLE t

(

ts1 TIMESTAMP NULL DEFAULT NULL,

ts2 TIMESTAMP NULL DEFAULT 0,

ts3 TIMESTAMP NULL DEFAULT CURRENT\_TIMESTAMP

);

A [TIMESTAMP](#_bookmark109) column that permits NULL values does *not* take on the current timestamp at insert time except under one of the following conditions:

• Its default value is defined as CURRENT\_TIMESTAMP and no value is specified for the column

• CURRENT\_TIMESTAMP or any of its synonyms such as NOW() is explicitly inserted into the column

In other words, a [TIMESTAMP](#_bookmark109) column defined to permit NULL values auto-initializes only if its definition includes DEFAULT CURRENT\_TIMESTAMP:

CREATE TABLE t (ts TIMESTAMP NULL DEFAULT CURRENT\_TIMESTAMP);

If the [TIMESTAMP](#_bookmark109) column permits NULL values but its definition does not include DEFAULT CURRENT\_TIMESTAMP, you must explicitly insert a value corresponding to the current date and time. Suppose that tables t1 and t2 have these definitions:

CREATE TABLE t1 (ts TIMESTAMP NULL DEFAULT '0000-00-00 00:00:00');

CREATE TABLE t2 (ts TIMESTAMP NULL DEFAULT NULL);

To set the [TIMESTAMP](#_bookmark109) column in either table to the current timestamp at insert time, explicitly assign it that value. For example:

INSERT INTO t2 VALUES (CURRENT\_TIMESTAMP);

INSERT INTO t1 VALUES (NOW());

If the explicit\_defaults\_for\_timestamp system variable is enabled, [TIMESTAMP](#_bookmark109) columns permit NULL values only if declared with the NULL attribute. Also, [TIMESTAMP](#_bookmark109) columns do not permit assigning NULL to assign the current timestamp, whether declared with the NULL or NOT NULL attribute. To assign the current timestamp, set the column to CURRENT\_TIMESTAMP or a synonym such as NOW().

**11.2.6** **Fractional** **Seconds** **in** **Time** **Values**

MySQL has fractional seconds support for [TIME](#_bookmark110), [DATETIME](#_bookmark109), and [TIMESTAMP](#_bookmark109) values, with up to microseconds (6 digits) precision:

• To define a column that includes a fractional seconds part, use the syntax *type\_name* (*fsp*), where *type\_name* is [TIME](#_bookmark110), [DATETIME](#_bookmark109), or [TIMESTAMP](#_bookmark109), and *fsp* is the fractional seconds precision. For example:

CREATE TABLE t1 (t TIME(3), dt DATETIME(6));

The *fsp* value, if given, must be in the range 0 to 6. A value of 0 signifies that there is no fractional part. If omitted, the default precision is 0. (This differs from the standard SQL default of 6, for compatibility with previous MySQL versions.)

• Inserting a [TIME](#_bookmark110), [DATE](#_bookmark109), or [TIMESTAMP](#_bookmark109) value with a fractional seconds part into a column of the same type but having fewer fractional digits results in rounding. Consider a table created and populated as follows:

CREATE TABLE fractest( c1 TIME(2), c2 DATETIME(2), c3 TIMESTAMP(2) );

INSERT INTO fractest VALUES

('17:51:04.777', '2018-09-08 17:51:04.777', '2018-09-08 17:51:04.777');

The temporal values are inserted into the table with rounding:

mysql> **SELECT** **\*** **FROM** **fractest;**

+-------------+------------------------+------------------------+

| c1 | c2 | c3 |

+-------------+------------------------+------------------------+

| 17:51:04 .78 | 2018-09-08 17:51:04 .78 | 2018-09-08 17:51:04 .78 |

+-------------+------------------------+------------------------+

No warning or error is given when such rounding occurs. This behavior follows the SQL standard. To insert the values with truncation instead, enable the TIME\_TRUNCATE\_FRACTIONAL SQL mode: SET @@sql\_mode = sys.list\_add(@@sql\_mode, 'TIME\_TRUNCATE\_FRACTIONAL');

With that SQL mode enabled, the temporal values are inserted with truncation:

mysql> **SELECT** **\*** **FROM** **fractest;**

+-------------+------------------------+------------------------+

| c1 | c2 | c3 |

+-------------+------------------------+------------------------+

| 17:51:04 .77 | 2018-09-08 17:51:04 .77 | 2018-09-08 17:51:04 .77 |

+ +------------------------+------------------------+

• Functions that take temporal arguments accept values with fractional seconds. Return values from temporal functions include fractional seconds as appropriate. For example, NOW() with no argument returns the current date and time with no fractional part, but takes an optional argument from 0 to 6 to specify that the return value includes a fractional seconds part of that many digits.

• Syntax for temporal literals produces temporal values: DATE '*str*', TIME '*str*', and TIMESTAMP '*str*', and the ODBC-syntax equivalents. The resulting value includes a trailing fractional seconds part if specified. Previously, the temporal type keyword was ignored and these constructs produced the string value. See Standard SQL and ODBC Date and Time Literals

**11.2.7** **Conversion** **Between** **Date** **and** **Time** **Types**

To some extent, you can convert a value from one temporal type to another. However, there may be some alteration of the value or loss of information. In all cases, conversion between temporal types is subject to the range of valid values for the resulting type. For example, although [DATE](#_bookmark109), [DATETIME](#_bookmark109), and [TIMESTAMP](#_bookmark109) values all can be specified using the same set of formats, the types do not all have the same range of values. [TIMESTAMP](#_bookmark109) values cannot be earlier than 1970 UTC or later than '2038-01-19 03:14:07' UTC. This means that a date such as '1968-01-01', while valid as a [DATE](#_bookmark109) or [DATETIME](#_bookmark109) value, is not valid as a [TIMESTAMP](#_bookmark109) value and is converted to 0.

Conversion of [DATE](#_bookmark109) values:

• Conversion to a [DATETIME](#_bookmark109) or [TIMESTAMP](#_bookmark109) value adds a time part of '00:00:00' because the [DATE](#_bookmark109) value contains no time information.

• Conversion to a [TIME](#_bookmark110) value is not useful; the result is '00:00:00'. Conversion of [DATETIME](#_bookmark109) and [TIMESTAMP](#_bookmark109) values:

• Conversion to a [DATE](#_bookmark109) value takes fractional seconds into account and rounds the time part. For example, '1999-12-31 23:59:59.499' becomes '1999-12-31', whereas '1999-12-31 23:59:59.500' becomes '2000-01-01'.

• Conversion to a [TIME](#_bookmark110) value discards the date part because the [TIME](#_bookmark110) type contains no date information.

For conversion of [TIME](#_bookmark110) values to other temporal types, the value of CURRENT\_DATE() is used for the date part. The [TIME](#_bookmark110) is interpreted as elapsed time (not time of day) and added to the date. This means that the date part of the result differs from the current date if the time value is outside the range from '00:00:00' to '23:59:59'.

Suppose that the current date is '2012-01-01'. [TIME](#_bookmark110) values of '12:00:00', '24:00:00', and '-12:00:00', when converted to [DATETIME](#_bookmark109) or [TIMESTAMP](#_bookmark109) values, result in '2012-01-01 12:00:00', '2012-01-02 00:00:00', and '2011-12-31 12:00:00', respectively.

Conversion of [TIME](#_bookmark110) to [DATE](#_bookmark109) is similar but discards the time part from the result: '2012-01-01', '2012-01-02', and '2011-12-31', respectively.

Explicit conversion can be used to override implicit conversion. For example, in comparison of [DATE](#_bookmark109) and [DATETIME](#_bookmark109) values, the [DATE](#_bookmark109) value is coerced to the [DATETIME](#_bookmark109) type by adding a time part of '00:00:00'. To perform the comparison by ignoring the time part of the [DATETIME](#_bookmark109) value instead, use the CAST() function in the following way:

*date\_col* = CAST(*datetime\_col* AS DATE)

Conversion of [TIME](#_bookmark110) and [DATETIME](#_bookmark109) values to numeric form (for example, by adding +0) depends on whether the value contains a fractional seconds part. [TIME(*N*](#_bookmark110)) or [DATETIME(*N*](#_bookmark109)) is converted to integer when *N* is 0 (or omitted) and to a DECIMAL value with *N*decimal digits when *N* is greater than 0:

mysql> **SELECT** **CURTIME(),** **CURTIME()+0,** **CURTIME(3)+0;**

+-----------+-------------+--------------+

| CURTIME() | CURTIME()+0 | CURTIME(3)+0 |

+-----------+-------------+--------------+

| 09:28:00 | 92800 | 92800.887 |

+-----------+-------------+--------------+

mysql> **SELECT** **NOW(),** **NOW()+0,** **NOW(3)+0;**

+---------------------+----------------+--------------------+

| NOW() | NOW()+0 | NOW(3)+0 |

+---------------------+----------------+--------------------+

| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |

+---------------------+----------------+--------------------+

**11.2.8** **2-Digit** **Years** **in** **Dates**

Date values with 2-digit years are ambiguous because the century is unknown. Such values must be interpreted into 4-digit form because MySQL stores years internally using 4 digits.

For [DATETIME](#_bookmark109), [DATE](#_bookmark109), and [TIMESTAMP](#_bookmark109) types, MySQL interprets dates specified with ambiguous year values using these rules:

• Year values in the range 00-69 become 2000-2069.

• Year values in the range 70-99 become 1970-1999.

For YEAR, the rules are the same, with this exception: A numeric 00 inserted into YEAR results in 0000 rather than 2000. To specify zero for YEAR and have it be interpreted as 2000, specify it as a string '0' or '00'.

Remember that these rules are only heuristics that provide reasonable guesses as to what your data values mean. If the rules used by MySQL do not produce the values you require, you must provide unambiguous input containing 4-digit year values.

ORDER BY properly sorts [YEAR](#_bookmark111) values that have 2-digit years.

Some functions like MIN() and MAX() convert a [YEAR](#_bookmark111) to a number. This means that a value with a 2- digit year does not work properly with these functions. The fix in this case is to convert the [YEAR](#_bookmark111) to 4- digit year format.

**11.3** **String** **Data** **Types**

The string data types are [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), [BINARY](#_bookmark73), [VARBINARY](#_bookmark73), [BLOB](#_bookmark61), [TEXT](#_bookmark61), [ENUM](#_bookmark62), and [SET](#_bookmark63).

For information about storage requirements of the string data types, see [Section 11.7, “Data Type](#_bookmark132) [Storage Requirements”](#_bookmark132) .

For descriptions of functions that operate on string values, see Section 12.8, “String Functions and Operators” .

**11.3.1** **String** **Data** **Type** **Syntax**

The string data types are [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), [BINARY](#_bookmark73), [VARBINARY](#_bookmark73), [BLOB](#_bookmark61), [TEXT](#_bookmark61), [ENUM](#_bookmark62), and [SET](#_bookmark63).

In some cases, MySQL may change a string column to a type different from that given in a CREATE TABLE or ALTER TABLE statement. See Section 13.1.20.7, “Silent Column Specification Changes” .

For definitions of character string columns ([CHAR](#_bookmark60), [VARCHAR](#_bookmark60), and the [TEXT](#_bookmark61) types), MySQL interprets length specifications in character units. For definitions of binary string columns ([BINARY](#_bookmark73), [VARBINARY](#_bookmark73), and the [BLOB](#_bookmark61) types), MySQL interprets length specifications in byte units.

Column definitions for character string data types [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), the [TEXT](#_bookmark61) types, [ENUM](#_bookmark62), [SET](#_bookmark63), and any synonyms) can specify the column character set and collation:

• CHARACTER SET specifies the character set. If desired, a collation for the character set can be specified with the COLLATE attribute, along with any other attributes. For example:

|  |  |
| --- | --- |
| CREATE  (  c1  c2  ); | TABLE t  VARCHAR(20) CHARACTER SET utf8mb4,  TEXT CHARACTER SET latin1 COLLATE latin1\_general\_cs |

This table definition creates a column named c1 that has a character set of utf8mb4 with the default collation for that character set, and a column named c2 that has a character set of latin1 and a case-sensitive (\_cs) collation.

The rules for assigning the character set and collation when either or both of CHARACTER SET and the COLLATE attribute are missing are described in [Section 10.3.5, “Column Character Set and](#_bookmark11) [Collation”](#_bookmark11) .

CHARSET is a synonym for CHARACTER SET.

• Specifying the CHARACTER SET binary attribute for a character string data type causes the column to be created as the corresponding binary string data type: [CHAR](#_bookmark60) becomes [BINARY](#_bookmark73), [VARCHAR](#_bookmark60) becomes [VARBINARY](#_bookmark73), and [TEXT](#_bookmark61) becomes [BLOB](#_bookmark61). For the [ENUM](#_bookmark62) and [SET](#_bookmark63) data types, this does not occur; they are created as declared. Suppose that you specify a table using this definition:

CREATE TABLE t

(

c1 VARCHAR(10) CHARACTER SET binary,

c2 TEXT CHARACTER SET binary,

c3 ENUM('a','b','c') CHARACTER SET binary

);

The resulting table has this definition:

CREATE TABLE t

(



c1 VARBINARY(10),

c2 BLOB,

c3 ENUM('a','b','c') CHARACTER SET binary

);

• The BINARY attribute is a nonstandard MySQL extension that is shorthand for specifying the binary (\_bin) collation of the column character set (or of the table default character set if no column character set is specified). In this case, comparison and sorting are based on numeric character code values. Suppose that you specify a table using this definition:

CREATE TABLE t

(

c1 VARCHAR(10) CHARACTER SET latin1 BINARY,

c2 TEXT BINARY

) CHARACTER SET utf8mb4;

The resulting table has this definition:

CREATE TABLE t (

c1 VARCHAR(10) CHARACTER SET latin1 COLLATE latin1\_bin,

c2 TEXT CHARACTER SET utf8mb4 COLLATE utf8mb4\_bin

) CHARACTER SET utf8mb4;

In MySQL 8.0, this nonstandard use of the BINARY attribute is ambiguous because the utf8mb4 character set has multiple \_bin collations. As of MySQL 8.0.17, the BINARY attribute is deprecated and you should expect support for it to be removed in a future version of MySQL. Applications should be adjusted to use an explicit \_bin collation instead.

The use of BINARY to specify a data type or character set remains unchanged.

• The ASCII attribute is shorthand for CHARACTER SET latin1. Supported in older MySQL releases, ASCII is deprecated in MySQL 8.0.28 and later; use CHARACTER SET instead.

• The UNICODE attribute is shorthand for CHARACTER SET ucs2. Supported in older MySQL releases, UNICODE is deprecated in MySQL 8.0.28 and later; use CHARACTER SET instead.

Character column comparison and sorting are based on the collation assigned to the column. For the [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), [TEXT](#_bookmark61), [ENUM](#_bookmark62), and [SET](#_bookmark63) data types, you can declare a column with a binary (\_bin) collation or the BINARY attribute to cause comparison and sorting to use the underlying character code values rather than a lexical ordering.

For additional information about use of character sets in MySQL, see [Chapter 10, *Character* *Sets,*](#_bookmark1) [*Collations,* *Unicode*](#_bookmark1).

• [NATIONAL] CHAR[(*M*)] [CHARACTER SET *charset\_name*] [COLLATE *collation\_name*]

A fixed-length string that is always right-padded with spaces to the specified length when stored. *M* represents the column length in characters. The range of *M* is 0 to 255. If *M* is omitted, the length is 1.

**Note**

Trailing spaces are removed when [CHAR](#_bookmark60) values are retrieved unless the PAD\_CHAR\_TO\_FULL\_LENGTH SQL mode is enabled.

[CHAR](#_bookmark60) is shorthand for [CHARACTER](#_bookmark60). [NATIONAL CHAR](#_bookmark60) (or its equivalent short form, [NCHAR](#_bookmark60)) is the standard SQL way to define that a [CHAR](#_bookmark60) column should use some predefined character set. MySQL uses utf8mb3 as this predefined character set. [Section 10.3.7, “The National Character Set”](#_bookmark13) .

The [CHAR BYTE](#_bookmark73) data type is an alias for the [BINARY](#_bookmark73) data type. This is a compatibility feature.

MySQL permits you to create a column of type CHAR(0). This is useful primarily when you must be compliant with old applications that depend on the existence of a column but that do not actually use its value. CHAR(0) is also quite nice when you need a column that can take only two values: A



column that is defined as CHAR(0) NULL occupies only one bit and can take only the values NULL and '' (the empty string).

• [NATIONAL] VARCHAR(*M*) [CHARACTER SET *charset\_name*] [COLLATE *collation\_name*]

A variable-length string. *M* represents the maximum column length in characters. The range of *M* is 0 to 65,535. The effective maximum length of a [VARCHAR](#_bookmark60) is subject to the maximum row size (65,535 bytes, which is shared among all columns) and the character set used. For example, utf8mb3 characters can require up to three bytes per character, so a [VARCHAR](#_bookmark60) column that uses the utf8mb3 character set can be declared to be a maximum of 21,844 characters. See Section 8.4.7, “Limits on Table Column Count and Row Size” .

MySQL stores [VARCHAR](#_bookmark60) values as a 1-byte or 2-byte length prefix plus data. The length prefix indicates the number of bytes in the value. A [VARCHAR](#_bookmark60) column uses one length byte if values require no more than 255 bytes, two length bytes if values may require more than 255 bytes.

**Note**

MySQL follows the standard SQL specification, and does *not* remove trailing spaces from [VARCHAR](#_bookmark60) values.

[VARCHAR](#_bookmark60) is shorthand for [CHARACTER VARYING](#_bookmark60). [NATIONAL VARCHAR](#_bookmark60) is the standard SQL way to define that a [VARCHAR](#_bookmark60) column should use some predefined character set. MySQL uses utf8mb3 as this predefined character set. [Section 10.3.7, “The National Character Set”](#_bookmark13) . [NVARCHAR](#_bookmark60) is shorthand for [NATIONAL VARCHAR](#_bookmark60).

• [BINARY[(*M*)]](#_bookmark73)

The [BINARY](#_bookmark73) type is similar to the [CHAR](#_bookmark60) type, but stores binary byte strings rather than nonbinary character strings. An optional length *M* represents the column length in bytes. If omitted, *M*defaults to

1.

• [VARBINARY(*M*](#_bookmark73))

The [VARBINARY](#_bookmark73) type is similar to the [VARCHAR](#_bookmark60) type, but stores binary byte strings rather than nonbinary character strings. *M* represents the maximum column length in bytes.

• [TINYBLOB](#_bookmark61)

A [BLOB](#_bookmark61) column with a maximum length of 255 (28 − 1) bytes. Each [TINYBLOB](#_bookmark61) value is stored using a 1-byte length prefix that indicates the number of bytes in the value.

• [TINYTEXT [CHARACTER SET *charset\_name*] [COLLATE *collation\_name*]](#_bookmark61)

A [TEXT](#_bookmark61) column with a maximum length of 255 (28 − 1) characters. The effective maximum length is less if the value contains multibyte characters. Each [TINYTEXT](#_bookmark61) value is stored using a 1-byte length prefix that indicates the number of bytes in the value.

• [BLOB[(*M*)]](#_bookmark61)

A [BLOB](#_bookmark61) column with a maximum length of 65,535 (216 − 1) bytes. Each [BLOB](#_bookmark61) value is stored using a 2-byte length prefix that indicates the number of bytes in the value.

An optional length *M*can be given for this type. If this is done, MySQL creates the column as the smallest [BLOB](#_bookmark61) type large enough to hold values *M* bytes long.

• [TEXT[(*M*)] [CHARACTER SET *charset\_name*] [COLLATE *collation\_name*]](#_bookmark61)

A [TEXT](#_bookmark61) column with a maximum length of 65,535 (216 − 1) characters. The effective maximum length is less if the value contains multibyte characters. Each [TEXT](#_bookmark61) value is stored using a 2-byte length prefix that indicates the number of bytes in the value.

An optional length *M*can be given for this type. If this is done, MySQL creates the column as the smallest [TEXT](#_bookmark61) type large enough to hold values *M*characters long.

• [MEDIUMBLOB](#_bookmark61)

A [BLOB](#_bookmark61) column with a maximum length of 16,777,215 (224 − 1) bytes. Each [MEDIUMBLOB](#_bookmark61) value is stored using a 3-byte length prefix that indicates the number of bytes in the value.

• [MEDIUMTEXT [CHARACTER SET *charset\_name*] [COLLATE *collation\_name*]](#_bookmark61)

A [TEXT](#_bookmark61) column with a maximum length of 16,777,215 (224 − 1) characters. The effective maximum length is less if the value contains multibyte characters. Each [MEDIUMTEXT](#_bookmark61) value is stored using a 3- byte length prefix that indicates the number of bytes in the value.

• [LONGBLOB](#_bookmark61)

A [BLOB](#_bookmark61) column with a maximum length of 4,294,967,295 or 4GB (232 − 1) bytes. The effective maximum length of [LONGBLOB](#_bookmark61) columns depends on the configured maximum packet size in the client/server protocol and available memory. Each [LONGBLOB](#_bookmark61) value is stored using a 4-byte length prefix that indicates the number of bytes in the value.

• [LONGTEXT [CHARACTER SET *charset\_name*] [COLLATE *collation\_name*]](#_bookmark61)

A [TEXT](#_bookmark61) column with a maximum length of 4,294,967,295 or 4GB (232 − 1) characters. The effective maximum length is less if the value contains multibyte characters. The effective maximum length of [LONGTEXT](#_bookmark61) columns also depends on the configured maximum packet size in the client/server protocol and available memory. Each [LONGTEXT](#_bookmark61) value is stored using a 4-byte length prefix that indicates the number of bytes in the value.

• [ENUM('*value1*','*value2*',...) [CHARACTER SET *charset\_name*] [COLLATE](#_bookmark62) [*collation\_name*]](#_bookmark62)

An enumeration. A string object that can have only one value, chosen from the list of values '*value1*', '*value2*', ..., NULL or the special '' error value. [ENUM](#_bookmark62) values are represented internally as integers.

An [ENUM](#_bookmark62) column can have a maximum of 65,535 distinct elements.

The maximum supported length of an individual ENUM element is *M*<= 255 and (*M*x *w*) <= 1020, where M is the element literal length and *w* is the number of bytes required for the maximum-length character in the character set.

• [SET('*value1*','*value2*',...) [CHARACTER SET *charset\_name*] [COLLATE](#_bookmark63) [*collation\_name*]](#_bookmark63)

A set. A string object that can have zero or more values, each of which must be chosen from the list of values '*value1*', '*value2*', ... [SET](#_bookmark63) values are represented internally as integers.

A [SET](#_bookmark63) column can have a maximum of 64 distinct members.

The maximum supported length of an individual SET element is *M*<= 255 and (*M*x *w*) <= 1020, where M is the element literal length and *w* is the number of bytes required for the maximum-length character in the character set.

**11.3.2** **The** **CHAR** **and** **VARCHAR** **Types**

The CHAR and VARCHAR types are similar, but differ in the way they are stored and retrieved. They also differ in maximum length and in whether trailing spaces are retained.

The CHAR and VARCHAR types are declared with a length that indicates the maximum number of characters you want to store. For example, CHAR(30) can hold up to 30 characters.

The length of a CHAR column is fixed to the length that you declare when you create the table. The length can be any value from 0 to 255. When CHAR values are stored, they are right-padded with spaces to the specified length. When CHAR values are retrieved, trailing spaces are removed unless the PAD\_CHAR\_TO\_FULL\_LENGTH SQL mode is enabled.

Values in VARCHAR columns are variable-length strings. The length can be specified as a value from 0 to 65,535. The effective maximum length of a VARCHAR is subject to the maximum row size (65,535 bytes, which is shared among all columns) and the character set used. See Section 8.4.7, “Limits on Table Column Count and Row Size” .

In contrast to CHAR, VARCHAR values are stored as a 1-byte or 2-byte length prefix plus data. The length prefix indicates the number of bytes in the value. A column uses one length byte if values require no more than 255 bytes, two length bytes if values may require more than 255 bytes.

If strict SQL mode is not enabled and you assign a value to a CHAR or VARCHAR column that exceeds the column's maximum length, the value is truncated to fit and a warning is generated. For truncation of nonspace characters, you can cause an error to occur (rather than a warning) and suppress insertion of the value by using strict SQL mode. See Section 5.1.11, “Server SQL Modes” .

For VARCHAR columns, trailing spaces in excess of the column length are truncated prior to insertion and a warning is generated, regardless of the SQL mode in use. For CHAR columns, truncation of excess trailing spaces from inserted values is performed silently regardless of the SQL mode.

VARCHAR values are not padded when they are stored. Trailing spaces are retained when values are stored and retrieved, in conformance with standard SQL.

The following table illustrates the differences between CHAR and VARCHAR by showing the result of storing various string values into CHAR(4) and VARCHAR(4) columns (assuming that the column uses a single-byte character set such as latin1).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **CHAR(4)** | **Storage** **Required** | **VARCHAR(4)** | **Storage** **Required** |
| '' | ' ' | 4 bytes | '' | 1 byte |
| 'ab' | 'ab ' | 4 bytes | 'ab' | 3 bytes |
| 'abcd' | 'abcd' | 4 bytes | 'abcd' | 5 bytes |
| 'abcdefgh' | 'abcd' | 4 bytes | 'abcd' | 5 bytes |

The values shown as stored in the last row of the table apply *only* *when* *not* *using* *strict* *SQL* *mode*; if strict mode is enabled, values that exceed the column length are *not* *stored*, and an error results.

InnoDB encodes fixed-length fields greater than or equal to 768 bytes in length as variable-length fields, which can be stored off-page. For example, a CHAR(255) column can exceed 768 bytes if the maximum byte length of the character set is greater than 3, as it is with utf8mb4.

If a given value is stored into the CHAR(4) and VARCHAR(4) columns, the values retrieved from the columns are not always the same because trailing spaces are removed from CHAR columns upon retrieval. The following example illustrates this difference:

mysql> **CREATE** **TABLE** **vc** **(v** **VARCHAR(4),** **c** **CHAR(4));**

Query OK, 0 rows affected (0.01 sec)

mysql> **INSERT** **INTO** **vc** **VALUES** **('ab** **',** **'ab** **');**

Query OK, 1 row affected (0.00 sec)

mysql> **SELECT** **CONCAT('(',** **v,** **')'),** **CONCAT('(',** **c,** **')')** **FROM** **vc;**

+---------------------+---------------------+

| CONCAT('(', v, ')') | CONCAT('(', c, ')') |

+---------------------+---------------------+

|  |  |  |
| --- | --- | --- |
| | (ab ) | | (ab) | | |

+---------------------+---------------------+



1 row in set (0 .06 sec)

Values in CHAR, VARCHAR, and TEXT columns are sorted and compared according to the character set collation assigned to the column.

MySQL collations have a pad attribute of PAD SPACE, other than Unicode collations based on UCA 9.0.0 and higher, which have a pad attribute of NO PAD. (see [Section 10.10.1, “Unicode Character](#_bookmark39) [Sets”](#_bookmark39)).

To determine the pad attribute for a collation, use the INFORMATION\_SCHEMA COLLATIONS table, which has a PAD\_ATTRIBUTE column.

For nonbinary strings (CHAR, VARCHAR, and TEXT values), the string collation pad attribute determines treatment in comparisons of trailing spaces at the end of strings. NO PAD collations treat trailing spaces as significant in comparisons, like any other character. PAD SPACE collations treat trailing spaces as insignificant in comparisons; strings are compared without regard to trailing spaces. See [Trailing Space](#_bookmark79) [Handling in Comparisons](#_bookmark79). The server SQL mode has no effect on comparison behavior with respect to trailing spaces.

**Note**

For more information about MySQL character sets and collations, see [Chapter 10, *Character* *Sets,* *Collations,* *Unicode*](#_bookmark1). For additional information about storage requirements, see [Section 11.7, “Data Type Storage](#_bookmark132) [Requirements”](#_bookmark132) .

For those cases where trailing pad characters are stripped or comparisons ignore them, if a column has an index that requires unique values, inserting into the column values that differ only in number of trailing pad characters results in a duplicate-key error. For example, if a table contains 'a', an attempt to store 'a ' causes a duplicate-key error.

**11.3.3** **The** **BINARY** **and** **VARBINARY** **Types**

The BINARY and VARBINARY types are similar to [CHAR](#_bookmark60) and [VARCHAR](#_bookmark60), except that they store binary strings rather than nonbinary strings. That is, they store byte strings rather than character strings. This means they have the binary character set and collation, and comparison and sorting are based on the numeric values of the bytes in the values.

The permissible maximum length is the same for BINARY and VARBINARY as it is for [CHAR](#_bookmark60) and [VARCHAR](#_bookmark60), except that the length for BINARY and VARBINARY is measured in bytes rather than characters.

The BINARY and VARBINARY data types are distinct from the CHAR BINARY and VARCHAR BINARY data types. For the latter types, the BINARY attribute does not cause the column to be treated as a binary string column. Instead, it causes the binary (\_bin) collation for the column character set (or the table default character set if no column character set is specified) to be used, and the column itself stores nonbinary character strings rather than binary byte strings. For example, if the default character set is utf8mb4, CHAR(5) BINARY is treated as CHAR(5) CHARACTER SET utf8mb4 COLLATE utf8mb4\_bin. This differs from BINARY(5), which stores 5-byte binary strings that have the binary character set and collation. For information about the differences between the binary collation of the binary character set and the \_bin collations of nonbinary character sets, see [Section 10.8.5, “The](#_bookmark26) [binary Collation Compared to \_bin Collations”](#_bookmark26) .

If strict SQL mode is not enabled and you assign a value to a BINARY or VARBINARY column that exceeds the column's maximum length, the value is truncated to fit and a warning is generated. For cases of truncation, to cause an error to occur (rather than a warning) and suppress insertion of the value, use strict SQL mode. See Section 5.1.11, “Server SQL Modes” .

When BINARY values are stored, they are right-padded with the pad value to the specified length. The pad value is 0x00 (the zero byte). Values are right-padded with 0x00 for inserts, and no trailing



bytes are removed for retrievals. All bytes are significant in comparisons, including ORDER BY and DISTINCT operations. 0x00 and space differ in comparisons, with 0x00 sorting before space.

Example: For a BINARY(3) column, 'a ' becomes 'a \0' when inserted. 'a\0' becomes 'a \0\0' when inserted. Both inserted values remain unchanged for retrievals.

For VARBINARY, there is no padding for inserts and no bytes are stripped for retrievals. All bytes are significant in comparisons, including ORDER BY and DISTINCT operations. 0x00 and space differ in comparisons, with 0x00 sorting before space.

For those cases where trailing pad bytes are stripped or comparisons ignore them, if a column has an index that requires unique values, inserting values into the column that differ only in number of trailing pad bytes results in a duplicate-key error. For example, if a table contains 'a', an attempt to store 'a \0' causes a duplicate-key error.

