

MySQL Globalization

Abstract

This is the MySQL Globalization extract from the MySQL 5.7 Reference Manual.

For legal information, see the [Legal Notices](#).

For help with using MySQL, please visit either the [MySQL Forums](#) or [MySQL Mailing Lists](#), where you can discuss your issues with other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the [MySQL Documentation Library](#).

Licensing information—MySQL 5.7. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL 5.7, see [this document](#) for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of MySQL 5.7, see [this document](#) for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Licensing information—MySQL Cluster. This product may include third-party software, used under license. If you are using a *Community* release of MySQL Cluster NDB 7.5, see [this document](#) for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2016-11-04 (revision: 49755)

Table of Contents

Preface and Legal Notices	v
1 Globalization	1
2 Character Set Support	3
2.1 Character Sets and Collations in General	4
2.2 Character Sets and Collations in MySQL	5
2.2.1 String Repertoire	6
2.2.2 UTF-8 for Metadata	8
2.3 Specifying Character Sets and Collations	9
2.3.1 Collation Naming Conventions	10
2.3.2 Server Character Set and Collation	11
2.3.3 Database Character Set and Collation	11
2.3.4 Table Character Set and Collation	13
2.3.5 Column Character Set and Collation	13
2.3.6 Character String Literal Character Set and Collation	15
2.3.7 The National Character Set	17
2.3.8 Character Set Introducers	17
2.3.9 Examples of Character Set and Collation Assignment	19
2.3.10 Compatibility with Other DBMSs	20
2.4 Connection Character Sets and Collations	20
2.5 Configuring Application Character Set and Collation	23
2.6 Error Message Character Set	24
2.7 Column Character Set Conversion	25
2.8 Collation Issues	27
2.8.1 Using COLLATE in SQL Statements	27
2.8.2 COLLATE Clause Precedence	28
2.8.3 Character Set and Collation Compatibility	28
2.8.4 Collation of Expressions	28
2.8.5 The binary Collation Compared to _bin Collations	29
2.8.6 Examples of the Effect of Collation	31
2.8.7 Using Collation in INFORMATION_SCHEMA Searches	33
2.9 Unicode Support	35
2.9.1 The utf8 Character Set (3-Byte UTF-8 Unicode Encoding)	36
2.9.2 The utf8mb3 Character Set (Alias for utf8)	37
2.9.3 The utf8mb4 Character Set (4-Byte UTF-8 Unicode Encoding)	37
2.9.4 The ucs2 Character Set (UCS-2 Unicode Encoding)	38
2.9.5 The utf16 Character Set (UTF-16 Unicode Encoding)	38
2.9.6 The utf16le Character Set (UTF-16LE Unicode Encoding)	39
2.9.7 The utf32 Character Set (UTF-32 Unicode Encoding)	39
2.9.8 Converting Between 3-Byte and 4-Byte Unicode Character Sets	39
2.10 Supported Character Sets and Collations	42
2.10.1 Unicode Character Sets	43
2.10.2 West European Character Sets	48
2.10.3 Central European Character Sets	49
2.10.4 South European and Middle East Character Sets	50
2.10.5 Baltic Character Sets	51
2.10.6 Cyrillic Character Sets	51
2.10.7 Asian Character Sets	52
2.10.8 The Binary Character Set	57
3 Character Set Configuration	59
4 Setting the Error Message Language	61
5 Adding a Character Set	63

5.1 Character Definition Arrays	65
5.2 String Collating Support for Complex Character Sets	66
5.3 Multi-Byte Character Support for Complex Character Sets	66
6 Adding a Collation to a Character Set	67
6.1 Collation Implementation Types	68
6.2 Choosing a Collation ID	71
6.3 Adding a Simple Collation to an 8-Bit Character Set	72
6.4 Adding a UCA Collation to a Unicode Character Set	73
6.4.1 Defining a UCA Collation Using LDML Syntax	73
6.4.2 LDML Syntax Supported in MySQL	75
6.4.3 Diagnostics During Index.xml Parsing	79
7 MySQL Server Time Zone Support	81
7.1 Staying Current with Time Zone Changes	83
7.2 Time Zone Leap Second Support	84
8 MySQL Server Locale Support	87

Preface and Legal Notices

This is the MySQL Globalization extract from the MySQL 5.7 Reference Manual.

Legal Notices

Copyright © 1997, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

Chapter 1 Globalization

This chapter covers issues of globalization, which includes internationalization (MySQL's capabilities for adapting to local use) and localization (selecting particular local conventions):

- MySQL support for character sets in SQL statements.
- How to configure the server to support different character sets.
- Selecting the language for error messages.
- How to set the server's time zone and enable per-connection time zone support.
- Selecting the locale for day and month names.

Chapter 2 Character Set Support

Table of Contents

2.1 Character Sets and Collations in General	4
2.2 Character Sets and Collations in MySQL	5
2.2.1 String Repertoire	6
2.2.2 UTF-8 for Metadata	8
2.3 Specifying Character Sets and Collations	9
2.3.1 Collation Naming Conventions	10
2.3.2 Server Character Set and Collation	11
2.3.3 Database Character Set and Collation	11
2.3.4 Table Character Set and Collation	13
2.3.5 Column Character Set and Collation	13
2.3.6 Character String Literal Character Set and Collation	15
2.3.7 The National Character Set	17
2.3.8 Character Set Introducers	17
2.3.9 Examples of Character Set and Collation Assignment	19
2.3.10 Compatibility with Other DBMSs	20
2.4 Connection Character Sets and Collations	20
2.5 Configuring Application Character Set and Collation	23
2.6 Error Message Character Set	24
2.7 Column Character Set Conversion	25
2.8 Collation Issues	27
2.8.1 Using COLLATE in SQL Statements	27
2.8.2 COLLATE Clause Precedence	28
2.8.3 Character Set and Collation Compatibility	28
2.8.4 Collation of Expressions	28
2.8.5 The binary Collation Compared to _bin Collations	29
2.8.6 Examples of the Effect of Collation	31
2.8.7 Using Collation in INFORMATION_SCHEMA Searches	33
2.9 Unicode Support	35
2.9.1 The utf8 Character Set (3-Byte UTF-8 Unicode Encoding)	36
2.9.2 The utf8mb3 Character Set (Alias for utf8)	37
2.9.3 The utf8mb4 Character Set (4-Byte UTF-8 Unicode Encoding)	37
2.9.4 The ucs2 Character Set (UCS-2 Unicode Encoding)	38
2.9.5 The utf16 Character Set (UTF-16 Unicode Encoding)	38
2.9.6 The utf16le Character Set (UTF-16LE Unicode Encoding)	39
2.9.7 The utf32 Character Set (UTF-32 Unicode Encoding)	39
2.9.8 Converting Between 3-Byte and 4-Byte Unicode Character Sets	39
2.10 Supported Character Sets and Collations	42
2.10.1 Unicode Character Sets	43
2.10.2 West European Character Sets	48
2.10.3 Central European Character Sets	49
2.10.4 South European and Middle East Character Sets	50
2.10.5 Baltic Character Sets	51
2.10.6 Cyrillic Character Sets	51
2.10.7 Asian Character Sets	52
2.10.8 The Binary Character Set	57

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. You can specify character sets at the server,

database, table, and column level. MySQL supports the use of character sets for the [MyISAM](#), [MEMORY](#), and [InnoDB](#) storage engines.

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.

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 [utf8](#) Unicode character set, issue this statement after connecting to the server:

```
SET NAMES 'utf8';
```

For more information about configuring character sets for application use and character set-related issues in client/server communication, see [Section 2.5, “Configuring Application Character Set and Collation”](#), and [Section 2.4, “Connection Character Sets and Collations”](#).

2.1 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.

2.2 Character Sets and Collations in MySQL

MySQL Server supports multiple 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 2.10, “Supported Character Sets and Collations”](#).

```
mysql> SHOW CHARACTER SET;
```

Charset	Description	Default collation	Maxlen
big5	Big5 Traditional Chinese	big5_chinese_ci	2
...			
latin1	cp1252 West European	latin1_swedish_ci	1
latin2	ISO 8859-2 Central European	latin2_general_ci	1
...			
utf8	UTF-8 Unicode	utf8_general_ci	3
ucs2	UCS-2 Unicode	ucs2_general_ci	2
...			
utf8mb4	UTF-8 Unicode	utf8mb4_general_ci	4
...			
binary	Binary pseudo charset	binary	1
...			

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. For example:

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
```

Charset	Description	Default collation	Maxlen
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

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 [latin1](#) (cp1252 West European) character set, use this statement:

```
mysql> SHOW COLLATION WHERE Charset = 'latin1';
```

Collation	Charset	Id	Default	Compiled	Sortlen
latin1_german1_ci	latin1	5		Yes	1

latin1_swedish_ci	latin1	8	Yes	Yes	1
latin1_danish_ci	latin1	15		Yes	1
latin1_german2_ci	latin1	31		Yes	2
latin1_bin	latin1	47		Yes	1
latin1_general_ci	latin1	48		Yes	1
latin1_general_cs	latin1	49		Yes	1
latin1_spanish_ci	latin1	94		Yes	1
+-----+-----+-----+-----+-----+					

The `latin1` collations have the following meanings.

Collation	Meaning
<code>latin1_bin</code>	Binary according to <code>latin1</code> encoding
<code>latin1_danish_ci</code>	Danish/Norwegian
<code>latin1_general_ci</code>	Multilingual (Western European)
<code>latin1_general_cs</code>	Multilingual (ISO Western European), case sensitive
<code>latin1_german1_ci</code>	German DIN-1 (dictionary order)
<code>latin1_german2_ci</code>	German DIN-2 (phone book order)
<code>latin1_spanish_ci</code>	Modern Spanish
<code>latin1_swedish_ci</code>	Swedish/Finnish

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 `latin1` and `utf8` are `latin1_swedish_ci` and `utf8_general_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 2.3.1, “Collation Naming Conventions”](#).

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.

2.2.1 String 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 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 supplemental characters outside the BMP range (`U+01000` 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 or to a character set that is a superset of **ASCII**. (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.

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

- The repertoire for string constants depends on string content:

```
SET NAMES utf8; SELECT 'abc';
SELECT _utf8'def';
SELECT N'MySQL';
```

Although the character set is `utf8` 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`.

- Columns having the `ascii` character set have `ASCII` repertoire because of their 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(_utf8'abc')` has `ASCII` repertoire because its argument has `ASCII` repertoire.
- 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 ([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 range of the [ascii](#) character set. For the second call, the repertoire is [UNICODE](#) because the second argument is outside the [ascii](#) character set range.

- The repertoire for function return values is determined based only on the repertoire of the 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.

2.2.2 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 | utf8 |
+-----+-----+
```

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 `latin1`. 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 2.4, “Connection Character Sets and Collations”](#).) 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 SQL standard document 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 2.8.4, “Collation of Expressions”](#).

