14.7 Date and Time Functions

This section describes the functions that can be used to manipulate temporal values. See Section 13.2, “Date and Time Data Types”, for a description of the range of values each date and time type has and the valid formats in which values may be specified.

Table 14.11 Date and Time Functions

Name	Description
`ADDDATE()`	Add time values (intervals) to a date value
`ADDTIME()`	Add time
`CONVERT_TZ()`	Convert from one time zone to another
`CURDATE()`	Return the current date
`CURRENT_DATE()`, `CURRENT_DATE`	Synonyms for CURDATE()
`CURRENT_TIME()`, `CURRENT_TIME`	Synonyms for CURTIME()
`CURRENT_TIMESTAMP()`, `CURRENT_TIMESTAMP`	Synonyms for NOW()
`CURTIME()`	Return the current time
`DATE()`	Extract the date part of a date or datetime expression
`DATE_ADD()`	Add time values (intervals) to a date value
`DATE_FORMAT()`	Format date as specified
`DATE_SUB()`	Subtract a time value (interval) from a date
`DATEDIFF()`	Subtract two dates
`DAY()`	Synonym for DAYOFMONTH()
`DAYNAME()`	Return the name of the weekday
`DAYOFMONTH()`	Return the day of the month (0-31)
`DAYOFWEEK()`	Return the weekday index of the argument
`DAYOFYEAR()`	Return the day of the year (1-366)
`EXTRACT()`	Extract part of a date
`FROM_DAYS()`	Convert a day number to a date
`FROM_UNIXTIME()`	Format Unix timestamp as a date
`GET_FORMAT()`	Return a date format string

Name	Description
HOUR()	Extract the hour
LAST_DAY	Return the last day of the month for the argument
LOCALTIME(), LOCALTIME	Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonym for NOW()
MAKEDATE()	Create a date from the year and day of year
MAKETIME()	Create time from hour, minute, second
MICROSECOND()	Return the microseconds from argument
MINUTE()	Return the minute from the argument
MONTH()	Return the month from the date passed
MONTHNAME()	Return the name of the month
NOW()	Return the current date and time
PERIOD_ADD()	Add a period to a year-month
PERIOD_DIFF()	Return the number of months between periods
QUARTER()	Return the quarter from a date argument
SEC_TO_TIME()	Converts seconds to 'hh:mm:ss' format
SECOND()	Return the second (0-59)
STR_TO_DATE()	Convert a string to a date
SUBDATE()	Synonym for DATE_SUB() when invoked with three arguments
SUBTIME()	Subtract times
SYSDATE()	Return the time at which the function executes
TIME()	Extract the time portion of the expression passed
TIME_FORMAT()	Format as time
TIME_TO_SEC()	Return the argument converted to seconds
TIMEDIFF()	Subtract time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments
TIMESTAMPADD()	Add an interval to a datetime expression
TIMESTAMPDIFF()	Return the difference of two datetime expressions, using the units specified
TO_DAYS()	Return the date argument converted to days
TO_SECONDS()	Return the date or datetime argument converted to seconds since Year 0
UNIX_TIMESTAMP()	Return a Unix timestamp
UTC_DATE()	Return the current UTC date

Name	Description
`UTC_TIME()`	Return the current UTC time
`UTC_TIMESTAMP()`	Return the current UTC date and time
`WEEK()`	Return the week number
`WEEKDAY()`	Return the weekday index
`WEEKOFYEAR()`	Return the calendar week of the date (1-53)
`YEAR()`	Return the year
`YEARWEEK()`	Return the year and week

Here is an example that uses date functions. The following query selects all rows with a ***`date_col`*** value from within the last 30 days:

```
mysql> SELECT something FROM tbl_name
    -> WHERE DATE_SUB(CURDATE(),INTERVAL 30 DAY) <= date_col;
```

The query also selects rows with dates that lie in the future.

Functions that expect date values usually accept datetime values and ignore the time part. Functions that expect time values usually accept datetime values and ignore the date part.