You should consider the preceding padding and stripping characteristics carefully if you plan to use the BINARY data type for storing binary data and you require that the value retrieved be exactly the same as the value stored. The following example illustrates how 0x00-padding of BINARY values affects column value comparisons:

mysql> **CREATE** **TABLE** **t** **(c** **BINARY(3));**

Query OK, 0 rows affected (0.01 sec)

mysql> **INSERT** **INTO** **t** **SET** **c** **=** **'a';**

Query OK, 1 row affected (0.01 sec)

mysql> **SELECT** **HEX(c),** **c** **=** **'a',** **c** **=** **'a\0\0'** **from** **t;**

+--------+---------+-------------+

| HEX(c) | c = 'a' | c = 'a\0\0' |

+--------+---------+-------------+

| 610000 | 0 | 1 |

+--------+---------+-------------+

1 row in set (0.09 sec)

If the value retrieved must be the same as the value specified for storage with no padding, it might be preferable to use VARBINARY or one of the [BLOB](#_bookmark61) data types instead.

**Note**

Within the mysql client, binary strings display using hexadecimal notation, depending on the value of the --binary-as-hex. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client” .

**11.3.4** **The** **BLOB** **and** **TEXT** **Types**

A BLOB is a binary large object that can hold a variable amount of data. The four BLOB types are TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB. These differ only in the maximum length of the values they can hold. The four TEXT types are TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT. These correspond to the four BLOB types and have the same maximum lengths and storage requirements. See [Section 11.7, “Data Type Storage Requirements”](#_bookmark132) .

BLOB values are treated as binary strings (byte strings). They have the binary character set and collation, and comparison and sorting are based on the numeric values of the bytes in column values. TEXT values are treated as nonbinary strings (character strings). They have a character set other than binary, and values are sorted and compared based on the collation of the character set.

If strict SQL mode is not enabled and you assign a value to a BLOB or TEXT column that exceeds the column's maximum length, the value is truncated to fit and a warning is generated. For truncation of nonspace characters, you can cause an error to occur (rather than a warning) and suppress insertion of the value by using strict SQL mode. See Section 5.1.11, “Server SQL Modes” .

Truncation of excess trailing spaces from values to be inserted into [TEXT](#_bookmark61) columns always generates a warning, regardless of the SQL mode.

For TEXT and BLOB columns, there is no padding on insert and no bytes are stripped on select.

If a TEXT column is indexed, index entry comparisons are space-padded at the end. This means that, if the index requires unique values, duplicate-key errors occur for values that differ only in the number of trailing spaces. For example, if a table contains 'a', an attempt to store 'a ' causes a duplicate-key error. This is not true for BLOB columns.

In most respects, you can regard a BLOB column as a [VARBINARY](#_bookmark73) column that can be as large as you like. Similarly, you can regard a TEXT column as a [VARCHAR](#_bookmark60) column. BLOB and TEXT differ from [VARBINARY](#_bookmark73) and [VARCHAR](#_bookmark60) in the following ways:

• For indexes on BLOB and TEXT columns, you must specify an index prefix length. For [CHAR](#_bookmark60) and [VARCHAR](#_bookmark60), a prefix length is optional. See Section 8.3.5, “Column Indexes” .

• BLOB and TEXT columns cannot have DEFAULT values.

If you use the BINARY attribute with a TEXT data type, the column is assigned the binary (\_bin) collation of the column character set.

LONG and LONG VARCHAR map to the MEDIUMTEXT data type. This is a compatibility feature.

MySQL Connector/ODBC defines BLOB values as LONGVARBINARY and TEXT values as

LONGVARCHAR.

Because BLOB and TEXT values can be extremely long, you might encounter some constraints in using them:

• Only the first max\_sort\_length bytes of the column are used when sorting. The default value of max\_sort\_length is 1024. You can make more bytes significant in sorting or grouping by increasing the value of max\_sort\_length at server startup or runtime. Any client can change the value of its session max\_sort\_length variable:

mysql> **SET** **max\_sort\_length** **=** **2000;**

mysql> **SELECT** **id,** **comment** **FROM** **t**

-> **ORDER** **BY** **comment;**

• Instances of BLOB or TEXT columns in the result of a query that is processed using a temporary table causes the server to use a table on disk rather than in memory because the MEMORY storage engine does not support those data types (see Section 8.4.4, “Internal Temporary Table Use in MySQL”). Use of disk incurs a performance penalty, so include BLOB or TEXT columns in the query result only if they are really needed. For example, avoid using SELECT \*, which selects all columns.

• The maximum size of a BLOB or TEXT object is determined by its type, but the largest value you actually can transmit between the client and server is determined by the amount of available memory and the size of the communications buffers. You can change the message buffer size by changing the value of the max\_allowed\_packet variable, but you must do so for both the server and your client program. For example, both mysql and mysqldump enable you to change the client-side max\_allowed\_packet value. See Section 5.1.1, “Configuring the Server” , Section 4.5.1, “mysql — The MySQL Command-Line Client” , and Section 4.5.4, “mysqldump — A Database Backup Program” . You may also want to compare the packet sizes and the size of the data objects you are storing with the storage requirements, see [Section 11.7, “Data Type Storage Requirements”](#_bookmark132)

Each BLOB or TEXT value is represented internally by a separately allocated object. This is in contrast to all other data types, for which storage is allocated once per column when the table is opened.

In some cases, it may be desirable to store binary data such as media files in BLOB or TEXT columns. You may find MySQL's string handling functions useful for working with such data. See Section 12.8, “String Functions and Operators” . For security and other reasons, it is usually preferable to do so using application code rather than giving application users the FILE privilege. You can discuss specifics for various languages and platforms in the MySQL Forums (<http://forums.mysql.com/>).



**Note**

Within the mysql client, binary strings display using hexadecimal notation, depending on the value of the --binary-as-hex. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client” .

**11.3.5** **The** **ENUM** **Type**

An ENUM is a string object with a value chosen from a list of permitted values that are enumerated explicitly in the column specification at table creation time.

See [Section 11.3.1, “String Data Type Syntax”](#_bookmark117) for [ENUM](#_bookmark62) type syntax and length limits.

The [ENUM](#_bookmark62) type has these advantages:

• Compact data storage in situations where a column has a limited set of possible values. The strings you specify as input values are automatically encoded as numbers. See [Section 11.7, “Data Type](#_bookmark132) [Storage Requirements”](#_bookmark132) for storage requirements for the ENUM type.

• Readable queries and output. The numbers are translated back to the corresponding strings in query results.

and these potential issues to consider:

• If you make enumeration values that look like numbers, it is easy to mix up the literal values with their internal index numbers, as explained in [Enumeration Limitations](#_bookmark135).

• Using ENUM columns in ORDER BY clauses requires extra care, as explained in [Enumeration Sorting](#_bookmark136).

• [Creating and Using ENUM Columns](#_bookmark137)

• [Index Values for Enumeration Literals](#_bookmark138)

• [Handling of Enumeration Literals](#_bookmark139)

• [Empty or NULL Enumeration Values](#_bookmark140)

• [Enumeration Sorting](#_bookmark136)

• [Enumeration Limitations](#_bookmark135)

**Creating** **and** **Using** **ENUM** **Columns**

An enumeration value must be a quoted string literal. For example, you can create a table with an ENUM column like this:

CREATE TABLE shirts (

name VARCHAR(40),

size ENUM('x-small', 'small', 'medium', 'large', 'x-large')

);

INSERT INTO shirts (name, size) VALUES ('dress shirt','large'), ('t-shirt','medium'),

('polo shirt','small');

SELECT name, size FROM shirts WHERE size = 'medium';

+---------+--------+

| name | size |

+---------+--------+

| t-shirt | medium |

+---------+--------+

UPDATE shirts SET size = 'small' WHERE size = 'large';

COMMIT;

Inserting 1 million rows into this table with a value of 'medium' would require 1 million bytes of storage, as opposed to 6 million bytes if you stored the actual string 'medium' in a VARCHAR column.

**Index** **Values** **for** **Enumeration** **Literals**

Each enumeration value has an index:

• The elements listed in the column specification are assigned index numbers, beginning with 1.

• The index value of the empty string error value is 0. This means that you can use the following SELECT statement to find rows into which invalid ENUM values were assigned:

mysql> **SELECT** **\*** **FROM** ***tbl\_name*** **WHERE** ***col*=0;**

• The index of the NULL value is NULL.

• The term “index” here refers to a position within the list of enumeration values. It has nothing to do with table indexes.

For example, a column specified as ENUM('Mercury', 'Venus', 'Earth') can have any of the values shown here. The index of each value is also shown.

|  |  |
| --- | --- |
| **Value** | **Index** |
| NULL | NULL |
| '' | 0 |
| 'Mercury' | 1 |
| 'Venus' | 2 |
| 'Earth' | 3 |

An [ENUM](#_bookmark62) column can have a maximum of 65,535 distinct elements.

If you retrieve an ENUM value in a numeric context, the column value's index is returned. For example, you can retrieve numeric values from an ENUM column like this:

mysql> **SELECT** ***col*+0** **FROM** ***tbl\_name*;**

Functions such as SUM() or AVG() that expect a numeric argument cast the argument to a number if necessary. For ENUM values, the index number is used in the calculation.

**Handling** **of** **Enumeration** **Literals**

Trailing spaces are automatically deleted from ENUM member values in the table definition when a table is created.

When retrieved, values stored into an ENUM column are displayed using the lettercase that was used in the column definition. Note that ENUM columns can be assigned a character set and collation. For binary or case-sensitive collations, lettercase is taken into account when assigning values to the column.

If you store a number into an ENUM column, the number is treated as the index into the possible values, and the value stored is the enumeration member with that index. (However, this does *not* work with LOAD DATA, which treats all input as strings.) If the numeric value is quoted, it is still interpreted as an index if there is no matching string in the list of enumeration values. For these reasons, it is not advisable to define an ENUM column with enumeration values that look like numbers, because this can easily become confusing. For example, the following column has enumeration members with string values of '0', '1', and '2', but numeric index values of 1, 2, and 3:

numbers ENUM('0','1','2')

If you store 2, it is interpreted as an index value, and becomes '1' (the value with index 2). If you store '2', it matches an enumeration value, so it is stored as '2'. If you store '3', it does not match any enumeration value, so it is treated as an index and becomes '2' (the value with index 3).

mysql> **INSERT** **INTO** **t** **(numbers)** **VALUES(2),('2'),('3');**

mysql> **SELECT** **\*** **FROM** **t;**

+---------+

| numbers |

+---------+

|

|

|

+---------+

To determine all possible values for an ENUM column, use SHOW COLUMNS FROM *tbl\_name* LIKE '*enum\_col*' and parse the ENUM definition in the Type column of the output.

1

2

2

|

|

|

In the C API, ENUM values are returned as strings. For information about using result set metadata to distinguish them from other strings, see [C API Basic Data Structures](https://dev.mysql.com/doc/c-api/8.0/en/c-api-data-structures.html).

**Empty** **or** **NULL** **Enumeration** **Values**

An enumeration value can also be the empty string ( '') or NULL under certain circumstances:

• If you insert an invalid value into an ENUM (that is, a string not present in the list of permitted values), the empty string is inserted instead as a special error value. This string can be distinguished from a “normal” empty string by the fact that this string has the numeric value 0. See [Index Values for](#_bookmark138) [Enumeration Literals](#_bookmark138) for details about the numeric indexes for the enumeration values.

If strict SQL mode is enabled, attempts to insert invalid ENUM values result in an error.

• If an ENUM column is declared to permit NULL, the NULL value is a valid value for the column, and the default value is NULL. If an ENUM column is declared NOT NULL, its default value is the first element of the list of permitted values.

**Enumeration** **Sorting**

ENUM values are sorted based on their index numbers, which depend on the order in which the enumeration members were listed in the column specification. For example, 'b' sorts before 'a' for ENUM('b', 'a'). The empty string sorts before nonempty strings, and NULL values sort before all other enumeration values.

To prevent unexpected results when using the ORDER BY clause on an ENUM column, use one of these techniques:

• Specify the ENUM list in alphabetic order.

• Make sure that the column is sorted lexically rather than by index number by coding ORDER BY CAST(*col* AS CHAR) or ORDER BY CONCAT(*col*).

**Enumeration** **Limitations**

An enumeration value cannot be an expression, even one that evaluates to a string value.

For example, this CREATE TABLE statement does *not* work because the CONCAT function cannot be used to construct an enumeration value:

CREATE TABLE sizes (

size ENUM('small', CONCAT('med','ium'), 'large')

);

You also cannot employ a user variable as an enumeration value. This pair of statements do *not* work:

SET @mysize = 'medium';

CREATE TABLE sizes (

size ENUM('small', @mysize, 'large')

);

We strongly recommend that you do *not* use numbers as enumeration values, because it does not save on storage over the appropriate [TINYINT](#_bookmark101) or [SMALLINT](#_bookmark101) type, and it is easy to mix up the strings and the underlying number values (which might not be the same) if you quote the ENUM values incorrectly. If you do use a number as an enumeration value, always enclose it in quotation marks. If the quotation marks are omitted, the number is regarded as an index. See [Handling of Enumeration](#_bookmark139) [Literals](#_bookmark139) to see how even a quoted number could be mistakenly used as a numeric index value.

Duplicate values in the definition cause a warning, or an error if strict SQL mode is enabled.

**11.3.6** **The** **SET** **Type**

A SET is a string object that can have zero or more values, each of which must be chosen from a list of permitted values specified when the table is created. SET column values that consist of multiple set members are specified with members separated by commas ( ,). A consequence of this is that SET member values should not themselves contain commas.

For example, a column specified as SET('one', 'two') NOT NULL can have any of these values:

''

'one'

'two'

'one,two'

A [SET](#_bookmark63) column can have a maximum of 64 distinct members.

Duplicate values in the definition cause a warning, or an error if strict SQL mode is enabled.

Trailing spaces are automatically deleted from SET member values in the table definition when a table is created.

See [String Type Storage Requirements](#_bookmark82) for storage requirements for the [SET](#_bookmark63) type. See [Section 11.3.1, “String Data Type Syntax”](#_bookmark117) for [SET](#_bookmark63) type syntax and length limits.

When retrieved, values stored in a SET column are displayed using the lettercase that was used in the column definition. Note that SET columns can be assigned a character set and collation. For binary or case-sensitive collations, lettercase is taken into account when assigning values to the column.

MySQL stores SET values numerically, with the low-order bit of the stored value corresponding to the first set member. If you retrieve a SET value in a numeric context, the value retrieved has bits set corresponding to the set members that make up the column value. For example, you can retrieve numeric values from a SET column like this:

mysql> **SELECT** ***set\_col*+0** **FROM** ***tbl\_name*;**

If a number is stored into a SET column, the bits that are set in the binary representation of the number determine the set members in the column value. For a column specified as SET('a','b','c','d'), the members have the following decimal and binary values.

|  |  |  |
| --- | --- | --- |
| **SET** **Member** | **Decimal** **Value** | **Binary** **Value** |
| 'a' | 1 | 0001 |
| 'b' | 2 | 0010 |
| 'c' | 4 | 0100 |
| 'd' | 8 | 1000 |

If you assign a value of 9 to this column, that is 1001 in binary, so the first and fourth SET value members 'a' and 'd' are selected and the resulting value is 'a,d'.

For a value containing more than one SET element, it does not matter what order the elements are listed in when you insert the value. It also does not matter how many times a given element is listed in the value. When the value is retrieved later, each element in the value appears once, with elements

listed according to the order in which they were specified at table creation time. Suppose that a column is specified as SET('a','b','c','d'):

mysql> **CREATE** **TABLE** **myset** **(col** **SET('a',** **'b',** **'c',** **'d'));**

If you insert the values 'a,d', 'd,a', 'a,d,d', 'a,d,a', and 'd,a,d':

mysql> **INSERT** **INTO** **myset** **(col)** **VALUES**

-> ('a,d'), ('d,a'), ('a,d,a'), ('a,d,d'), ('d,a,d');

Query OK, 5 rows affected (0.01 sec)

Records: 5 Duplicates: 0 Warnings: [0](#_bookmark141)

Then all these values appear as 'a,d' when retrieved:

mysql> **SELECT** **col** **FROM** **myset;**

+------+

| col |

+------+

| a,d |

| a,d |

| a,d |

| a,d |

| a,d |

+------+

5 rows in set (0.04 sec)

If you set a SET column to an unsupported value, the value is ignored and a warning is issued:

mysql> **INSERT** **INTO** **myset** **(col)** **VALUES** **('a,d,d,s');**

Query OK, 1 row affected, 1 warning (0.03 sec)

mysql> **SHOW** **WARNINGS;**

+---------+------+------------------------------------------+

| Level | Code | Message |

+---------+------+------------------------------------------+

| Warning | 1265 | Data truncated for column 'col' at row 1 |

+---------+------+------------------------------------------+

1 row in set (0.04 sec)

mysql> **SELECT** **col** **FROM** **myset;**

+------+

| col |

+------+

| a,d |

| a,d |

| a,d |

| a,d |

| a,d |

| a,d |

+------+

6 rows in set (0.01 sec)

If strict SQL mode is enabled, attempts to insert invalid SET values result in an error. SET values are sorted numerically. NULL values sort before non-NULL SET values.

Functions such as SUM() or AVG() that expect a numeric argument cast the argument to a number if necessary. For SET values, the cast operation causes the numeric value to be used.

Normally, you search for SET values using the FIND\_IN\_SET() function or the LIKE operator:

mysql> **SELECT** **\*** **FROM** ***tbl\_name*** **WHERE** **FIND\_IN\_SET('*value*** **',*set\_col*)>0;**

mysql> **SELECT** **\*** **FROM** ***tbl\_name*** **WHERE** ***set\_col*** **LIKE** **'%*value*%';**

The first statement finds rows where *set\_col* contains the *value* set member. The second is similar, but not the same: It finds rows where *set\_col* contains *value* anywhere, even as a substring of another set member.

The following statements also are permitted:

mysql> **SELECT** **\*** **FROM** ***tbl\_name*** **WHERE** ***set\_col*** **&** **1;**

mysql> **SELECT** **\*** **FROM** ***tbl\_name*** **WHERE** ***set\_col*** **=** **'*val1*,*val2*';**

The first of these statements looks for values containing the first set member. The second looks for an exact match. Be careful with comparisons of the second type. Comparing set values to '*val1*,*val2*' returns different results than comparing values to '*val2*,*val1*'. You should specify the values in the same order they are listed in the column definition.

To determine all possible values for a SET column, use SHOW COLUMNS FROM *tbl\_name* LIKE *set\_col* and parse the SET definition in the Type column of the output.

In the C API, SET values are returned as strings. For information about using result set metadata to distinguish them from other strings, see [C API Basic Data Structures](https://dev.mysql.com/doc/c-api/8.0/en/c-api-data-structures.html).

**11.4** **Spatial** **Data** **Types**

The [Open Geospatial Consortium](http://www.opengeospatial.org) (OGC) is an international consortium of more than 250 companies, agencies, and universities participating in the development of publicly available conceptual solutions that can be useful with all kinds of applications that manage spatial data.

The Open Geospatial Consortium publishes the *OpenGIS®* *Implementation* *Standard* *for* *Geographic* *information* *-* *Simple* *feature* *access* *-* *Part* *2:* *SQL* *option*, a document that proposes several conceptual ways for extending an SQL RDBMS to support spatial data. This specification is available from the OGC website at <http://www.opengeospatial.org/standards/sfs>.

Following the OGC specification, MySQL implements spatial extensions as a subset of the **SQL** **with** **Geometry** **Types** environment. This term refers to an SQL environment that has been extended with a set of geometry types. A geometry-valued SQL column is implemented as a column that has a geometry type. The specification describes a set of SQL geometry types, as well as functions on those types to create and analyze geometry values.

MySQL spatial extensions enable the generation, storage, and analysis of geographic features:

• Data types for representing spatial values

• Functions for manipulating spatial values

• Spatial indexing for improved access times to spatial columns

The spatial data types and functions are available for MyISAM, InnoDB, NDB, and ARCHIVE tables. For indexing spatial columns, MyISAM and InnoDB support both SPATIAL and non-SPATIAL indexes. The other storage engines support non-SPATIAL indexes, as described in Section 13.1.15, “CREATE

INDEX Statement” .

A **geographic** **feature** is anything in the world that has a location. A feature can be:

• An entity. For example, a mountain, a pond, a city.

• A space. For example, town district, the tropics.

• A definable location. For example, a crossroad, as a particular place where two streets intersect.

Some documents use the term **geospatial** **feature** to refer to geographic features.

**Geometry** is another word that denotes a geographic feature. Originally the word **geometry** meant measurement of the earth. Another meaning comes from cartography, referring to the geometric features that cartographers use to map the world.

The discussion here considers these terms synonymous: **geographic** **feature**, **geospatial** **feature**, **feature**, or **geometry**. The term most commonly used is **geometry**, defined as *a* *point* *or* *an* *aggregate* *of* *points* *representing* *anything* *in* *the* *world* *that* *has* *a* *location*.

The following material covers these topics:

• The spatial data types implemented in MySQL model

• The basis of the spatial extensions in the OpenGIS geometry model

• Data formats for representing spatial data

• How to use spatial data in MySQL

• Use of indexing for spatial data

• MySQL differences from the OpenGIS specification

For information about functions that operate on spatial data, see Section 12.17, “Spatial Analysis

Functions” .

**Additional** **Resources**

These standards are important for the MySQL implementation of spatial operations:

• SQL/MM Part 3: Spatial.

• The [Open Geospatial Consortium](http://www.opengeospatial.org) publishes the *OpenGIS®* *Implementation* *Standard* *for* *Geographic* *information*, a document that proposes several conceptual ways for extending an SQL RDBMS to support spatial data. See in particular Simple Feature Access - Part 1: Common Architecture, and Simple Feature Access - Part 2: SQL Option. The Open Geospatial Consortium (OGC) maintains a website at [. .](http://www.opengeospatial.org/)http://wwwopengeospatialorg/. The specification is available there at [http://](http://www.opengeospatial.org/standards/sfs) [www.opengeospatial.org/standards/sfs](http://www.opengeospatial.org/standards/sfs). It contains additional information relevant to the material here.

• The grammar for [spatial reference system](#_bookmark123) (SRS) definitions is based on the grammar defined in *OpenGIS* *Implementation* *Specification:* *Coordinate* *Transformation* *Services*, Revision 1.00, OGC 01-009, January 12, 2001, Section 7.2. This specification is available at [http://](http://www.opengeospatial.org/standards/ct) [www.opengeospatial.org/standards/ct](http://www.opengeospatial.org/standards/ct). For differences from that specification in SRS definitions as implemented in MySQL, see Section 13.1.19, “CREATE SPATIAL REFERENCE SYSTEM

Statement” .

If you have questions or concerns about the use of the spatial extensions to MySQL, you can discuss them in the GIS forum: <https://forums.mysql.com/list.php?23>.

**11.4.1** **Spatial** **Data** **Types**

MySQL has spatial data types that correspond to OpenGIS classes. The basis for these types is described in [Section 11.4.2, “The OpenGIS Geometry Model”](#_bookmark120) .

Some spatial data types hold single geometry values:

• GEOMETRY

• POINT

• LINESTRING

• POLYGON

GEOMETRY can store geometry values of any type. The other single-value types (POINT, LINESTRING,

and POLYGON) restrict their values to a particular geometry type.

The other spatial data types hold collections of values:

• MULTIPOINT

• MULTILINESTRING

• MULTIPOLYGON

• GEOMETRYCOLLECTION

GEOMETRYCOLLECTION can store a collection of objects of any type. The other collection types (MULTIPOINT, MULTILINESTRING, and MULTIPOLYGON) restrict collection members to those having a particular geometry type.

Example: To create a table named geom that has a column named g that can store values of any geometry type, use this statement:

CREATE TABLE geom (g GEOMETRY);

Columns with a spatial data type can have an SRID attribute, to explicitly indicate the spatial reference system (SRS) for values stored in the column. For example:

CREATE TABLE geom (

p POINT SRID 0,

g GEOMETRY NOT NULL SRID 4326

);

SPATIAL indexes can be created on spatial columns if they are NOT NULL and have a specific SRID, so if you plan to index the column, declare it with the NOT NULL and SRID attributes:

CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326);

InnoDB tables permit SRID values for Cartesian and geographic SRSs. MyISAM tables permit SRID values for Cartesian SRSs.

The SRID attribute makes a spatial column SRID-restricted, which has these implications:

• The column can contain only values with the given SRID. Attempts to insert values with a different SRID produce an error.

• The optimizer can use SPATIAL indexes on the column. See Section 8.3.3, “SPATIAL Index Optimization” .

Spatial columns with no SRID attribute are not SRID-restricted and accept values with any SRID. However, the optimizer cannot use SPATIAL indexes on them until the column definition is modified to include an SRID attribute, which may require that the column contents first be modified so that all values have the same SRID.

For other examples showing how to use spatial data types in MySQL, see [Section 11.4.6, “Creating](#_bookmark124) [Spatial Columns”](#_bookmark124) . For information about spatial reference systems, see [Section 11.4.5, “Spatial](#_bookmark123) [Reference System Support”](#_bookmark123) .

**11.4.2** **The** **OpenGIS** **Geometry** **Model**

The set of geometry types proposed by OGC's **SQL** **with** **Geometry** **Types** environment is based on the **OpenGIS** **Geometry** **Model**. In this model, each geometric object has the following general properties:

• It is associated with a spatial reference system, which describes the coordinate space in which the object is defined.

• It belongs to some geometry class.

**11.4.2.1** **The** **Geometry** **Class** **Hierarchy**

The geometry classes define a hierarchy as follows:

• Geometry (noninstantiable)

• Point (instantiable)

• Curve (noninstantiable)

• LineString (instantiable)

• Line

• LinearRing

• Surface (noninstantiable)

• Polygon (instantiable)

• GeometryCollection (instantiable)

• MultiPoint (instantiable)

• MultiCurve (noninstantiable)

• MultiLineString (instantiable)

• MultiSurface (noninstantiable)

• MultiPolygon (instantiable)

It is not possible to create objects in noninstantiable classes. It is possible to create objects in instantiable classes. All classes have properties, and instantiable classes may also have assertions (rules that define valid class instances).

Geometry is the base class. It is an abstract class. The instantiable subclasses of Geometry are restricted to zero-, one-, and two-dimensional geometric objects that exist in two-dimensional coordinate space. All instantiable geometry classes are defined so that valid instances of a geometry class are topologically closed (that is, all defined geometries include their boundary).

The base Geometry class has subclasses for Point, Curve, Surface, and GeometryCollection:

• Point represents zero-dimensional objects.

• Curve represents one-dimensional objects, and has subclass LineString, with sub-subclasses Line and LinearRing.

• Surface is designed for two-dimensional objects and has subclass Polygon.

• GeometryCollection has specialized zero-, one-, and two-dimensional collection classes named MultiPoint, MultiLineString, and MultiPolygon for modeling geometries corresponding to collections of Points, LineStrings, and Polygons, respectively. MultiCurve and MultiSurface are introduced as abstract superclasses that generalize the collection interfaces to handle Curves and Surfaces.

Geometry, Curve, Surface, MultiCurve, and MultiSurface are defined as noninstantiable classes. They define a common set of methods for their subclasses and are included for extensibility.

Point, LineString, Polygon, GeometryCollection, MultiPoint, MultiLineString, and MultiPolygon are instantiable classes.

**11.4.2.2** **Geometry** **Class**

Geometry is the root class of the hierarchy. It is a noninstantiable class but has a number of properties, described in the following list, that are common to all geometry values created from any of the Geometry subclasses. Particular subclasses have their own specific properties, described later.

**Geometry** **Properties**

A geometry value has the following properties:

• Its **type**. Each geometry belongs to one of the instantiable classes in the hierarchy.

• Its **SRID**, or spatial reference identifier. This value identifies the geometry's associated spatial reference system that describes the coordinate space in which the geometry object is defined.

In MySQL, the SRID value is an integer associated with the geometry value. The maximum usable

32

SRID value is 2 − 1. If a larger value is given, only the lower 32 bits are used.

SRID 0 represents an infinite flat Cartesian plane with no units assigned to its axes. To ensure SRID 0 behavior, create geometry values using SRID 0. SRID 0 is the default for new geometry values if no SRID is specified.

For computations on multiple geometry values, all values must have the same SRID or an error occurs.

• Its **coordinates** in its spatial reference system, represented as double-precision (8-byte) numbers. All nonempty geometries include at least one pair of (X,Y) coordinates. Empty geometries contain no coordinates.

Coordinates are related to the SRID. For example, in different coordinate systems, the distance between two objects may differ even when objects have the same coordinates, because the distance on the **planar** coordinate system and the distance on the **geodetic** system (coordinates on the Earth's surface) are different things.

• Its **interior**, **boundary**, and **exterior**.

Every geometry occupies some position in space. The exterior of a geometry is all space not occupied by the geometry. The interior is the space occupied by the geometry. The boundary is the interface between the geometry's interior and exterior.

• Its **MBR** (minimum bounding rectangle), or envelope. This is the bounding geometry, formed by the minimum and maximum (X,Y) coordinates:

((MINX MINY, MAXX MINY, MAXX MAXY, MINX MAXY, MINX MINY))

• Whether the value is **simple** or **nonsimple**. Geometry values of types (LineString, MultiPoint, MultiLineString) are either simple or nonsimple. Each type determines its own assertions for being simple or nonsimple.

• Whether the value is **closed** or **not** **closed**. Geometry values of types (LineString, MultiString) are either closed or not closed. Each type determines its own assertions for being closed or not closed.

• Whether the value is **empty** or **nonempty** A geometry is empty if it does not have any points. Exterior, interior, and boundary of an empty geometry are not defined (that is, they are represented by a NULL value). An empty geometry is defined to be always simple and has an area of 0.

• Its **dimension**. A geometry can have a dimension of −1, 0, 1, or 2:

• − 1 for an empty geometry.

• 0 for a geometry with no length and no area.

• 1 for a geometry with nonzero length and zero area.

• 2 for a geometry with nonzero area.

Point objects have a dimension of zero. LineString objects have a dimension of 1. Polygon objects have a dimension of 2. The dimensions of MultiPoint, MultiLineString, and MultiPolygon objects are the same as the dimensions of the elements they consist of.

**11.4.2.3** **Point** **Class**

A Point is a geometry that represents a single location in coordinate space.

**Point** **Examples**

• Imagine a large-scale map of the world with many cities. A Point object could represent each city.

• On a city map, a Point object could represent a bus stop.

**Point** **Properties**

• X-coordinate value.

• Y-coordinate value.

• Point is defined as a zero-dimensional geometry.

• The boundary of a Point is the empty set.

**11.4.2.4** **Curve** **Class**

A Curve is a one-dimensional geometry, usually represented by a sequence of points. Particular subclasses of Curve define the type of interpolation between points. Curve is a noninstantiable class.

**Curve** **Properties**

• A Curve has the coordinates of its points.

• A Curve is defined as a one-dimensional geometry.

• A Curve is simple if it does not pass through the same point twice, with the exception that a curve can still be simple if the start and end points are the same.

• A Curve is closed if its start point is equal to its endpoint.