2.3 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 **utf8** Unicode character set, issue this statement after connecting to the server:

```
SET NAMES 'utf8';
```

For more information about character set-related issues in client/server communication, see [Section 2.4, “Connection Character Sets and Collations”](#).

2.3.1 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, **utf8_general_ci** and **latin1_swedish_ci** are collations for the **utf8** 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 language name. For example, **utf8_turkish_ci** and **utf8_hungarian_ci** sort characters for the **utf8** character set using the rules of Turkish and Hungarian, respectively.
- Collation suffixes indicate whether a collation is case and accent sensitive, or binary. The following table shows the suffixes used to indicate these characteristics.

Table 2.1 Collation Case Sensitivity Suffixes

Suffix	Meaning
_ai	Accent insensitive
_as	Accent sensitive
_ci	Case insensitive
_cs	Case 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, and **latin1_general_cs** is explicitly case sensitive and implicitly accent sensitive.

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 more information, see [Section 2.8.5, “The binary Collation Compared to _bin Collations”](#).

- For Unicode character sets, collation names 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:
 - **utf8_unicode_520_ci** is based on UCA 5.2.0 weight keys (<http://www.unicode.org/Public/UCA/5.2.0/allkeys.txt>).

- `utf8_unicode_ci` (with no version named) is based on UCA 4.0.0 weight keys (<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. For more information, see [Checking Whether Tables or Indexes Must Be Rebuilt](#), and [Rebuilding or Repairing Tables or Indexes](#).

2.3.2 Server Character Set and Collation

MySQL Server has a server character set and a server collation. These can be set 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=latin1`. If you specify only a character set (for example, `latin1`) but not a collation, that is the same as saying `--character-set-server=latin1 --collation-server=latin1_swedish_ci` because `latin1_swedish_ci` is the default collation for `latin1`. Therefore, the following three commands all have the same effect:

```
shell> mysqld
shell> mysqld --character-set-server=latin1
shell> mysqld --character-set-server=latin1 \
       --collation-server=latin1_swedish_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:

```
shell> cmake . -DDEFAULT_CHARSET=latin1
```

Or:

```
shell> 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.

2.3.3 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**.

All database options are stored in a text file named **db.opt** that can be found in the database directory.

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

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.
- 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 **COLLATE** attribute. To override this, provide explicit **CHARACTER SET** and **COLLATE** attributes.

2.3.4 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.
- 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.

2.3.5 Column Character Set and Collation

Every “character” column (that is, a column of type **CHAR**, **VARCHAR**, or **TEXT**) 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** and **SET** 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 utf8 COLLATE utf8_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 `utf8` and collation `utf8_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 utf8  
) CHARACTER SET latin1 COLLATE latin1_bin;
```

The character set is specified for the column, but the collation is not. The column has character set `utf8` and the default collation for `utf8`, which is `utf8_general_ci`. To see the default collation for each character set, use the `SHOW CHARACTER SET` statement.

- 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 utf8_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 `utf8_polish_ci` and the character set is the one associated with the collation, which is `utf8`.

- 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.

2.3.6 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]
```

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

Examples:

```
SELECT 'abc';
SELECT _latin1'abc';
SELECT _binary'abc';
SELECT _utf8'abc' COLLATE utf8_danish_ci;
```

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.

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.
- 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 `utf8` character set and its default collation (that is, `utf8_general_ci`):

```
SELECT _utf8'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 `utf8_general_ci` collation (fails if the connection character set is not `utf8`):

```
SELECT 'Müller' COLLATE utf8_general_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 2.4, “Connection Character Sets and Collations”](#)), 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.

2.3.7 The National Character Set

Standard SQL defines `NCHAR` or `NATIONAL CHAR` as a way to indicate that a `CHAR` 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';
```

2.3.8 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]
```

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

Examples:

```
SELECT 'abc';
SELECT _latin1'abc';
SELECT _binary'abc';
SELECT _utf8'abc' COLLATE utf8_danish_ci;
SELECT _latin1 X'4D7953514C';
SELECT _utf8 0x4D7953514C COLLATE utf8_danish_ci;
SELECT _latin1 b'1000001';
SELECT _utf8 0b1000001 COLLATE utf8_danish_ci;
```

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.

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 unnecessary.

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.
- 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 `utf8` character set and its default collation (that is, `utf8_general_ci`):

```
SELECT _utf8'Müller';
SELECT _utf8 X'0A0D';
SELECT _utf8 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 `utf8_general_ci` collation (fails if the connection character set is not `utf8`):

```
SELECT 'Müller' COLLATE utf8_general_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 `utf8_general_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';
```

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 2.3.6, “Character String Literal Character Set and Collation”](#).

2.3.9 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_ci;
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_ci`.

2.3.10 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);
```

2.4 Connection Character Sets and Collations

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 server character set and collation are the values of the `character_set_server` and `collation_server` system variables.
- The character set and collation of the default database are the values of the `character_set_database` and `collation_database` system variables.

Additional character set and collation system variables are involved in handling traffic for the connection between a client and the server. Every client has connection-related character set and collation system variables.

A “connection” is what you make when you connect to the server. The client sends SQL statements, such as queries, over the connection to the server. The server sends responses, such as result sets or error messages, over the connection back to the client. This leads to several questions about character set and collation handling for client connections, each of which can be answered in terms of system variables:

- What character set is the statement in when it leaves the client?

The server takes the `character_set_client` system variable to be the character set in which statements are sent by the client.

- What character set should the server translate a statement to after receiving it?

For this, the server uses the `character_set_connection` and `collation_connection` system variables. It converts statements sent by the client from `character_set_client` to `character_set_connection` (except for string literals that have an introducer such as `_latin1` or `_utf8`). `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.

- What character set should the server translate to before shipping result sets or error messages 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, and result metadata such as column names and error messages.

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*.

Two statements affect the connection-related character set variables as a group:

- `SET NAMES 'charset_name' [COLLATE 'collation_name']`

`SET NAMES` indicates what character set the client will use 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.)

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, use the optional `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`. 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

`ucs2`, `utf16`, `utf16le`, and `utf32` cannot be used as a client character set, which means that they do not work for `SET NAMES` or `SET CHARACTER SET`.

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

- In the absence of other information, the programs use the compiled-in default character set, usually `latin1`.
- The programs 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, `ucs2` is not supported as a connection character set.

C applications can use character set autodetection based on the OS setting by invoking `mysql_options()` as follows before connecting to the server:

```
mysql_options(mysql,
               MYSQL_SET_CHARSET_NAME,
               MYSQL_AUTODETECT_CHARSET_NAME);
```

- The programs support a `--default-character-set` option, which enables users to specify the character set explicitly to override whatever default the client otherwise determines.

When a client connects to the server, it sends the name of the character set that it wants to use. The server uses the name to set the `character_set_client`, `character_set_results`, and `character_set_connection` system variables. In effect, the server performs a `SET NAMES` operation using the character set name.

With the `mysql` client, to use a character set different from the default, you could explicitly execute `SET NAMES` every time you start up. To accomplish the same result more easily, add the `--default-character-set` option setting to your `mysql` command line or in your option file. For example, the following option file setting changes the three connection-related character set 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 utf8
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.

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 `latin1` just before sending results back. Conversion may be lossy if there are characters that are not in both character sets.

If you want 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;
```

To see the values of the character set and collation system variables that apply to your connection, use these statements:

```
SHOW VARIABLES LIKE 'character_set%';
SHOW VARIABLES LIKE 'collation%';
```

You must also consider the environment within which your MySQL applications execute. See [Section 2.5, “Configuring Application Character Set and Collation”](#).

For more information about character sets and error messages, see [Section 2.6, “Error Message Character Set”](#).

2.5 Configuring Application Character Set and Collation

For applications that store data using the default MySQL character set and collation ([latin1](#), [latin1_swedish_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 require [utf8](#), whereas applications that use another database might require [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 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 [utf8](#) character set and [utf8_general_ci](#) collation.

Specify character settings per database. To create a database such that its tables will use a given default character set and collation for data storage, use a [CREATE DATABASE](#) statement like this:

```
CREATE DATABASE mydb
  DEFAULT CHARACTER SET utf8
  DEFAULT COLLATE utf8_general_ci;
```

Tables created in the database will use [utf8](#) and [utf8_general_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 'utf8'](#) 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, for connections made using [mysql](#), you can specify the [--default-character-set=utf8](#) command-line option to achieve the same effect as [SET NAMES 'utf8'](#).

For more information about configuring client connections, see [Section 2.4, “Connection Character Sets and Collations”](#).

If you change the default character set or collation for a database, stored routines that use the database 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 [CREATE PROCEDURE and CREATE FUNCTION Syntax](#).)

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=utf8
collation-server=utf8_general_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 'utf8'"` option to cause `SET NAMES` to be executed automatically for each client that connects. However, this will yield inconsistent results because the `init_connect` value is not executed for users who have the `SUPER` privilege.

Specify character settings at MySQL configuration time. To select a character set and collation when you configure and build MySQL from source, use the `DEFAULT_CHARSET` and `DEFAULT_COLLATION` options for `CMake`:

```
shell> cmake . -DDEFAULT_CHARSET=utf8 \
          -DDEFAULT_COLLATION=utf8_general_ci
```

The resulting server uses `utf8` and `utf8_general_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. If you will 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" />
```

2.6 Error Message Character Set

This section describes how the MySQL server uses character sets for constructing error messages and returning them to clients. For information about the language of error messages (rather than the character set), see [Chapter 4, Setting the Error Message Language](#).