Functions that return the current date or time each are evaluated only once per query at the start of query execution. This means that multiple references to a function such as `NOW()` within a single query always produce the same result. (For our purposes, a single query also includes a call to a stored program (stored routine, trigger, or event) and all subprograms called by that program.) This principle also applies to `CURDATE()`, `CURTIME()`, `UTC_DATE()`, `UTC_TIME()`, `UTC_TIMESTAMP()`, and to any of their synonyms.

The `CURRENT_TIMESTAMP()`, `CURRENT_TIME()`, `CURRENT_DATE()`, and `FROM_UNIXTIME()` functions return values in the current session time zone, which is available as the session value of the `time_zone` system variable. In addition, `UNIX_TIMESTAMP()` assumes that its argument is a datetime value in the session time zone. See Section 7.1.15, “MySQL Server Time Zone Support”.

Some date functions can be used with “zero” dates or incomplete dates such as `'2001-11-00'`, whereas others cannot. Functions that extract parts of dates typically work with incomplete dates and thus can return 0 when you might otherwise expect a nonzero value. For example:

```
mysql> SELECT DAYOFMONTH('2001-11-00'), MONTH('2005-00-00');
        -> 0, 0
```

Other functions expect complete dates and return `NULL` for incomplete dates. These include functions that perform date arithmetic or that map parts of dates to names. For example:

```
mysql> SELECT DATE_ADD('2006-05-00',INTERVAL 1 DAY);
        -> NULL
mysql> SELECT DAYNAME('2006-05-00');
        -> NULL
```

Several functions are strict when passed a `DATE()` function value as their argument and reject incomplete dates with a day part of zero: `CONVERT_TZ()`, `DATE_ADD()`, `DATE_SUB()`, `DAYOFYEAR()`, `TIMESTAMPDIFF()`, `TO_DAYS()`, `TO_SECONDS()`, `WEEK()`, `WEEKDAY()`, `WEEKOFYEAR()`, `YEARWEEK()`.

Fractional seconds for `TIME`, `DATETIME`, and `TIMESTAMP` values are supported, with up to microsecond precision. Functions that take temporal arguments accept values with fractional seconds. Return values from temporal functions include fractional seconds as appropriate.

- `ADDDATE(`***`date`***`,INTERVAL `***`expr`*** ***`unit`***`)`, `ADDDATE(`***`date`***`,`***`days`***`)`

  When invoked with the `INTERVAL` form of the second argument, `ADDDATE()` is a synonym for `DATE_ADD()`. The related function `SUBDATE()` is a synonym for `DATE_SUB()`. For information on the `INTERVAL` ***`unit`*** argument, see Temporal Intervals.

  ```
  mysql> SELECT DATE_ADD('2008-01-02', INTERVAL 31 DAY);
          -> '2008-02-02'
  mysql> SELECT ADDDATE('2008-01-02', INTERVAL 31 DAY);
          -> '2008-02-02'
  ```

  When invoked with the ***`days`*** form of the second argument, MySQL treats it as an integer number of days to be added to ***`expr`***.

  ```
  mysql> SELECT ADDDATE('2008-01-02', 31);
          -> '2008-02-02'
  ```

  This function returns `NULL` if ***`date`*** or ***`days`*** is `NULL`.

- `ADDTIME(`***`expr1`***`,`***`expr2`***`)`

  `ADDTIME()` adds ***`expr2`*** to ***`expr1`*** and returns the result. ***`expr1`*** is a time or datetime expression, and ***`expr2`*** is a time expression. Returns `NULL` if ***`expr1`***or ***`expr2`*** is `NULL`.

  Beginning with MySQL 8.0.28, the return type of this function and of the `SUBTIME()` function is determined as follows:

- If the first argument is a dynamic parameter (such as in a prepared statement), the return type is `TIME`.
- Otherwise, the resolved type of the function is derived from the resolved type of the first argument.

```
mysql> SELECT ADDTIME('2007-12-31 23:59:59.999999', '1 1:1:1.000002');
        -> '2008-01-02 01:01:01.000001'
mysql> SELECT ADDTIME('01:00:00.999999', '02:00:00.999998');
        -> '03:00:01.999997'
```

- `CONVERT_TZ(`***`dt`***`,`***`from_tz`***`,`***`to_tz`***`)`

  `CONVERT_TZ()` converts a datetime value ***`dt`*** from the time zone given by ***`from_tz`*** to the time zone given by ***`to_tz`*** and returns the resulting value. Time zones are specified as described in Section 7.1.15, “MySQL Server Time Zone Support”. This function returns `NULL` if any of the arguments are invalid, or if any of them are `NULL`.