• The boundary of a closed Curve is empty.

• The boundary of a nonclosed Curve consists of its two endpoints.

• A Curve that is simple and closed is a LinearRing.

**11.4.2.5** **LineString** **Class**

A LineString is a Curve with linear interpolation between points.

**LineString** **Examples**

• On a world map, LineString objects could represent rivers.

• In a city map, LineString objects could represent streets.

**LineString** **Properties**

• A LineString has coordinates of segments, defined by each consecutive pair of points.

• A LineString is a Line if it consists of exactly two points.

• A LineString is a LinearRing if it is both closed and simple.

**11.4.2.6** **Surface** **Class**

A Surface is a two-dimensional geometry. It is a noninstantiable class. Its only instantiable subclass is

Polygon.

**Surface** **Properties**

• A Surface is defined as a two-dimensional geometry.

• The OpenGIS specification defines a simple Surface as a geometry that consists of a single “patch” that is associated with a single exterior boundary and zero or more interior boundaries.

• The boundary of a simple Surface is the set of closed curves corresponding to its exterior and interior boundaries.

**11.4.2.7** **Polygon** **Class**

A Polygon is a planar Surface representing a multisided geometry. It is defined by a single exterior boundary and zero or more interior boundaries, where each interior boundary defines a hole in the

Polygon.

**Polygon** **Examples**

• On a region map, Polygon objects could represent forests, districts, and so on.

**Polygon** **Assertions**

• The boundary of a Polygon consists of a set of LinearRing objects (that is, LineString objects that are both simple and closed) that make up its exterior and interior boundaries.

• A Polygon has no rings that cross. The rings in the boundary of a Polygon may intersect at a Point, but only as a tangent.

• A Polygon has no lines, spikes, or punctures.

• A Polygon has an interior that is a connected point set.

• A Polygon may have holes. The exterior of a Polygon with holes is not connected. Each hole defines a connected component of the exterior.

The preceding assertions make a Polygon a simple geometry.

**11.4.2.8** **GeometryCollection** **Class**

A GeomCollection is a geometry that is a collection of zero or more geometries of any class.

GeomCollection and GeometryCollection are synonymous, with GeomCollection the preferred type name.

All the elements in a geometry collection must be in the same spatial reference system (that is, in the same coordinate system). There are no other constraints on the elements of a geometry collection, although the subclasses of GeomCollection described in the following sections may restrict membership. Restrictions may be based on:

• Element type (for example, a MultiPoint may contain only Point elements)

• Dimension

• Constraints on the degree of spatial overlap between elements

**11.4.2.9** **MultiPoint** **Class**

A MultiPoint is a geometry collection composed of Point elements. The points are not connected or ordered in any way.

**MultiPoint** **Examples**

• On a world map, a MultiPoint could represent a chain of small islands.

• On a city map, a MultiPoint could represent the outlets for a ticket office.

**MultiPoint** **Properties**

• A MultiPoint is a zero-dimensional geometry.

• A MultiPoint is simple if no two of its Point values are equal (have identical coordinate values).

• The boundary of a MultiPoint is the empty set.

**11.4.2.10** **MultiCurve** **Class**

A MultiCurve is a geometry collection composed of Curve elements. MultiCurve is a noninstantiable class.

**MultiCurve** **Properties**

• A MultiCurve is a one-dimensional geometry.

• A MultiCurve is simple if and only if all of its elements are simple; the only intersections between any two elements occur at points that are on the boundaries of both elements.

• A MultiCurve boundary is obtained by applying the “mod 2 union rule” (also known as the “odd- even rule”): A point is in the boundary of a MultiCurve if it is in the boundaries of an odd number of

Curve elements.

• A MultiCurve is closed if all of its elements are closed.

• The boundary of a closed MultiCurve is always empty.

**11.4.2.11** **MultiLineString** **Class**

A MultiLineString is a MultiCurve geometry collection composed of LineString elements.

**MultiLineString** **Examples**

• On a region map, a MultiLineString could represent a river system or a highway system.

**11.4.2.12** **MultiSurface** **Class**

A MultiSurface is a geometry collection composed of surface elements. MultiSurface is a noninstantiable class. Its only instantiable subclass is MultiPolygon.

**MultiSurface** **Assertions**

• Surfaces within a MultiSurface have no interiors that intersect.

• Surfaces within a MultiSurface have boundaries that intersect at most at a finite number of points.

**11.4.2.13** **MultiPolygon** **Class**

A MultiPolygon is a MultiSurface object composed of Polygon elements.

**MultiPolygon** **Examples**

• On a region map, a MultiPolygon could represent a system of lakes.

**MultiPolygon** **Assertions**

• A MultiPolygon has no two Polygon elements with interiors that intersect.

• A MultiPolygon has no two Polygon elements that cross (crossing is also forbidden by the previous assertion), or that touch at an infinite number of points.

• A MultiPolygon may not have cut lines, spikes, or punctures. A MultiPolygon is a regular, closed point set.

• A MultiPolygon that has more than one Polygon has an interior that is not connected. The number of connected components of the interior of a MultiPolygon is equal to the number of Polygon values in the MultiPolygon.

**MultiPolygon** **Properties**

• A MultiPolygon is a two-dimensional geometry.

• A MultiPolygon boundary is a set of closed curves (LineString values) corresponding to the boundaries of its Polygon elements.

• Each Curve in the boundary of the MultiPolygon is in the boundary of exactly one Polygon element.

• Every Curve in the boundary of an Polygon element is in the boundary of the MultiPolygon.

**11.4.3** **Supported** **Spatial** **Data** **Formats**

Two standard spatial data formats are used to represent geometry objects in queries:

• Well-Known Text (WKT) format

• Well-Known Binary (WKB) format

Internally, MySQL stores geometry values in a format that is not identical to either WKT or WKB format. (Internal format is like WKB but with an initial 4 bytes to indicate the SRID.)

There are functions available to convert between different data formats; see Section 12.17.6, “Geometry Format Conversion Functions” .

The following sections describe the spatial data formats MySQL uses:

• [Well-Known Text (WKT) Format](#_bookmark142)

• [Well-Known Binary (WKB) Format](#_bookmark143)

• [Internal Geometry Storage Format](#_bookmark144)

**Well-Known** **Text** **(WKT)** **Format**

The Well-Known Text (WKT) representation of geometry values is designed for exchanging geometry data in ASCII form. The OpenGIS specification provides a Backus-Naur grammar that specifies the formal production rules for writing WKT values (see [Section 11.4, “Spatial Data Types”](#_bookmark118)).

Examples of WKT representations of geometry objects:

• A Point:

POINT(15 20)

The point coordinates are specified with no separating comma. This differs from the syntax for the SQL Point() function, which requires a comma between the coordinates. Take care to use the syntax appropriate to the context of a given spatial operation. For example, the following statements both use ST\_X() to extract the X-coordinate from a Point object. The first produces the object directly using the Point() function. The second uses a WKT representation converted to a Point with ST\_GeomFromText().

mysql> **SELECT** **ST\_X(Point(15,** **20));**

+---------------------+

| ST\_X(POINT(15, 20)) |

+---------------------+

| 15 |

+---------------------+

mysql> **SELECT** **ST\_X(ST\_GeomFromText('POINT(15** **20)'));**

+---------------------------------------+

| ST\_X(ST\_GeomFromText('POINT(15 20)')) |

+---------------------------------------+

| 15 |

+ +

• A LineString with four points:

LINESTRING(0 0, 10 10, 20 25, 50 60)

The point coordinate pairs are separated by commas.

• A Polygon with one exterior ring and one interior ring:

POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))

• A MultiPoint with three Point values:

MULTIPOINT(0 0, 20 20, 60 60)

Spatial functions such as ST\_MPointFromText() and ST\_GeomFromText() that accept WKT-format representations of MultiPoint values permit individual points within values to be surrounded by parentheses. For example, both of the following function calls are valid:

ST\_MPointFromText('MULTIPOINT (1 1, 2 2, 3 3)')

ST\_MPointFromText('MULTIPOINT ((1 1), (2 2), (3 3))')

• A MultiLineString with two LineString values:

MULTILINESTRING((10 10, 20 20), (15 15, 30 15))

• A MultiPolygon with two Polygon values:

MULTIPOLYGON(((0 0,10 0,10 10,0 10,0 0)),((5 5,7 5,7 7,5 7, 5 5)))

• A GeometryCollection consisting of two Point values and one LineString:

GEOMETRYCOLLECTION(POINT(10 10), POINT(30 30), LINESTRING(15 15, 20 20))

**Well-Known** **Binary** **(WKB)** **Format**

The Well-Known Binary (WKB) representation of geometric values is used for exchanging geometry data as binary streams represented by [BLOB](#_bookmark61) values containing geometric WKB information. This format is defined by the OpenGIS specification (see [Section 11.4, “Spatial Data Types”](#_bookmark118)). It is also defined in the ISO *SQL/MM* *Part* *3:* *Spatial* standard.

WKB uses 1-byte unsigned integers, 4-byte unsigned integers, and 8-byte double-precision numbers (IEEE 754 format). A byte is eight bits.

For example, a WKB value that corresponds to POINT(1 -1) consists of this sequence of 21 bytes, each represented by two hexadecimal digits:

0101000000000000000000F03F000000000000F0BF

The sequence consists of the components shown in the following table.

**Table** **11.2** **WKB** **Components** **Example**

|  |  |  |
| --- | --- | --- |
| **Component** | **Size** | **Value** |
| Byte order | 1 byte | 01 |

|  |  |  |
| --- | --- | --- |
| **Component** | **Size** | **Value** |
| WKB type | 4 bytes | 01000000 |
| X coordinate | 8 bytes | 000000000000F03F |
| Y coordinate | 8 bytes | 000000000000F0BF |

Component representation is as follows:

• The byte order indicator is either 1 or 0 to signify little-endian or big-endian storage. The little-endian and big-endian byte orders are also known as Network Data Representation (NDR) and External Data Representation (XDR), respectively.

• The WKB type is a code that indicates the geometry type. MySQL uses values from 1 through 7 to indicate Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection.

• A Point value has X and Y coordinates, each represented as a double-precision value.

WKB values for more complex geometry values have more complex data structures, as detailed in the OpenGIS specification.

**Internal** **Geometry** **Storage** **Format**

MySQL stores geometry values using 4 bytes to indicate the SRID followed by the WKB representation

of the value. For a description of WKB format, see [Well-Known Binary (WKB) Format](#_bookmark143). For the WKB part, these MySQL-specific considerations apply:

• The byte-order indicator byte is 1 because MySQL stores geometries as little-endian values.

• MySQL supports geometry types of Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. Other geometry types are not supported.

• Only GeometryCollection can be empty. Such a value is stored with 0 elements.

• Polygon rings can be specified both clockwise and counterclockwise. MySQL flips the rings automatically when reading data.

Cartesian coordinates are stored in the length unit of the spatial reference system, with X values in the X coordinates and Y values in the Y coordinates. Axis directions are those specified by the spatial reference system.

Geographic coordinates are stored in the angle unit of the spatial reference system, with longitudes in the X coordinates and latitudes in the Y coordinates. Axis directions and the meridian are those specified by the spatial reference system.

The LENGTH() function returns the space in bytes required for value storage. Example:

mysql> **SET** **@g** **=** **ST\_GeomFromText('POINT(1** **-1)');**

mysql> **SELECT** **LENGTH(@g);**

+------------+

| LENGTH(@g) |

+------------+

| 25 |

+------------+

mysql> **SELECT** **HEX(@g);**

+----------------------------------------------------+

| HEX(@g) |

+----------------------------------------------------+

| 000000000101000000000000000000F03F000000000000F0BF |

+----------------------------------------------------+

The value length is 25 bytes, made up of these components (as can be seen from the hexadecimal value):

• 4 bytes for integer SRID (0)

• 1 byte for integer byte order (1 = little-endian)

• 4 bytes for integer type information (1 = Point)

• 8 bytes for double-precision X coordinate (1)

• 8 bytes for double-precision Y coordinate (−1)

**11.4.4** **Geometry** **Well-Formedness** **and** **Validity**

For geometry values, MySQL distinguishes between the concepts of syntactically well-formed and geometrically valid.

A geometry is syntactically well-formed if it satisfies conditions such as those in this (nonexhaustive) list:

• Linestrings have at least two points

• Polygons have at least one ring

• Polygon rings are closed (first and last points the same)

• Polygon rings have at least 4 points (minimum polygon is a triangle with first and last points the same)

• Collections are not empty (except GeometryCollection)

A geometry is geometrically valid if it is syntactically well-formed and satisfies conditions such as those in this (nonexhaustive) list:

• Polygons are not self-intersecting

• Polygon interior rings are inside the exterior ring

• Multipolygons do not have overlapping polygons

Spatial functions fail if a geometry is not syntactically well-formed. Spatial import functions that parse WKT or WKB values raise an error for attempts to create a geometry that is not syntactically well- formed. Syntactic well-formedness is also checked for attempts to store geometries into tables.

It is permitted to insert, select, and update geometrically invalid geometries, but they must be syntactically well-formed. Due to the computational expense, MySQL does not check explicitly for geometric validity. Spatial computations may detect some cases of invalid geometries and raise an error, but they may also return an undefined result without detecting the invalidity. Applications that require geometrically-valid geometries should check them using the ST\_IsValid() function.

**11.4.5** **Spatial** **Reference** **System** **Support**

A spatial reference system (SRS) for spatial data is a coordinate-based system for geographic locations.

There are different types of spatial reference systems:

• A projected SRS is a projection of a globe onto a flat surface; that is, a flat map. For example, a light bulb inside a globe that shines on a paper cylinder surrounding the globe projects a map onto the paper. The result is georeferenced: Each point maps to a place on the globe. The coordinate system on that plane is Cartesian using a length unit (meters, feet, and so forth), rather than degrees of longitude and latitude.

The globes in this case are ellipsoids; that is, flattened spheres. Earth is a bit shorter in its North- South axis than its East-West axis, so a slightly flattened sphere is more correct, but perfect spheres permit faster calculations.

• A geographic SRS is a nonprojected SRS representing longitude-latitude (or latitude-longitude) coordinates on an ellipsoid, in any angular unit.

• The SRS denoted in MySQL by SRID 0 represents an infinite flat Cartesian plane with no units assigned to its axes. Unlike projected SRSs, it is not georeferenced and it does not necessarily represent Earth. It is an abstract plane that can be used for anything. SRID 0 is the default SRID for spatial data in MySQL.

MySQL maintains information about available spatial reference systems for spatial data in the data dictionary mysql.st\_spatial\_reference\_systems table, which can store entries for projected and geographic SRSs. This data dictionary table is invisible, but SRS entry contents are available through the INFORMATION\_SCHEMA ST\_SPATIAL\_REFERENCE\_SYSTEMS table, implemented as a view on mysql.st\_spatial\_reference\_systems (see Section 26.3.36, “The INFORMATION\_SCHEMA ST\_SPATIAL\_REFERENCE\_SYSTEMS Table” ).

The following example shows what an SRS entry looks like:

mysql> **SELECT** **\***

**FROM** **INFORMATION\_SCHEMA.ST\_SPATIAL\_REFERENCE\_SYSTEMS**

**WHERE** **SRS\_ID** **=** **4326\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SRS\_NAME: WGS 84

SRS\_ID: 4326

ORGANIZATION: EPSG

ORGANIZATION\_COORDSYS\_ID: 4326

DEFINITION: GEOGCS["WGS 84",DATUM["World Geodetic System 1984",

SPHEROID["WGS 84",6378137,298.257223563,

AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],

PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],

UNIT["degree",0.017453292519943278,

AUTHORITY["EPSG","9122"]],

AXIS["Lat",NORTH],AXIS["Long",EAST],

AUTHORITY["EPSG","4326"]]

DESCRIPTION:

This entry describes the SRS used for GPS systems. It has the name (SRS\_NAME) WGS 84 and the ID (SRS\_ID) 4326, which is the ID used by the [European Petroleum Survey Group](http://epsg.org) (EPSG).

SRS definitions in the DEFINITION column are WKT values, represented as specified in the [Open](http://www.opengeospatial.org) [Geospatial Consortium](http://www.opengeospatial.org) document [OGC 12-063r5](http://docs.opengeospatial.org/is/12-063r5/12-063r5.html).

SRS\_ID values represent the same kind of values as the SRID of geometry values or passed as the SRID argument to spatial functions. SRID 0 (the unitless Cartesian plane) is special. It is always a legal spatial reference system ID and can be used in any computations on spatial data that depend on SRID values.

For computations on multiple geometry values, all values must have the same SRID or an error occurs.

SRS definition parsing occurs on demand when definitions are needed by GIS functions. Parsed definitions are stored in the data dictionary cache to enable reuse and avoid incurring parsing overhead for every statement that needs SRS information.

To enable manipulation of SRS entries stored in the data dictionary, MySQL provides these SQL statements:

• CREATE SPATIAL REFERENCE SYSTEM: See Section 13.1.19, “CREATE SPATIAL REFERENCE SYSTEM Statement” . The description for this statement includes additional information about SRS components.

• DROP SPATIAL REFERENCE SYSTEM: See Section 13.1.31, “DROP SPATIAL REFERENCE SYSTEM Statement” .

**11.4.6** **Creating** **Spatial** **Columns**

MySQL provides a standard way of creating spatial columns for geometry types, for example, with CREATE TABLE or ALTER TABLE. Spatial columns are supported for MyISAM, InnoDB, NDB, and ARCHIVE tables. See also the notes about spatial indexes under [Section 11.4.10, “Creating Spatial](#_bookmark128) [Indexes”](#_bookmark128) .

Columns with a spatial data type can have an SRID attribute, to explicitly indicate the spatial reference system (SRS) for values stored in the column. For implications of an SRID-restricted column, see [Section 11.4.1, “Spatial Data Types”](#_bookmark119) .

• Use the CREATE TABLE statement to create a table with a spatial column:

CREATE TABLE geom (g GEOMETRY);

• Use the ALTER TABLE statement to add or drop a spatial column to or from an existing table:

ALTER TABLE geom ADD pt POINT;

ALTER TABLE geom DROP pt;

**11.4.7** **Populating** **Spatial** **Columns**

After you have created spatial columns, you can populate them with spatial data.

Values should be stored in internal geometry format, but you can convert them to that format

from either Well-Known Text (WKT) or Well-Known Binary (WKB) format. The following examples demonstrate how to insert geometry values into a table by converting WKT values to internal geometry format:

• Perform the conversion directly in the INSERT statement: INSERT INTO geom VALUES (ST\_GeomFromText('POINT(1 1)'));

SET @g = 'POINT(1 1)';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

• Perform the conversion prior to the INSERT:

SET @g = ST\_GeomFromText('POINT(1 1)');

INSERT INTO geom VALUES (@g);

The following examples insert more complex geometries into the table:

SET @g = 'LINESTRING(0 0,1 1,2 2)';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

SET @g =

'GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';

INSERT INTO geom VALUES (ST\_GeomFromText(@g));

The preceding examples use ST\_GeomFromText() to create geometry values. You can also use type-specific functions:

SET @g = 'POINT(1 1)';

INSERT INTO geom VALUES (ST\_PointFromText(@g));

SET @g = 'LINESTRING(0 0,1 1,2 2)';

INSERT INTO geom VALUES (ST\_LineStringFromText(@g));

SET @g = 'POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))';

INSERT INTO geom VALUES (ST\_PolygonFromText(@g));

SET @g =

'GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,1 1,2 2,3 3,4 4))';

INSERT INTO geom VALUES (ST\_GeomCollFromText(@g));

A client application program that wants to use WKB representations of geometry values is responsible for sending correctly formed WKB in queries to the server. There are several ways to satisfy this requirement. For example:

• Inserting a POINT(1 1) value with hex literal syntax:

INSERT INTO geom VALUES

(ST\_GeomFromWKB(X'0101000000000000000000F03F000000000000F03F'));

• An ODBC application can send a WKB representation, binding it to a placeholder using an argument of [BLOB](#_bookmark61) type:

INSERT INTO geom VALUES (ST\_GeomFromWKB(?))

Other programming interfaces may support a similar placeholder mechanism.

• In a C program, you can escape a binary value using [mysql\_real\_escape\_string\_quote()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-escape-string-quote.html) and include the result in a query string that is sent to the server. See [mysql\_real\_escape\_string\_quote()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-escape-string-quote.html).

**11.4.8** **Fetching** **Spatial** **Data**

Geometry values stored in a table can be fetched in internal format. You can also convert them to WKT or WKB format.

• Fetching spatial data in internal format:

Fetching geometry values using internal format can be useful in table-to-table transfers: CREATE TABLE geom2 (g GEOMETRY) SELECT g FROM geom;

• Fetching spatial data in WKT format:

The ST\_AsText() function converts a geometry from internal format to a WKT string. SELECT ST\_AsText(g) FROM geom;

• Fetching spatial data in WKB format:

The ST\_AsBinary() function converts a geometry from internal format to a [BLOB](#_bookmark61) containing the WKB value.

SELECT ST\_AsBinary(g) FROM geom;

**11.4.9** **Optimizing** **Spatial** **Analysis**

For MyISAM and InnoDB tables, search operations in columns containing spatial data can be optimized using SPATIAL indexes. The most typical operations are:

• Point queries that search for all objects that contain a given point

• Region queries that search for all objects that overlap a given region

MySQL uses **R-Trees** **with** **quadratic** **splitting** for SPATIAL indexes on spatial columns. A SPATIAL index is built using the minimum bounding rectangle (MBR) of a geometry. For most geometries, the MBR is a minimum rectangle that surrounds the geometries. For a horizontal or a vertical linestring, the

MBR is a rectangle degenerated into the linestring. For a point, the MBR is a rectangle degenerated into the point.

It is also possible to create normal indexes on spatial columns. In a non-SPATIAL index, you must declare a prefix for any spatial column except for POINT columns.

MyISAM and InnoDB support both SPATIAL and non-SPATIAL indexes. Other storage engines support non-SPATIAL indexes, as described in Section 13.1.15, “CREATE INDEX Statement” .

**11.4.10** **Creating** **Spatial** **Indexes**

For InnoDB and MyISAM tables, MySQL can create spatial indexes using syntax similar to that for creating regular indexes, but using the SPATIAL keyword. Columns in spatial indexes must be declared NOT NULL. The following examples demonstrate how to create spatial indexes:

• With CREATE TABLE: CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326, SPATIAL INDEX(g));

• With ALTER TABLE:

CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326);

ALTER TABLE geom ADD SPATIAL INDEX(g);

• With CREATE INDEX:

CREATE TABLE geom (g GEOMETRY NOT NULL SRID 4326);

CREATE SPATIAL INDEX g ON geom (g);

SPATIAL INDEX creates an R-tree index. For storage engines that support nonspatial indexing of spatial columns, the engine creates a B-tree index. A B-tree index on spatial values is useful for exact- value lookups, but not for range scans.

The optimizer can use spatial indexes defined on columns that are SRID-restricted. For more information, see [Section 11.4.1, “Spatial Data Types”](#_bookmark119) , and Section 8.3.3, “SPATIAL Index Optimization” .

For more information on indexing spatial columns, see Section 13.1.15, “CREATE INDEX Statement” . To drop spatial indexes, use ALTER TABLE or DROP INDEX:

• With ALTER TABLE: ALTER TABLE geom DROP INDEX g;

• With DROP INDEX: DROP INDEX g ON geom;

Example: Suppose that a table geom contains more than 32,000 geometries, which are stored in the column g of type GEOMETRY. The table also has an AUTO\_INCREMENT column fid for storing object ID values.

mysql> **DESCRIBE** **geom;**

+-------+----------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+-------+----------+------+-----+---------+----------------+

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | fid  g | | int(11) |  | geometry | | |  | | PRI | | NULL  | | |  | | auto\_increment |  | |

+-------+----------+------+-----+---------+----------------+

2 rows in set (0.00 sec)

mysql> **SELECT** **COUNT(\*)** **FROM** **geom;**

+----------+

| count(\*) |

+----------+

| 32376 |

+----------+

1 row in set (0.00 sec)

To add a spatial index on the column g, use this statement:

mysql> **ALTER** **TABLE** **geom** **ADD** **SPATIAL** **INDEX(g);**

Query OK, 32376 rows affected (4.05 sec)

Records: 32376 Duplicates: 0 Warnings: [0](#_bookmark95)

**11.4.11** **Using** **Spatial** **Indexes**

The optimizer investigates whether available spatial indexes can be involved in the search for queries that use a function such as MBRContains() or MBRWithin() in the WHERE clause. The following query finds all objects that are in the given rectangle:

mysql> **SET** **@poly** **=**

-> **'Polygon((30000** **15000,**

**31000** **15000,**

**31000** **16000,**

**30000** **16000,**

**30000** **15000))';**

mysql> **SELECT** **fid,ST\_AsText(g)** **FROM** **geom** **WHERE**

-> **MBRContains(ST\_GeomFromText(@poly),g);**

+-----+---------------------------------------------------------------+

| fid | ST\_AsText(g) |

+-----+---------------------------------------------------------------+

| 21 | LINESTRING(30350 .4 15828 .8,30350 .6 15845,30333 .8 15845,30 ... |

| 22 | LINESTRING(30350 .6 15871 .4,30350 .6 15887 .8,30334 15887 .8, ... |

| 23 | LINESTRING(30350 .6 15914 .2,30350 .6 15930 .4,30334 15930 .4, ... |

| 24 | LINESTRING(30290 .2 15823,30290 .2 15839 .4,30273 .4 15839 .4, ... |

| 25 | LINESTRING(30291 .4 15866 .2,30291 .6 15882 .4,30274 .8 15882. ... |

| 26 | LINESTRING(30291 .6 15918 .2,30291 .6 15934 .4,30275 15934 .4, ... |

| 249 | LINESTRING(30337 .8 15938 .6,30337 .8 15946 .8,30320 .4 15946. ... |

| 1 | LINESTRING(30250 .4 15129 .2,30248 .8 15138 .4,30238 .2 15136. ... |

| 2 | LINESTRING(30220 .2 15122 .8,30217 .2 15137 .8,30207 .6 15136, ... |

| 3 | LINESTRING(30179 15114 .4,30176 .6 15129 .4,30167 15128,3016 ... |

| 4 | LINESTRING(30155 .2 15121 .4,30140 .4 15118 .6,30142 15109,30 ... |

| 5 | LINESTRING(30192 .4 15085,30177 .6 15082 .2,30179 .2 15072 .4, ... |

| 6 | LINESTRING(30244 15087,30229 15086 .2,30229 .4 15076 .4,3024 ... |

| 7 | LINESTRING(30200 .6 15059 .4,30185 .6 15058 .6,30186 15048 .8, ... |

| 10 | LINESTRING(30179 .6 15017 .8,30181 15002 .8,30190 .8 15003 .6, ... |

| 11 | LINESTRING(30154 .2 15000 .4,30168 .6 15004 .8,30166 15014 .2, ... |

| 13 | LINESTRING(30105 15065 .8,30108 .4 15050 .8,30118 15053,3011 ... |

| 154 | LINESTRING(30276 .2 15143 .8,30261 .4 15141,30263 15131 .4,30 ... |

| 155 | LINESTRING(30269 .8 15084,30269 .4 15093 .4,30258 .6 15093,30 ... |

| 157 | LINESTRING(30128 .2 15011,30113 .2 15010 .2,30113 .6 15000 .4, ... |

+-----+---------------------------------------------------------------+

20 rows in set (0.00 sec)

Use EXPLAIN to check the way this query is executed:

mysql> **SET** **@poly** **=**

-> **'Polygon((30000** **15000,**

**31000** **15000,**

**31000** **16000,**

**30000** **16000,**

**30000** **15000))';**

mysql> **EXPLAIN** **SELECT** **fid,ST\_AsText(g)** **FROM** **geom** **WHERE**

-> **MBRContains(ST\_GeomFromText(@poly),g)\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: geom

type: range

possible\_keys: g

key: g

key\_len: 32

ref: NULL

rows: 50

Extra: Using where

1 row in set (0.00 sec)

Check what would happen without a spatial index:

mysql> **SET** **@poly** **=**

-> **'Polygon((30000** **15000,**

**31000** **15000,**

**31000** **16000,**

**30000** **16000,**

**30000** **15000))';**

mysql> **EXPLAIN** **SELECT** **fid,ST\_AsText(g)** **FROM** **g** **IGNORE** **INDEX** **(g)** **WHERE**

-> **MBRContains(ST\_GeomFromText(@poly),g)\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: geom

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 32376

Extra: Using where

1 row in set (0.00 sec)

Executing the SELECT statement without the spatial index yields the same result but causes the execution time to rise from 0.00 seconds to 0.46 seconds:

mysql> **SET** **@poly** **=**

-> **'Polygon((30000** **15000,**

**31000** **15000,**

**31000** **16000,**

**30000** **16000,**

**30000** **15000))';**

mysql> **SELECT** **fid,ST\_AsText(g)** **FROM** **geom** **IGNORE** **INDEX** **(g)** **WHERE**

-> **MBRContains(ST\_GeomFromText(@poly),g);**

+-----+---------------------------------------------------------------+

| fid | ST\_AsText(g) |

+-----+---------------------------------------------------------------+

| 1 | LINESTRING(30250 .4 15129 .2,30248 .8 15138 .4,30238 .2 15136. ... |

| 2 | LINESTRING(30220 .2 15122 .8,30217 .2 15137 .8,30207 .6 15136, ... |

| 3 | LINESTRING(30179 15114 .4,30176 .6 15129 .4,30167 15128,3016 ... |

| 4 | LINESTRING(30155 .2 15121 .4,30140 .4 15118 .6,30142 15109,30 ... |

| 5 | LINESTRING(30192 .4 15085,30177 .6 15082 .2,30179 .2 15072 .4, ... |

| 6 | LINESTRING(30244 15087,30229 15086 .2,30229 .4 15076 .4,3024 ... |

| 7 | LINESTRING(30200 .6 15059 .4,30185 .6 15058 .6,30186 15048 .8, ... |

| 10 | LINESTRING(30179 .6 15017 .8,30181 15002 .8,30190 .8 15003 .6, ... |

| 11 | LINESTRING(30154 .2 15000 .4,30168 .6 15004 .8,30166 15014 .2, ... |

| 13 | LINESTRING(30105 15065 .8,30108 .4 15050 .8,30118 15053,3011 ... |

| 21 | LINESTRING(30350 .4 15828 .8,30350 .6 15845,30333 .8 15845,30 ... |

| 22 | LINESTRING(30350 .6 15871 .4,30350 .6 15887 .8,30334 15887 .8, ... |

| 23 | LINESTRING(30350 .6 15914 .2,30350 .6 15930 .4,30334 15930 .4, ... |

| 24 | LINESTRING(30290 .2 15823,30290 .2 15839 .4,30273 .4 15839 .4, ... |

| 25 | LINESTRING(30291 .4 15866 .2,30291 .6 15882 .4,30274 .8 15882. ... |

| 26 | LINESTRING(30291 .6 15918 .2,30291 .6 15934 .4,30275 15934 .4, ... |

| 154 | LINESTRING(30276 .2 15143 .8,30261 .4 15141,30263 15131 .4,30 ... |

| 155 | LINESTRING(30269 .8 15084,30269 .4 15093 .4,30258 .6 15093,30 ... |

| 157 | LINESTRING(30128 .2 15011,30113 .2 15010 .2,30113 .6 15000 .4, ... |

| 249 | LINESTRING(30337 .8 15938 .6,30337 .8 15946 .8,30320 .4 15946. ... |

+-----+---------------------------------------------------------------+

20 rows in set (0.46 sec)

**11.5** **The** **JSON** **Data** **Type**

• [Creating JSON Values](#_bookmark145)

• [Normalization, Merging, and Autowrapping of JSON Values](#_bookmark146)

• [Searching and Modifying JSON Values](#_bookmark147)



• [JSON Path Syntax](#_bookmark148)

• [Comparison and Ordering of JSON Values](#_bookmark149)

• [Converting between JSON and non-JSON values](#_bookmark150)

• [Aggregation of JSON Values](#_bookmark151)

MySQL supports a native JSON data type defined by [RFC 7159](https://tools.ietf.org/html/rfc7159) that enables efficient access to data in JSON (JavaScript Object Notation) documents. The JSON data type provides these advantages over storing JSON-format strings in a string column:

• Automatic validation of JSON documents stored in JSON columns. Invalid documents produce an error.