The server constructs error messages using UTF-8 and returns them to clients in the character set specified by the `character_set_results` system variable.

The server constructs error messages as follows:

- The message template uses UTF-8.
- 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` unique column, the resulting error message uses UTF-8 with some bytes hexadecimal encoded:

```
Duplicate entry 'A\xC3\x9F' for key 1
```

To return a message to the client after it has been constructed, 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 `utf8`, either, because that matches the original error message character set.

For characters that 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 (`0x01000` 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 2.4, “Connection Character Sets and Collations”](#).

The encoding that occurs during the conversion to `character_set_results` before returning error messages to clients can result in different message content compared to earlier versions (before MySQL 5.5). For example, if an error occurs for an attempt to drop a table named `ペ` (KATAKANA LETTER PE) and `character_set_results` is a character set such as `latin1` that does not contain that character, the resulting message sent to the client has an encoded table name:

```
ERROR 1051 (42S02): Unknown table '\30DA'
```

Before MySQL 5.5, the name is not encoded:

```
ERROR 1051 (42S02): Unknown table 'ペ'
```

2.7 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](#), [VARBINARY](#), [BLOB](#)), 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](#), [VARCHAR](#), [TEXT](#)), 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 will be 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 [utf8](#) so that you can store values from many languages. The following statement accomplishes this:

```
ALTER TABLE t MODIFY col1 CHAR(50) CHARACTER SET utf8;
```

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 [latin1](#). 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 later. In that case, MySQL would store new values in the column using [latin1](#), and the column will 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 will 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 [ALTER TABLE Syntax](#).

2.8 Collation Issues

The following sections discuss various aspects of character set collations.

2.8.1 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;
```

2.8.2 COLLATE Clause Precedence

The **COLLATE** clause has high precedence (higher than `||`), so the following two expressions are equivalent:

```
x || y COLLATE z
x || (y COLLATE z)
```

2.8.3 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'
```

2.8.4 Collation of 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 **charset_name**:

```
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:

```
SELECT x FROM T WHERE x = 'Y';
```

Should the comparison use the collation of the column **x**, or of the string literal **'Y'**? Both **x** and **'Y'** have collations, so which collation takes precedence?

Standard SQL resolves such questions using what used to be called “coercibility” rules. 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.
- **NULL** or an expression that is derived from **NULL** has a coercibility of 5.

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(utf8_column, latin1_column) FROM t1;
```

It returns a result that has a character set of `utf8` and the same collation as `utf8_column`. Values of `latin1_column` are automatically converted to `utf8` 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, except that it is for collations rather than data types.

Although automatic conversion is not in the SQL standard, the SQL standard document 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.

Examples:

Comparison	Collation Used
<code>column1 = 'A'</code>	Use collation of <code>column1</code>
<code>column1 = 'A' COLLATE x</code>	Use collation of <code>'A' COLLATE x</code>
<code>column1 COLLATE x = 'A' COLLATE y</code>	Error

The `COERCIBILITY()` function can be used to determine the coercibility of a string expression:

```
mysql> SELECT COERCIBILITY('A' COLLATE latin1_swedish_ci);
-> 0
mysql> SELECT COERCIBILITY(VERSION());
-> 3
mysql> SELECT COERCIBILITY('A');
-> 4
```

See [Information Functions](#).

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 [Type Conversion in Expression Evaluation](#).

2.8.5 The binary Collation Compared to `_bin` Collations

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

Binary strings (as stored using the `BINARY`, `VARBINARY`, and `BLOB` 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.

Nonbinary strings (as stored using the `CHAR`, `VARCHAR`, and `TEXT` 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. One of these is the binary collation for the character set, indicated by a `_bin` suffix in the collation name. For example, the binary collations for `latin1` and `utf8` are named `latin1_bin` and `utf8_bin`, respectively.

The `binary` collation differs from the `_bin` collations in several respects.

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 the `_bin` collation, 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 from another column that has a different character set:

```
UPDATE t1 SET utf8_bin_column=latin1_column;
INSERT INTO t1 (latin1_column) SELECT utf8_bin_column FROM t2;
```

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

```
SET NAMES latin1;
INSERT INTO t1 (utf8_bin_column) VALUES ('string-in-latin1');
```

- When sending results from the server to a client:

```
SET NAMES latin1;
SELECT utf8_bin_column FROM t2;
```

For binary string columns, no conversion occurs. For the preceding cases, 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 latin1 COLLATE latin1_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 be converted to a nonbinary string:

```
mysql> SET NAMES binary;
mysql> SELECT LOWER('aA'), LOWER(CONVERT('aA' USING latin1));
+-----+-----+
| LOWER('aA') | LOWER(CONVERT('aA' USING latin1)) |
+-----+-----+
| aA          | aa                                |
+-----+-----+
```

Trailing space handling in comparisons. Nonbinary strings have [PADSPACE](#) behavior for all collations, including [_bin](#) collations. Trailing spaces are insignificant in comparisons:

```
mysql> SET NAMES utf8 COLLATE utf8_bin;
mysql> SELECT 'a ' = 'a';
+-----+
| 'a ' = 'a' |
+-----+
|          1 |
+-----+
```

For binary strings, all characters are significant in comparisons, including trailing spaces:

```
mysql> SET NAMES binary;
mysql> SELECT 'a ' = 'a';
+-----+
| 'a ' = 'a' |
+-----+
|          0 |
+-----+
```

Trailing space handling for inserts and retrievals. [CHAR\(N\)](#) columns store nonbinary strings. Values shorter than *N* characters are extended with spaces on insertion. For retrieval, trailing spaces are removed.

[BINARY\(N\)](#) columns store binary strings. Values shorter than *N* bytes are extended with [0x00](#) bytes on insertion. For retrieval, nothing is removed; a value of the declared length is always returned.

```
mysql> CREATE TABLE t1 (
    a CHAR(10) CHARACTER SET utf8 COLLATE utf8_bin,
    b BINARY(10)
);
mysql> INSERT INTO t1 VALUES ('a','a');
mysql> SELECT HEX(a), HEX(b) FROM t1;
+-----+-----+
| HEX(a) | HEX(b) |
+-----+-----+
| 61     | 61000000000000000000000000000000 |
+-----+-----+
```

2.8.6 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 the U with two dots over it (ü), which the Germans call “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 utf8;
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 utf8 COLLATE utf8_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, you'll get these results in comparisons:

```
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 `utf8_unicode_ci` (the sorting shown is done according to the German DIN 5007 standard).

2.8.7 Using Collation in INFORMATION_SCHEMA Searches

String columns in `INFORMATION_SCHEMA` tables have a collation of `utf8_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 insensitive, depending on the characteristics of the underlying file system and the value of the `lower_case_table_names` system variable. 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; see also Bug #34921.

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. Changing the value of `lower_case_table_names` to 1 or 2 causes the second query to return the same (nonempty) result as the first query.

On Windows or OS X, 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 `utf8_general_ci` collation is not used for `INFORMATION_SCHEMA` queries when searching for values that correspond to objects represented in the file system. It is a result of file system-scanning optimizations implemented for `INFORMATION_SCHEMA` searches. For information about these optimizations, see [Optimizing INFORMATION_SCHEMA Queries](#).

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 2.8.1, “Using COLLATE in SQL Statements”](#)). 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 utf8_general_ci = 'test';
+-----+
| SCHEMA_NAME |
+-----+
| test       |
+-----+
mysql> SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA
        WHERE SCHEMA_NAME COLLATE utf8_general_ci = 'TEST';
+-----+
| SCHEMA_NAME |
+-----+
| test       |
+-----+
```

In the preceding queries, it is important to apply the `COLLATE` clause to the `INFORMATION_SCHEMA` column name. Applying `COLLATE` to the comparison value has no effect.

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 `utf8_bin`) is case sensitive; the others are not:

```
WHERE TABLE_NAME COLLATE utf8_bin = 'City'
WHERE TABLE_NAME COLLATE utf8_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 `utf8_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 |
+-----+

```

2.9 Unicode Support

The initial implementation of Unicode support (in MySQL 4.1) included two character sets for storing Unicode data:

- **utf8**, a UTF-8 encoding of the Unicode character set using one to three bytes per character.
- **ucs2**, the UCS-2 encoding of the Unicode character set using 16 bits per character.

These two character sets support the characters from the Basic Multilingual Plane (BMP) of Unicode Version 3.0. BMP characters have these characteristics:

- Their code values are between 0 and 65535 (or **U+0000** .. **U+FFFF**).
- They can be encoded with 8, 16, or 24 bits, as in **utf8**.
- They can be encoded with a fixed 16-bit word, as in **ucs2**.
- They are sufficient for almost all characters in major languages.

Characters not supported by the aforementioned character sets include supplementary characters that lie outside the BMP. Characters outside the BMP compare as REPLACEMENT CHARACTER and convert to '?' when converted to a Unicode character set.