  On 32-bit platforms, the supported range of values for this function is the same as for the `TIMESTAMP` type (see Section 13.2.1, “Date and Time Data Type Syntax”, for range information). On 64-bit platforms, beginning with MySQL 8.0.28, the maximum supported value is `'3001-01-18 23:59:59.999999'` UTC.

  Regardless of platform or MySQL version, if the value falls out of the supported range when converted from ***`from_tz`*** to UTC, no conversion occurs.

```
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET');
        -> '2004-01-01 13:00:00'
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','+00:00','+10:00');
        -> '2004-01-01 22:00:00'
```

> **Note**
>
> To use named time zones such as `'MET'` or `'Europe/Amsterdam'`, the time zone tables must be properly set up. For instructions, see Section 7.1.15, “MySQL Server Time Zone Support”.

- `CURDATE()`

  Returns the current date as a value in `'`***`YYYY-MM-DD`***`'` or ***`YYYYMMDD`*** format, depending on whether the function is used in string or numeric context.

```
mysql> SELECT CURDATE();
        -> '2008-06-13'
mysql> SELECT CURDATE() + 0;
        -> 20080613
```

- `CURRENT_DATE`, `CURRENT_DATE()`

  `CURRENT_DATE` and `CURRENT_DATE()` are synonyms for `CURDATE()`.

- `CURRENT_TIME`, `CURRENT_TIME([`***`fsp`***`])`

  `CURRENT_TIME` and `CURRENT_TIME()` are synonyms for `CURTIME()`.

- `CURRENT_TIMESTAMP`, `CURRENT_TIMESTAMP([`***`fsp`***`])`

  `CURRENT_TIMESTAMP` and `CURRENT_TIMESTAMP()` are synonyms for `NOW()`.

- `CURTIME([`***`fsp`***`])`

  Returns the current time as a value in ***`'hh:mm:ss'`*** or ***`hhmmss`*** format, depending on whether the function is used in string or numeric context. The value is expressed in the session time zone.

  If the ***`fsp`*** argument is given to specify a fractional seconds precision from 0 to 6, the return value includes a fractional seconds part of that many digits.

```
mysql> SELECT CURTIME();
+-----------+
| CURTIME() |
+-----------+
| 19:25:37  |
+-----------+

mysql> SELECT CURTIME() + 0;
+---------------+
| CURTIME() + 0 |
+---------------+
|        192537 |
+---------------+

mysql> SELECT CURTIME(3);
+--------------+
| CURTIME(3)   |
+--------------+
| 19:25:37.840 |
+--------------+
```

- `DATE(`***`expr`***`)`

Extracts the date part of the date or datetime expression ***expr***. Returns NULL if ***expr*** is NULL.

```
mysql> SELECT DATE('2003-12-31 01:02:03');
        -> '2003-12-31'
```

- DATEDIFF(***expr1***,***expr2***)

  DATEDIFF() returns ***expr1*** − ***expr2*** expressed as a value in days from one date to the other. ***expr1*** and ***expr2*** are date or date-and-time expressions. Only the date parts of the values are used in the calculation.

```
mysql> SELECT DATEDIFF('2007-12-31 23:59:59','2007-12-30');
        -> 1
mysql> SELECT DATEDIFF('2010-11-30 23:59:59','2010-12-31');
        -> -31
```

  This function returns NULL if ***expr1*** or ***expr2*** is NULL.

- DATE_ADD(***date***,INTERVAL ***expr*** ***unit***), DATE_SUB(***date***,INTERVAL ***expr*** ***unit***)

  These functions perform date arithmetic. The ***date*** argument specifies the starting date or datetime value. ***expr*** is an expression specifying the interval value to be added or subtracted from the starting date. ***expr*** is evaluated as a string; it may start with a - for negative intervals. ***unit*** is a keyword indicating the units in which the expression should be interpreted.

  For more information about temporal interval syntax, including a full list of ***unit*** specifiers, the expected form of the ***expr*** argument for each ***unit*** value, and rules for operand interpretation in temporal arithmetic, see Temporal Intervals.