• Optimized storage format. JSON documents stored in JSON columns are converted to an internal format that permits quick read access to document elements. When the server later must read a JSON value stored in this binary format, the value need not be parsed from a text representation. The binary format is structured to enable the server to look up subobjects or nested values directly by key or array index without reading all values before or after them in the document.

MySQL 8.0 also supports the *JSON* *Merge* *Patch* format defined in [RFC 7396](https://tools.ietf.org/html/rfc7396), using the JSON\_MERGE\_PATCH() function. See the description of this function, as well as [Normalization,](#_bookmark146) [Merging, and Autowrapping of JSON Values](#_bookmark146), for examples and further information.

**Note**

This discussion uses JSON in monotype to indicate specifically the JSON data type and “JSON” in regular font to indicate JSON data in general.

The space required to store a JSON document is roughly the same as for [LONGBLOB](#_bookmark61) or [LONGTEXT](#_bookmark61); see [Section 11.7, “Data Type Storage Requirements”](#_bookmark132) , for more information. It is important to keep in mind that the size of any JSON document stored in a JSON column is limited to the value of the max\_allowed\_packet system variable. (When the server is manipulating a JSON value internally in memory, it can be larger than this; the limit applies when the server stores it.) You can obtain the amount of space required to store a JSON document using the JSON\_STORAGE\_SIZE() function; note that for a [JSON](#_bookmark130) column, the storage size—and thus the value returned by this function— is that used by the column prior to any partial updates that may have been performed on it (see the discussion of the JSON partial update optimization later in this section).

Prior to MySQL 8.0.13, a JSON column cannot have a non-NULL default value.

Along with the JSON data type, a set of SQL functions is available to enable operations on JSON values, such as creation, manipulation, and searching. The following discussion shows examples of these operations. For details about individual functions, see Section 12.18, “JSON Functions” .

A set of spatial functions for operating on GeoJSON values is also available. See Section 12.17.11, “Spatial GeoJSON Functions” .

JSON columns, like columns of other binary types, are not indexed directly; instead, you can create an index on a generated column that extracts a scalar value from the JSON column. See Indexing a Generated Column to Provide a JSON Column Index, for a detailed example.

The MySQL optimizer also looks for compatible indexes on virtual columns that match JSON expressions.

In MySQL 8.0.17 and later, the InnoDB storage engine supports multi-valued indexes on JSON arrays.

See Multi-Valued Indexes.

MySQL NDB Cluster 8.0 supports JSON columns and MySQL JSON functions, including creation of an index on a column generated from a JSON column as a workaround for being unable to index a JSON column. A maximum of 3 JSON columns per NDB table is supported.

**Partial** **Updates** **of** **JSON** **Values**

In MySQL 8.0, the optimizer can perform a partial, in-place update of a JSON column instead of removing the old document and writing the new document in its entirety to the column. This optimization can be performed for an update that meets the following conditions:

• The column being updated was declared as JSON.

• The UPDATE statement uses any of the three functions JSON\_SET(), JSON\_REPLACE(), or JSON\_REMOVE() to update the column. A direct assignment of the column value (for example, UPDATE mytable SET jcol = '{"a": 10, "b": 25}') cannot be performed as a partial update.

Updates of multiple JSON columns in a single UPDATE statement can be optimized in this fashion; MySQL can perform partial updates of only those columns whose values are updated using the three functions just listed.

• The input column and the target column must be the same column; a statement such as UPDATE mytable SET jcol1 = JSON\_SET(jcol2, '$.a', 100) cannot be performed as a partial update.

The update can use nested calls to any of the functions listed in the previous item, in any combination, as long as the input and target columns are the same.

• All changes replace existing array or object values with new ones, and do not add any new elements to the parent object or array.

• The value being replaced must be at least as large as the replacement value. In other words, the new value cannot be any larger than the old one.

A possible exception to this requirement occurs when a previous partial update has left sufficient space for the larger value. You can use the function JSON\_STORAGE\_FREE() see how much space has been freed by any partial updates of a JSON column.

Such partial updates can be written to the binary log using a compact format that saves space; this can be enabled by setting the binlog\_row\_value\_options system variable to PARTIAL\_JSON.

It is important to distinguish the partial update of a JSON column value stored in a table from writing the partial update of a row to the binary log. It is possible for the complete update of a JSON column to be recorded in the binary log as a partial update. This can happen when either (or both) of the last two conditions from the previous list is not met but the other conditions are satisfied.

See also the description of binlog\_row\_value\_options.

The next few sections provide basic information regarding the creation and manipulation of JSON values.

**Creating** **JSON** **Values**

A JSON array contains a list of values separated by commas and enclosed within [ and ] characters:

["abc", 10, null, true, false]

A JSON object contains a set of key-value pairs separated by commas and enclosed within { and } characters:

{"k1": "value", "k2": 10}

As the examples illustrate, JSON arrays and objects can contain scalar values that are strings or numbers, the JSON null literal, or the JSON boolean true or false literals. Keys in JSON objects must be strings. Temporal (date, time, or datetime) scalar values are also permitted:

["12:18:29.000000", "2015-07-29", "2015-07-29 12:18:29.000000"]

Nesting is permitted within JSON array elements and JSON object key values:

[99, {"id": "HK500", "cost": 75.99}, ["hot", "cold"]]

{"k1": "value", "k2": [10, 20]}

You can also obtain JSON values from a number of functions supplied by MySQL for this purpose (see Section 12.18.2, “Functions That Create JSON Values” ) as well as by casting values of other types to the JSON type using CAST(*value* AS JSON) (see [Converting between JSON and non-JSON values](#_bookmark150)). The next several paragraphs describe how MySQL handles JSON values provided as input.

In MySQL, JSON values are written as strings. MySQL parses any string used in a context that requires a JSON value, and produces an error if it is not valid as JSON. These contexts include inserting a value into a column that has the JSON data type and passing an argument to a function that expects a JSON value (usually shown as *json\_doc* or *json\_val* in the documentation for MySQL JSON functions), as the following examples demonstrate:

• Attempting to insert a value into a JSON column succeeds if the value is a valid JSON value, but fails if it is not:

mysql> **CREATE** **TABLE** **t1** **(jdoc** **JSON);**

Query OK, 0 rows affected (0.20 sec)

mysql> **INSERT** **INTO** **t1** **VALUES('{"key1":** **"value1",** **"key2":** **"value2"}');**

Query OK, 1 row affected (0.01 sec)

mysql> **INSERT** **INTO** **t1** **VALUES('[1,** **2,');**

ERROR 3140 (22032) at line 2: Invalid JSON text:

"Invalid value." at position 6 in value (or column) '[1, 2,'.

Positions for “at position *N*” in such error messages are 0-based, but should be considered rough indications of where the problem in a value actually occurs.

• The JSON\_TYPE() function expects a JSON argument and attempts to parse it into a JSON value. It returns the value's JSON type if it is valid and produces an error otherwise:

mysql> **SELECT** **JSON\_TYPE('["a",** **"b",** **1]');**

+----------------------------+

| JSON\_TYPE('["a", "b", 1]') |

+----------------------------+

| ARRAY |

+----------------------------+

mysql> **SELECT** **JSON\_TYPE('"hello"');**

+----------------------+

| JSON\_TYPE('"hello"') |

+----------------------+

| STRING |

+ +

mysql> **SELECT** **JSON\_TYPE('hello');**

ERROR 3146 (22032): Invalid data type for JSON data in argument 1

to function json\_type; a JSON string or JSON type is required.

MySQL handles strings used in JSON context using the utf8mb4 character set and utf8mb4\_bin collation. Strings in other character sets are converted to utf8mb4 as necessary. (For strings in the ascii or utf8mb3 character sets, no conversion is needed because ascii and utf8mb3 are subsets of utf8mb4.)

As an alternative to writing JSON values using literal strings, functions exist for composing JSON values from component elements. JSON\_ARRAY() takes a (possibly empty) list of values and returns a JSON array containing those values:

mysql> **SELECT** **JSON\_ARRAY('a',** **1,** **NOW());**

+----------------------------------------+

| JSON\_ARRAY('a', 1, NOW()) |

+----------------------------------------+

| ["a", 1, "2015-07-27 09:43:47.000000"] |

+----------------------------------------+

JSON\_OBJECT() takes a (possibly empty) list of key-value pairs and returns a JSON object containing those pairs:

mysql> **SELECT** **JSON\_OBJECT('key1',** **1,** **'key2',** **'abc');**

+---------------------------------------+

| JSON\_OBJECT('key1', 1, 'key2', 'abc') |

+---------------------------------------+

| {"key1": 1, "key2": "abc"} |

+---------------------------------------+

JSON\_MERGE\_PRESERVE() takes two or more JSON documents and returns the combined result:

mysql> **SELECT** **JSON\_MERGE\_PRESERVE('["a",** **1]',** **'{"key":** **"value"}');**

+-----------------------------------------------------+

| JSON\_MERGE\_PRESERVE('["a", 1]', '{"key": "value"}') |

+-----------------------------------------------------+

| ["a", 1, {"key": "value"}] |

+-----------------------------------------------------+

1 row in set (0.00 sec)

For information about the merging rules, see [Normalization, Merging, and Autowrapping of JSON](#_bookmark146) [Values](#_bookmark146).

(MySQL 8.0.3 and later also support JSON\_MERGE\_PATCH(), which has somewhat different behavior. See JSON\_MERGE\_PATCH() compared with JSON\_MERGE\_PRESERVE(), for information about the differences between these two functions.)

JSON values can be assigned to user-defined variables:

mysql> **SET** **@j** **=** **JSON\_OBJECT('key',** **'value');**

mysql> **SELECT** **@j;**

+------------------+

| @j |

+------------------+

| {"key": "value"} |

+------------------+

However, user-defined variables cannot be of JSON data type, so although @j in the preceding example looks like a JSON value and has the same character set and collation as a JSON value, it does *not* have the JSON data type. Instead, the result from JSON\_OBJECT() is converted to a string when assigned to the variable.

Strings produced by converting JSON values have a character set of utf8mb4 and a collation of utf8mb4\_bin:

mysql> **SELECT** **CHARSET(@j),** **COLLATION(@j);**

+-------------+---------------+

| CHARSET(@j) | COLLATION(@j) |

+-------------+---------------+

| utf8mb4 | utf8mb4\_bin |

+-------------+---------------+

Because utf8mb4\_bin is a binary collation, comparison of JSON values is case-sensitive.

mysql> **SELECT** **JSON\_ARRAY('x')** **=** **JSON\_ARRAY('X');**

+-----------------------------------+

| JSON\_ARRAY('x') = JSON\_ARRAY('X') |

+-----------------------------------+

| 0 |

+-----------------------------------+

Case sensitivity also applies to the JSON null, true, and false literals, which always must be written in lowercase:

mysql> **SELECT** **JSON\_VALID('null'),** **JSON\_VALID('Null'),** **JSON\_VALID('NULL');**

+--------------------+--------------------+--------------------+

| JSON\_VALID('null') | JSON\_VALID('Null') | JSON\_VALID('NULL') |

+--------------------+--------------------+--------------------+

| 1 | 0 | 0 |

+--------------------+--------------------+--------------------+

mysql> **SELECT** **CAST('null'** **AS** **JSON);**

+----------------------+

| CAST('null' AS JSON) |

+----------------------+

| null |

+----------------------+

1 row in set (0.00 sec)

mysql> **SELECT** **CAST('NULL'** **AS** **JSON);**

ERROR 3141 (22032): Invalid JSON text in argument 1 to function cast\_as\_json:

"Invalid value." at position 0 in 'NULL'.

Case sensitivity of the JSON literals differs from that of the SQL NULL, TRUE, and FALSE literals, which can be written in any lettercase:

mysql> **SELECT** **ISNULL(null),** **ISNULL(Null),** **ISNULL(NULL);**

+--------------+--------------+--------------+

| ISNULL(null) | ISNULL(Null) | ISNULL(NULL) |

+--------------+--------------+--------------+

| 1 | 1 | 1 |

+--------------+--------------+--------------+

Sometimes it may be necessary or desirable to insert quote characters ( " or ') into a JSON document. Assume for this example that you want to insert some JSON objects containing strings representing sentences that state some facts about MySQL, each paired with an appropriate keyword, into a table created using the SQL statement shown here:

mysql> **CREATE** **TABLE** **facts** **(sentence** **JSON);**

Among these keyword-sentence pairs is this one:

mascot: The MySQL mascot is a dolphin named "Sakila".

One way to insert this as a JSON object into the facts table is to use the MySQL JSON\_OBJECT() function. In this case, you must escape each quote character using a backslash, as shown here:

mysql> **INSERT** **INTO** **facts** **VALUES**

> **(JSON\_OBJECT("mascot",** **"Our** **mascot** **is** **a** **dolphin** **named** **\"Sakila\"** **."));**

This does not work in the same way if you insert the value as a JSON object literal, in which case, you must use the double backslash escape sequence, like this:

mysql> **INSERT** **INTO** **facts** **VALUES**

> **('{"mascot":** **"Our** **mascot** **is** **a** **dolphin** **named** **\\"Sakila\\"** **."}');**

Using the double backslash keeps MySQL from performing escape sequence processing, and instead causes it to pass the string literal to the storage engine for processing. After inserting the JSON object in either of the ways just shown, you can see that the backslashes are present in the JSON column value by doing a simple SELECT, like this:

mysql> **SELECT** **sentence** **FROM** **facts;**

+---------------------------------------------------------+

| sentence |

+---------------------------------------------------------+

| {"mascot": "Our mascot is a dolphin named \"Sakila\" ."} |

+---------------------------------------------------------+

To look up this particular sentence employing mascot as the key, you can use the column-path operator ->, as shown here:

mysql> SELECT col->"$.mascot" FROM qtest;

+---------------------------------------------+

|  |  |
| --- | --- |
| | col->"$.mascot" | | |



+---------------------------------------------+

| "Our mascot is a dolphin named \"Sakila\" ." |

+---------------------------------------------+

1 row in set (0.00 sec)

This leaves the backslashes intact, along with the surrounding quote marks. To display the desired value using mascot as the key, but without including the surrounding quote marks or any escapes, use the inline path operator ->>, like this:

mysql> **SELECT** **sentence->>"$** **.mascot"** **FROM** **facts;**

+-----------------------------------------+

| sentence->>"$ .mascot" |

+-----------------------------------------+

| Our mascot is a dolphin named "Sakila" . |

+-----------------------------------------+

**Note**

The previous example does not work as shown if the NO\_BACKSLASH\_ESCAPES server SQL mode is enabled. If this mode is

set, a single backslash instead of double backslashes can be used to insert the JSON object literal, and the backslashes are preserved. If you use the JSON\_OBJECT() function when performing the insert and this mode is set, you must alternate single and double quotes, like this:

mysql> **INSERT** **INTO** **facts** **VALUES**

> **(JSON\_OBJECT('mascot',** **'Our** **mascot** **is** **a** **dolphin** **named** **"Sakila"** **.** **'));**

See the description of the JSON\_UNQUOTE() function for more information about the effects of this mode on escaped characters in JSON values.

**Normalization,** **Merging,** **and** **Autowrapping** **of** **JSON** **Values**

When a string is parsed and found to be a valid JSON document, it is also normalized. This means that members with keys that duplicate a key found later in the document, reading from left to right, are discarded. The object value produced by the following JSON\_OBJECT() call includes only the second key1 element because that key name occurs earlier in the value, as shown here:

mysql> **SELECT** **JSON\_OBJECT('key1',** **1,** **'key2',** **'abc',** **'key1',** **'def');**

+------------------------------------------------------+

| JSON\_OBJECT('key1', 1, 'key2', 'abc', 'key1', 'def') |

+------------------------------------------------------+

|  |  |
| --- | --- |
| | {"key1": "def", "key2": "abc"} | | |

+------------------------------------------------------+

Normalization is also performed when values are inserted into JSON columns, as shown here:

mysql> **CREATE** **TABLE** **t1** **(c1** **JSON);**

mysql> **INSERT** **INTO** **t1** **VALUES**

>

>

mysql> **SELECT** **c1** **FROM** **t1;**

+------------------+

| c1 |

+------------------+

| {"x": "red"} |

| {"x": [3, 5, 7]} |

+------------------+

**('{"x":** **17,** **"x":** **"red"}'),**

**('{"x":** **17,** **"x":** **"red",** **"x":** **[3,** **5,** **7]}');**

This “last duplicate key wins” behavior is suggested by [RFC 7159](https://tools.ietf.org/html/rfc7159) and is implemented by most JavaScript parsers. (Bug #86866, Bug #26369555)

In versions of MySQL prior to 8.0.3, members with keys that duplicated a key found earlier in the document were discarded. The object value produced by the following JSON\_OBJECT() call does not include the second key1 element because that key name occurs earlier in the value:

in previous versions of MySQL (renamed in MySQL 8.0.3). JSON\_MERGE() is still supported as an alias for JSON\_MERGE\_PRESERVE() in MySQL 8.0, but is deprecated and subject to removal in a future release.

**Merging** **arrays.** In contexts that combine multiple arrays, the arrays are merged into a single array. JSON\_MERGE\_PRESERVE() does this by concatenating arrays named later to the end of the first array. JSON\_MERGE\_PATCH() considers each argument as an array consisting of a single element (thus having 0 as its index) and then applies “last duplicate key wins” logic to select only the last argument. You can compare the results shown by this query:

mysql> **SELECT** **JSON\_OBJECT('key1',** **1,** **'key2',** **'abc',** **'key1',** **'def');**

+------------------------------------------------------+

| JSON\_OBJECT('key1', 1, 'key2', 'abc', 'key1', 'def') |

+------------------------------------------------------+

|  |  |
| --- | --- |
| | {"key1": 1, "key2": "abc"} | | |

+------------------------------------------------------+

Prior to MySQL 8.0.3, this “first duplicate key wins” normalization was also performed when inserting values into JSON columns.

mysql> **CREATE** **TABLE** **t1** **(c1** **JSON);**

mysql> **INSERT** **INTO** **t1** **VALUES**

>

>

mysql> **SELECT** **c1** **FROM** **t1;**

+-----------+

| c1 |

+-----------+

| {"x": 17} |

| {"x": 17} |

+-----------+

**('{"x":** **17,** **"x":** **"red"}'),**

**('{"x":** **17,** **"x":** **"red",** **"x":** **[3,** **5,** **7]}');**

MySQL also discards extra whitespace between keys, values, or elements in the original JSON document, and leaves (or inserts, when necessary) a single space following each comma ( ,) or colon (:) when displaying it. This is done to enhance readability.

MySQL functions that produce JSON values (see Section 12.18.2, “Functions That Create JSON Values” ) always return normalized values.

To make lookups more efficient, MySQL also sorts the keys of a JSON object. *You* *should* *be* *aware* *that* *the* *result* *of* *this* *ordering* *is* *subject* *to* *change* *and* *not* *guaranteed* *to* *be* *consistent* *across* *releases*.

**Merging** **JSON** **Values**

Two merging algorithms are supported in MySQL 8.0.3 (and later), implemented by the functions JSON\_MERGE\_PRESERVE() and JSON\_MERGE\_PATCH(). These differ in how they handle duplicate keys: JSON\_MERGE\_PRESERVE() retains values for duplicate keys, while JSON\_MERGE\_PATCH() discards all but the last value. The next few paragraphs explain how each of these two functions handles the merging of different combinations of JSON documents (that is, of objects and arrays).

 **Note**

 JSON\_MERGE\_PRESERVE() is the same as the JSON\_MERGE() function found



|  |  |
| --- | --- |
| mysql> **SELECT**  -> **JSON\_MERGE\_PRESERVE('[1,** **2]',** **'["a",** **"b",** **"c"]',** **'[true,** **false]')**  -> **JSON\_MERGE\_PATCH('[1,** **2]',** **'["a",** **"b",** **"c"]',** **'[true,** **false]')** **AS**  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Preserve: [1, 2, "a", "b", "c", true, false]  Patch: [true, false] | **AS** **Preserve,**  **Patch\G** |

Multiple objects when merged produce a single object. JSON\_MERGE\_PRESERVE() handles multiple objects having the same key by combining all unique values for that key in an array; this array is then

used as the value for that key in the result. JSON\_MERGE\_PATCH() discards values for which duplicate keys are found, working from left to right, so that the result contains only the last value for that key. The following query illustrates the difference in the results for the duplicate key a:

mysql> **SELECT**

-> **JSON\_MERGE\_PRESERVE('{"a":** **1,** **"b":** **2}',** **'{"c":** **3,** **"a":** **4}',** **'{"c":** **5,** **"d":** **3}')** **AS** **Preserve,**

-> **JSON\_MERGE\_PATCH('{"a":** **3,** **"b":** **2}',** **'{"c":** **3,** **"a":** **4}',** **'{"c":** **5,** **"d":** **3}')** **AS** **Patch\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Preserve: {"a": [1, 4], "b": 2, "c": [3, 5], "d": 3}

Patch: {"a": 4, "b": 2, "c": 5, "d": 3}

Nonarray values used in a context that requires an array value are autowrapped: The value is surrounded by [ and ] characters to convert it to an array. In the following statement, each argument is autowrapped as an array ( [1], [2]). These are then merged to produce a single result array; as in the previous two cases, JSON\_MERGE\_PRESERVE() combines values having the same key while JSON\_MERGE\_PATCH() discards values for all duplicate keys except the last, as shown here:

mysql> **SELECT**

-> **JSON\_MERGE\_PRESERVE('1',** **'2')** **AS** **Preserve,**

-> **JSON\_MERGE\_PATCH('1',** **'2')** **AS** **Patch\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Preserve: [1, 2]

Patch: 2

Array and object values are merged by autowrapping the object as an array and merging the arrays by combining values or by “last duplicate key wins” according to the choice of merging function (JSON\_MERGE\_PRESERVE() or JSON\_MERGE\_PATCH(), respectively), as can be seen in this example:

mysql> **SELECT**

-> **JSON\_MERGE\_PRESERVE('[10,** **20]',** **'{"a":** **"x",** **"b":** **"y"}')** **AS** **Preserve,**

-> **JSON\_MERGE\_PATCH('[10,** **20]',** **'{"a":** **"x",** **"b":** **"y"}')** **AS** **Patch\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Preserve: [10, 20, {"a": "x", "b": "y"}]

Patch: {"a": "x", "b": "y"}

**Searching** **and** **Modifying** **JSON** **Values**

A JSON path expression selects a value within a JSON document.

Path expressions are useful with functions that extract parts of or modify a JSON document, to specify where within that document to operate. For example, the following query extracts from a JSON document the value of the member with the name key:

mysql> **SELECT** **JSON\_EXTRACT('{"id":** **14,** **"name":** **"Aztalan"}',** **'$** **.name');**

+---------------------------------------------------------+

| JSON\_EXTRACT('{"id": 14, "name": "Aztalan"}', '$ .name') |

+---------------------------------------------------------+

| "Aztalan" |

+---------------------------------------------------------+

Path syntax uses a leading $ character to represent the JSON document under consideration, optionally followed by selectors that indicate successively more specific parts of the document:

• A period followed by a key name names the member in an object with the given key. The key name must be specified within double quotation marks if the name without quotes is not legal within path expressions (for example, if it contains a space).

• [*N*] appended to a *path* that selects an array names the value at position *N*within the array. Array positions are integers beginning with zero. If *path* does not select an array value, *path*[0] evaluates to the same value as *path*:

mysql> **SELECT** **JSON\_SET('"x"',** **'$[0]',** **'a');**

+------------------------------+

| JSON\_SET('"x"', '$[0]', 'a') |

+------------------------------+

|  |  |
| --- | --- |
| | "a" | | |

+------------------------------+

1 row in set (0.00 sec)

• [*M* to *N*] specifies a subset or range of array values starting with the value at position *M*, and ending with the value at position *N*.

last is supported as a synonym for the index of the rightmost array element. Relative addressing of array elements is also supported. If *path* does not select an array value, *path*[last] evaluates to the same value as *path*, as shown later in this section (see [Rightmost array element](#_bookmark152)).

• Paths can contain \* or \*\* wildcards:

• . [\*] evaluates to the values of all members in a JSON object.

• [\*] evaluates to the values of all elements in a JSON array.

• *prefix*\*\**suffix* evaluates to all paths that begin with the named prefix and end with the named suffix.

• A path that does not exist in the document (evaluates to nonexistent data) evaluates to NULL. Let $ refer to this JSON array with three elements:

[3, {"a": [5, 6], "b": 10}, [99, 100]]

Then:

• $[0] evaluates to 3.

• $[1] evaluates to {"a": [5, 6], "b": 10}.

• $[2] evaluates to [99, 100].

• $[3] evaluates to NULL (it refers to the fourth array element, which does not exist).

Because $[1] and $[2] evaluate to nonscalar values, they can be used as the basis for more-specific path expressions that select nested values. Examples:

• $[1].a evaluates to [5, 6].

• $[1].a[1] evaluates to 6.

• $[1].b evaluates to 10.

• $[2][0] evaluates to 99.

As mentioned previously, path components that name keys must be quoted if the unquoted key name is not legal in path expressions. Let $ refer to this value:

{"a fish": "shark", "a bird": "sparrow"}

The keys both contain a space and must be quoted:

• $."a fish" evaluates to shark.

• $."a bird" evaluates to sparrow.

Paths that use wildcards evaluate to an array that can contain multiple values:

mysql> **SELECT** **JSON\_EXTRACT('{"a":** **1,** **"b":** **2,** **"c":** **[3,** **4,** **5]}',** **'$** **.\*');**

+---------------------------------------------------------+

| JSON\_EXTRACT('{"a": 1, "b": 2, "c": [3, 4, 5]}', '$ .\*') |

+---------------------------------------------------------+

| [1, 2, [3, 4, 5]] |

+---------------------------------------------------------+

mysql> **SELECT** **JSON\_EXTRACT('{"a":** **1,** **"b":** **2,** **"c":** **[3,** **4,** **5]}',** **'$** **.c[\*]');**

+------------------------------------------------------------+

| JSON\_EXTRACT('{"a": 1, "b": 2, "c": [3, 4, 5]}', '$ .c[\*]') |

+------------------------------------------------------------+

| [3, 4, 5] |

+------------------------------------------------------------+

In the following example, the path $\*\*.b evaluates to multiple paths ($.a.b and $.c.b) and produces an array of the matching path values:

mysql> **SELECT** **JSON\_EXTRACT('{"a":** **{"b":** **1},** **"c":** **{"b":** **2}}',** **'$\*\*** **.b');**

+---------------------------------------------------------+

| JSON\_EXTRACT('{"a": {"b": 1}, "c": {"b": 2}}', '$\*\* .b') |

+---------------------------------------------------------+

| [1, 2] |

+---------------------------------------------------------+

**Ranges** **from** **JSON** **arrays.**

arrays. For example, $[1 to here:

You can use ranges with the to keyword to specify subsets of JSON 3] includes the second, third, and fourth elements of an array, as shown

mysql> **SELECT** **JSON\_EXTRACT('[1,** **2,** **3,** **4,** **5]',** **'$[1** **to** **3]');**

+----------------------------------------------+

| JSON\_EXTRACT('[1, 2, 3, 4, 5]', '$[1 to 3]') |

+----------------------------------------------+

| [2, 3, 4] |

+----------------------------------------------+

1 row in set (0.00 sec)

The syntax is *M* to *N*, where *M*and *N*are, respectively, the first and last indexes of a range of elements from a JSON array. *N* must be greater than *M*; *M* must be greater than or equal to 0. Array elements are indexed beginning with 0.

You can use ranges in contexts where wildcards are supported.

**Rightmost** **array** **element.** The last keyword is supported as a synonym for the index of the last element in an array. Expressions of the form last - *N*can be used for relative addressing, and within range definitions, like this:

mysql> **SELECT** **JSON\_EXTRACT('[1,** **2,** **3,** **4,** **5]',** **'$[last-3** **to** **last-1]');**

+--------------------------------------------------------+

| JSON\_EXTRACT('[1, 2, 3, 4, 5]', '$[last-3 to last-1]') |

+--------------------------------------------------------+

| [2, 3, 4] |

+--------------------------------------------------------+

1 row in set (0.01 sec)

If the path is evaluated against a value that is not an array, the result of the evaluation is the same as if the value had been wrapped in a single-element array:

mysql> SELECT JSON\_REPLACE('"Sakila"', '$[last]', 10);

+-----------------------------------------+

| JSON\_REPLACE('"Sakila"', '$[last]', 10) |

+-----------------------------------------+

| 10 |

+-----------------------------------------+

1 row in set (0.00 sec)

You can use *column*->*path* with a JSON column identifier and JSON path expression as a synonym for JSON\_EXTRACT(*column*, *path*). See Section 12.18.3, “Functions That Search JSON Values” , for more information. See also Indexing a Generated Column to Provide a JSON Column Index.

Some functions take an existing JSON document, modify it in some way, and return the resulting modified document. Path expressions indicate where in the document to make changes. For example, the JSON\_SET(), JSON\_INSERT(), and JSON\_REPLACE() functions each take a JSON document, plus one or more path-value pairs that describe where to modify the document and the values to use. The functions differ in how they handle existing and nonexisting values within the document.

Consider this document:

mysql> SET @j = '["a", {"b": [true, false]}, [10, 20]]';

JSON\_SET() replaces values for paths that exist and adds values for paths that do not exist:.

mysql> **SELECT** **JSON\_SET(@j,** **'$[1]** **.b[0]',** **1,** **'$[2][2]',** **2);**

+--------------------------------------------+

| JSON\_SET(@j, '$[1] .b[0]', 1, '$[2][2]', 2) |

+--------------------------------------------+

| ["a", {"b": [1, false]}, [10, 20, 2]] |

+--------------------------------------------+

In this case, the path $[1].b[0] selects an existing value (true), which is replaced with the value following the path argument (1). The path $[2][2] does not exist, so the corresponding value (2) is added to the value selected by $[2].