Unicode support for supplementary characters requires character sets that have a broader range (including non-BMP characters) and therefore take more space. The following table shows a brief feature comparison of the original and expanded Unicode support.

Before MySQL 5.5	MySQL 5.5 and up
All Unicode 3.0 characters	All Unicode 5.0 and 6.0 characters
No supplementary characters	With supplementary characters
utf8 character set for up to three bytes, BMP only	No change
ucs2 character set, BMP only	No change
	utf8mb4 character set for up to four bytes, BMP or supplemental
	utf16 character set, BMP or supplemental
	utf16le character set, BMP or supplemental
	utf32 character set, BMP or supplemental

If you want to use the character sets that are “wider” than the original **utf8** and **ucs2** character sets, there are potential incompatibility issues for your applications; see [Section 2.9.8, “Converting Between 3-Byte and 4-Byte Unicode Character Sets”](#). That section also describes how to convert tables from **utf8** to the (4-byte) **utf8mb4** character set, and what constraints may apply in doing so.

MySQL supports these Unicode character sets:

- [utf8](#), a UTF-8 encoding of the Unicode character set using one to three bytes per character.
- [utf8mb4](#), a UTF-8 encoding of the Unicode character set using one to four bytes per character.
- [ucs2](#), the UCS-2 encoding of the Unicode character set using 16 bits per character.
- [utf16](#), the UTF-16 encoding for the Unicode character set; 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 32 bits per character.

[utf8](#) and [ucs2](#) support BMP characters. [utf8mb4](#), [utf16](#), [utf16le](#), and [utf32](#) support BMP and supplementary characters.

A similar set of collations is available for most Unicode character sets. For example, each has a Danish collation, the names of which are [ucs2_danish_ci](#), [utf16_danish_ci](#), [utf32_danish_ci](#), [utf8_danish_ci](#), and [utf8mb4_danish_ci](#). The exception is [utf16le](#), which has only two collations. For a description of Unicode collations and their differentiating properties, including collation properties for supplementary characters, see [Section 2.10.1, “Unicode Character Sets”](#).

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 will need 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 need to communicate with the server using Unicode should set the client character set accordingly; for example, by issuing a `SET NAMES 'utf8'` statement. [ucs2](#), [utf16](#), [utf16le](#), and [utf32](#) cannot be used as a client character set, which means that they do not work for `SET NAMES` or `SET CHARACTER SET`. (See [Section 2.4, “Connection Character Sets and Collations”](#).)

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

2.9.1 The utf8 Character Set (3-Byte UTF-8 Unicode Encoding)

UTF-8 (Unicode Transformation Format with 8-bit units) is an alternative way to store Unicode data. It is implemented according to RFC 3629, which describes encoding sequences that take from one to four bytes. (An older standard for UTF-8 encoding, RFC 2279, describes UTF-8 sequences that take from one to six bytes. RFC 3629 renders RFC 2279 obsolete; for this reason, sequences with five and six bytes are no longer used.)

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.

The [utf8](#) character set in MySQL has these characteristics:

- No support for supplementary characters (BMP characters only).
- A maximum of three bytes per multibyte character.

Exactly the same set of characters is available in [utf8](#) and [ucs2](#). That is, they have the same repertoire.

Tip

To save space with UTF-8, use [VARCHAR](#) instead of [CHAR](#). Otherwise, MySQL must reserve three bytes for each character in a [CHAR CHARACTER SET utf8](#) column because that is the maximum possible character length. For example, MySQL must reserve 30 bytes for a [CHAR\(10\) CHARACTER SET utf8](#) column.

For additional information about data type storage, see [Data Type Storage Requirements](#). For information about [InnoDB](#) physical row storage, including how [InnoDB](#) tables that use [COMPACT](#) row format handle UTF-8 [CHAR\(N\)](#) columns internally, see [Physical Row Structure of InnoDB Tables](#).

2.9.2 The utf8mb3 Character Set (Alias for utf8)

The [utf8](#) character set uses a maximum of three bytes per character. To make this character limit explicit (much as the limit of four bytes per character is explicit in the [utf8mb4](#) character set name), use the character set name [utf8mb3](#), which is an alias for [utf8](#). [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 utf8) COLLATE utf8_czech_ci;
```

MySQL immediately converts instances of [utf8mb3](#) in an alias to [utf8](#), so 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 will see the true name, [utf8](#) or [utf8_collation_substring](#).

The [utf8mb3](#) alias is also valid in certain places other than [CHARACTER SET](#) clauses. For example, these are legal:

```
mysqld --character-set-server=utf8mb3
SET NAMES 'utf8mb3'; /* and other SET statements that have similar effect */
SELECT _utf8mb3 'a';
```

There is no [utf8mb3](#) alias for the corresponding [utf8](#) collation for collation names that include a version number to indicate the Unicode Collation Algorithm version on which the collation is based (for example, [utf8_unicode520_ci](#)).

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

The character set named [utf8](#) uses a maximum of three bytes per character and contains only BMP characters. The [utf8mb4](#) character set uses a maximum of four bytes per character supports supplemental characters:

- For a BMP character, [utf8](#) and [utf8mb4](#) have identical storage characteristics: same code values, same encoding, same length.

- For a supplementary character, `utf8` cannot store the character at all, whereas `utf8mb4` requires four bytes to store it. Because `utf8` cannot store the character at all, you have no supplementary characters in `utf8` columns and need not worry about converting characters or losing data when upgrading `utf8` data from older versions of MySQL.

`utf8mb4` is a superset of `utf8`, so for an operation such as the following concatenation, the result has character set `utf8mb4` and the collation of `utf8mb4_col`:

```
SELECT CONCAT(utf8_col, utf8mb4_col);
```

Similarly, the following comparison in the `WHERE` clause works according to the collation of `utf8mb4_col`:

```
SELECT * FROM utf8_tbl, utf8mb4_tbl
WHERE utf8_tbl.utf8_col = utf8mb4_tbl.utf8mb4_col;
```

Tip: To save space with `utf8mb4`, use `VARCHAR` instead of `CHAR`. Otherwise, MySQL must reserve four bytes for each character in a `CHAR CHARACTER SET utf8mb4` column because that is the maximum possible length. For example, MySQL must reserve 40 bytes for a `CHAR(10) CHARACTER SET utf8mb4` column.

2.9.4 The ucs2 Character Set (UCS-2 Unicode Encoding)

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 Home Page](#).

In MySQL, the `ucs2` character set is a fixed-length 16-bit encoding for Unicode BMP characters.

2.9.5 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](#) 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);
```

2.9.6 The utf16le Character Set (UTF-16LE Unicode Encoding)

This is the same as `utf16` but is little-endian rather than big-endian.

2.9.7 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 */
```

2.9.8 Converting Between 3-Byte and 4-Byte Unicode Character Sets

This section describes issues that you may face when converting from the `utf8` character set to the `utf8mb4` character set, or vice versa.

Note

The discussion here focuses primarily on converting between `utf8` and `utf8mb4`, but similar principles apply to converting between the `ucs2` character set and character sets such as `utf16` or `utf32`.

The `utf8` and `utf8mb4` character sets differ as follows:

- `utf8` supports only characters in the Basic Multilingual Plane (BMP). `utf8mb4` additionally supports supplementary characters that lie outside the BMP.
- `utf8` uses a maximum of three bytes per character. `utf8mb4` uses a maximum of four bytes per character.

One advantage of converting from `utf8` to `utf8mb4` is that this enables applications to use supplementary characters. One tradeoff is that this may increase data storage space requirements.

In most respects, converting from `utf8` to `utf8mb4` should present few problems. These are the primary potential areas of incompatibility:

- For the variable-length character data types (`VARCHAR` and the `TEXT` types), the maximum permitted length in characters is less for `utf8mb4` columns than for `utf8` columns.
- For all character data types (`CHAR`, `VARCHAR`, and the `TEXT` types), the maximum number of characters that can be indexed is less for `utf8mb4` columns than for `utf8` columns.

Consequently, to convert tables from `utf8` to `utf8mb4`, it may be necessary to change some column or index definitions.

Tables can be converted from `utf8` to `utf8mb4` by using `ALTER TABLE`. Suppose that a table was originally defined as follows:

```
CREATE TABLE t1 (
  col1 CHAR(10) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
  col2 CHAR(10) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL
) CHARACTER SET utf8;
```

The following statement converts `t1` to use `utf8mb4`:

```
ALTER TABLE t1
  DEFAULT CHARACTER SET utf8mb4,
  MODIFY col1 CHAR(10)
    CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL,
  MODIFY col2 CHAR(10)
    CHARACTER SET utf8mb4 COLLATE utf8mb4_bin NOT NULL;
```

In terms of table content, conversion from `utf8` to `utf8mb4` presents no problems:

- For a BMP character, `utf8` and `utf8mb4` have identical storage characteristics: same code values, same encoding, same length.
- For a supplementary character, `utf8` cannot store the character at all, whereas `utf8mb4` requires four bytes. Because `utf8` cannot store the character at all, `utf8` columns have no supplementary characters and you need not worry about converting characters or losing data when converting to `utf8mb4`.

In terms of table structure, the catch when converting from `utf8` 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`, `VARCHAR`, and `TEXT` data types, watch for these issues when converting your MySQL tables:

- Check all definitions of `utf8` columns and make sure they will not exceed the maximum length for the storage engine.

- Check all indexes on `utf8` columns and make sure they will 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 `utf8` rather than `utf8mb4`.

Here are some examples where structural changes may be needed:

- A `TINYTEXT` 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` column that uses `utf8` 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`.

Similarly, a very long `VARCHAR` column may need to be changed to one of the longer `TEXT` types if you want to convert it from `utf8` to `utf8mb4`.

- `InnoDB` has a maximum index length of 767 bytes for tables that use `COMPACT` or `REDUNDANT` row format, so for `utf8` or `utf8mb4` columns, you can index a maximum of 255 or 191 characters, respectively. If you currently have `utf8` 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 utf8, 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, you can enable the `innodb_large_prefix` option to permit `index key prefixes` longer than 767 bytes (up to 3072 bytes). Creating such tables also requires the option values `innodb_file_format=barracuda` and `innodb_file_per_table=true`.) In this case, enabling the `innodb_large_prefix` option enables you to index a maximum of 1024 or 768 characters for `utf8` or `utf8mb4` columns, respectively. For related information, see [Limits on InnoDB Tables](#).

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 `utf8` to `utf8mb4` without problems, using `ALTER TABLE` as described previously.

The following items summarize other potential areas of incompatibility:

- Performance of 4-byte UTF-8 (`utf8mb4`) is slower than for 3-byte UTF-8 (`utf8`). To avoid this penalty, continue to use `utf8`.
- `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. More generally, applications cannot send `utf8mb4`, `utf16`, `utf16le`, or `utf32` data to an older server that does not understand it:

- `utf8mb4`, `utf16`, and `utf32` are not recognized before MySQL 5.5.3.
- `utf16le` is not recognized before MySQL 5.6.1.
- For replication, if character sets that support supplementary characters are to be used on the master, all slaves must understand them as well. If you attempt to replicate from a newer master to an older slave, `utf8` data will be seen as `utf8` by the slave and should replicate correctly. But you cannot send `utf8mb4`, `utf16`, `utf16le`, or `utf32` data to an older slave that does not understand it:
 - `utf8mb4`, `utf16`, and `utf32` are not recognized before MySQL 5.5.3.
 - `utf16le` is not recognized before MySQL 5.6.1.

Also, keep in mind the general principle that if a table has different definitions on the master and slave, this can lead to unexpected results. For example, the differences in maximum index key length make it risky to use `utf8` on the master and `utf8mb4` on the slave.

If you have converted to `utf8mb4`, `utf16`, `utf16le`, or `utf32`, and then decide to convert back to `utf8` or `ucs2` (for example, to downgrade to an older version of MySQL), these considerations apply:

- `utf8` 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 will see `utf8` in the dump file object definitions and create new objects that use the (3-byte) `utf8` character set.

2.10 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
big5	Big5 Traditional Chinese	big5_chinese_ci	2
dec8	DEC West European	dec8_swedish_ci	1
cp850	DOS West European	cp850_general_ci	1
hp8	HP West European	hp8_english_ci	1
koi8r	KOI8-R Relcom Russian	koi8r_general_ci	1
latin1	cp1252 West European	latin1_swedish_ci	1
latin2	ISO 8859-2 Central European	latin2_general_ci	1
swe7	7bit Swedish	swe7_swedish_ci	1
ascii	US ASCII	ascii_general_ci	1
ujis	EUC-JP Japanese	ujis_japanese_ci	3
sjis	Shift-JIS Japanese	sjis_japanese_ci	2
hebrew	ISO 8859-8 Hebrew	hebrew_general_ci	1
tis620	TIS620 Thai	tis620_thai_ci	1
euckr	EUC-KR Korean	euckr_korean_ci	2
koi8u	KOI8-U Ukrainian	koi8u_general_ci	1
gb2312	GB2312 Simplified Chinese	gb2312_chinese_ci	2

greek	ISO 8859-7 Greek	greek_general_ci	1	
cp1250	Windows Central European	cp1250_general_ci	1	
gbk	GBK Simplified Chinese	gbk_chinese_ci	2	
latin5	ISO 8859-9 Turkish	latin5_turkish_ci	1	
armSCII8	ARMSCII-8 Armenian	armSCII8_general_ci	1	
utf8	UTF-8 Unicode	utf8_general_ci	3	
ucs2	UCS-2 Unicode	ucs2_general_ci	2	
cp866	DOS Russian	cp866_general_ci	1	
keybcs2	DOS Kamenicky Czech-Slovak	keybcs2_general_ci	1	
macce	Mac Central European	macce_general_ci	1	
macroman	Mac West European	macroman_general_ci	1	
cp852	DOS Central European	cp852_general_ci	1	
latin7	ISO 8859-13 Baltic	latin7_general_ci	1	
utf8mb4	UTF-8 Unicode	utf8mb4_general_ci	4	
cp1251	Windows Cyrillic	cp1251_general_ci	1	
utf16	UTF-16 Unicode	utf16_general_ci	4	
utf16le	UTF-16LE Unicode	utf16le_general_ci	4	
cp1256	Windows Arabic	cp1256_general_ci	1	
cp1257	Windows Baltic	cp1257_general_ci	1	
utf32	UTF-32 Unicode	utf32_general_ci	4	
binary	Binary pseudo charset	binary	1	
geostd8	GEOSTD8 Georgian	geostd8_general_ci	1	
cp932	SJIS for Windows Japanese	cp932_japanese_ci	2	
eucjpms	UJIS for Windows Japanese	eucjpms_japanese_ci	3	
gb18030	China National Standard GB18030	gb18030_chinese_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.

2.10.1 Unicode Character Sets

MySQL supports multiple Unicode character sets:

- [utf8](#), a UTF-8 encoding of the Unicode character set using one to three bytes per character.
- [utf8mb4](#), a UTF-8 encoding of the Unicode character set using one to four bytes per character.
- [ucs2](#), the UCS-2 encoding of the Unicode character set using 16 bits per character.
- [utf16](#), the UTF-16 encoding for the Unicode character set; 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 32 bits per character.

[utf8](#) and [ucs2](#) support Basic Multilingual Plane (BMP) characters. [utf8mb4](#), [utf16](#), [utf16le](#), and [utf32](#) support BMP and supplementary characters.

This section describes the collations available for Unicode character sets and their differentiating properties. For general information about Unicode, see [Section 2.9, “Unicode Support”](#).

Most Unicode character sets have a general collection (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 [utf8](#), [utf8_general_ci](#) and [utf8_bin](#) are its general and binary collations, and [utf8_danish_ci](#) is one of its language-specific collations.

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`.

A language name shown in the following table indicates a language-specific collation. Unicode character sets may include collations for one or more of these languages.

Table 2.2 Unicode Collation Language Specifiers

Language	Language Specifier
Classical Latin	<code>roman</code>
Croatian	<code>croatian</code>
Czech	<code>czech</code>
Danish	<code>danish</code>
Esperanto	<code>esperanto</code>
Estonian	<code>estonian</code>
German phone book order	<code>german2</code>
Hungarian	<code>hungarian</code>
Icelandic	<code>icelandic</code>
Latvian	<code>latvian</code>
Lithuanian	<code>lithuanian</code>
Persian	<code>persian</code>
Polish	<code>polish</code>
Romanian	<code>romanian</code>
Sinhala	<code>sinhala</code>
Slovak	<code>slovak</code>
Slovenian	<code>slovenian</code>
Modern Spanish	<code>spanish</code>
Traditional Spanish	<code>spanish2</code>
Swedish	<code>swedish</code>
Turkish	<code>turkish</code>
Vietnamese	<code>vietnamese</code>

Croatian collations are tailored for these Croatian letters: Č, Ć, Dž, Đ, Lj, Nj, Š, Ž.

Danish collations may also be used for Norwegian.

For Classical Latin collations, `I` and `J` compare as equal, and `U` and `V` compare as equal.

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.

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 = Υ < Ö
```

For questions about particular language orderings, [unicode.org](http://www.unicode.org) provides Common Locale Data Repository (CLDR) collation charts at <http://www.unicode.org/cldr/charts/30/collation/index.html>.

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. For more information, see [Checking Whether Tables or Indexes Must Be Rebuilt](#), and [Rebuilding or Repairing Tables or Indexes](#).

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/UC4.0.0/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 primarily Vietnamese, Yoruba, and some smaller languages such as 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 later than 4.0.0 include the version in the collation name. Thus, `utf8_unicode_520_ci` is based on UCA 5.2.0 weight keys (<http://www.unicode.org/Public/UC5.2.0/allkeys.txt>). For collations of `utf8` that include a UCA version, there is no `utf8mb3` alias; see [Section 2.9.2, “The utf8mb3 Character Set \(Alias for utf8\)”](#).

MySQL implements language-specific Unicode collations if the ordering based only on UCA does not work well for a language. Language-specific collations are UCA-based, with additional language tailoring rules.

`LOWER()` and `UPPER()` perform case folding according to the collation of their argument. A character that has uppercase and lowercase versions only in a Unicode version more recent than 4.0.0 is converted by these functions only if the argument has a collation that uses a recent enough UCA version.

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 `utf8_general_ci` collation are faster, but slightly less correct, than comparisons for `utf8_unicode_ci`. The reason for this is that `utf8_unicode_ci` supports mappings such as expansions; that is, when one character compares as equal to combinations of other characters. For example, in German and some other languages ß is equal to ss. `utf8_unicode_ci` also supports contractions and ignorable characters. `utf8_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 `utf8_general_ci` and `utf8_unicode_ci` (for the effect of this in comparisons or searches, see [Section 2.8.6, “Examples of the Effect of Collation”](#)):

```
Ä = A
Ö = O
Ü = U
```

A difference between the collations is that this is true for `utf8_general_ci`:

```
ß = s
```

Whereas this is true for `utf8_unicode_ci`, which supports the German DIN-1 ordering (also known as dictionary order):

```
ß = ss
```

MySQL implements `utf8` language-specific collations if the ordering with `utf8_unicode_ci` does not work well for a language. For example, `utf8_unicode_ci` works fine for German dictionary order and French, so there is no need to create special `utf8` collations.