  The return value depends on the arguments:

  - If ***date*** is NULL, the function returns NULL.
  - DATE if the ***date*** argument is a DATE value and your calculations involve only YEAR, MONTH, and DAY parts (that is, no time parts).
  - (*MySQL 8.0.28 and later*:) TIME if the ***date*** argument is a TIME value and the calculations involve only HOURS, MINUTES, and SECONDS parts (that is, no date parts).
  - DATETIME if the first argument is a DATETIME (or TIMESTAMP) value, or if the first argument is a DATE and the ***unit*** value uses HOURS, MINUTES, or SECONDS, or if the first argument is of type TIME and the ***unit*** value uses YEAR, MONTH, or DAY.

- (*MySQL 8.0.28 and later*:) If the first argument is a dynamic parameter (for example, of a prepared statement), its resolved type is `DATE` if the second argument is an interval that contains some combination of `YEAR`, `MONTH`, or `DAY` values only; otherwise, its type is `DATETIME`.

- String otherwise (type `VARCHAR`).

> **Note**
>
> In MySQL 8.0.22 through 8.0.27, when used in prepared statements, these functions returned `DATETIME` values regardless of argument types. (Bug #103781)

To ensure that the result is `DATETIME`, you can use `CAST()` to convert the first argument to `DATETIME`.

```
mysql> SELECT DATE_ADD('2018-05-01',INTERVAL 1 DAY);
        -> '2018-05-02'
mysql> SELECT DATE_SUB('2018-05-01',INTERVAL 1 YEAR);
        -> '2017-05-01'
mysql> SELECT DATE_ADD('2020-12-31 23:59:59',
    ->                 INTERVAL 1 SECOND);
        -> '2021-01-01 00:00:00'
mysql> SELECT DATE_ADD('2018-12-31 23:59:59',
    ->                 INTERVAL 1 DAY);
        -> '2019-01-01 23:59:59'
mysql> SELECT DATE_ADD('2100-12-31 23:59:59',
    ->                 INTERVAL '1:1' MINUTE_SECOND);
        -> '2101-01-01 00:01:00'
mysql> SELECT DATE_SUB('2025-01-01 00:00:00',
    ->                 INTERVAL '1 1:1:1' DAY_SECOND);
        -> '2024-12-30 22:58:59'
mysql> SELECT DATE_ADD('1900-01-01 00:00:00',
    ->                 INTERVAL '-1 10' DAY_HOUR);
        -> '1899-12-30 14:00:00'
mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
        -> '1997-12-02'
mysql> SELECT DATE_ADD('1992-12-31 23:59:59.000002',
    ->            INTERVAL '1.999999' SECOND_MICROSECOND);
        -> '1993-01-01 00:00:01.000001'
```

When adding a `MONTH` interval to a `DATE` or `DATETIME` value, and the resulting date includes a day that does not exist in the given month, the day is adjusted to the last day of the month, as shown here:

```
mysql> SELECT DATE_ADD('2024-03-30', INTERVAL 1 MONTH) AS d1,
     >        DATE_ADD('2024-03-31', INTERVAL 1 MONTH) AS d2;
+------------+------------+
| d1         | d2         |
+------------+------------+
| 2024-04-30 | 2024-04-30 |
+------------+------------+
1 row in set (0.00 sec)
```

- `DATE_FORMAT(`***`date`***`,`***`format`***`)`

  Formats the ***`date`*** value according to the ***`format`*** string. If either argument is `NULL`, the function returns `NULL`.

  The specifiers shown in the following table may be used in the ***`format`*** string. The `%` character is required before format specifier characters. The specifiers apply to other functions as well: `STR_TO_DATE()`, `TIME_FORMAT()`, `UNIX_TIMESTAMP()`.