JSON\_INSERT() adds new values but does not replace existing values:

mysql> **SELECT** **JSON\_INSERT(@j,** **'$[1]** **.b[0]',** **1,** **'$[2][2]',** **2);**

+-----------------------------------------------+

| JSON\_INSERT(@j, '$[1] .b[0]', 1, '$[2][2]', 2) |

+-----------------------------------------------+

| ["a", {"b": [true, false]}, [10, 20, 2]] |

+-----------------------------------------------+

JSON\_REPLACE() replaces existing values and ignores new values:

mysql> **SELECT** **JSON\_REPLACE(@j,** **'$[1]** **.b[0]',** **1,** **'$[2][2]',** **2);**

+------------------------------------------------+

| JSON\_REPLACE(@j, '$[1] .b[0]', 1, '$[2][2]', 2) |

+------------------------------------------------+

|

+------------------------------------------------+

| ["a", {"b": [1, false]}, [10, 20]]

The path-value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

JSON\_REMOVE() takes a JSON document and one or more paths that specify values to be removed from the document. The return value is the original document minus the values selected by paths that exist within the document:

mysql> **SELECT** **JSON\_REMOVE(@j,** **'$[2]',** **'$[1]** **.b[1]',** **'$[1]** **.b[1]');**

+---------------------------------------------------+

| JSON\_REMOVE(@j, '$[2]', '$[1] .b[1]', '$[1] .b[1]') |

+---------------------------------------------------+

|  |  |
| --- | --- |
| | ["a", {"b": [true]}] | | |

+---------------------------------------------------+

The paths have these effects:

• $[2] matches [10, 20] and removes it.

• The first instance of $[1].b[1] matches false in the b element and removes it.

• The second instance of $[1].b[1] matches nothing: That element has already been removed, the path no longer exists, and has no effect.

**JSON** **Path** **Syntax**

Many of the JSON functions supported by MySQL and described elsewhere in this Manual (see Section 12.18, “JSON Functions” ) require a path expression in order to identify a specific element in a JSON document. A path consists of the path's scope followed by one or more path legs. For paths used in MySQL JSON functions, the scope is always the document being searched or otherwise operated on, represented by a leading $ character. Path legs are separated by period characters ( .). Cells in arrays are represented by [*N*], where *N* is a non-negative integer. Names of keys must be double-quoted strings or valid ECMAScript identifiers (see [*Identifier* *Names* *and* *Identifiers*](http://www.ecma-international.org/ecma-262/5.1/#sec-7.6), in the



*ECMAScript* *Language* *Specification*). Path expressions, like JSON text, should be encoded using the ascii, utf8mb3, or utf8mb4 character set. Other character encodings are implicitly coerced to utf8mb4. The complete syntax is shown here:

*pathExpression*:

*scope* [(*pathLeg*)\*]

*pathLeg*:

*member* | *arrayLocation* | *doubleAsterisk*

*member*:

*period* ( *keyName* | *asterisk* )

*arrayLocation*:

*leftBracket* ( *nonNegativeInteger* | *asterisk* ) *rightBracket*

*keyName*:

[*ESIdentifier*](http://www.ecma-international.org/ecma-262/5.1/#sec-7.6) | *doubleQuotedString*

*doubleAsterisk*:

'\*\*'

*period*:

'.'

*asterisk*:

'\*'

*leftBracket*:

'['

*rightBracket*:

']'

As noted previously, in MySQL, the scope of the path is always the document being operated on, represented as $. You can use '$' as a synonym for the document in JSON path expressions.

 **Note**

  Some implementations support column references for scopes of JSON paths;

MySQL 8.0 does not support these.

The wildcard \* and \*\* tokens are used as follows:

• .\* represents the values of all members in the object.

• [\*] represents the values of all cells in the array.

• [*prefix*]\*\**suffix* represents all paths beginning with *prefix* and ending with *suffix*. *prefix* is optional, while *suffix* is required; in other words, a path may not end in \*\*.

In addition, a path may not contain the sequence \*\*\*.

For path syntax examples, see the descriptions of the various JSON functions that take paths as arguments, such as JSON\_CONTAINS\_PATH(), JSON\_SET(), and JSON\_REPLACE(). For examples which include the use of the \* and \*\* wildcards, see the description of the JSON\_SEARCH() function.

MySQL 8.0 also supports range notation for subsets of JSON arrays using the to keyword (such as $[2 to 10]), as well as the last keyword as a synonym for the rightmost element of an array. See [Searching and Modifying JSON Values](#_bookmark147), for more information and examples.

**Comparison** **and** **Ordering** **of** **JSON** **Values**

JSON values can be compared using the [=](#_bookmark153), [<](#_bookmark154), [<=](#_bookmark155), [>](#_bookmark156), [>=](#_bookmark157), [<>](#_bookmark158), [!=](#_bookmark158), and [<=>](#_bookmark159) operators. The following comparison operators and functions are not yet supported with JSON values:

• [BETWEEN](#_bookmark160)

• [IN()](#_bookmark161)

• [GREATEST()](#_bookmark162)

• [LEAST()](#_bookmark163)

A workaround for the comparison operators and functions just listed is to cast JSON values to a native MySQL numeric or string data type so they have a consistent non-JSON scalar type.

Comparison of JSON values takes place at two levels. The first level of comparison is based on the JSON types of the compared values. If the types differ, the comparison result is determined solely by which type has higher precedence. If the two values have the same JSON type, a second level of comparison occurs using type-specific rules.

The following list shows the precedences of JSON types, from highest precedence to the lowest. (The type names are those returned by the JSON\_TYPE() function.) Types shown together on a line have the same precedence. Any value having a JSON type listed earlier in the list compares greater than any value having a JSON type listed later in the list.

BLOB

BIT

OPAQUE

DATETIME

TIME

DATE

BOOLEAN

ARRAY

OBJECT

STRING

INTEGER, DOUBLE

NULL

For JSON values of the same precedence, the comparison rules are type specific:

• BLOB

The first *N* bytes of the two values are compared, where *N* is the number of bytes in the shorter value. If the first *N* bytes of the two values are identical, the shorter value is ordered before the longer value.

• BIT

Same rules as for BLOB.

• OPAQUE

Same rules as for BLOB. OPAQUE values are values that are not classified as one of the other types.

• DATETIME

A value that represents an earlier point in time is ordered before a value that represents a later point in time. If two values originally come from the MySQL DATETIME and TIMESTAMP types, respectively, they are equal if they represent the same point in time.

• TIME

The smaller of two time values is ordered before the larger one.

• DATE

The earlier date is ordered before the more recent date.

• ARRAY

Two JSON arrays are equal if they have the same length and values in corresponding positions in the arrays are equal.

If the arrays are not equal, their order is determined by the elements in the first position where there is a difference. The array with the smaller value in that position is ordered first. If all values of the shorter array are equal to the corresponding values in the longer array, the shorter array is ordered first.

Example: [] < ["a"] < ["ab"] < ["ab", "cd", "ef"] < ["ab", "ef"]

• BOOLEAN

The JSON false literal is less than the JSON true literal.

• OBJECT

Two JSON objects are equal if they have the same set of keys, and each key has the same value in both objects.

Example: {"a": 1, "b": 2} = {"b": 2, "a": 1} The order of two objects that are not equal is unspecified but deterministic.

• STRING

Strings are ordered lexically on the first *N* bytes of the utf8mb4 representation of the two strings being compared, where *N* is the length of the shorter string. If the first *N* bytes of the two strings are identical, the shorter string is considered smaller than the longer string.

Example: "a" < "ab" < "b" < "bc"

This ordering is equivalent to the ordering of SQL strings with collation utf8mb4\_bin. Because utf8mb4\_bin is a binary collation, comparison of JSON values is case-sensitive:

"A" < "a"

• INTEGER, DOUBLE

JSON values can contain exact-value numbers and approximate-value numbers. For a general discussion of these types of numbers, see Section 9.1.2, “Numeric Literals” .

The rules for comparing native MySQL numeric types are discussed in [Section 12.3, “Type](#_bookmark75) [Conversion in Expression Evaluation”](#_bookmark75) , but the rules for comparing numbers within JSON values differ somewhat:

• In a comparison between two columns that use the native MySQL [INT](#_bookmark101) and [DOUBLE](#_bookmark103) numeric types, respectively, it is known that all comparisons involve an integer and a double, so the integer is converted to double for all rows. That is, exact-value numbers are converted to approximate- value numbers.

• On the other hand, if the query compares two JSON columns containing numbers, it cannot be known in advance whether numbers are integer or double. To provide the most consistent behavior across all rows, MySQL converts approximate-value numbers to exact-value numbers. The resulting ordering is consistent and does not lose precision for the exact-value numbers. For example, given the scalars 9223372036854775805, 9223372036854775806,

9223372036854775807 and 9.223372036854776e18, the order is such as this:

9223372036854775805 < 9223372036854775806 < 9223372036854775807

< 9.223372036854776e18 = 9223372036854776000 < 9223372036854776001

Were JSON comparisons to use the non-JSON numeric comparison rules, inconsistent ordering could occur. The usual MySQL comparison rules for numbers yield these orderings:

• Integer comparison: 9223372036854775805 < 9223372036854775806 < 9223372036854775807 (not defined for 9.223372036854776e18)

• Double comparison: 9223372036854775805 = 9223372036854775806 = 9223372036854775807 = 9.223372036854776e18

For comparison of any JSON value to SQL NULL, the result is UNKNOWN.

For comparison of JSON and non-JSON values, the non-JSON value is converted to JSON according to the rules in the following table, then the values compared as described previously.

**Converting** **between** **JSON** **and** **non-JSON** **values**

The following table provides a summary of the rules that MySQL follows when casting between JSON values and values of other types:

**Table** **11.3** **JSON** **Conversion** **Rules**

|  |  |  |
| --- | --- | --- |
| **other** **type** | **CAST(other** **type** **AS** **JSON)** | **CAST(JSON** **AS** **other** **type)** |
| JSON | No change | No change |
| utf8 character type (utf8mb4, utf8mb3, ascii) | The string is parsed into a JSON value. | The JSON value is serialized into a utf8mb4 string. |
| Other character types | Other character encodings are implicitly converted to utf8mb4 and treated as described for this character type. | The JSON value is serialized into a utf8mb4 string, then cast to the other character encoding. The result may not be meaningful. |
| NULL | Results in a NULL value of type  JSON. | Not applicable. |
| Geometry types | The geometry value is converted into a JSON document by calling ST\_AsGeoJSON(). | Illegal operation. Workaround: Pass the result of CAST(*json\_val* AS CHAR) to ST\_GeomFromGeoJSON(). |
| All other types | Results in a JSON document consisting of a single scalar value. | Succeeds if the JSON document consists of a single scalar value of the target type and that scalar value can be cast to the target type. Otherwise, returns NULL and produces a warning. |

ORDER BY and GROUP BY for JSON values works according to these principles:

• Ordering of scalar JSON values uses the same rules as in the preceding discussion.

• For ascending sorts, SQL NULL orders before all JSON values, including the JSON null literal; for descending sorts, SQL NULL orders after all JSON values, including the JSON null literal.

• Sort keys for JSON values are bound by the value of the max\_sort\_length system variable, so keys that differ only after the first max\_sort\_length bytes compare as equal.

• Sorting of nonscalar values is not currently supported and a warning occurs.

For sorting, it can be beneficial to cast a JSON scalar to some other native MySQL type. For example, if a column named jdoc contains JSON objects having a member consisting of an id key and a nonnegative value, use this expression to sort by id values:

ORDER BY CAST(JSON\_EXTRACT(jdoc, '$.id') AS UNSIGNED)

If there happens to be a generated column defined to use the same expression as in the ORDER BY, the MySQL optimizer recognizes that and considers using the index for the query execution plan. See Section 8.3.11, “Optimizer Use of Generated Column Indexes” .

**Aggregation** **of** **JSON** **Values**

For aggregation of JSON values, SQL NULL values are ignored as for other data types.

Non-NULL values are converted to a numeric type and aggregated, except for MIN(), MAX(), and GROUP\_CONCAT(). The conversion to number should produce a meaningful result for JSON values that are numeric scalars, although (depending on the values) truncation and loss of precision may occur. Conversion to number of other JSON values may not produce a meaningful result.

**11.6** **Data** **Type** **Default** **Values**

Data type specifications can have explicit or implicit default values.

A DEFAULT *value* clause in a data type specification explicitly indicates a default value for a column. Examples:

CREATE TABLE t1 (

i INT DEFAULT -1,

c VARCHAR(10) DEFAULT '',

price DOUBLE(16,2) DEFAULT 0.00

);

SERIAL DEFAULT VALUE is a special case. In the definition of an integer column, it is an alias for NOT

NULL AUTO\_INCREMENT UNIQUE.

Some aspects of explicit DEFAULT clause handling are version dependent, as described following.

• [Explicit Default Handling as of MySQL 8.0.13](#_bookmark164)

• [Explicit Default Handling Prior to MySQL 8.0.13](#_bookmark165)

• [Implicit Default Handling](#_bookmark166)

**Explicit** **Default** **Handling** **as** **of** **MySQL** **8.0.13**

The default value specified in a DEFAULT clause can be a literal constant or an expression. With one exception, enclose expression default values within parentheses to distinguish them from literal constant default values. Examples:

CREATE TABLE t1 (

-- literal defaults

|  |  |
| --- | --- |
| i INT  c VARCHAR(10)  -- expression  f FLOAT  b BINARY(16)  d DATE  p POINT  j JSON  ); | DEFAULT 0,  DEFAULT '',  defaults  DEFAULT (RAND() \* RAND()),  DEFAULT (UUID\_TO\_BIN(UUID())),  DEFAULT (CURRENT\_DATE + INTERVAL 1 YEAR),  DEFAULT (Point(0,0)),  DEFAULT (JSON\_ARRAY()) |

different results may occur for different uses of the table unless the SQL mode is the same during all uses.

For CREATE TABLE ... LIKE and CREATE TABLE ... SELECT, the destination table preserves expression default values from the original table.

If an expression default value refers to a nondeterministic function, any statement that causes the expression to be evaluated is unsafe for statement-based replication. This includes statements such as INSERT and UPDATE. In this situation, if binary logging is disabled, the statement is executed as normal. If binary logging is enabled and binlog\_format is set to STATEMENT, the statement is logged and executed but a warning message is written to the error log, because replication slaves might diverge. When binlog\_format is set to MIXED or ROW, the statement is executed as normal.

When inserting a new row, the default value for a column with an expression default can be inserted either by omitting the column name or by specifying the column as DEFAULT (just as for columns with literal defaults):

The exception is that, for [TIMESTAMP](#_bookmark109) and [DATETIME](#_bookmark109) columns, you can specify the CURRENT\_TIMESTAMP function as the default, without enclosing parentheses. See [Section 11.2.5,](#_bookmark112) [“Automatic Initialization and Updating for TIMESTAMP and DATETIME”](#_bookmark112) .

The [BLOB](#_bookmark61), [TEXT](#_bookmark61), GEOMETRY, and [JSON](#_bookmark130) data types can be assigned a default value only if the value is written as an expression, even if the expression value is a literal:

• This is permitted (literal default specified as expression): CREATE TABLE t2 (b BLOB DEFAULT ('abc'));

• This produces an error (literal default not specified as expression): CREATE TABLE t2 (b BLOB DEFAULT 'abc');

Expression default values must adhere to the following rules. An error occurs if an expression contains disallowed constructs.

• Literals, built-in functions (both deterministic and nondeterministic), and operators are permitted.

• Subqueries, parameters, variables, stored functions, and loadable functions are not permitted.

• An expression default value cannot depend on a column that has the AUTO\_INCREMENT attribute.

• An expression default value for one column can refer to other table columns, with the exception that references to generated columns or columns with expression default values must be to columns that occur earlier in the table definition. That is, expression default values cannot contain forward references to generated columns or columns with expression default values.

The ordering constraint also applies to the use of ALTER TABLE to reorder table columns. If the resulting table would have an expression default value that contains a forward reference to a generated column or column with an expression default value, the statement fails.

 **Note**

 If any component of an expression default value depends on the SQL mode,



mysql> **CREATE** **TABLE** **t4** **(uid** **BINARY(16)** **DEFAULT** **(UUID\_TO\_BIN(UUID())));**

mysql> **INSERT** **INTO** **t4** **()** **VALUES();**

mysql> **INSERT** **INTO** **t4** **()** **VALUES(DEFAULT);**

mysql> **SELECT** **BIN\_TO\_UUID(uid)** **AS** **uid** **FROM** **t4;**

+--------------------------------------+

| uid |

+--------------------------------------+

| f1109174-94c9-11e8-971d-3bf1095aa633 |

| f110cf9a-94c9-11e8-971d-3bf1095aa633 |

+--------------------------------------+

However, the use of DEFAULT(*col\_name*) to specify the default value for a named column is permitted only for columns that have a literal default value, not for columns that have an expression default value.

Not all storage engines permit expression default values. For those that do not, an [ER\_UNSUPPORTED\_ACTION\_ON\_DEFAULT\_VAL\_GENERATED](https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html#error_er_unsupported_action_on_default_val_generated) error occurs.

If a default value evaluates to a data type that differs from the declared column type, implicit coercion to the declared type occurs according to the usual MySQL type-conversion rules. See [Section 12.3, “Type](#_bookmark75) [Conversion in Expression Evaluation”](#_bookmark75) .

**Explicit** **Default** **Handling** **Prior** **to** **MySQL** **8.0.13**

With one exception, the default value specified in a DEFAULT clause must be a literal constant; it cannot be a function or an expression. This means, for example, that you cannot set the default for a date column to be the value of a function such as NOW() or CURRENT\_DATE. The exception is that, for [TIMESTAMP](#_bookmark109) and [DATETIME](#_bookmark109) columns, you can specify CURRENT\_TIMESTAMP as the default. See [Section 11.2.5, “Automatic Initialization and Updating for TIMESTAMP and DATETIME”](#_bookmark112) .

The [BLOB](#_bookmark61), [TEXT](#_bookmark61), GEOMETRY, and [JSON](#_bookmark130) data types cannot be assigned a default value.

If a default value evaluates to a data type that differs from the declared column type, implicit coercion to the declared type occurs according to the usual MySQL type-conversion rules. See [Section 12.3, “Type](#_bookmark75) [Conversion in Expression Evaluation”](#_bookmark75) .

**Implicit** **Default** **Handling**

If a data type specification includes no explicit DEFAULT value, MySQL determines the default value as follows:

If the column can take NULL as a value, the column is defined with an explicit DEFAULT NULL clause.

If the column cannot take NULL as a value, MySQL defines the column with no explicit DEFAULT clause.

For data entry into a NOT NULL column that has no explicit DEFAULT clause, if an INSERT or REPLACE statement includes no value for the column, or an UPDATE statement sets the column to NULL, MySQL handles the column according to the SQL mode in effect at the time:

• If strict SQL mode is enabled, an error occurs for transactional tables and the statement is rolled back. For nontransactional tables, an error occurs, but if this happens for the second or subsequent row of a multiple-row statement, the preceding rows are inserted.

• If strict mode is not enabled, MySQL sets the column to the implicit default value for the column data type.

Suppose that a table t is defined as follows:

CREATE TABLE t (i INT NOT NULL);

In this case, i has no explicit default, so in strict mode each of the following statements produce an error and no row is inserted. When not using strict mode, only the third statement produces an error; the implicit default is inserted for the first two statements, but the third fails because DEFAULT(i) cannot produce a value:

INSERT INTO t VALUES();

INSERT INTO t VALUES(DEFAULT);

INSERT INTO t VALUES(DEFAULT(i));

See Section 5.1.11, “Server SQL Modes” .

For a given table, the SHOW CREATE TABLE statement displays which columns have an explicit DEFAULT clause.



Implicit defaults are defined as follows:

• For numeric types, the default is 0, with the exception that for integer or floating-point types declared with the AUTO\_INCREMENT attribute, the default is the next value in the sequence.

• For date and time types other than [TIMESTAMP](#_bookmark109), the default is the appropriate “zero” value for the type. This is also true for [TIMESTAMP](#_bookmark109) if the explicit\_defaults\_for\_timestamp

system variable is enabled (see Section 5. 1.8, “Server System Variables” ). Otherwise, for the first [TIMESTAMP](#_bookmark109) column in a table, the default value is the current date and time. See [Section 11.2, “Date](#_bookmark107) [and Time Data Types”](#_bookmark107) .

• For string types other than [ENUM](#_bookmark62), the default value is the empty string. For [ENUM](#_bookmark62), the default is the first enumeration value.

**11.7** **Data** **Type** **Storage** **Requirements**

• [InnoDB Table Storage Requirements](#_bookmark167)

• [NDB Table Storage Requirements](#_bookmark168)

• [Numeric Type Storage Requirements](#_bookmark169)

• [Date and Time Type Storage Requirements](#_bookmark170)

• [String Type Storage Requirements](#_bookmark82)

• [Spatial Type Storage Requirements](#_bookmark171)

• [JSON Storage Requirements](#_bookmark172)

The storage requirements for table data on disk depend on several factors. Different storage engines represent data types and store raw data differently. Table data might be compressed, either for a column or an entire row, complicating the calculation of storage requirements for a table or column.

Despite differences in storage layout on disk, the internal MySQL APIs that communicate and exchange information about table rows use a consistent data structure that applies across all storage engines.

This section includes guidelines and information for the storage requirements for each data type supported by MySQL, including the internal format and size for storage engines that use a fixed-size representation for data types. Information is listed by category or storage engine.

The internal representation of a table has a maximum row size of 65,535 bytes, even if the storage engine is capable of supporting larger rows. This figure excludes [BLOB](#_bookmark61) or [TEXT](#_bookmark61) columns, which contribute only 9 to 12 bytes toward this size. For [BLOB](#_bookmark61) and [TEXT](#_bookmark61) data, the information is stored internally in a different area of memory than the row buffer. Different storage engines handle the allocation and storage of this data in different ways, according to the method they use for handling the corresponding types. For more information, see Chapter 16, *Alternative* *Storage* *Engines*, and Section 8.4.7, “Limits on Table Column Count and Row Size” .

**InnoDB** **Table** **Storage** **Requirements**

See Section 15.10, “InnoDB Row Formats” for information about storage requirements for InnoDB tables.

**NDB** **Table** **Storage** **Requirements**

**Important**

NDB tables use *4-byte* *alignment*; all NDB data storage is done in multiples of 4 bytes. Thus, a column value that would typically take 15 bytes requires 16

bytes in an NDB table. For example, in NDB tables, the [TINYINT](#_bookmark101), [SMALLINT](#_bookmark101), [MEDIUMINT](#_bookmark101), and [INTEGER](#_bookmark101) ([INT](#_bookmark101)) column types each require 4 bytes storage per record due to the alignment factor.

Each BIT(*M*) column takes *M* bits of storage space. Although an individual [BIT](#_bookmark104) column is *not* 4-byte aligned, NDB reserves 4 bytes (32 bits) per row for the first 1-32 bits needed for BIT columns, then another 4 bytes for bits 33-64, and so on.

While a NULL itself does not require any storage space, NDB reserves 4 bytes per row if the table definition contains any columns allowing NULL, up to 32 NULL columns. (If an NDB Cluster table is defined with more than 32 NULL columns up to 64 NULL columns, then 8 bytes per row are reserved.)

Every table using the NDB storage engine requires a primary key; if you do not define a primary key, a “hidden” primary key is created by NDB. This hidden primary key consumes 31-35 bytes per table record.

You can use the ndb\_size.pl Perl script to estimate NDB storage requirements. It connects to a current MySQL (not NDB Cluster) database and creates a report on how much space that database would require if it used the NDB storage engine. See Section 23.5.28, “ndb\_size.pl — NDBCLUSTER Size Requirement Estimator” for more information.

**Numeric** **Type** **Storage** **Requirements**

|  |  |
| --- | --- |
| **Data** **Type** | **Storage** **Required** |
| [TINYINT](#_bookmark101) | 1 byte |
| [SMALLINT](#_bookmark101) | 2 bytes |
| [MEDIUMINT](#_bookmark101) | 3 bytes |
| [INT](#_bookmark101), [INTEGER](#_bookmark101) | 4 bytes |
| [BIGINT](#_bookmark101) | 8 bytes |
| FLOAT(*p*) | 4 bytes if 0 <= *p* <= 24, 8 bytes if 25 <= *p* <= 53 |
| [FLOAT](#_bookmark103) | 4 bytes |
| DOUBLE [PRECISION], [REAL](#_bookmark103) | 8 bytes |
| DECIMAL(*M*,*D*), NUMERIC(*M*,*D*) | Varies; see following discussion |
| BIT(*M*) | approximately (*M*+7)/8 bytes |

Values for [DECIMAL](#_bookmark102) (and [NUMERIC](#_bookmark102)) columns are represented using a binary format that packs nine decimal (base 10) digits into four bytes. Storage for the integer and fractional parts of each value are determined separately. Each multiple of nine digits requires four bytes, and the “leftover” digits require some fraction of four bytes. The storage required for excess digits is given by the following table.

|  |  |
| --- | --- |
| **Leftover** **Digits** | **Number** **of** **Bytes** |
| 0 | 0 |
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 6 | 3 |
| 7 | 4 |
| 8 | 4 |

**Date** **and** **Time** **Type** **Storage** **Requirements**

For [TIME](#_bookmark110), [DATETIME](#_bookmark109), and [TIMESTAMP](#_bookmark109) columns, the storage required for tables created before MySQL 5.6.4 differs from tables created from 5.6.4 on. This is due to a change in 5.6.4 that permits these types to have a fractional part, which requires from 0 to 3 bytes.

|  |  |  |
| --- | --- | --- |
| **Data** **Type** | **Storage** **Required** **Before**  **MySQL** **5.6.4** | **Storage** **Required** **as** **of** **MySQL**  **5.6.4** |
| [YEAR](#_bookmark111) | 1 byte | 1 byte |
| [DATE](#_bookmark109) | 3 bytes | 3 bytes |
| [TIME](#_bookmark110) | 3 bytes | 3 bytes + fractional seconds storage |
| [DATETIME](#_bookmark109) | 8 bytes | 5 bytes + fractional seconds storage |
| [TIMESTAMP](#_bookmark109) | 4 bytes | 4 bytes + fractional seconds storage |

As of MySQL 5.6.4, storage for [YEAR](#_bookmark111) and [DATE](#_bookmark109) remains unchanged. However, [TIME](#_bookmark110), [DATETIME](#_bookmark109), and [TIMESTAMP](#_bookmark109) are represented differently. [DATETIME](#_bookmark109) is packed more efficiently, requiring 5 rather than 8 bytes for the nonfractional part, and all three parts have a fractional part that requires from 0 to 3 bytes, depending on the fractional seconds precision of stored values.

|  |  |
| --- | --- |
| **Fractional** **Seconds** **Precision** | **Storage** **Required** |
| 0 | 0 bytes |
| 1, 2 | 1 byte |
| 3, 4 | 2 bytes |
| 5, 6 | 3 bytes |

For example, [TIME(0)](#_bookmark110), [TIME(2)](#_bookmark110), [TIME(4)](#_bookmark110), and [TIME(6)](#_bookmark110) use 3, 4, 5, and 6 bytes, respectively. [TIME](#_bookmark110) and [TIME(0)](#_bookmark110) are equivalent and require the same storage.

For details about internal representation of temporal values, see [MySQL Internals: Important](https://dev.mysql.com/doc/internals/en/algorithms.html) [Algorithms and Structures](https://dev.mysql.com/doc/internals/en/algorithms.html).

**String** **Type** **Storage** **Requirements**

In the following table, *M* represents the declared column length in characters for nonbinary string types and bytes for binary string types. *L* represents the actual length in bytes of a given string value.

|  |  |
| --- | --- |
| **Data** **Type** | **Storage** **Required** |
| CHAR(*M*) | The compact family of InnoDB row formats optimize storage for variable-length character sets. See COMPACT Row Format Storage Characteristics. Otherwise, *M* × *w* bytes, <= *M* <= 255, where *w* is the number of bytes required for the maximum-length character in the character set. |
| BINARY(*M*) | *M* bytes, 0 <= *M* <= 255 |
| VARCHAR(*M*), VARBINARY(*M*) | *L* + 1 bytes if column values require 0 − 255 bytes, *L* + 2 bytes if values may require more than 255 bytes |
| [TINYBLOB](#_bookmark61), [TINYTEXT](#_bookmark61) | *L* + 1 bytes, where *L* < 28 |

|  |  |
| --- | --- |
| **Data** **Type** | **Storage** **Required** |
| [BLOB](#_bookmark61), [TEXT](#_bookmark61) | *L* + 2 bytes, where *L* < 216 |
| [MEDIUMBLOB](#_bookmark61), [MEDIUMTEXT](#_bookmark61) | *L* + 3 bytes, where *L* < 224 |
| [LONGBLOB](#_bookmark61), [LONGTEXT](#_bookmark61) | *L* + 4 bytes, where *L* < 232 |
| ENUM('*value1*','*value2*',...) | 1 or 2 bytes, depending on the number of enumeration values (65,535 values maximum) |
| SET('*value1*','*value2*',...) | 1, 2, 3, 4, or 8 bytes, depending on the number of set members (64 members maximum) |

Variable-length string types are stored using a length prefix plus data. The length prefix requires from one to four bytes depending on the data type, and the value of the prefix is *L* (the byte length of the string). For example, storage for a [MEDIUMTEXT](#_bookmark61) value requires *L* bytes to store the value plus three bytes to store the length of the value.

To calculate the number of bytes used to store a particular [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), or [TEXT](#_bookmark61) column value, you must take into account the character set used for that column and whether the value contains multibyte characters. In particular, when using a UTF-8 Unicode character set, you must keep in mind that not all characters use the same number of bytes. utf8mb3 and utf8mb4 character sets can require up to three and four bytes per character, respectively. For a breakdown of the storage used for different categories of utf8mb3 or utf8mb4 characters, see [Section 10.9, “Unicode Support”](#_bookmark29) .

[VARCHAR](#_bookmark60), [VARBINARY](#_bookmark73), and the [BLOB](#_bookmark61) and [TEXT](#_bookmark61) types are variable-length types. For each, the storage requirements depend on these factors:

• The actual length of the column value

• The column's maximum possible length

• The character set used for the column, because some character sets contain multibyte characters

For example, a VARCHAR(255) column can hold a string with a maximum length of 255 characters. Assuming that the column uses the latin1 character set (one byte per character), the actual storage required is the length of the string (*L*), plus one byte to record the length of the string. For the string 'abcd', *L* is 4 and the storage requirement is five bytes. If the same column is instead declared to use the ucs2 double-byte character set, the storage requirement is 10 bytes: The length of 'abcd' is eight bytes and the column requires two bytes to store lengths because the maximum length is greater than 255 (up to 510 bytes).