`utf8_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 `utf8_general_ci` because it is faster. If this is not acceptable (for example, if you require German dictionary order), use `utf8_unicode_ci` because it is more accurate.

If you require German DIN-2 (phone book) ordering, use the `utf8_german2_ci` collation, which compares the following sets of characters equal:

```
Ä = Æ = AE
Ö = Œ = OE
Ü = UE
ß = ss
```

`utf8_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 `utf8_german_ci` corresponding to `latin1_german_ci` for German dictionary order because `utf8_general_ci` suffices.

For all Unicode collations except the binary (`_bin`) collations, MySQL performs a table lookup to find a character's collating weight. This weight can be displayed using the `WEIGHT_STRING()` function. (See [String Functions](#).) If 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`), weight = 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`. With UCA 5.2.0 collations, all palochkas sort together.

- For supplementary characters in general collations, the weight is the weight for `0xffff REPLACEMENT CHARACTER`. For supplementary characters in UCA 4.0.0 collations, their collating weight is `0xffff`. 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 (0xffffd); /* 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 [0xffffd](#). (Were the [utf32_general_ci](#) collation used instead, the result is 1 because all three characters have a weight of [0xffffd](#) 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](#). [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 collation 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 [utf8_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	utf8	utf16
0FF9D	HALFWIDTH KATAKANA LETTER N	EF BE 9D	FF 9D
10384	UGARITIC LETTER DELTA	F0 90 8E 84	D8 00 DF 84

The two characters in the chart are in order by code point value because `0xff9d < 0x10384`. And they are in order by `utf8` 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, `utf8_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.

2.10.2 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)
- `hp8` (HP Western European) collations:
 - `hp8_bin`
 - `hp8_english_ci` (default)
- `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)

`latin1` is the default character set. 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 2.8.6, “Examples of the Effect of Collation”](#).

- `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)
- `swe7` (7bit Swedish) collations:
 - `swe7_bin`
 - `swe7_swedish_ci` (default)

2.10.3 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)

2.10.4 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` (ARMSII-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)

2.10.5 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](#)

2.10.6 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)

2.10.7 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 2.10.7.1, “The cp932 Character Set”](#), for additional information about the [cp932](#) and [sjis](#) character sets. See [Section 2.10.7.2, “The gb18030 Character Set”](#), 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 [MySQL 5.7 FAQ: MySQL Chinese, Japanese, and Korean Character Sets](#).

- [big5](#) (Big5 Traditional Chinese) collations:
 - [big5_bin](#)
 - [big5_chinese_ci](#) (default)
- [cp932](#) (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` (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.

2.10.7.1 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-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://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 **eucjpm**s, and solves the problems with **sjis**/**ujis** conversion. For details, please refer to <http://www.sjfaq.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
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 [Server Command Options](#).

2.10.7.2 The gb18030 Character Set

In MySQL, the `gb18030` character set, introduced in MySQL 5.7.4, 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 `utf8` 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.

2.10.8 The Binary Character Set

The `binary` character set is the character set of 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. The effect is that 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 information about the differences between the `binary` collation of the `binary` character set and the `_bin` collations of nonbinary character sets, see [Section 2.8.5, “The binary Collation Compared to _bin Collations”](#).

To convert a string expression to a binary string, any of these constructs are equivalent:

```
BINARY expr
CAST(expr AS BINARY)
CONVERT(expr USING BINARY)
```

If *expr* 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 2.3.8, “Character Set Introducers”](#).

Chapter 3 Character Set Configuration

You can change the default server character set and collation with the `--character-set-server` and `--collation-server` options when you start the server. The collation must be a legal collation for the default character set. (Use the `SHOW COLLATION` statement to determine which collations are available for each character set.) See [Server Command Options](#).

If you try to use a character set that is not compiled into your binary, you might run into the following problems:

- 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 in question. 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 5.7/share/charsets"
```

- The character set is a complex character set that cannot be loaded dynamically. In this case, 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 6.4, “Adding a UCA Collation to a Unicode Character Set”](#).

- The character set is a dynamic character set, but you do not have a configuration file for it. In this case, you should install the configuration file for the character set from a new MySQL distribution.
- If your character set index file does not contain the name for the character set, your program displays an error message. The file is named `Index.xml` and the message is:

```
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.

For **MyISAM** tables, you can check the character set name and number for a table with `myisamchk -dvv tbl_name`.

Chapter 4 Setting the Error Message Language

By default, `mysqld` produces error messages in English, but they can also be displayed 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.

You can select which language the server uses for error messages using the instructions in this section.

The server searches for the error message file in two locations:

- It tries to find 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:

```
shell> 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.

- 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.

The `lc_messages_dir` system variable has only a global value and is read only. `lc_messages` has global and session values and can be modified at runtime, so the error message language can be changed while the server is running, and individual clients each can have a different error message language by changing their session `lc_messages` value to a different 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:

```
mysql> SET lc_messages = 'en_US';
```

By default, the language files are located in the `share/mysql/LANGUAGE` directory under the MySQL base directory.

For information about changing the character set for error messages (rather than the language), see [Section 2.6, “Error Message Character Set”](#).

You can change the content of the error messages produced by the server using the instructions in the MySQL Internals manual, available at [MySQL Internals: Error Messages](#). If you do change the content of error messages, remember to repeat your changes after each upgrade to a newer version of MySQL.

Chapter 5 Adding a Character Set

Table of Contents

5.1 Character Definition Arrays	65
5.2 String Collating Support for Complex Character Sets	66
5.3 Multi-Byte Character Support for Complex Character Sets	66

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 5.1, “Character Definition Arrays”](#).
- 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 5.1, “Character Definition Arrays”](#).
 - 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 5.2, “String Collating Support for Complex Character Sets”](#).
 - If the character set requires multibyte character support, see [Section 5.3, “Multi-Byte Character Support for Complex Character Sets”](#).
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 `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, or `CMake` will not recognize `-DDEFAULT_CHARSET=MYSET`.
4. Reconfigure, recompile, and test.

5.1 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 5.2, “String Collating Support for Complex Character Sets”](#).

5.2 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.

5.3 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.

Chapter 6 Adding a Collation to a Character Set

Table of Contents

6.1 Collation Implementation Types	68
6.2 Choosing a Collation ID	71
6.3 Adding a Simple Collation to an 8-Bit Character Set	72
6.4 Adding a UCA Collation to a Unicode Character Set	73
6.4.1 Defining a UCA Collation Using LDML Syntax	73
6.4.2 LDML Syntax Supported in MySQL	75
6.4.3 Diagnostics During Index.xml Parsing	79

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 6.1, “Collation Implementation Types”](#). 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 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.

Summary of the procedure for adding a new 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 collation is present.

The instructions here cover only 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 [Chapter 5, Adding a Character Set](#). 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 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. For further information, see [Checking Whether Tables or Indexes Must Be Rebuilt](#).

Additional Resources

- 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/>

6.1 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 6.3, “Adding a Simple Collation to an 8-Bit Character Set”](#).

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 [Chapter 5, Adding a Character Set](#).

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
mysql> SELECT c1, HEX(c1), HEX(WEIGHT_STRING(c1)) FROM t1;
+-----+-----+-----+
| c1    | HEX(c1) | HEX(WEIGHT_STRING(c1)) |
+-----+-----+-----+
| a     | 61      | 41                     |
| A     | 41      | 41                     |
| ぢ     | 82C0    | 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 '臚' 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
mysql> SELECT c1, HEX(c1), HEX(WEIGHT_STRING(c1)) FROM t1;
+-----+-----+-----+
| c1    | HEX(c1) | HEX(WEIGHT_STRING(c1)) |
+-----+-----+-----+
| a     | 61      | 41                     |
| A     | 41      | 41                     |
| 臚     | 81B0    | C286                   |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

For implementation instructions, see [Chapter 5, Adding a Character Set](#).

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. `utf8_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 'utf8' COLLATE 'utf8_general_ci';
Query OK, 0 rows affected (0.00 sec)
mysql> CREATE TABLE t1
      (c1 CHAR(1) CHARACTER SET UTF8 COLLATE utf8_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
mysql> SELECT c1, HEX(c1), HEX(WEIGHT_STRING(c1)) FROM t1;
+-----+-----+-----+
| c1   | HEX(c1) | HEX(WEIGHT_STRING(c1)) |
+-----+-----+-----+
| a    | 61      | 0041                   |
| A    | 41      | 0041                   |
| À    | C380    | 0041                   |
| á    | C3A1    | 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 'utf8' COLLATE 'utf8_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 'utf8' COLLATE 'utf8_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 'utf8' COLLATE 'utf8_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 [Chapter 5, Adding a Character Set](#). For a UCA collation, see [Section 6.4, “Adding a UCA Collation to a Unicode Character Set”](#).

Miscellaneous collations

There are also a few collations that do not fall into any of the previous categories.

6.2 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 will appear 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()` C API function.

To determine the largest currently used ID, issue the following statement:

```
mysql> SELECT MAX(ID) FROM INFORMATION_SCHEMA.COLLATIONS;
+-----+
| MAX(ID) |
+-----+
|      210 |
+-----+
```

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 |
| ... |
|210 |
+-----+
```

Warning

Before MySQL 5.5, which provides for a range of user-defined collation IDs, you must choose an ID in the range from 1 to 254. In this case, if you upgrade MySQL, you may find that the collation ID you choose has been assigned to a collation included in the new MySQL distribution. In this case, you will need to choose a new value for your own collation.

In addition, before upgrading, you should save the configuration files that you change. If you upgrade in place, the process will replace the your modified files.