Specifier	**Description**
`%a`	Abbreviated weekday name (`Sun`..`Sat`)
`%b`	Abbreviated month name (`Jan`..`Dec`)
`%c`	Month, numeric (`0`..`12`)
`%D`	Day of the month with English suffix (`0th`, `1st`, `2nd`, `3rd`, …)
`%d`	Day of the month, numeric (`00`..`31`)
`%e`	Day of the month, numeric (`0`..`31`)
`%f`	Microseconds (`000000`..`999999`)
`%H`	Hour (`00`..`23`)
`%h`	Hour (`01`..`12`)
`%I`	Hour (`01`..`12`)
`%i`	Minutes, numeric (`00`..`59`)
`%j`	Day of year (`001`..`366`)
`%k`	Hour (`0`..`23`)
`%l`	Hour (`1`..`12`)
`%M`	Month name (`January`..`December`)
`%m`	Month, numeric (`00`..`12`)
`%p`	`AM` or `PM`
`%r`	Time, 12-hour (***`hh:mm:ss`*** followed by `AM` or `PM`)
`%S`	Seconds (`00`..`59`)

Specifier	Description
`%s`	Seconds (`00`..`59`)
`%T`	Time, 24-hour (***hh:mm:ss***)
`%U`	Week (`00`..`53`), where Sunday is the first day of the week; `WEEK()` mode 0
`%u`	Week (`00`..`53`), where Monday is the first day of the week; `WEEK()` mode 1
`%V`	Week (`01`..`53`), where Sunday is the first day of the week; `WEEK()` mode 2; used with `%X`
`%v`	Week (`01`..`53`), where Monday is the first day of the week; `WEEK()` mode 3; used with `%x`
`%W`	Weekday name (`Sunday`..`Saturday`)
`%w`	Day of the week (`0`=Sunday..`6`=Saturday)
`%X`	Year for the week where Sunday is the first day of the week, numeric, four digits; used with `%V`
`%x`	Year for the week, where Monday is the first day of the week, numeric, four digits; used with `%v`
`%Y`	Year, numeric, four digits
`%y`	Year, numeric (two digits)
`%%`	A literal `%` character
`%`***x***	***x***, for any "***x***" not listed above

Ranges for the month and day specifiers begin with zero due to the fact that MySQL permits the storing of incomplete dates such as `'2014-00-00'`.

The language used for day and month names and abbreviations is controlled by the value of the `lc_time_names` system variable (Section 12.16, "MySQL Server Locale Support").

For the `%U`, `%u`, `%V`, and `%v` specifiers, see the description of the `WEEK()` function for information about the mode values. The mode affects how week numbering occurs.

`DATE_FORMAT()` returns a string with a character set and collation given by `character_set_connection` and `collation_connection` so that it can return month and weekday names containing non-ASCII characters.

```
mysql> SELECT DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y');
        -> 'Sunday October 2009'
mysql> SELECT DATE_FORMAT('2007-10-04 22:23:00', '%H:%i:%s');
        -> '22:23:00'
mysql> SELECT DATE_FORMAT('1900-10-04 22:23:00',
    ->                 '%D %y %a %d %m %b %j');
```

```
        -> '4th 00 Thu 04 10 Oct 277'
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00',
    ->                 '%H %k %I %r %T %S %w');
        -> '22 22 10 10:23:00 PM 22:23:00 00 6'
mysql> SELECT DATE_FORMAT('1999-01-01', '%X %V');
        -> '1998 52'
mysql> SELECT DATE_FORMAT('2006-06-00', '%d');
        -> '00'
```

- `DATE_SUB(`***`date`***`,INTERVAL `***`expr unit`***`)`

  See the description for `DATE_ADD()`.

- `DAY(`***`date`***`)`

  `DAY()` is a synonym for `DAYOFMONTH()`.

- `DAYNAME(`***`date`***`)`

  Returns the name of the weekday for ***`date`***. The language used for the name is controlled by the value of the `lc_time_names` system variable (see Section 12.16, “MySQL Server Locale Support”). Returns `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT DAYNAME('2007-02-03');
        -> 'Saturday'
```

- `DAYOFMONTH(`***`date`***`)`

  Returns the day of the month for ***`date`***, in the range `1` to `31`, or `0` for dates such as `'0000-00-00'` or `'2008-00-00'` that have a zero day part. Returns `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT DAYOFMONTH('2007-02-03');
        -> 3
```