The effective maximum number of *bytes* that can be stored in a [VARCHAR](#_bookmark60) or [VARBINARY](#_bookmark73) column is subject to the maximum row size of 65,535 bytes, which is shared among all columns. For a [VARCHAR](#_bookmark60) column that stores multibyte characters, the effective maximum number of *characters* is less. For example, utf8mb4 characters can require up to four bytes per character, so a [VARCHAR](#_bookmark60) column that uses the utf8mb4 character set can be declared to be a maximum of 16,383 characters. See Section 8.4.7, “Limits on Table Column Count and Row Size” .

InnoDB encodes fixed-length fields greater than or equal to 768 bytes in length as variable-length fields, which can be stored off-page. For example, a CHAR(255) column can exceed 768 bytes if the maximum byte length of the character set is greater than 3, as it is with utf8mb4.

The NDB storage engine supports variable-width columns. This means that a [VARCHAR](#_bookmark60) column in an NDB Cluster table requires the same amount of storage as would any other storage engine, with the exception that such values are 4-byte aligned. Thus, the string 'abcd' stored in a VARCHAR(50) column using the latin1 character set requires 8 bytes (rather than 5 bytes for the same column value in a MyISAM table).

[TEXT](#_bookmark61), [BLOB](#_bookmark61), and [JSON](#_bookmark130) columns are implemented differently in the NDB storage engine, wherein each row in the column is made up of two separate parts. One of these is of fixed size (256 bytes for TEXT and BLOB, 4000 bytes for JSON), and is actually stored in the original table. The other consists of any

data in excess of 256 bytes, which is stored in a hidden blob parts table. The size of the rows in this second table are determined by the exact type of the column, as shown in the following table:

|  |  |
| --- | --- |
| **Type** | **Blob** **Part** **Size** |
| [BLOB](#_bookmark61), [TEXT](#_bookmark61) | 2000 |
| [MEDIUMBLOB](#_bookmark61), [MEDIUMTEXT](#_bookmark61) | 4000 |
| [LONGBLOB](#_bookmark61), [LONGTEXT](#_bookmark61) | 13948 |
| [JSON](#_bookmark130) | 8100 |

This means that the size of a [TEXT](#_bookmark61) column is 256 if *size* <= 256 (where *size* represents the size of the row); otherwise, the size is 256 + *size* + (2000 × (*size* − 256) % 2000).

No blob parts are stored separately by NDB for TINYBLOB or TINYTEXT column values.

You can increase the size of an NDB blob column's blob part to the maximum of 13948 using NDB\_COLUMN in a column comment when creating or altering the parent table. In NDB 8.0.30 and later, it is also possible to set the inline size for a TEXT, BLOB, or JSON column, using NDB\_TABLE in a column comment. See NDB\_COLUMN Options, for more information.

The size of an [ENUM](#_bookmark62) object is determined by the number of different enumeration values. One byte is used for enumerations with up to 255 possible values. Two bytes are used for enumerations having between 256 and 65,535 possible values. See [Section 11.3.5, “The ENUM Type”](#_bookmark62) .

The size of a [SET](#_bookmark63) object is determined by the number of different set members. If the set size is *N*, the object occupies (*N*+7)/8 bytes, rounded up to 1, 2, 3, 4, or 8 bytes. A [SET](#_bookmark63) can have a maximum of 64 members. See [Section 11.3.6, “The SET Type”](#_bookmark63) .

**Spatial** **Type** **Storage** **Requirements**

MySQL stores geometry values using 4 bytes to indicate the SRID followed by the WKB representation of the value. The LENGTH() function returns the space in bytes required for value storage.

For descriptions of WKB and internal storage formats for spatial values, see [Section 11.4.3, “Supported](#_bookmark121) [Spatial Data Formats”](#_bookmark121) .

**JSON** **Storage** **Requirements**

In general, the storage requirement for a [JSON](#_bookmark130) column is approximately the same as for a LONGBLOB or LONGTEXT column; that is, the space consumed by a JSON document is roughly the same as it would be for the document's string representation stored in a column of one of these types. However, there is an overhead imposed by the binary encoding, including metadata and dictionaries needed for lookup, of the individual values stored in the JSON document. For example, a string stored in a JSON document requires 4 to 10 bytes additional storage, depending on the length of the string and the size of the object or array in which it is stored.

In addition, MySQL imposes a limit on the size of any JSON document stored in a JSON column such that it cannot be any larger than the value of max\_allowed\_packet.

**11.8** **Choosing** **the** **Right** **Type** **for** **a** **Column**

For optimum storage, you should try to use the most precise type in all cases. For example, if an integer column is used for values in the range from 1 to 99999, MEDIUMINT UNSIGNED is the best type. Of the types that represent all the required values, this type uses the least amount of storage.

All basic calculations (+, -, \*, and /) with [DECIMAL](#_bookmark102) columns are done with precision of 65 decimal (base 10) digits. See [Section 11.1.1, “Numeric Data Type Syntax”](#_bookmark100) .

If accuracy is not too important or if speed is the highest priority, the [DOUBLE](#_bookmark103) type may be good enough. For high precision, you can always convert to a fixed-point type stored in a [BIGINT](#_bookmark101). This enables you to do all calculations with 64-bit integers and then convert results back to floating-point values as necessary.

**11.9** **Using** **Data** **Types** **from** **Other** **Database** **Engines**

To facilitate the use of code written for SQL implementations from other vendors, MySQL maps data types as shown in the following table. These mappings make it easier to import table definitions from

other database systems into MySQL.

|  |  |
| --- | --- |
| **Other** **Vendor** **Type** | **MySQL** **Type** |
| [BOOL](#_bookmark101) | [TINYINT](#_bookmark101) |
| [BOOLEAN](#_bookmark101) | [TINYINT](#_bookmark101) |
| CHARACTER VARYING(*M*) | VARCHAR(*M*) |
| [FIXED](#_bookmark102) | [DECIMAL](#_bookmark102) |
| [FLOAT4](#_bookmark103) | [FLOAT](#_bookmark103) |
| [FLOAT8](#_bookmark103) | [DOUBLE](#_bookmark103) |
| [INT1](#_bookmark101) | [TINYINT](#_bookmark101) |
| [INT2](#_bookmark101) | [SMALLINT](#_bookmark101) |
| [INT3](#_bookmark101) | [MEDIUMINT](#_bookmark101) |
| [INT4](#_bookmark101) | [INT](#_bookmark101) |
| INT8 | [BIGINT](#_bookmark101) |
| LONG VARBINARY | [MEDIUMBLOB](#_bookmark61) |
| LONG VARCHAR | [MEDIUMTEXT](#_bookmark61) |
| LONG | [MEDIUMTEXT](#_bookmark61) |
| [MIDDLEINT](#_bookmark101) | [MEDIUMINT](#_bookmark101) |
| [NUMERIC](#_bookmark102) | [DECIMAL](#_bookmark102) |

Data type mapping occurs at table creation time, after which the original type specifications are discarded. If you create a table with types used by other vendors and then issue a DESCRIBE *tbl\_name* statement, MySQL reports the table structure using the equivalent MySQL types. For example:

mysql> **CREATE** **TABLE** **t** **(a** **BOOL,** **b** **FLOAT8,** **c** **LONG** **VARCHAR,** **d** **NUMERIC);**

Query OK, 0 rows affected (0.00 sec)

mysql> **DESCRIBE** **t;**

+-------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+-------+---------------+------+-----+---------+-------+

| a | tinyint(1) | YES | | NULL | |

| b | double | YES | | NULL | |

| c | mediumtext | YES | | NULL | |

| d | decimal(10,0) | YES | | NULL | |

+-------+---------------+------+-----+---------+-------+

4 rows in set (0.01 sec)

Chapter 12 Functions and Operators

**Table** **of** **Contents**

[12.1 Built-In Function Operator Reference](#_bookmark173)and [2133](#_bookmark173)

[12.2 Loadable Function Reference](#_bookmark174) [2153](#_bookmark174)

[12.3 Type Conversion Expression Evaluation](#_bookmark75)in [2157](#_bookmark75)

[12.4 Operators](#_bookmark175) [2160](#_bookmark175)

[12.4.1 Operator Precedence](#_bookmark176) [2161](#_bookmark176)

[12.4.2 Comparison Functions Operators](#_bookmark177)and [2162](#_bookmark177)

[12.4.3 Logical Operators](#_bookmark178) [2168](#_bookmark178)

[12.4.4 Assignment Operators](#_bookmark179) [2170](#_bookmark179)

[12.5 Flow Control Functions](#_bookmark180) [2171](#_bookmark180)

12.6 Numeric Functions and Operators 2175

12.6.1 Arithmetic Operators 2176

12.6.2 Mathematical Functions 2178

12.7 Date and Time Functions 2187

12.8 String Functions and Operators 2209

12.8.1 String Comparison Functions and Operators 2225

12.8.2 Regular Expressions 2228

12.8.3 Character Set and Collation of Function Results 2238

12.9 What Calendar Is Used By MySQL? 2239

12.10 Full-Text Search Functions 2239

12.10.1 Natural Language Full-Text Searches 2241

12.10.2 Boolean Full-Text Searches 2244

12.10.3 Full-Text Searches with Query Expansion 2249

12.10.4 Full-Text Stopwords 2250

12.10.5 Full-Text Restrictions 2254

12.10.6 Fine-Tuning MySQL Full-Text Search 2255

12.10.7 Adding a User-Defined Collation for Full-Text Indexing 2258

12.10.8 ngram Full-Text Parser 2260

12.10.9 MeCab Full-Text Parser Plugin 2262

12.11 Cast Functions and Operators 2266

12.12 XML Functions 2280

12.13 Bit Functions and Operators 2290

12.14 Encryption and Compression Functions 2301

12.15 Locking Functions 2310

12.16 Information Functions 2312

12.17 Spatial Analysis Functions 2323

12.17.1 Spatial Function Reference 2324

12.17.2 Argument Handling by Spatial Functions 2327

12.17.3 Functions That Create Geometry Values from WKT Values 2328

12.17.4 Functions That Create Geometry Values from WKB Values 2330

12.17.5 MySQL-Specific Functions That Create Geometry Values 2332

12.17.6 Geometry Format Conversion Functions 2333

12.17.7 Geometry Property Functions 2335

12.17.8 Spatial Operator Functions 2347

12.17.9 Functions That Test Spatial Relations Between Geometry Objects 2355

12.17.10 Spatial Geohash Functions 2362

12.17.11 Spatial GeoJSON Functions 2364

12.17.12 Spatial Aggregate Functions 2366

12.17.13 Spatial Convenience Functions 2368

12.18 JSON Functions 2373

12.18.1 JSON Function Reference 2373

12.18.2 Functions That Create JSON Values 2375

12.18.3 Functions That Search JSON Values 2376



12.18.4 Functions That Modify JSON Values 2391

12.18.5 Functions That Return JSON Value Attributes 2400

12.18.6 JSON Table Functions 2402

12.18.7 JSON Schema Validation Functions 2407

12.18.8 JSON Utility Functions 2413

12.19 Functions Used with Global Transaction Identifiers (GTIDs) 2418

12.20 Aggregate Functions 2420

12.20.1 Aggregate Function Descriptions 2420

12.20.2 GROUP BY Modifiers 2430

12.20.3 MySQL Handling of GROUP BY 2435

12.20.4 Detection of Functional Dependence 2439

12.21 Window Functions 2442

12.21.1 Window Function Descriptions 2442

12.21.2 Window Function Concepts and Syntax 2448

12.21.3 Window Function Frame Specification 2452

12.21.4 Named Windows 2455

12.21.5 Window Function Restrictions 2456

12.22 Performance Schema Functions 2457

12.23 Internal Functions 2460

12.24 Miscellaneous Functions 2461

12.25 Precision Math 2477

12.25.1 Types of Numeric Values 2478

12.25.2 DECIMAL Data Type Characteristics 2478

12.25.3 Expression Handling 2479

12.25.4 Rounding Behavior 2481

12.25.5 Precision Math Examples 2482

Expressions can be used at several points in SQL statements, such as in the ORDER BY or HAVING clauses of SELECT statements, in the WHERE clause of a SELECT, DELETE, or UPDATE statement, or in SET statements. Expressions can be written using values from several sources, such as literal values, column values, NULL, variables, built-in functions and operators, loadable functions, and stored functions (a type of stored object).

This chapter describes the built-in functions and operators that are permitted for writing expressions in MySQL. For information about loadable functions and stored functions, see Section 5.7, “MySQL Server Loadable Functions” , and Section 25.2, “Using Stored Routines” . For the rules describing how the server interprets references to different kinds of functions, see Section 9.2.5, “Function Name Parsing and Resolution” .

An expression that contains NULL always produces a NULL value unless otherwise indicated in the documentation for a particular function or operator.

**Note**

By default, there must be no whitespace between a function name and the parenthesis following it. This helps the MySQL parser distinguish between function calls and references to tables or columns that happen to have the same name as a function. However, spaces around function arguments are permitted.

To tell the MySQL server to accept spaces after function names by starting it with the --sql-mode=IGNORE\_SPACE option. (See Section 5.1.11, “Server SQL Modes” .) Individual client programs can request this behavior by using the CLIENT\_IGNORE\_SPACE option for [mysql\_real\_connect()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-connect.html). In either case, all function names become reserved words.

For the sake of brevity, some examples in this chapter display the output from the mysql program in abbreviated form. Rather than showing examples in this format:

mysql> **SELECT** **MOD(29,9);**

+-----------+

| mod(29,9) |

+-----------+

| 2 |

+-----------+

1 rows in set (0.00 sec)

This format is used instead:

mysql> **SELECT** **MOD(29,9);**

-> 2

**12.1** **Built-In** **Function** **and** **Operator** **Reference**

The following table lists each built-in (native) function and operator and provides a short description of each one. For a table listing functions that are loadable at runtime, see [Section 12.2, “Loadable](#_bookmark174) [Function Reference”](#_bookmark174) .

**Table** **12.1** **Built-In** **Functions** **and** **Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| & | Bitwise AND |  |  |
| [>](#_bookmark156) | Greater than operator |  |  |
| >> | Right shift |  |  |
| [>=](#_bookmark157) | Greater than or equal operator |  |  |
| [<](#_bookmark154) | Less than operator |  |  |
| [<>, !=](#_bookmark158) | Not equal operator |  |  |
| << | Left shift |  |  |
| [<=](#_bookmark155) | Less than or equal  operator |  |  |
| [<=>](#_bookmark159) | NULL-safe equal to  operator |  |  |
| %, MOD | Modulo operator |  |  |
| \* | Multiplication operator |  |  |
| + | Addition operator |  |  |
| - | Minus operator |  |  |
| - | Change the sign of the argument |  |  |
| -> | Return value from JSON column after evaluating path; equivalent to JSON\_EXTRACT(). |  |  |
| ->> | Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON\_UNQUOTE(JSON | \_EXTRACT()). |  |
| / | Division operator |  |  |
| [:=](#_bookmark181) | Assign a value |  |  |
| [=](#_bookmark182) | Assign a value (as part of a SET statement, or as part of the SET |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
|  | clause in an UPDATE statement) |  |  |
| [=](#_bookmark153) | Equal operator |  |  |
| ^ | Bitwise XOR |  |  |
| ABS() | Return the absolute  value |  |  |
| ACOS() | Return the arc cosine |  |  |
| ADDDATE() | Add time values  (intervals) to a date  value |  |  |
| ADDTIME() | Add time |  |  |
| AES\_DECRYPT() | Decrypt using AES |  |  |
| AES\_ENCRYPT() | Encrypt using AES |  |  |
| [AND, &&](#_bookmark183) | Logical AND |  |  |
| ANY\_VALUE() | Suppress ONLY\_FULL\_GROUP\_BY value rejection |  |  |
| ASCII() | Return numeric value of left-most character |  |  |
| ASIN() | Return the arc sine |  |  |
| ATAN() | Return the arc tangent |  |  |
| ATAN2(), ATAN() | Return the arc tangent of the two arguments |  |  |
| AVG() | Return the average value of the argument |  |  |
| BENCHMARK() | Repeatedly execute an expression |  |  |
| [BETWEEN ...](#_bookmark160)  [AND ...](#_bookmark160) | Whether a value is within a range of values |  |  |
| BIN() | Return a string  containing binary  representation of a  number |  |  |
| BIN\_TO\_UUID() | Convert binary UUID to string |  |  |
| BINARY | Cast a string to a binary string |  | 8.0.27 |
| BIT\_AND() | Return bitwise AND |  |  |
| BIT\_COUNT() | Return the number of bits that are set |  |  |
| BIT\_LENGTH() | Return length of  argument in bits |  |  |
| BIT\_OR() | Return bitwise OR |  |  |
| BIT\_XOR() | Return bitwise XOR |  |  |
| CAN\_ACCESS\_COLUMN() | Internal use only |  |  |
| CAN\_ACCESS\_DATABASE | I(n)ternal use only |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| CAN\_ACCESS\_TABLE() | Internal use only |  |  |
| CAN\_ACCESS\_USER() | Internal use only | 8.0.22 |  |
| CAN\_ACCESS\_VIEW() | Internal use only |  |  |
| [CASE](#_bookmark184) | Case operator |  |  |
| CAST() | Cast a value as a  certain type |  |  |
| CEIL() | Return the smallest integer value not less than the argument |  |  |
| CEILING() | Return the smallest integer value not less than the argument |  |  |
| CHAR() | Return the character for each integer passed |  |  |
| CHAR\_LENGTH() | Return number of characters in argument |  |  |
| CHARACTER\_LENGTH() | Synonym for  CHAR\_LENGTH() |  |  |
| CHARSET() | Return the character set of the argument |  |  |
| [COALESCE()](#_bookmark185) | Return the first non-  NULL argument |  |  |
| COERCIBILITY() | Return the collation coercibility value of the string argument |  |  |
| COLLATION() | Return the collation of the string argument |  |  |
| COMPRESS() | Return result as a binary string |  |  |
| CONCAT() | Return concatenated string |  |  |
| CONCAT\_WS() | Return concatenate with separator |  |  |
| CONNECTION\_ID() | Return the connection ID (thread ID) for the connection |  |  |
| CONV() | Convert numbers  between different  number bases |  |  |
| CONVERT() | Cast a value as a  certain type |  |  |
| CONVERT\_TZ() | Convert from one time zone to another |  |  |
| COS() | Return the cosine |  |  |
| COT() | Return the cotangent |  |  |
| COUNT() | Return a count of the number of rows returned |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| COUNT(DISTINCT) | Return the count of a number of different values |  |  |
| CRC32() | Compute a cyclic redundancy check value |  |  |
| CUME\_DIST() | Cumulative distribution value |  |  |
| CURDATE() | Return the current date |  |  |
| CURRENT\_DATE(),  CURRENT\_DATE | Synonyms for  CURDATE() |  |  |
| CURRENT\_ROLE() | Return the current active roles |  |  |
| CURRENT\_TIME(),  CURRENT\_TIME | Synonyms for  CURTIME() |  |  |
| CURRENT\_TIMESTAMP()  CURRENT\_TIMESTAMP | S,ynonyms for NOW() |  |  |
| CURRENT\_USER(),  CURRENT\_USER | The authenticated user name and host name |  |  |
| CURTIME() | Return the current time |  |  |
| DATABASE() | Return the default (current) database name |  |  |
| DATE() | Extract the date part of a date or datetime expression |  |  |
| DATE\_ADD() | Add time values  (intervals) to a date  value |  |  |
| DATE\_FORMAT() | Format date as specified |  |  |
| DATE\_SUB() | Subtract a time value (interval) from a date |  |  |
| DATEDIFF() | Subtract two dates |  |  |
| DAY() | Synonym for  DAYOFMONTH() |  |  |
| DAYNAME() | Return the name of the weekday |  |  |
| DAYOFMONTH() | Return the day of the month (0-31) |  |  |
| DAYOFWEEK() | Return the weekday index of the argument |  |  |
| DAYOFYEAR() | Return the day of the year (1-366) |  |  |
| DEFAULT() | Return the default value for a table column |  |  |
| DEGREES() | Convert radians to  degrees |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| DENSE\_RANK() | Rank of current row  within its partition,  without gaps |  |  |
| DIV | Integer division |  |  |
| ELT() | Return string at index number |  |  |
| EXP() | Raise to the power of |  |  |
| EXPORT\_SET() | Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string |  |  |
| EXTRACT() | Extract part of a date |  |  |
| ExtractValue() | Extract a value from an XML string using XPath notation |  |  |
| FIELD() | Index (position) of first argument in subsequent arguments |  |  |
| FIND\_IN\_SET() | Index (position) of first argument within second argument |  |  |
| FIRST\_VALUE() | Value of argument from first row of window frame |  |  |
| FLOOR() | Return the largest integer value not greater than the argument |  |  |
| FORMAT() | Return a number formatted to specified number of decimal places |  |  |
| FORMAT\_BYTES() | Convert byte count to value with units | 8.0.16 |  |
| FORMAT\_PICO\_TIME() | Convert time in picoseconds to value with units | 8.0.16 |  |
| FOUND\_ROWS() | For a SELECT with a LIMIT clause, the number of rows that would be returned were there no LIMIT clause |  |  |
| FROM\_BASE64() | Decode base64  encoded string and  return result |  |  |
| FROM\_DAYS() | Convert a day number to a date |  |  |
| FROM\_UNIXTIME() | Format Unix timestamp as a date |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| GeomCollection() | Construct geometry  collection from  geometries |  |  |
| GeometryCollection () | Construct geometry  collection from  geometries |  |  |
| GET\_DD\_COLUMN\_PRIVI | LInrEnSal(se only |  |  |
| GET\_DD\_CREATE\_OPTIO | NIne(r)nal use only |  |  |
| GET\_DD\_INDEX\_SUB\_PA | RIne\_rLnEaNl o()nly |  |  |
| GET\_FORMAT() | Return a date format string |  |  |
| GET\_LOCK() | Get a named lock |  |  |
| [GREATEST()](#_bookmark162) | Return the largest  argument |  |  |
| GROUP\_CONCAT() | Return a concatenated string |  |  |
| GROUPING() | Distinguish super- aggregate ROLLUP rows from regular rows |  |  |
| GTID\_SUBSET() | Return true if all GTIDs in subset are also in set; otherwise false. |  |  |
| GTID\_SUBTRACT() | Return all GTIDs in set that are not in subset. |  |  |
| HEX() | Hexadecimal  representation of decimal or string value |  |  |
| HOUR() | Extract the hour |  |  |
| ICU\_VERSION() | ICU library version |  |  |
| IF() | If/else construct |  |  |
| IFNULL() | Null if/else construct |  |  |
| [IN()](#_bookmark161) | Whether a value is within a set of values |  |  |
| INET\_ATON() | Return the numeric value of an IP address |  |  |
| INET\_NTOA() | Return the IP address from a numeric value |  |  |
| INET6\_ATON() | Return the numeric  value of an IPv6  address |  |  |
| INET6\_NTOA() | Return the IPv6 address from a numeric value |  |  |
| INSERT() | Insert substring at specified position up to specified number of characters |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| INSTR() | Return the index of the first occurrence of substring |  |  |
| INTERNAL\_AUTO\_INCRE | MInrTna()luse only |  |  |
| INTERNAL\_AVG\_ROW\_LE | NInrHna()luse only |  |  |
| INTERNAL\_CHECK\_TIME | I(n)ternal use only |  |  |
| INTERNAL\_CHECKSUM() | Internal use only |  |  |
| INTERNAL\_DATA\_FREE () | Internal use only |  |  |
| INTERNAL\_DATA\_LENGT | HIne)rnal use only |  |  |
| INTERNAL\_DD\_CHAR\_LE | NInrHna()luse only |  |  |
| INTERNAL\_GET\_COMMEN | TInt\_rRna\_El l)y |  |  |
| INTERNAL\_GET\_ENABLE | DInt\_rOnLaElu\_eSly() | 8.0.19 |  |
| INTERNAL\_GET\_HOSTNA | MIne(r)nal use only | 8.0.19 |  |
| INTERNAL\_GET\_USERNA | MIne(r)nal use only | 8.0.19 |  |
| INTERNAL\_GET\_VIEW\_W | AInrInNaGlu\_o\_RlyROR() |  |  |
| INTERNAL\_INDEX\_COLU | MIne\_rCnAaRl AolIyTY() |  |  |
| INTERNAL\_INDEX\_LENG | TIne(r)nal use only |  |  |
| INTERNAL\_IS\_ENABLED | InRt\_rLnEal(se only | 8.0.19 |  |
| INTERNAL\_IS\_MANDATO | RIne\_rRnOaLl s(e)only | 8.0.19 |  |
| INTERNAL\_KEYS\_DISAB | LInr(n)al use only |  |  |
| INTERNAL\_MAX\_DATA\_L | EInrTnHal(se only |  |  |
| INTERNAL\_TABLE\_ROWS | I(n)ternal use only |  |  |
| INTERNAL\_UPDATE\_TIM | EIne)rnal use only |  |  |
| [INTERVAL()](#_bookmark186) | Return the index of the argument that is less than the first argument |  |  |
| [IS](#_bookmark187) | Test a value against a boolean |  |  |
| IS\_FREE\_LOCK() | Whether the named lock is free |  |  |
| IS\_IPV4() | Whether argument is an IPv4 address |  |  |
| IS\_IPV4\_COMPAT() | Whether argument is an IPv4-compatible address |  |  |
| IS\_IPV4\_MAPPED() | Whether argument is an IPv4-mapped address |  |  |
| IS\_IPV6() | Whether argument is an IPv6 address |  |  |
| [IS NOT](#_bookmark188) | Test a value against a boolean |  |  |
| [IS NOT NULL](#_bookmark189) | NOT NULL value test |  |  |
| [IS NULL](#_bookmark190) | NULL value test |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| IS\_USED\_LOCK() | Whether the named lock is in use; return connection identifier if true |  |  |
| IS\_UUID() | Whether argument is a valid UUID |  |  |
| [ISNULL()](#_bookmark191) | Test whether the  argument is NULL |  |  |
| JSON\_ARRAY() | Create JSON array |  |  |
| JSON\_ARRAY\_APPEND() | Append data to JSON document |  |  |
| JSON\_ARRAY\_INSERT() | Insert into JSON array |  |  |
| JSON\_ARRAYAGG() | Return result set as a single JSON array |  |  |
| JSON\_CONTAINS() | Whether JSON  document contains specific object at path |  |  |
| JSON\_CONTAINS\_PATH () | Whether JSON  document contains any data at path |  |  |
| JSON\_DEPTH() | Maximum depth of  JSON document |  |  |
| JSON\_EXTRACT() | Return data from JSON document |  |  |
| JSON\_INSERT() | Insert data into JSON document |  |  |
| JSON\_KEYS() | Array of keys from  JSON document |  |  |
| JSON\_LENGTH() | Number of elements in JSON document |  |  |
| JSON\_MERGE() | Merge JSON documents, preserving duplicate keys. Deprecated synonym for JSON\_MERGE\_PRESE | RVE() | Yes |
| JSON\_MERGE\_PATCH() | Merge JSON documents, replacing values of duplicate keys |  |  |
| JSON\_MERGE\_PRESERVE | M()erge JSON documents, preserving duplicate keys |  |  |
| JSON\_OBJECT() | Create JSON object |  |  |
| JSON\_OBJECTAGG() | Return result set as a single JSON object |  |  |
| JSON\_OVERLAPS() | Compares two JSON documents, returns TRUE (1) if these have any key-value pairs or array elements in | 8.0.17 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
|  | common, otherwise  FALSE (0) |  |  |
| JSON\_PRETTY() | Print a JSON document in human-readable format |  |  |
| JSON\_QUOTE() | Quote JSON document |  |  |
| JSON\_REMOVE() | Remove data from  JSON document |  |  |
| JSON\_REPLACE() | Replace values in JSON document |  |  |
| JSON\_SCHEMA\_VALID() | Validate JSON  document against JSON schema; returns TRUE/1 if document validates against schema, or FALSE/0 if it does not | 8.0.17 |  |
| JSON\_SCHEMA\_VALIDAT | Nlid\_aRtPJOSROTN() document against JSON schema; returns report in JSON format on outcome on validation including success or failure and reasons for failure | 8.0.17 |  |
| JSON\_SEARCH() | Path to value within  JSON document |  |  |
| JSON\_SET() | Insert data into JSON document |  |  |
| JSON\_STORAGE\_FREE() | Freed space within binary representation of JSON column value following partial update |  |  |
| JSON\_STORAGE\_SIZE() | Space used for storage of binary representation of a JSON document |  |  |
| JSON\_TABLE() | Return data from a JSON expression as a relational table |  |  |
| JSON\_TYPE() | Type of JSON value |  |  |
| JSON\_UNQUOTE() | Unquote JSON value |  |  |
| JSON\_VALID() | Whether JSON value is valid |  |  |
| JSON\_VALUE() | Extract value from  JSON document at  location pointed to  by path provided;  return this value as  VARCHAR(512) or  specified type | 8.0.21 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| LAG() | Value of argument from row lagging current row within partition |  |  |
| LAST\_DAY | Return the last day  of the month for the  argument |  |  |
| LAST\_INSERT\_ID() | Value of the  AUTOINCREMENT  column for the last  INSERT |  |  |
| LAST\_VALUE() | Value of argument from last row of window frame |  |  |
| LCASE() | Synonym for LOWER() |  |  |
| LEAD() | Value of argument from row leading current row within partition |  |  |
| [LEAST()](#_bookmark163) | Return the smallest  argument |  |  |
| LEFT() | Return the leftmost number of characters as specified |  |  |
| LENGTH() | Return the length of a string in bytes |  |  |
| LIKE | Simple pattern matching |  |  |
| LineString() | Construct LineString from Point values |  |  |
| LN() | Return the natural  logarithm of the  argument |  |  |
| LOAD\_FILE() | Load the named file |  |  |
| LOCALTIME(),  LOCALTIME | Synonym for NOW() |  |  |
| LOCALTIMESTAMP, LOCALTIMESTAMP() | Synonym for NOW() |  |  |
| LOCATE() | Return the position of the first occurrence of substring |  |  |
| LOG() | Return the natural  logarithm of the first  argument |  |  |
| LOG10() | Return the base-10  logarithm of the  argument |  |  |
| LOG2() | Return the base-2  logarithm of the  argument |  |  |
| LOWER() | Return the argument in lowercase |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| LPAD() | Return the string argument, left-padded with the specified string |  |  |
| LTRIM() | Remove leading spaces |  |  |
| MAKE\_SET() | Return a set of comma- separated strings that have the corresponding bit in bits set |  |  |
| MAKEDATE() | Create a date from the year and day of year |  |  |
| MAKETIME() | Create time from hour, minute, second |  |  |
| MASTER\_POS\_WAIT() | Block until the replica has read and applied all updates up to the specified position |  | 8.0.26 |
| MATCH() | Perform full-text search |  |  |
| MAX() | Return the maximum value |  |  |
| MBRContains() | Whether MBR of one  geometry contains MBR of another |  |  |
| MBRCoveredBy() | Whether one MBR is covered by another |  |  |
| MBRCovers() | Whether one MBR  covers another |  |  |
| MBRDisjoint() | Whether MBRs of two geometries are disjoint |  |  |
| MBREquals() | Whether MBRs of two geometries are equal |  |  |
| MBRIntersects() | Whether MBRs of two geometries intersect |  |  |
| MBROverlaps() | Whether MBRs of two geometries overlap |  |  |
| MBRTouches() | Whether MBRs of two geometries touch |  |  |
| MBRWithin() | Whether MBR of one geometry is within MBR of another |  |  |
| MD5() | Calculate MD5  checksum |  |  |
| MEMBER OF() | Returns true (1) if first operand matches any element of JSON array passed as second operand, otherwise returns false (0) | 8.0.17 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| MICROSECOND() | Return the  microseconds from  argument |  |  |
| MID() | Return a substring  starting from the  specified position |  |  |
| MIN() | Return the minimum value |  |  |
| MINUTE() | Return the minute from the argument |  |  |
| MOD() | Return the remainder |  |  |
| MONTH() | Return the month from the date passed |  |  |
| MONTHNAME() | Return the name of the month |  |  |
| MultiLineString() | Contruct MultiLineString from LineString values |  |  |
| MultiPoint() | Construct MultiPoint  from Point values |  |  |
| MultiPolygon() | Construct MultiPolygon from Polygon values |  |  |
| NAME\_CONST() | Cause the column to have the given name |  |  |
| [NOT, !](#_bookmark192) | Negates value |  |  |
| [NOT BETWEEN ...](#_bookmark193)  [AND ...](#_bookmark193) | Whether a value is not within a range of values |  |  |
| [NOT IN()](#_bookmark194) | Whether a value is not within a set of values |  |  |
| NOT LIKE | Negation of simple  pattern matching |  |  |
| NOT REGEXP | Negation of REGEXP |  |  |
| NOW() | Return the current date and time |  |  |
| NTH\_VALUE() | Value of argument from N-th row of window frame |  |  |
| NTILE() | Bucket number of current row within its partition. |  |  |
| NULLIF() | Return NULL if expr1 = expr2 |  |  |
| OCT() | Return a string  containing octal  representation of a  number |  |  |
| OCTET\_LENGTH() | Synonym for LENGTH() |  |  |
| [OR, ||](#_bookmark74) | Logical OR |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| ORD() | Return character code for leftmost character of the argument |  |  |
| PERCENT\_RANK() | Percentage rank value |  |  |
| PERIOD\_ADD() | Add a period to a year- month |  |  |
| PERIOD\_DIFF() | Return the number of months between periods |  |  |
| PI() | Return the value of pi |  |  |
| Point() | Construct Point from coordinates |  |  |
| Polygon() | Construct Polygon from LineString arguments |  |  |
| POSITION() | Synonym for LOCATE() |  |  |
| POW() | Return the argument raised to the specified power |  |  |
| POWER() | Return the argument raised to the specified power |  |  |
| PS\_CURRENT\_THREAD\_I | e(formance Schema thread ID for current thread | 8.0.16 |  |
| PS\_THREAD\_ID() | Performance Schema  thread ID for given  thread | 8.0.16 |  |
| QUARTER() | Return the quarter from a date argument |  |  |
| QUOTE() | Escape the argument for use in an SQL statement |  |  |
| RADIANS() | Return argument converted to radians |  |  |
| RAND() | Return a random  floating-point value |  |  |
| RANDOM\_BYTES() | Return a random byte vector |  |  |
| RANK() | Rank of current row within its partition, with gaps |  |  |
| REGEXP | Whether string matches regular expression |  |  |
| REGEXP\_INSTR() | Starting index of  substring matching  regular expression |  |  |
| REGEXP\_LIKE() | Whether string matches regular expression |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| REGEXP\_REPLACE() | Replace substrings  matching regular  expression |  |  |
| REGEXP\_SUBSTR() | Return substring  matching regular  expression |  |  |
| RELEASE\_ALL\_LOCKS() | Release all current  named locks |  |  |
| RELEASE\_LOCK() | Release the named lock |  |  |
| REPEAT() | Repeat a string the  specified number of  times |  |  |
| REPLACE() | Replace occurrences of a specified string |  |  |
| REVERSE() | Reverse the characters in a string |  |  |
| RIGHT() | Return the specified rightmost number of characters |  |  |
| RLIKE | Whether string matches regular expression |  |  |
| ROLES\_GRAPHML() | Return a GraphML document representing memory role subgraphs |  |  |
| ROUND() | Round the argument |  |  |
| ROW\_COUNT() | The number of rows  updated |  |  |
| ROW\_NUMBER() | Number of current row within its partition |  |  |
| RPAD() | Append string the  specified number of  times |  |  |
| RTRIM() | Remove trailing spaces |  |  |
| SCHEMA() | Synonym for  DATABASE() |  |  |
| SEC\_TO\_TIME() | Converts seconds to 'hh:mm:ss' format |  |  |
| SECOND() | Return the second  (0-59) |  |  |
| SESSION\_USER() | Synonym for USER() |  |  |
| SHA1(), SHA() | Calculate an SHA-1  160-bit checksum |  |  |
| SHA2() | Calculate an SHA-2  checksum |  |  |
| SIGN() | Return the sign of the argument |  |  |
| SIN() | Return the sine of the argument |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| SLEEP() | Sleep for a number of seconds |  |  |
| SOUNDEX() | Return a soundex string |  |  |
| SOUNDS LIKE | Compare sounds |  |  |
| SOURCE\_POS\_WAIT() | Block until the replica has read and applied all updates up to the specified position | 8.0.26 |  |
| SPACE() | Return a string of the specified number of spaces |  |  |
| SQRT() | Return the square root of the argument |  |  |
| ST\_Area() | Return Polygon or  MultiPolygon area |  |  |
| ST\_AsBinary(),  ST\_AsWKB() | Convert from internal geometry format to WKB |  |  |
| ST\_AsGeoJSON() | Generate GeoJSON  object from geometry |  |  |
| ST\_AsText(),  ST\_AsWKT() | Convert from internal geometry format to WKT |  |  |
| ST\_Buffer() | Return geometry of points within given distance from geometry |  |  |
| ST\_Buffer\_Strategy () | Produce strategy option for ST\_Buffer() |  |  |
| ST\_Centroid() | Return centroid as a point |  |  |
| ST\_Collect() | Aggregate spatial values into collection | 8.0.24 |  |
| ST\_Contains() | Whether one geometry contains another |  |  |
| ST\_ConvexHull() | Return convex hull of geometry |  |  |
| ST\_Crosses() | Whether one geometry crosses another |  |  |
| ST\_Difference() | Return point set  difference of two  geometries |  |  |
| ST\_Dimension() | Dimension of geometry |  |  |
| ST\_Disjoint() | Whether one geometry is disjoint from another |  |  |
| ST\_Distance() | The distance of one geometry from another |  |  |
| ST\_Distance\_Sphere () | Minimum distance on earth between two geometries |  |  |
| ST\_EndPoint() | End Point of LineString |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Description** | | **Introduced** | **Deprecated** |
| ST\_Envelope() | Return MBR of  geometry | |  |  |
| ST\_Equals() | Whether one geometry is equal to another | |  |  |
| ST\_ExteriorRing() | Return exterior ring of Polygon | |  |  |
| ST\_FrechetDistance () | The discrete Fréchet distance of one geometry from another | | 8.0.23 |  |
| ST\_GeoHash() | Produce a geohash  value | |  |  |
| ST\_GeomCollFromText ST\_GeometryCollecti ST\_GeomCollFromTxt () | R()e,turn geometry lFlercotiotm()W,KT | |  |  |
| ST\_GeomCollFromWKB () ST\_GeometryCollecti | R,eturn geometry lFlercotio(m)WKB | |  |  |
| ST\_GeometryN() | Return N-th geometry from geometry collection | |  |  |
| ST\_GeometryType() | Return name of  geometry type | |  |  |
| ST\_GeomFromGeoJSON () | Generate geometry from GeoJSON object | |  |  |
| ST\_GeomFromText(), ST\_GeometryFromText | Return geometry from W()KT | |  |  |
| ST\_GeomFromWKB(), ST\_GeometryFromWKB () | Return geometry from  WKB | |  |  |
| ST\_HausdorffDistanc | e) discrete Hausdorff distance of one geometry from another | | 8.0.23 |  |
| ST\_InteriorRingN() | Return N-th interior of Polygon | ring |  |  |
| ST\_Intersection() | Return point set  intersection of two  geometries | |  |  |
| ST\_Intersects() | Whether one geometry intersects another | |  |  |
| ST\_IsClosed() | Whether a geometry is closed and simple | |  |  |
| ST\_IsEmpty() | Whether a geometry is empty | |  |  |
| ST\_IsSimple() | Whether a geometry is simple | |  |  |
| ST\_IsValid() | Whether a geometry is valid | |  |  |
| ST\_LatFromGeoHash() | Return latitude from  geohash value | |  |  |
| ST\_Latitude() | Return latitude of Point | | 8.0.12 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| ST\_Length() | Return length of  LineString |  |  |
| ST\_LineFromText(), ST\_LineStringFromTe | Construct LineString rm()WKT |  |  |
| ST\_LineFromWKB(), ST\_LineStringFromWK | Construct LineString ro(m) WKB |  |  |
| ST\_LineInterpolateP | tpo(nt a given  percentage along a  LineString | 8.0.24 |  |
| ST\_LineInterpolateP | tpi(n)ts a given  percentage along a  LineString | 8.0.24 |  |
| ST\_LongFromGeoHash () | Return longitude from geohash value |  |  |
| ST\_Longitude() | Return longitude of  Point | 8.0.12 |  |
| ST\_MakeEnvelope() | Rectangle around two points |  |  |
| ST\_MLineFromText(), ST\_MultiLineStringF | Construct  oumltTiLeietS(t)ring from  WKT |  |  |
| ST\_MLineFromWKB(), ST\_MultiLineStringF | Construct  oumltWiLKie(S)tring from  WKB |  |  |
| ST\_MPointFromText()  ST\_MultiPointFromTe | ,  Construct MultiPoint  rm()WKT |  |  |
| ST\_MPointFromWKB(), ST\_MultiPointFromWK | Construct MultiPoint  ro(m) WKB |  |  |
| ST\_MPolyFromText(), ST\_MultiPolygonFrom | Construct MultiPolygon rtW()KT |  |  |
| ST\_MPolyFromWKB(), ST\_MultiPolygonFrom | Construct MultiPolygon r(W)KB |  |  |
| ST\_NumGeometries() | Return number of geometries in geometry collection |  |  |
| ST\_NumInteriorRing () ST\_NumInteriorRings  Return number of  i(n)terior rings in Polygon | , |  |  |
| ST\_NumPoints() | Return number of points in LineString |  |  |
| ST\_Overlaps() | Whether one geometry overlaps another |  |  |
| ST\_PointAtDistance () | The point a given  distance along a  LineString | 8.0.24 |  |
| ST\_PointFromGeoHash | C()onvert geohash value to POINT value |  |  |
| ST\_PointFromText() | Construct Point from  WKT |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| ST\_PointFromWKB() | Construct Point from  WKB |  |  |
| ST\_PointN() | Return N-th point from LineString |  |  |
| ST\_PolyFromText(), ST\_PolygonFromText () | Construct Polygon from  WKT |  |  |
| ST\_PolyFromWKB(), ST\_PolygonFromWKB() | Construct Polygon from  WKB |  |  |
| ST\_Simplify() | Return simplified  geometry |  |  |
| ST\_SRID() | Return spatial reference system ID for geometry |  |  |
| ST\_StartPoint() | Start Point of LineString |  |  |
| ST\_SwapXY() | Return argument with X/ Y coordinates swapped |  |  |
| ST\_SymDifference() | Return point set symmetric difference of two geometries |  |  |
| ST\_Touches() | Whether one geometry touches another |  |  |
| ST\_Transform() | Transform coordinates of geometry | 8.0.13 |  |
| ST\_Union() | Return point set union of two geometries |  |  |
| ST\_Validate() | Return validated  geometry |  |  |
| ST\_Within() | Whether one geometry is within another |  |  |
| ST\_X() | Return X coordinate of  Point |  |  |
| ST\_Y() | Return Y coordinate of  Point |  |  |
| STATEMENT\_DIGEST() | Compute statement  digest hash value |  |  |
| STATEMENT\_DIGEST\_TE | m()pute normalized statement digest |  |  |
| STD() | Return the population standard deviation |  |  |
| STDDEV() | Return the population standard deviation |  |  |
| STDDEV\_POP() | Return the population standard deviation |  |  |
| STDDEV\_SAMP() | Return the sample  standard deviation |  |  |
| STR\_TO\_DATE() | Convert a string to a date |  |  |
| STRCMP() | Compare two strings |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Description** | | **Introduced** | **Deprecated** |
| SUBDATE() | Synonym for  DATE\_SUB() when  invoked with three  arguments | |  |  |
| SUBSTR() | Return the specified | substring as |  |  |
| SUBSTRING() | Return the specified | substring as |  |  |
| SUBSTRING\_INDEX() | Return a substring from a string before the specified number of occurrences of the delimiter | |  |  |
| SUBTIME() | Subtract times | |  |  |
| SUM() | Return the sum | |  |  |
| SYSDATE() | Return the time at which the function executes | |  |  |
| SYSTEM\_USER() | Synonym for USER() | |  |  |
| TAN() | Return the tangent of the argument | |  |  |
| TIME() | Extract the time portion of the expression passed | |  |  |
| TIME\_FORMAT() | Format as time | |  |  |
| TIME\_TO\_SEC() | Return the argument converted to seconds | |  |  |
| TIMEDIFF() | Subtract time | |  |  |
| TIMESTAMP() | With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments | |  |  |
| TIMESTAMPADD() | Add an interval to a datetime expression | |  |  |
| TIMESTAMPDIFF() | Return the difference of two datetime expressions, using the units specified | |  |  |
| TO\_BASE64() | Return the argument converted to a base-64 string | |  |  |
| TO\_DAYS() | Return the date argument converted to days | |  |  |
| TO\_SECONDS() | Return the date or datetime argument converted to seconds since Year 0 | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| TRIM() | Remove leading and trailing spaces |  |  |
| TRUNCATE() | Truncate to specified number of decimal places |  |  |
| UCASE() | Synonym for UPPER() |  |  |
| UNCOMPRESS() | Uncompress a string compressed |  |  |
| UNCOMPRESSED\_LENGTH | R()eturn the length  of a string before  compression |  |  |
| UNHEX() | Return a string  containing hex  representation of a  number |  |  |
| UNIX\_TIMESTAMP() | Return a Unix  timestamp |  |  |
| UpdateXML() | Return replaced XML fragment |  |  |
| UPPER() | Convert to uppercase |  |  |
| USER() | The user name and host name provided by the client |  |  |
| UTC\_DATE() | Return the current UTC date |  |  |
| UTC\_TIME() | Return the current UTC time |  |  |
| UTC\_TIMESTAMP() | Return the current UTC date and time |  |  |
| UUID() | Return a Universal Unique Identifier (UUID) |  |  |
| UUID\_SHORT() | Return an integer-  valued universal  identifier |  |  |
| UUID\_TO\_BIN() | Convert string UUID to binary |  |  |
| VALIDATE\_PASSWORD\_S | ReEteNrTie()strength of password |  |  |
| VALUES() | Define the values to be used during an INSERT |  |  |
| VAR\_POP() | Return the population standard variance |  |  |
| VAR\_SAMP() | Return the sample  variance |  |  |
| VARIANCE() | Return the population standard variance |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| VERSION() | Return a string that indicates the MySQL server version |  |  |
| WAIT\_FOR\_EXECUTED\_G | IaDit\_tTil h)e given GTIDs have executed on the replica. |  |  |
| WAIT\_UNTIL\_SQL\_THRE | e\_AFTER\_GTIDS()  WAIT\_FOR\_EXECUTED\_G | TID\_SET(). | 8.0.18 |
| WEEK() | Return the week number |  |  |
| WEEKDAY() | Return the weekday  index |  |  |
| WEEKOFYEAR() | Return the calendar week of the date (1-53) |  |  |
| WEIGHT\_STRING() | Return the weight string for a string |  |  |
| [XOR](#_bookmark195) | Logical XOR |  |  |
| YEAR() | Return the year |  |  |
| YEARWEEK() | Return the year and  week |  |  |
| | | Bitwise OR |  |  |
| ~ | Bitwise inversion |  |  |