6.3 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 6.2, “Choosing a Collation ID”](#). 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/mysqlCharsets/ |
+-----+-----+
```

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>
```

- 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       |
+-----+-----+-----+-----+-----+-----+
```

6.4 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 6.1 MySQL Character Sets Available for User-Defined UCA Collations

Character Set	Base Collation
<code>utf8</code>	<code>utf8_unicode_ci</code>
<code>ucs2</code>	<code>ucs2_unicode_ci</code>
<code>utf16</code>	<code>utf16_unicode_ci</code>
<code>utf32</code>	<code>utf32_unicode_ci</code>

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.

6.4.1 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 6.4.2, “LDML Syntax Supported in MySQL”](#).

The example adds a collation named `utf8_phone_ci` to the `utf8` 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.

- Choose a collation ID, as shown in [Section 6.2, “Choosing a Collation ID”](#). The following steps use an ID of 1029.

- 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/ |
+-----+-----+
```

- 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="utf8">
...
  <collation name="utf8_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>
```

- 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.
- Restart the server and use this statement to verify that the collation is present:

```
mysql> SHOW COLLATION WHERE Collation = 'utf8_phone_ci';
+-----+-----+-----+-----+-----+-----+
| Collation | Charset | Id   | Default | Compiled | Sortlen |
+-----+-----+-----+-----+-----+-----+
| utf8_phone_ci | utf8   | 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 utf8 COLLATE utf8_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;
+-----+
| name | phone                |
+-----+
| Sanja | +380 (912) 8008005 |
| Bar   | +7-912-800-80-01  |
| Svoj  | +7 912 800 80 02   |
| Ramil | (7912) 800 80 03    |
| Hf    | +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)
```

6.4.2 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 6.4.3, "Diagnostics During Index.xml Parsing"](#).

Character Representation

Characters named in LDML rules can be written literally or in `\unnnn` 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 6.2 Abbreviated Shift Syntax

Abbreviated Syntax	Nonabbreviated Syntax
<code><pc>xyz</pc></code>	<code><p>x</p><p>y</p><p>z</p></code>
<code><sc>xyz</sc></code>	<code><s>x</s><s>y</s><s>z</s></code>
<code><tc>xyz</tc></code>	<code><t>x</t><t>y</t><t>z</t></code>
<code><ic>xyz</ic></code>	<code><i>x</i><i>y</i><i>z</i></code>

- 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 6.3 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_ignorable/>	U+0000	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 0E29+1
'b' has weight 0E29+2
'c' has weight 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="utf8_xxx_ci" version="5.2.0">
...
</collation>
```

6.4.3 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 _utf8'test' COLLATE utf8_test_ci;
ERROR 1273 (HY000): Unknown collation: 'utf8_test_ci'
mysql> SHOW WARNINGS;
```

Level	Code	Message
Error	1273	Unknown collation: 'utf8_test_ci'
Warning	1273	Expansion is too long at 'abcdefghi=x'

Chapter 7 MySQL Server Time Zone Support

Table of Contents

7.1 Staying Current with Time Zone Changes	83
7.2 Time Zone Leap Second Support	84

MySQL Server maintains several time zone settings:

- The system time zone. When the server starts, it attempts to determine the time zone of the host machine and uses it to set the `system_time_zone` system variable. The value does not change thereafter.

You can set the system time zone for MySQL Server at startup with the `--timezone=timezone_name` option to `mysqld_safe`. You can also set it by setting the `TZ` environment variable before you start `mysqld`. The permissible values for `--timezone` or `TZ` are system dependent. Consult your operating system documentation to see what values are acceptable.

- The server's current time zone. The global `time_zone` system variable indicates the time zone the server currently is operating in. The initial value for `time_zone` is 'SYSTEM', which indicates that the server time zone is the same as the system time zone.

The initial global server time zone value can be specified explicitly at startup with the `--default-time-zone=timezone` option on the command line, or you can use the following line in an option file:

```
default-time-zone='timezone'
```

If you have the `SUPER` privilege, you can set the global server time zone value at runtime with this statement:

```
mysql> SET GLOBAL time_zone = timezone;
```

- Per-connection time zones. Each client that connects has its own time zone setting, given by the session `time_zone` variable. Initially, the session variable takes its value from the global `time_zone` variable, but the client can change its own time zone with this statement:

```
mysql> SET time_zone = timezone;
```

The current session time zone setting affects display and storage of time values that are zone-sensitive. This includes the values displayed by functions such as `NOW()` or `CURTIME()`, and values stored in and retrieved from `TIMESTAMP` columns. Values for `TIMESTAMP` columns are converted from the current time zone to UTC for storage, and from UTC to the current time zone for retrieval.

The current time zone setting does not affect values displayed by functions such as `UTC_TIMESTAMP()` or values in `DATE`, `TIME`, or `DATETIME` columns. Nor are values in those data types stored in UTC; the time zone applies for them only when converting from `TIMESTAMP` values. If you want locale-specific arithmetic for `DATE`, `TIME`, or `DATETIME` values, convert them to UTC, perform the arithmetic, and then convert back.

The current values of the global and client-specific time zones can be retrieved like this:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
```

timezone values can be given in several formats, none of which are case sensitive:

- The value 'SYSTEM' indicates that the time zone should be the same as the system time zone.
- The value can be given as a string indicating an offset from UTC, such as '+10:00' or '-6:00'.
- The value can be given as a named time zone, such as 'Europe/Helsinki', 'US/Eastern', or 'MET'. Named time zones can be used only if the time zone information tables in the `mysql` database have been created and populated.

Populating the Time Zone Tables

Several tables in the `mysql` system database exist to maintain time zone information (see [The mysql System Database](#)). The MySQL installation procedure creates the time zone tables, but does not load them. You must do so manually using the following instructions.

Note

Loading the time zone information is not necessarily a one-time operation because the information changes occasionally. When such changes occur, applications that use the old rules become out of date and you may find it necessary to reload the time zone tables to keep the information used by your MySQL server current. See the notes at the end of this section.

If your system has its own *zoneinfo* database (the set of files describing time zones), you should use the `mysql_tzinfo_to_sql` program for filling the time zone tables. Examples of such systems are Linux, FreeBSD, Solaris, and OS X. One likely location for these files is the `/usr/share/zoneinfo` directory. If your system does not have a *zoneinfo* database, you can use the downloadable package described later in this section.

The `mysql_tzinfo_to_sql` program is used to load the time zone tables. On the command line, pass the *zoneinfo* directory path name to `mysql_tzinfo_to_sql` and send the output into the `mysql` program. For example:

```
shell> mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -u root mysql
```

`mysql_tzinfo_to_sql` reads your system's time zone files and generates SQL statements from them. `mysql` processes those statements to load the time zone tables.

`mysql_tzinfo_to_sql` also can be used to load a single time zone file or to generate leap second information:

- To load a single time zone file *tz_file* that corresponds to a time zone name *tz_name*, invoke `mysql_tzinfo_to_sql` like this:

```
shell> mysql_tzinfo_to_sql tz_file tz_name | mysql -u root mysql
```

With this approach, you must execute a separate command to load the time zone file for each named zone that the server needs to know about.

- If your time zone needs to account for leap seconds, initialize the leap second information like this, where *tz_file* is the name of your time zone file:

```
shell> mysql_tzinfo_to_sql --leap tz_file | mysql -u root mysql
```

- After running `mysql_tzinfo_to_sql`, it is best to restart the server so that it does not continue to use any previously cached time zone data.

If your system is one that has no zoneinfo database (for example, Windows), you can use a package that is available for download at the MySQL Developer Zone:

```
http://dev.mysql.com/downloads/timezones.html
```

Download a time zone package that contains SQL statements and unpack it, then load the package file contents into the time zone tables:

```
shell> mysql -u root mysql < file_name
```

Then restart the server.

Warning

Do *not* use a downloadable package that contains **MyISAM** tables. MySQL 5.7.5 and up uses **InnoDB** for the time zone tables. Trying to replace them with **MyISAM** tables will cause problems.

Warning

Do not use a downloadable package if your system has a zoneinfo database. Use the `mysql_tzinfo_to_sql` utility instead. Otherwise, you may cause a difference in datetime handling between MySQL and other applications on your system.

For information about time zone settings in replication setup, please see [Replication Features and Issues](#).

7.1 Staying Current with Time Zone Changes

When time zone rules change, applications that use the old rules become out of date. To stay current, it is necessary to make sure that your system uses current time zone information is used. For MySQL, there are two factors to consider in staying current:

- The operating system time affects the value that the MySQL server uses for times if its time zone is set to **SYSTEM**. Make sure that your operating system is using the latest time zone information. For most operating systems, the latest update or service pack prepares your system for the time changes. Check the Web site for your operating system vendor for an update that addresses the time changes.
- If you replace the system's `/etc/localtime` timezone file with a version that uses rules differing from those in effect at `mysqld` startup, you should restart `mysqld` so that it uses the updated rules. Otherwise, `mysqld` might not notice when the system changes its time.
- If you use named time zones with MySQL, make sure that the time zone tables in the `mysql` database are up to date. If your system has its own zoneinfo database, you should reload the MySQL time zone tables whenever the zoneinfo database is updated. For systems that do not have their own zoneinfo database, check the MySQL Developer Zone for updates. When a new update is available, download it and use it to replace the content of your current time zone tables. For instructions for both methods, see [Populating the Time Zone Tables](#). `mysqld` caches time zone information that it looks up, so after updating the time zone tables, you should restart `mysqld` to make sure that it does not continue to serve outdated time zone data.

If you are uncertain whether named time zones are available, for use either as the server's time zone setting or by clients that set their own time zone, check whether your time zone tables are empty. The following query determines whether the table that contains time zone names has any rows:

```
mysql> SELECT COUNT(*) FROM mysql.time_zone_name;
+-----+
| COUNT(*) |
+-----+
|          0 |
+-----+
```

A count of zero indicates that the table is empty. In this case, no one can be using named time zones, and you don't need to update the tables. A count greater than zero indicates that the table is not empty and that its contents are available to be used for named time zone support. In this case, you should be sure to reload your time zone tables so that anyone who uses named time zones will get correct query results.

To check whether your MySQL installation is updated properly for a change in Daylight Saving Time rules, use a test like the one following. The example uses values that are appropriate for the 2007 DST 1-hour change that occurs in the United States on March 11 at 2 a.m.

The test uses these two queries:

```
SELECT CONVERT_TZ('2007-03-11 2:00:00', 'US/Eastern', 'US/Central');
SELECT CONVERT_TZ('2007-03-11 3:00:00', 'US/Eastern', 'US/Central');
```

The two time values indicate the times at which the DST change occurs, and the use of named time zones requires that the time zone tables be used. The desired result is that both queries return the same result (the input time, converted to the equivalent value in the 'US/Central' time zone).

Before updating the time zone tables, you would see an incorrect result like this:

```
mysql> SELECT CONVERT_TZ('2007-03-11 2:00:00', 'US/Eastern', 'US/Central');
+-----+
| CONVERT_TZ('2007-03-11 2:00:00', 'US/Eastern', 'US/Central') |
+-----+
| 2007-03-11 01:00:00 |
+-----+
mysql> SELECT CONVERT_TZ('2007-03-11 3:00:00', 'US/Eastern', 'US/Central');
+-----+
| CONVERT_TZ('2007-03-11 3:00:00', 'US/Eastern', 'US/Central') |
+-----+
| 2007-03-11 02:00:00 |
+-----+
```

After updating the tables, you should see the correct result:

```
mysql> SELECT CONVERT_TZ('2007-03-11 2:00:00', 'US/Eastern', 'US/Central');
+-----+
| CONVERT_TZ('2007-03-11 2:00:00', 'US/Eastern', 'US/Central') |
+-----+
| 2007-03-11 01:00:00 |
+-----+
mysql> SELECT CONVERT_TZ('2007-03-11 3:00:00', 'US/Eastern', 'US/Central');
+-----+
| CONVERT_TZ('2007-03-11 3:00:00', 'US/Eastern', 'US/Central') |
+-----+
| 2007-03-11 01:00:00 |
+-----+
```

7.2 Time Zone Leap Second Support

Leap second values are returned with a time part that ends with `:59:59`. This means that a function such as `NOW()` can return the same value for two or three consecutive seconds during the leap second.

It remains true that literal temporal values having a time part that ends with `:59:60` or `:59:61` are considered invalid.

If it is necessary to search for `TIMESTAMP` values one second before the leap second, anomalous results may be obtained if you use a comparison with `'YYYY-MM-DD hh:mm:ss'` values. The following example demonstrates this. It changes the local time zone to UTC so there is no difference between internal values (which are in UTC) and displayed values (which have time zone correction applied).

```
mysql> CREATE TABLE t1 (
  a INT,
  ts TIMESTAMP DEFAULT NOW(),
  PRIMARY KEY (ts)
);
Query OK, 0 rows affected (0.01 sec)
mysql> -- change to UTC
mysql> SET time_zone = '+00:00';
Query OK, 0 rows affected (0.00 sec)
mysql> -- Simulate NOW() = '2008-12-31 23:59:59'
mysql> SET timestamp = 1230767999;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO t1 (a) VALUES (1);
Query OK, 1 row affected (0.00 sec)
mysql> -- Simulate NOW() = '2008-12-31 23:59:60'
mysql> SET timestamp = 1230768000;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO t1 (a) VALUES (2);
Query OK, 1 row affected (0.00 sec)
mysql> -- values differ internally but display the same
mysql> SELECT a, ts, UNIX_TIMESTAMP(ts) FROM t1;
+-----+-----+-----+
| a    | ts                | UNIX_TIMESTAMP(ts) |
+-----+-----+-----+
| 1    | 2008-12-31 23:59:59 | 1230767999         |
| 2    | 2008-12-31 23:59:59 | 1230768000         |
+-----+-----+-----+
2 rows in set (0.00 sec)
mysql> -- only the non-leap value matches
mysql> SELECT * FROM t1 WHERE ts = '2008-12-31 23:59:59';
+-----+-----+
| a    | ts                |
+-----+-----+
| 1    | 2008-12-31 23:59:59 |
+-----+-----+
1 row in set (0.00 sec)
mysql> -- the leap value with seconds=60 is invalid
mysql> SELECT * FROM t1 WHERE ts = '2008-12-31 23:59:60';
Empty set, 2 warnings (0.00 sec)
```

To work around this, you can use a comparison based on the UTC value actually stored in column, which has the leap second correction applied:

```
mysql> -- selecting using UNIX_TIMESTAMP value return leap value
mysql> SELECT * FROM t1 WHERE UNIX_TIMESTAMP(ts) = 1230768000;
+-----+-----+
| a    | ts                |
+-----+-----+
| 2    | 2008-12-31 23:59:59 |
+-----+-----+
1 row in set (0.00 sec)
```

Chapter 8 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/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 if you have the `SUPER` privilege. 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 'utf8';
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('2010-01-01'), MONTHNAME('2010-01-01');
+-----+-----+
| DAYNAME('2010-01-01') | MONTHNAME('2010-01-01') |
+-----+-----+
| Friday                | January                  |
+-----+-----+
1 row in set (0.00 sec)
mysql> SELECT DATE_FORMAT('2010-01-01', '%W %a %M %b');
+-----+
| DATE_FORMAT('2010-01-01', '%W %a %M %b') |
+-----+
| Friday Fri 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('2010-01-01'), MONTHNAME('2010-01-01');
+-----+-----+
| DAYNAME('2010-01-01') | MONTHNAME('2010-01-01') |
+-----+-----+
| viernes                | enero                    |
+-----+-----+
1 row in set (0.00 sec)
mysql> SELECT DATE_FORMAT('2010-01-01', '%W %a %M %b');
+-----+
| DATE_FORMAT('2010-01-01', '%W %a %M %b') |
+-----+
```

```
| viernes vie enero ene |
+-----+
1 row in set (0.00 sec)
```

The day or month name for each of the affected functions is converted from `utf8` 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.

<code>ar_AE</code> : Arabic - United Arab Emirates	<code>ar_BH</code> : Arabic - Bahrain
<code>ar_DZ</code> : Arabic - Algeria	<code>ar_EG</code> : Arabic - Egypt
<code>ar_IN</code> : Arabic - India	<code>ar_IQ</code> : Arabic - Iraq
<code>ar_JO</code> : Arabic - Jordan	<code>ar_KW</code> : Arabic - Kuwait
<code>ar_LB</code> : Arabic - Lebanon	<code>ar_LY</code> : Arabic - Libya
<code>ar_MA</code> : Arabic - Morocco	<code>ar_OM</code> : Arabic - Oman
<code>ar_QA</code> : Arabic - Qatar	<code>ar_SA</code> : Arabic - Saudi Arabia
<code>ar_SD</code> : Arabic - Sudan	<code>ar_SY</code> : Arabic - Syria
<code>ar_TN</code> : Arabic - Tunisia	<code>ar_YE</code> : Arabic - Yemen
<code>be_BY</code> : Belarusian - Belarus	<code>bg_BG</code> : Bulgarian - Bulgaria
<code>ca_ES</code> : Catalan - Spain	<code>cs_CZ</code> : Czech - Czech Republic
<code>da_DK</code> : Danish - Denmark	<code>de_AT</code> : German - Austria
<code>de_BE</code> : German - Belgium	<code>de_CH</code> : German - Switzerland
<code>de_DE</code> : German - Germany	<code>de_LU</code> : German - Luxembourg
<code>el_GR</code> : Greek - Greece	<code>en_AU</code> : English - Australia
<code>en_CA</code> : English - Canada	<code>en_GB</code> : English - United Kingdom
<code>en_IN</code> : English - India	<code>en_NZ</code> : English - New Zealand
<code>en_PH</code> : English - Philippines	<code>en_US</code> : English - United States
<code>en_ZA</code> : English - South Africa	<code>en_ZW</code> : English - Zimbabwe
<code>es_AR</code> : Spanish - Argentina	<code>es_BO</code> : Spanish - Bolivia
<code>es_CL</code> : Spanish - Chile	<code>es_CO</code> : Spanish - Columbia
<code>es_CR</code> : Spanish - Costa Rica	<code>es_DO</code> : Spanish - Dominican Republic
<code>es_EC</code> : Spanish - Ecuador	<code>es_ES</code> : Spanish - Spain
<code>es_GT</code> : Spanish - Guatemala	<code>es_HN</code> : Spanish - Honduras
<code>es_MX</code> : Spanish - Mexico	<code>es_NI</code> : Spanish - Nicaragua
<code>es_PA</code> : Spanish - Panama	<code>es_PE</code> : Spanish - Peru
<code>es_PR</code> : Spanish - Puerto Rico	<code>es_PY</code> : Spanish - Paraguay
<code>es_SV</code> : Spanish - El Salvador	<code>es_US</code> : Spanish - United States
<code>es_UY</code> : Spanish - Uruguay	<code>es_VE</code> : Spanish - Venezuela
<code>et_EE</code> : Estonian - Estonia	<code>eu_ES</code> : Basque - Basque
<code>fi_FI</code> : Finnish - Finland	<code>fo_FO</code> : Faroese - Faroe Islands
<code>fr_BE</code> : French - Belgium	<code>fr_CA</code> : French - Canada
<code>fr_CH</code> : French - Switzerland	<code>fr_FR</code> : 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 - FYROM
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	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 - Yugoslavia	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 - Viet Nam
zh_CN : Chinese - China	zh_HK : Chinese - Hong Kong
zh_TW : Chinese - Taiwan Province of China	