**12.2** **Loadable** **Function** **Reference**

The following table lists each function that is loadable at runtime and provides a short description of each one. For a table listing built-in functions and operators, see [Section 12.1, “Built-In Function and](#_bookmark173) [Operator Reference”](#_bookmark173)

For general information about loadable functions, see Section 5.7, “MySQL Server Loadable

Functions” .

**Table** **12.2** **Loadable** **Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| asymmetric\_decrypt () | Decrypt ciphertext using private or public key |  |  |
| asymmetric\_derive() | Derive symmetric key from asymmetric keys |  |  |
| asymmetric\_encrypt () | Encrypt cleartext using private or public key |  |  |
| asymmetric\_sign() | Generate signature from digest |  |  |
| asymmetric\_verify() | Verify that signature  matches digest |  |  |
| asynchronous\_connec | na\_rfeapilitvioenrs\_oaudre\_m server in a managed group to the source list | n.0a.d() |  |
| asynchronous\_connec | na\_rfeapilitvioenrs\_oaudre\_s server to the source list | u.0r.2e () |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| asynchronous\_connec | omnov\_feaerd\_delet group of replication source servers from the source list | .0m.\_aged() |  |
| asynchronous\_connec | omnov\_feaaieopvliecti\_let source server from the source list | .0s.\_rce() |  |
| audit\_api\_message\_e | \_sfs(a)ge event to audit log |  |  |
| audit\_log\_encryptio | e\_cahsasuwdoitrlget() encryption password |  |  |
| audit\_log\_encryptio | e\_ptadsiwtolorgdens\_cerp(ton password |  |  |
| audit\_log\_filter\_fl | sluhs)audit log filter  tables |  |  |
| audit\_log\_filter\_re | vmeov\_feialutdeitrlo(g) filter |  |  |
| audit\_log\_filter\_re | onsi\_earud(t log filter from user |  |  |
| audit\_log\_filter\_se | tDe\_fi taeui)log filter |  |  |
| audit\_log\_filter\_se | Ats\_ira(u)dit log filter to user |  |  |
| audit\_log\_read() | Return audit log records |  |  |
| audit\_log\_read\_book | km()ark for most recent audit log event |  |  |
| audit\_log\_rotate() | Rotate audit log file |  |  |
| create\_asymmetric\_p | at\_epiv()ate key |  |  |
| create\_asymmetric\_p | breakt\_eyp(u)blic key |  |  |
| create\_dh\_parameter | e()nerate shared DH secret |  |  |
| create\_digest() | Generate digest from string |  |  |
| firewall\_group\_deli | m()ove account from firewall group profile | 8.0.23 |  |
| firewall\_group\_enli | d()account to firewall group profile | 8.0.23 |  |
| gen\_blacklist() | Perform dictionary term replacement |  | 8.0.23 |
| gen\_blocklist() | Perform dictionary term replacement | 8.0.23 |  |
| gen\_dictionary() | Return random term from dictionary |  |  |
| gen\_dictionary\_drop | R()emove dictionary from registry |  |  |
| gen\_dictionary\_load | L(o)ad dictionary into  registry |  |  |
| gen\_range() | Generate random number within range |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| gen\_rnd\_email() | Generate random email address |  |  |
| gen\_rnd\_pan() | Generate random payment card Primary  Account Number |  |  |
| gen\_rnd\_ssn() | Generate random US Social Security number |  |  |
| gen\_rnd\_us\_phone() | Generate random US phone number |  |  |
| group\_replication\_d | ab**l**eea\_**mem**b\_io so that the member does not take it in the specified situation | n() |  |
| group\_replication\_e | lee am\_mebmebrer\_aa**c**tin for the member to take in the specified situation | () |  |
| group\_replication\_g | tucr\_mGmruonucation\_pr Replication protocol version | otocol() |  |
| group\_replication\_g | tuwr\_ixi\_ounmcurrenc number of consensus instances executable in parallel | y() |  |
| group\_replication\_r | t\_tmhememrb\_aetions actions configuration to the default settings | () |  |
| group\_replication\_s | si\_\_gprreym(b)er as new primary |  |  |
| group\_replication\_s | t\_puiecpaliait\_pr protocol version | otocol() |  |
| group\_replication\_s | t\_ratiu\_onucrenc of consensus instances executable in parallel | y() |  |
| group\_replication\_s | ihe\_tgo\_pur\_prim single-primary to multi- primary mode | ary\_mode() |  |
| group\_replication\_s | ihe\_tgo\_uspire\_pri multi-primary to single- primary mode | mary\_mode() |  |
| keyring\_aws\_rotate\_ | moa(te) AWS customer master key |  |  |
| keyring\_aws\_rotate\_ | aste()keys in keyring\_aws storage file |  |  |
| keyring\_hashicorp\_u | er\_tnig()  keyring\_hashicorp  reconfiguration |  |  |
| keyring\_key\_fetch() | Fetch keyring key value |  |  |
| keyring\_key\_generat | e()nerate random  keyring key |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| keyring\_key\_length\_ | etucr (k)eyring key  length |  |  |
| keyring\_key\_remove () | Remove keyring key |  |  |
| keyring\_key\_store() | Store key in keyring |  |  |
| keyring\_key\_type\_fe | u(r keyring key type |  |  |
| load\_rewrite\_rules () | Rewriter plugin helper routine |  |  |
| mask\_inner() | Mask interior part of  string |  |  |
| mask\_outer() | Mask left and right parts of string |  |  |
| mask\_pan() | Mask payment card Primary Account Number part of string |  |  |
| mask\_pan\_relaxed() | Mask payment card Primary Account Number part of string |  |  |
| mask\_ssn() | Mask US Social Security number |  |  |
| mysql\_firewall\_flus | hRe\_staftiruea()ll status variables |  |  |
| mysql\_query\_attribu | etc\_shtqruienr a(tribute value | 8.0.23 |  |
| normalize\_statement | N()ormalize SQL statement to digest form |  |  |
| read\_firewall\_group | U\_apewflirtal(l)group profile recorded- statement cache | 8.0.23 |  |
| read\_firewall\_group | p(d)ate firewall group profile cache | 8.0.23 |  |
| read\_firewall\_users | U()pdate firewall account profile cache |  | 8.0.26 |
| read\_firewall\_white | ipt(e)firewall account profile recorded- statement cache |  | 8.0.26 |
| service\_get\_read\_lo | u(i)re locking service shared locks |  |  |
| service\_get\_write\_l | cir(e)locking service exclusive locks |  |  |
| service\_release\_loc | l(e)ase locking service locks |  |  |
| set\_firewall\_group\_ | tdaebl(is)h firewall group profile operational mode | 8.0.23 |  |
| set\_firewall\_mode() | Establish firewall  account profile  operational mode |  | 8.0.26 |
| version\_tokens\_dele | el(e)te tokens from  version tokens list |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| version\_tokens\_edit | M()odify version tokens list |  |  |
| version\_tokens\_lock | A\_ieuls(i)ve locks on version tokens |  |  |
| version\_tokens\_lock | A\_ireh(a)red locks on version tokens |  |  |
| version\_tokens\_set () | Set version tokens list |  |  |
| version\_tokens\_show | R()eturn version tokens list |  |  |
| version\_tokens\_unlo | l(e)ase version tokens locks |  |  |

**12.3** **Type** **Conversion** **in** **Expression** **Evaluation**

When an operator is used with operands of different types, type conversion occurs to make the operands compatible. Some conversions occur implicitly. For example, MySQL automatically converts strings to numbers as necessary, and vice versa.

mysql> **SELECT** **1+'1';**

-> 2

mysql> **SELECT** **CONCAT(2,'** **test');**

-> '2 test'

It is also possible to convert a number to a string explicitly using the CAST() function. Conversion occurs implicitly with the CONCAT() function because it expects string arguments.

mysql> **SELECT** **38** **.8,** **CAST(38** **.8** **AS** **CHAR);**

-> 38 .8, '38 .8'

mysql> **SELECT** **38** **.8,** **CONCAT(38** **.8);**

-> 38.8, '38.8'

See later in this section for information about the character set of implicit number-to-string conversions, and for modified rules that apply to CREATE TABLE ... SELECT statements.

The following rules describe how conversion occurs for comparison operations:

• If one or both arguments are NULL, the result of the comparison is NULL, except for the NULL-safe [<=>](#_bookmark159) equality comparison operator. For NULL <=> NULL, the result is true. No conversion is needed.

• If both arguments in a comparison operation are strings, they are compared as strings.

• If both arguments are integers, they are compared as integers.

• Hexadecimal values are treated as binary strings if not compared to a number.

• If one of the arguments is a [TIMESTAMP](#_bookmark109) or [DATETIME](#_bookmark109) column and the other argument is a constant, the constant is converted to a timestamp before the comparison is performed. This is done to be more ODBC-friendly. This is not done for the arguments to [IN()](#_bookmark161). To be safe, always use complete datetime, date, or time strings when doing comparisons. For example, to achieve best results when using [BETWEEN](#_bookmark160) with date or time values, use CAST() to explicitly convert the values to the desired data type.

A single-row subquery from a table or tables is not considered a constant. For example, if a subquery returns an integer to be compared to a [DATETIME](#_bookmark109) value, the comparison is done as two integers. The integer is not converted to a temporal value. To compare the operands as [DATETIME](#_bookmark109) values, use CAST() to explicitly convert the subquery value to [DATETIME](#_bookmark109).

• If one of the arguments is a decimal value, comparison depends on the other argument. The arguments are compared as decimal values if the other argument is a decimal or integer value, or as floating-point values if the other argument is a floating-point value.

• In all other cases, the arguments are compared as floating-point (double-precision) numbers. For example, a comparison of string and numeric operands takes place as a comparison of floating-point numbers.

For information about conversion of values from one temporal type to another, see [Section 11.2.7,](#_bookmark114) [“Conversion Between Date and Time Types”](#_bookmark114) .

Comparison of JSON values takes place at two levels. The first level of comparison is based on the JSON types of the compared values. If the types differ, the comparison result is determined solely by which type has higher precedence. If the two values have the same JSON type, a second level of comparison occurs using type-specific rules. For comparison of JSON and non-JSON values, the non-JSON value is converted to JSON and the values compared as JSON values. For details, see [Comparison and Ordering of JSON Values](#_bookmark149).

The following examples illustrate conversion of strings to numbers for comparison operations:

|  |  |
| --- | --- |
| mysql> **SELECT**  -> 0  mysql> **SELECT**  -> 1  mysql> **SELECT**  -> 0  mysql> **SELECT**  -> 1 | **1** **>** **'6x';**  **7** **>** **'6x';**  **0** **>** **'x6';**  **0** **=** **'x6';** |

For comparisons of a string column with a number, MySQL cannot use an index on the column to look up the value quickly. If *str\_col* is an indexed string column, the index cannot be used when performing the lookup in the following statement:

SELECT \* FROM *tbl\_name* WHERE *str\_col*=1;

The reason for this is that there are many different strings that may convert to the value 1, such as '1', ' 1', or '1a'.

Comparisons between floating-point numbers and large values of INTEGER type are approximate because the integer is converted to double-precision floating point before comparison, which is not capable of representing all 64-bit integers exactly. For example, the integer value 253 + 1 is not representable as a float, and is rounded to 253 or 253 + 2 before a float comparison, depending on the platform.

To illustrate, only the first of the following comparisons compares equal values, but both comparisons return true (1):

mysql> **SELECT** **'9223372036854775807'** **=** **9223372036854775807;**

-> 1

mysql> **SELECT** **'9223372036854775807'** **=** **9223372036854775806;**

-> 1

When conversions from string to floating-point and from integer to floating-point occur, they do not necessarily occur the same way. The integer may be converted to floating-point by the CPU, whereas the string is converted digit by digit in an operation that involves floating-point multiplications. Also, results can be affected by factors such as computer architecture or the compiler version or optimization level. One way to avoid such problems is to use CAST() so that a value is not converted implicitly to a float-point number:

mysql> **SELECT** **CAST('9223372036854775807'** **AS** **UNSIGNED)** **=** **9223372036854775806;**

-> 0

For more information about floating-point comparisons, see Section B.3.4.8, “Problems with Floating-

Point Values” .

The server includes dtoa, a conversion library that provides the basis for improved conversion between string or [DECIMAL](#_bookmark102) values and approximate-value ([FLOAT](#_bookmark103)/[DOUBLE](#_bookmark103)) numbers:

• Consistent conversion results across platforms, which eliminates, for example, Unix versus Windows conversion differences.

• Accurate representation of values in cases where results previously did not provide sufficient precision, such as for values close to IEEE limits.

• Conversion of numbers to string format with the best possible precision. The precision of dtoa is always the same or better than that of the standard C library functions.

Because the conversions produced by this library differ in some cases from non-dtoa results, the potential exists for incompatibilities in applications that rely on previous results. For example, applications that depend on a specific exact result from previous conversions might need adjustment to accommodate additional precision.

The dtoa library provides conversions with the following properties. *D* represents a value with a [DECIMAL](#_bookmark102) or string representation, and *F* represents a floating-point number in native binary (IEEE) format.

• *F*-> *D* conversion is done with the best possible precision, returning *D* as the shortest string that yields *F*when read back in and rounded to the nearest value in native binary format as specified by IEEE.

• *D* -> *F*conversion is done such that *F* is the nearest native binary number to the input decimal string *D*.

These properties imply that *F* -> *D* -> *F* conversions are lossless unless *F* is -inf, +inf, or NaN. The

latter values are not supported because the SQL standard defines them as invalid values for [FLOAT](#_bookmark103) or [DOUBLE](#_bookmark103).

For *D* -> *F*-> *D* conversions, a sufficient condition for losslessness is that *D* uses 15 or fewer digits of precision, is not a denormal value, -inf, +inf, or NaN. In some cases, the conversion is lossless even if *D* has more than 15 digits of precision, but this is not always the case.

Implicit conversion of a numeric or temporal value to string produces a value that has a character set and collation determined by the character\_set\_connection and collation\_connection system variables. (These variables commonly are set with SET NAMES. For information about connection character sets, see [Section 10.4, “Connection Character Sets and Collations”](#_bookmark17) .)

This means that such a conversion results in a character (nonbinary) string (a [CHAR](#_bookmark60), [VARCHAR](#_bookmark60), or [LONGTEXT](#_bookmark61) value), except in the case that the connection character set is set to binary. In that case, the conversion result is a binary string (a [BINARY](#_bookmark73), [VARBINARY](#_bookmark73), or [LONGBLOB](#_bookmark61) value).

For integer expressions, the preceding remarks about expression *evaluation* apply somewhat differently for expression *assignment*; for example, in a statement such as this:

CREATE TABLE t SELECT *integer\_expr*;

In this case, the table in the column resulting from the expression has type [INT](#_bookmark101) or [BIGINT](#_bookmark101) depending on the length of the integer expression. If the maximum length of the expression does not fit in an [INT](#_bookmark101), [BIGINT](#_bookmark101) is used instead. The length is taken from the max\_length value of the SELECT result set metadata (see [C API Basic Data Structures](https://dev.mysql.com/doc/c-api/8.0/en/c-api-data-structures.html)). This means that you can force a [BIGINT](#_bookmark101) rather than [INT](#_bookmark101) by use of a sufficiently long expression:

CREATE TABLE t SELECT 000000000000000000000;

**12.4** **Operators**

**Table** **12.3** **Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| & | Bitwise AND |  |  |
| [>](#_bookmark156) | Greater than operator |  |  |
| >> | Right shift |  |  |
| [>=](#_bookmark157) | Greater than or equal operator |  |  |
| [<](#_bookmark154) | Less than operator |  |  |
| [<>, !=](#_bookmark158) | Not equal operator |  |  |
| << | Left shift |  |  |
| [<=](#_bookmark155) | Less than or equal  operator |  |  |
| [<=>](#_bookmark159) | NULL-safe equal to  operator |  |  |
| %, MOD | Modulo operator |  |  |
| \* | Multiplication operator |  |  |
| + | Addition operator |  |  |
| - | Minus operator |  |  |
| - | Change the sign of the argument |  |  |
| -> | Return value from JSON column after evaluating path; equivalent to JSON\_EXTRACT(). |  |  |
| ->> | Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON\_UNQUOTE(JSON | \_EXTRACT()). |  |
| / | Division operator |  |  |
| [:=](#_bookmark181) | Assign a value |  |  |
| [=](#_bookmark182) | Assign a value (as part of a SET statement, or as part of the SET clause in an UPDATE statement) |  |  |
| [=](#_bookmark153) | Equal operator |  |  |
| ^ | Bitwise XOR |  |  |
| [AND, &&](#_bookmark183) | Logical AND |  |  |
| [BETWEEN ...](#_bookmark160)  [AND ...](#_bookmark160) | Whether a value is within a range of values |  |  |
| BINARY | Cast a string to a binary string |  | 8.0.27 |
| [CASE](#_bookmark184) | Case operator |  |  |
| DIV | Integer division |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Introduced** | **Deprecated** |
| [IN()](#_bookmark161) | Whether a value is within a set of values |  |  |
| [IS](#_bookmark187) | Test a value against a boolean |  |  |
| [IS NOT](#_bookmark188) | Test a value against a boolean |  |  |
| [IS NOT NULL](#_bookmark189) | NOT NULL value test |  |  |
| [IS NULL](#_bookmark190) | NULL value test |  |  |
| LIKE | Simple pattern matching |  |  |
| MEMBER OF() | Returns true (1) if first operand matches any element of JSON array passed as second operand, otherwise returns false (0) | 8.0.17 |  |
| [NOT, !](#_bookmark192) | Negates value |  |  |
| [NOT BETWEEN ...](#_bookmark193)  [AND ...](#_bookmark193) | Whether a value is not within a range of values |  |  |
| [NOT IN()](#_bookmark194) | Whether a value is not within a set of values |  |  |
| NOT LIKE | Negation of simple  pattern matching |  |  |
| NOT REGEXP | Negation of REGEXP |  |  |
| [OR, ||](#_bookmark74) | Logical OR |  |  |
| REGEXP | Whether string matches regular expression |  |  |
| RLIKE | Whether string matches regular expression |  |  |
| SOUNDS LIKE | Compare sounds |  |  |
| [XOR](#_bookmark195) | Logical XOR |  |  |
| | | Bitwise OR |  |  |
| ~ | Bitwise inversion |  |  |

**12.4.1** **Operator** **Precedence**

Operator precedences are shown in the following list, from highest precedence to the lowest. Operators that are shown together on a line have the same precedence.

INTERVAL

BINARY, COLLATE

!

- (unary minus), ~ (unary bit inversion)

^

\*, /, DIV, %, MOD

-, +

<<, >>

&

|

= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN, MEMBER OF

BETWEEN, CASE, WHEN, THEN, ELSE

NOT

AND, &&

XOR

OR, ||

= (assignment), :=

The precedence of = depends on whether it is used as a comparison operator ([=](#_bookmark153)) or as an assignment operator ([=](#_bookmark182)). When used as a comparison operator, it has the same precedence as [<=>](#_bookmark159), [>=](#_bookmark157), [>](#_bookmark156), [<=](#_bookmark155), [<](#_bookmark154), [<>](#_bookmark158), [!=](#_bookmark158), [IS](#_bookmark187), LIKE, REGEXP, and [IN()](#_bookmark161). When used as an assignment operator, it has the same precedence as [:=](#_bookmark181). Section 13.7.6.1, “SET Syntax for Variable Assignment” , and Section 9.4, “User- Defined Variables” , explain how MySQL determines which interpretation of = should apply.

For operators that occur at the same precedence level within an expression, evaluation proceeds left to right, with the exception that assignments evaluate right to left.

The precedence and meaning of some operators depends on the SQL mode:

• By default, [||](#_bookmark74) is a logical [OR](#_bookmark74) operator. With PIPES\_AS\_CONCAT enabled, [||](#_bookmark74) is string concatenation, with a precedence between ^ and the unary operators.

• By default, [!](#_bookmark192) has a higher precedence than NOT. With HIGH\_NOT\_PRECEDENCE enabled, [!](#_bookmark192) and NOT have the same precedence.

See Section 5.1.11, “Server SQL Modes” .

The precedence of operators determines the order of evaluation of terms in an expression. To override this order and group terms explicitly, use parentheses. For example:

mysql> **SELECT** **1+2\*3;**

-> 7

mysql> **SELECT** **(1+2)\*3;**

-> 9

**12.4.2** **Comparison** **Functions** **and** **Operators**

**Table** **12.4** **Comparison** **Operators**

|  |  |
| --- | --- |
| **Name** | **Description** |
| [>](#_bookmark156) | Greater than operator |
| [>=](#_bookmark157) | Greater than or equal operator |
| [<](#_bookmark154) | Less than operator |
| [<>, !=](#_bookmark158) | Not equal operator |
| [<=](#_bookmark155) | Less than or equal operator |
| [<=>](#_bookmark159) | NULL-safe equal to operator |
| [=](#_bookmark153) | Equal operator |
| [BETWEEN ... AND ...](#_bookmark160) | Whether a value is within a range of values |
| [COALESCE()](#_bookmark185) | Return the first non-NULL argument |
| [GREATEST()](#_bookmark162) | Return the largest argument |
| [IN()](#_bookmark161) | Whether a value is within a set of values |
| [INTERVAL()](#_bookmark186) | Return the index of the argument that is less than the first argument |
| [IS](#_bookmark187) | Test a value against a boolean |
| [IS NOT](#_bookmark188) | Test a value against a boolean |
| [IS NOT NULL](#_bookmark189) | NOT NULL value test |
| [IS NULL](#_bookmark190) | NULL value test |
| [ISNULL()](#_bookmark191) | Test whether the argument is NULL |
| [LEAST()](#_bookmark163) | Return the smallest argument |
| LIKE | Simple pattern matching |



|  |  |
| --- | --- |
| **Name** | **Description** |
| [NOT BETWEEN ... AND ...](#_bookmark193) | Whether a value is not within a range of values |
| [NOT IN()](#_bookmark194) | Whether a value is not within a set of values |
| NOT LIKE | Negation of simple pattern matching |
| STRCMP() | Compare two strings |

Comparison operations result in a value of 1 (TRUE), 0 (FALSE), or NULL. These operations work for both numbers and strings. Strings are automatically converted to numbers and numbers to strings as necessary.

The following relational comparison operators can be used to compare not only scalar operands, but row operands:

= > < >= <= <> !=

The descriptions for those operators later in this section detail how they work with row operands. For additional examples of row comparisons in the context of row subqueries, see Section 13.2.15.5, “Row Subqueries” .

Some of the functions in this section return values other than 1 (TRUE), 0 (FALSE), or NULL. [LEAST()](#_bookmark163) and [GREATEST()](#_bookmark162) are examples of such functions; [Section 12.3, “Type Conversion Expression](#_bookmark75)in [Evaluation”](#_bookmark75) , describes the rules for comparison operations performed by these and similar functions for determining their return values.

**Note**

In previous versions of MySQL, when evaluating an expression containing LEAST() or GREATEST(), the server attempted to guess the context in which the function was used, and to coerce the function's arguments to the data type of the expression as a whole. For example, the arguments to LEAST("11", "45", "2") are evaluated and sorted as strings, so that this expression returns "11". In MySQL 8.0.3 and earlier, when evaluating the expression LEAST("11", "45", "2") + 0, the server converted the arguments to integers (anticipating the addition of integer 0 to the result) before sorting them, thus returning 2.

Beginning with MySQL 8.0.4, the server no longer attempts to infer context in this fashion. Instead, the function is executed using the arguments as provided, performing data type conversions to one or more of the arguments if and

only if they are not all of the same type. Any type coercion mandated by an expression that makes use of the return value is now performed following function execution. This means that, in MySQL 8.0.4 and later, LEAST("11", "45", "2") + 0 evaluates to "11" + 0 and thus to integer 11. (Bug #83895, Bug #25123839)

To convert a value to a specific type for comparison purposes, you can use the CAST() function. String values can be converted to a different character set using CONVERT(). See Section 12.11, “Cast Functions and Operators” .

By default, string comparisons are not case-sensitive and use the current character set. The default is

utf8mb4.

• [=](#_bookmark153) Equal:

mysql> **SELECT** **1** **=** **0;**

-> 0

mysql> **SELECT** **'0'** **=** **0;**

-> 1

mysql> **SELECT** **'0** **.0'** **=** **0;**

-> 1

mysql> **SELECT** **'0** **.01'** **=** **0;**

-> 0

mysql> **SELECT** **'** **.01'** **=** **0** **.01;**

-> 1

For row comparisons, (a, b) = (x, y) is equivalent to:

(a = x) AND (b = y)

• [<=>](#_bookmark159)

NULL-safe equal. This operator performs an equality comparison like the [=](#_bookmark153) operator, but returns 1

rather than NULL if both operands are NULL, and 0 rather than NULL if one operand is NULL. The [<=>](#_bookmark159) operator is equivalent to the standard SQL IS NOT DISTINCT FROM operator.

mysql> **SELECT** **1** **<=>** **1,** **NULL** **<=>** **NULL,** **1** **<=>** **NULL;**

-> 1, 1, 0

mysql> **SELECT** **1** **=** **1,** **NULL** **=** **NULL,** **1** **=** **NULL;**

-> 1, NULL, NULL

For row comparisons, (a, b) <=> (x, y) is equivalent to:

(a <=> x) AND (b <=> y)

• [<>](#_bookmark158), [!=](#_bookmark158) Not equal:

mysql> **SELECT** **'** **.01'** **<>** **'0** **.01';**

-> 1

mysql> **SELECT** **.01** **<>** **'0** **.01';**

-> 0

mysql> **SELECT** **'zapp'** **<>** **'zappp';**

-> 1

For row comparisons, (a, b) <> (x, y) and (a, b) != (x, y) are equivalent to: (a <> x) OR (b <> y)

• [<=](#_bookmark155)

Less than or equal:

mysql> **SELECT** **0.1** **<=** **2;**

-> 1

For row comparisons, (a, b) <= (x, y) is equivalent to:

(a < x) OR ((a = x) AND (b <= y))

• [<](#_bookmark154)

Less than:

mysql> **SELECT** **2** **<** **2;**

-> 0

For row comparisons, (a, b) < (x, y) is equivalent to:

(a < x) OR ((a = x) AND (b < y))

• [>=](#_bookmark157)

Greater than or equal:

mysql> **SELECT** **2** **>=** **2;**

-> 1

For row comparisons, (a, b) >= (x, y) is equivalent to:

(a > x) OR ((a = x) AND (b >= y))

• [>](#_bookmark156)

Greater than:

mysql> **SELECT** **2** **>** **2;**

-> 0

For row comparisons, (a, b) > (x, y) is equivalent to:

(a > x) OR ((a = x) AND (b > y))

• [*expr* BETWEEN *min* AND *max*](#_bookmark160)

If *expr* is greater than or equal to *min* and *expr* is less than or equal to *max*, [BETWEEN](#_bookmark160) returns 1, otherwise it returns 0. This is equivalent to the expression (*min* <= *expr* AND *expr* <= *max*) if all the arguments are of the same type. Otherwise type conversion takes place according to the rules described in [Section 12.3, “Type Conversion in Expression Evaluation”](#_bookmark75) , but applied to all the three arguments.

mysql> **SELECT** **2** **BETWEEN** **1** **AND** **3,** **2** **BETWEEN** **3** **and** **1;**

-> 1, 0

mysql> **SELECT** **1** **BETWEEN** **2** **AND** **3;**

-> 0

mysql> **SELECT** **'b'** **BETWEEN** **'a'** **AND** **'c';**

-> 1

mysql> **SELECT** **2** **BETWEEN** **2** **AND** **'3';**

-> 1

mysql> **SELECT** **2** **BETWEEN** **2** **AND** **'x-3';**

-> 0

For best results when using [BETWEEN](#_bookmark160) with date or time values, use CAST() to explicitly convert the values to the desired data type. Examples: If you compare a [DATETIME](#_bookmark109) to two [DATE](#_bookmark109) values, convert the [DATE](#_bookmark109) values to [DATETIME](#_bookmark109) values. If you use a string constant such as '2001-1-1' in a comparison to a [DATE](#_bookmark109), cast the string to a [DATE](#_bookmark109).

• [*expr* NOT BETWEEN *min* AND *max*](#_bookmark193)

This is the same as NOT (*expr* BETWEEN *min* AND *max*).

• [COALESCE(*value*,...)](#_bookmark185)

Returns the first non-NULL value in the list, or NULL if there are no non-NULL values. The return type of [COALESCE()](#_bookmark185) is the aggregated type of the argument types.

mysql> **SELECT** **COALESCE(NULL,1);**

-> 1

mysql> **SELECT** **COALESCE(NULL,NULL,NULL);**

-> NULL

• [GREATEST(*value1*,*value2*,...)](#_bookmark162)

With two or more arguments, returns the largest (maximum-valued) argument. The arguments are compared using the same rules as for [LEAST()](#_bookmark163).

mysql> **SELECT** **GREATEST(2,0);**

-> 2

mysql> **SELECT** **GREATEST(34** **.0,3** **.0,5** **.0,767** **.0);**

-> 767.0

mysql> **SELECT** **GREATEST('B','A','C');**

-> 'C'

[GREATEST()](#_bookmark162) returns NULL if any argument is NULL.

• [*expr* IN (*value*,...)](#_bookmark161)

Returns 1 (true) if *expr* is equal to any of the values in the IN() list, else returns 0 (false).

Type conversion takes place according to the rules described in [Section 12.3, “Type Conversion in](#_bookmark75) [Expression Evaluation”](#_bookmark75) , applied to all the arguments. If no type conversion is needed for the values in the IN() list, they are all non-JSON constants of the same type, and *expr* can be compared to each of them as a value of the same type (possibly after type conversion), an optimization takes place. The values the list are sorted and the search for *expr* is done using a binary search, which makes the IN() operation very quick.

mysql> **SELECT** **2** **IN** **(0,3,5,7);**

-> 0

mysql> **SELECT** **'wefwf'** **IN** **('wee','wefwf','weg');**

-> 1

IN() can be used to compare row constructors:

mysql> **SELECT** **(3,4)** **IN** **((1,2),** **(3,4));**

-> 1

mysql> **SELECT** **(3,4)** **IN** **((1,2),** **(3,5));**

-> 0

You should never mix quoted and unquoted values in an IN() list because the comparison rules for quoted values (such as strings) and unquoted values (such as numbers) differ. Mixing types may therefore lead to inconsistent results. For example, do not write an IN() expression like this:

SELECT val1 FROM tbl1 WHERE val1 IN (1,2,'a');

Instead, write it like this:

SELECT val1 FROM tbl1 WHERE val1 IN ('1','2','a');

Implicit type conversion may produce nonintuitive results:

mysql> **SELECT** **'a'** **IN** **(0),** **0** **IN** **('b');**

-> 1, 1

In both cases, the comparison values are converted to floating-point values, yielding 0.0 in each case, and a comparison result of 1 (true).

The number of values in the IN() list is only limited by the max\_allowed\_packet value.

To comply with the SQL standard, IN() returns NULL not only if the expression on the left hand side is NULL, but also if no match is found in the list and one of the expressions in the list is NULL.

IN() syntax can also be used to write certain types of subqueries. See Section 13.2.15.3, “Subqueries with ANY, IN, or SOME” .

• [*expr* NOT IN (*value*,...)](#_bookmark194)

This is the same as NOT (*expr* IN (*value*,...)).

• [INTERVAL(*N*,*N1*,*N2*,*N3*,...)](#_bookmark186)

Returns 0 if *N* ≤ *N1*, 1 if *N* ≤ *N2* and so on, or -1 if *N* is NULL. All arguments are treated as integers. It is required that *N1* ≤ *N2* ≤ *N3* ≤ ... ≤ *Nn* for this function to work correctly. This is because a binary search is used (very fast).

mysql> **SELECT** **INTERVAL(23,** **1,** **15,** **17,** **30,** **44,** **200);**

-> 3

mysql> **SELECT** **INTERVAL(10,** **1,** **10,** **100,** **1000);**

-> 2

mysql> **SELECT** **INTERVAL(22,** **23,** **30,** **44,** **200);**

-> 0

• [IS *boolean\_value*](#_bookmark187)

Tests a value against a boolean value, where *boolean\_value* can be TRUE, FALSE, or UNKNOWN.

mysql> **SELECT** **1** **IS** **TRUE,** **0** **IS** **FALSE,** **NULL** **IS** **UNKNOWN;**

-> 1, 1, 1

• [IS NOT *boolean\_value*](#_bookmark188)

Tests a value against a boolean value, where *boolean\_value* can be TRUE, FALSE, or UNKNOWN.

mysql> **SELECT** **1** **IS** **NOT** **UNKNOWN,** **0** **IS** **NOT** **UNKNOWN,** **NULL** **IS** **NOT** **UNKNOWN;**

-> 1, 1, 0

• [IS NULL](#_bookmark190)

Tests whether a value is NULL.

mysql> **SELECT** **1** **IS** **NULL,** **0** **IS** **NULL,** **NULL** **IS** **NULL;**

-> 0, 0, 1

To work well with ODBC programs, MySQL supports the following extra features when using [IS](#_bookmark190) [NULL](#_bookmark190):

• If sql\_auto\_is\_null variable is set to 1, then after a statement that successfully inserts an automatically generated AUTO\_INCREMENT value, you can find that value by issuing a statement of the following form:

SELECT \* FROM *tbl\_name* WHERE *auto\_col* IS NULL

If the statement returns a row, the value returned is the same as if you invoked the LAST\_INSERT\_ID() function. For details, including the return value after a multiple-row insert, see Section 12.16, “Information Functions” . If no AUTO\_INCREMENT value was successfully inserted, the SELECT statement returns no row.

The behavior of retrieving an AUTO\_INCREMENT value by using an [IS NULL](#_bookmark190) comparison can be disabled by setting sql\_auto\_is\_null = 0. See Section 5.1.8, “Server System Variables” .

The default value of sql\_auto\_is\_null is 0.

• For [DATE](#_bookmark109) and [DATETIME](#_bookmark109) columns that are declared as NOT NULL, you can find the special date '0000-00-00' by using a statement like this:

SELECT \* FROM *tbl\_name* WHERE *date\_column* IS NULL

This is needed to get some ODBC applications to work because ODBC does not support a '0000-00-00' date value.

See [Obtaining Auto-Increment Values](https://dev.mysql.com/doc/connector-odbc/en/connector-odbc-usagenotes-functionality-last-insert-id.html), and the description for the FLAG\_AUTO\_IS\_NULL option at [Connector/ODBC Connection Parameters](https://dev.mysql.com/doc/connector-odbc/en/connector-odbc-configuration-connection-parameters.html).

• [IS NOT NULL](#_bookmark190)

Tests whether a value is not NULL.

mysql> **SELECT** **1** **IS** **NOT** **NULL,** **0** **IS** **NOT** **NULL,** **NULL** **IS** **NOT** **NULL;**

-> 1, 1, 0

• [ISNULL(*expr*](#_bookmark191))

If *expr* is NULL, [ISNULL()](#_bookmark191) returns 1, otherwise it returns 0.

mysql> **SELECT** **ISNULL(1+1);**

-> 0

mysql> **SELECT** **ISNULL(1/0);**

-> 1

[ISNULL()](#_bookmark191) can be used instead of [=](#_bookmark153) to test whether a value is NULL. (Comparing a value to NULL using [=](#_bookmark153) always yields NULL.)

The [ISNULL()](#_bookmark191) function shares some special behaviors with the [IS NULL](#_bookmark190) comparison operator. See the description of [IS NULL](#_bookmark190).

• [LEAST(*value1*,*value2*,...)](#_bookmark163)

With two or more arguments, returns the smallest (minimum-valued) argument. The arguments are compared using the following rules:

• If any argument is NULL, the result is NULL. No comparison is needed.

• If all arguments are integer-valued, they are compared as integers.

• If at least one argument is double precision, they are compared as double-precision values. Otherwise, if at least one argument is a [DECIMAL](#_bookmark102) value, they are compared as [DECIMAL](#_bookmark102) values.

• If the arguments comprise a mix of numbers and strings, they are compared as strings.

• If any argument is a nonbinary (character) string, the arguments are compared as nonbinary strings.

• In all other cases, the arguments are compared as binary strings.

The return type of [LEAST()](#_bookmark163) is the aggregated type of the comparison argument types.

mysql> **SELECT** **LEAST(2,0);**

-> 0

mysql> **SELECT** **LEAST(34** **.0,3** **.0,5** **.0,767** **.0);**

-> 3.0

mysql> **SELECT** **LEAST('B','A','C');**

-> 'A'

**12.4.3** **Logical** **Operators**

**Table** **12.5** **Logical** **Operators**

|  |  |
| --- | --- |
| **Name** | **Description** |
| [AND, &&](#_bookmark183) | Logical AND |
| [NOT, !](#_bookmark192) | Negates value |
| [OR, ||](#_bookmark74) | Logical OR |
| [XOR](#_bookmark195) | Logical XOR |

In SQL, all logical operators evaluate to TRUE, FALSE, or NULL (UNKNOWN). In MySQL, these are implemented as 1 (TRUE), 0 (FALSE), and NULL. Most of this is common to different SQL database servers, although some servers may return any nonzero value for TRUE.

MySQL evaluates any nonzero, non-NULL value to TRUE. For example, the following statements all assess to TRUE:

mysql> **SELECT** **10** **IS** **TRUE;**

-> 1

mysql> **SELECT** **-10** **IS** **TRUE;**

-> 1

mysql> **SELECT** **'string'** **IS** **NOT** **NULL;**

-> 1

• [NOT](#_bookmark192), [!](#_bookmark192)

Logical NOT. Evaluates to 1 if the operand is 0, to 0 if the operand is nonzero, and NOT NULL returns NULL.

mysql> **SELECT** **NOT** **10;**

-> 0

mysql> **SELECT** **NOT** **0;**

-> 1

mysql> **SELECT** **NOT** **NULL;**

-> NULL

mysql> **SELECT** **!** **(1+1);**

-> 0

mysql> **SELECT** **!** **1+1;**

-> 1

The last example produces 1 because the expression evaluates the same way as (!1)+1.

The [!](#_bookmark192), operator is a nonstandard MySQL extension. As of MySQL 8.0.17, this operator is deprecated; expect it to be removed in a future version of MySQL. Applications should be adjusted to use the standard SQL [NOT](#_bookmark192) operator.

• [AND](#_bookmark183), [&&](#_bookmark183)

Logical AND. Evaluates to 1 if all operands are nonzero and not NULL, to 0 if one or more operands are 0, otherwise NULL is returned.

mysql> **SELECT** **1** **AND** **1;**

-> 1

mysql> **SELECT** **1** **AND** **0;**

-> 0

mysql> **SELECT** **1** **AND** **NULL;**

-> NULL

mysql> **SELECT** **0** **AND** **NULL;**

-> 0

mysql> **SELECT** **NULL** **AND** **0;**

-> 0

The [&&](#_bookmark183), operator is a nonstandard MySQL extension. As of MySQL 8.0.17, this operator is deprecated; expect support for it to be removed in a future version of MySQL. Applications should be adjusted to use the standard SQL [AND](#_bookmark183) operator.

• [OR](#_bookmark74), [||](#_bookmark74)

Logical OR. When both operands are non-NULL, the result is 1 if any operand is nonzero, and 0 otherwise. With a NULL operand, the result is 1 if the other operand is nonzero, and NULL otherwise. If both operands are NULL, the result is NULL.

mysql> **SELECT** **1** **OR** **1;**

-> 1

mysql> **SELECT** **1** **OR** **0;**

-> 1

mysql> **SELECT** **0** **OR** **0;**

-> 0

mysql> **SELECT** **0** **OR** **NULL;**

-> NULL

mysql> **SELECT** **1** **OR** **NULL;**



-> 1

**Note**

If the PIPES\_AS\_CONCAT SQL mode is enabled, || signifies the SQL- standard string concatenation operator (like CONCAT()).

The ||, operator is a nonstandard MySQL extension. As of MySQL 8.0.17, this operator is deprecated; expect support for it to be removed in a future version of MySQL. Applications should be adjusted to use the standard SQL OR operator. Exception: Deprecation does not apply if PIPES\_AS\_CONCAT is enabled because, in that case, || signifies string concatenation.

• XOR

Logical XOR. Returns NULL if either operand is NULL. For non-NULL operands, evaluates to 1 if an odd number of operands is nonzero, otherwise 0 is returned.

mysql> **SELECT** **1** **XOR** **1;**

-> 0

mysql> **SELECT** **1** **XOR** **0;**

-> 1

mysql> **SELECT** **1** **XOR** **NULL;**

-> NULL

mysql> **SELECT** **1** **XOR** **1** **XOR** **1;**

-> 1

a XOR b is mathematically equal to (a AND (NOT b)) OR ((NOT a) and b).

**12.4.4** **Assignment** **Operators**

**Table** **12.6** **Assignment** **Operators**

|  |  |
| --- | --- |
| **Name** | **Description** |
| [:=](#_bookmark181) | Assign a value |
| [=](#_bookmark182) | Assign a value (as part of a SET statement, or as part of the SET clause in an UPDATE statement) |

• [:=](#_bookmark181)

Assignment operator. Causes the user variable on the left hand side of the operator to take on the value to its right. The value on the right hand side may be a literal value, another variable storing a value, or any legal expression that yields a scalar value, including the result of a query (provided that this value is a scalar value). You can perform multiple assignments in the same SET statement. You can perform multiple assignments in the same statement.

Unlike [=](#_bookmark182), the [:=](#_bookmark181) operator is never interpreted as a comparison operator. This means you can use [:=](#_bookmark181) in any valid SQL statement (not just in SET statements) to assign a value to a variable.

mysql> **SELECT** **@var1,** **@var2;**

-> NULL, NULL

mysql> **SELECT** **@var1** **:=** **1,** **@var2;**

-> 1, NULL

mysql> **SELECT** **@var1,** **@var2;**

-> 1, NULL

mysql> **SELECT** **@var1,** **@var2** **:=** **@var1;**

-> 1, 1

mysql> **SELECT** **@var1,** **@var2;**

-> 1, 1

mysql> **SELECT** **@var1:=COUNT(\*)** **FROM** **t1;**

-> 4

mysql> **SELECT** **@var1;**

-> 4

You can make value assignments using [:=](#_bookmark181) in other statements besides SELECT, such as UPDATE, as shown here:

mysql> **SELECT** **@var1;**

-> 4

mysql> **SELECT** **\*** **FROM** **t1;**

-> 1, 3, 5, 7

mysql> **UPDATE** **t1** **SET** **c1** **=** **2** **WHERE** **c1** **=** **@var1:=** **1;**

Query OK, 1 row affected (0 .00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> **SELECT** **@var1;**

-> 1

mysql> **SELECT** **\*** **FROM** **t1;**

-> 2, 3, 5, 7

While it is also possible both to set and to read the value of the same variable in a single SQL statement using the [:=](#_bookmark181) operator, this is not recommended. Section 9.4, “User-Defined Variables” , explains why you should avoid doing this.

• [=](#_bookmark182)

This operator is used to perform value assignments in two cases, described in the next two paragraphs.

Within a SET statement, = is treated as an assignment operator that causes the user variable on the left hand side of the operator to take on the value to its right. (In other words, when used in a SET statement, = is treated identically to [:=](#_bookmark181).) The value on the right hand side may be a literal value, another variable storing a value, or any legal expression that yields a scalar value, including the result of a query (provided that this value is a scalar value). You can perform multiple assignments in the same SET statement.

In the SET clause of an UPDATE statement, = also acts as an assignment operator; in this case, however, it causes the column named on the left hand side of the operator to assume the value given to the right, provided any WHERE conditions that are part of the UPDATE are met. You can make multiple assignments in the same SET clause of an UPDATE statement.

In any other context, = is treated as a [comparison operator](#_bookmark153).

mysql> **SELECT** **@var1,** **@var2;**

-> NULL, NULL

mysql> **SELECT** **@var1** **:=** **1,** **@var2;**

-> 1, NULL

mysql> **SELECT** **@var1,** **@var2;**

-> 1, NULL

mysql> **SELECT** **@var1,** **@var2** **:=** **@var1;**

-> 1, 1

mysql> **SELECT** **@var1,** **@var2;**

-> 1, 1

For more information, see Section 13.7.6.1, “SET Syntax for Variable Assignment” , Section 13.2.17, “UPDATE Statement” , and Section 13.2.15, “Subqueries” .

**12.5** **Flow** **Control** **Functions**

**Table** **12.7** **Flow** **Control** **Operators**

|  |  |
| --- | --- |
| **Name** | **Description** |
| [CASE](#_bookmark184) | Case operator |
| IF() | If/else construct |
| IFNULL() | Null if/else construct |



|  |  |
| --- | --- |
| **Name** | **Description** |
| NULLIF() | Return NULL if expr1 = expr2 |

• [CASE *value* WHEN *compare\_value* THEN *result* [WHEN *compare\_value* THEN *result*](#_bookmark184) [...] [ELSE *result*] END](#_bookmark184)

[CASE WHEN *condition* THEN *result* [WHEN *condition* THEN *result* ...] [ELSE](#_bookmark184) [*result*] END](#_bookmark184)

The first [CASE](#_bookmark184) syntax returns the *result* for the first *value*=*compare\_value* comparison that is true. The second syntax returns the result for the first condition that is true. If no comparison or condition is true, the result after ELSE is returned, or NULL if there is no ELSE part.

**Note**

The syntax of the [CASE](#_bookmark184) *operator* described here differs slightly from that of the SQL CASE *statement* described in Section 13.6.5.1, “CASE Statement” ,