- `DAYOFWEEK(`***`date`***`)`

  Returns the weekday index for ***`date`*** (`1` = Sunday, `2` = Monday, ..., `7` = Saturday). These index values correspond to the ODBC standard. Returns `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT DAYOFWEEK('2007-02-03');
        -> 7
```

- `DAYOFYEAR(`***`date`***`)`

Returns the day of the year for ***`date`***, in the range `1` to `366`. Returns `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT DAYOFYEAR('2007-02-03');
        -> 34
```

- `EXTRACT(`***`unit`***` FROM `***`date`***`)`

  The `EXTRACT()` function uses the same kinds of ***`unit`*** specifiers as `DATE_ADD()` or `DATE_SUB()`, but extracts parts from the date rather than performing date arithmetic. For information on the ***`unit`*** argument, see Temporal Intervals. Returns `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT EXTRACT(YEAR FROM '2019-07-02');
        -> 2019
mysql> SELECT EXTRACT(YEAR_MONTH FROM '2019-07-02 01:02:03');
        -> 201907
mysql> SELECT EXTRACT(DAY_MINUTE FROM '2019-07-02 01:02:03');
        -> 20102
mysql> SELECT EXTRACT(MICROSECOND
    ->                FROM '2003-01-02 10:30:00.000123');
        -> 123
```

- `FROM_DAYS(`***`N`***`)`

  Given a day number ***`N`***, returns a `DATE` value. Returns `NULL` if ***`N`*** is `NULL`.

```
mysql> SELECT FROM_DAYS(730669);
        -> '2000-07-03'
```

  Use `FROM_DAYS()` with caution on old dates. It is not intended for use with values that precede the advent of the Gregorian calendar (1582). See Section 13.2.7, “What Calendar Is Used By MySQL?”.

- `FROM_UNIXTIME(`***`unix_timestamp`***`[,`***`format`***`])`

  Returns a representation of ***`unix_timestamp`*** as a datetime or character string value. The value returned is expressed using the session time zone. (Clients can set the session time zone as described in Section 7.1.15, “MySQL Server Time Zone Support”.) ***`unix_timestamp`*** is an internal timestamp value representing seconds since `'1970-01-01 00:00:00'` UTC, such as produced by the `UNIX_TIMESTAMP()` function.

  If ***`format`*** is omitted, this function returns a `DATETIME` value.

  If ***`unix_timestamp`*** or ***`format`*** is `NULL`, this function returns `NULL`.

If ***unix_timestamp*** is an integer, the fractional seconds precision of the `DATETIME` is zero. When ***unix_timestamp*** is a decimal value, the fractional seconds precision of the `DATETIME` is the same as the precision of the decimal value, up to a maximum of 6. When ***unix_timestamp*** is a floating point number, the fractional seconds precision of the datetime is 6.

On 32-bit platforms, the maximum useful value for ***unix_timestamp*** is 2147483647.999999, which returns `'2038-01-19 03:14:07.999999'` UTC. On 64-bit platforms running MySQL 8.0.28 or later, the effective maximum is 32536771199.999999, which returns `'3001-01-18 23:59:59.999999'` UTC. Regardless of platform or version, a greater value for ***unix_timestamp*** than the effective maximum returns `0`.

format is used to format the result in the same way as the format string used for the `DATE_FORMAT()` function. If ***format*** is supplied, the value returned is a `VARCHAR`.

```
mysql> SELECT FROM_UNIXTIME(1447430881);
        -> '2015-11-13 10:08:01'
mysql> SELECT FROM_UNIXTIME(1447430881) + 0;
        -> 20151113100801
mysql> SELECT FROM_UNIXTIME(1447430881,
    ->                      '%Y %D %M %h:%i:%s %x');
        -> '2015 13th November 10:08:01 2015'
```

Note

If you use `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` to convert between values in a non-UTC time zone and Unix timestamp values, the conversion is lossy because the mapping is not one-to-one in both directions. For details, see the description of the `UNIX_TIMESTAMP()` function.

- `GET_FORMAT({DATE|TIME|DATETIME}, {'EUR'|'USA'|'JIS'|'ISO'|'INTERNAL'})`

Returns a format string. This function is useful in combination with the `DATE_FORMAT()` and the `STR_TO_DATE()` functions.

If ***format*** is `NULL`, this function returns `NULL`.

The possible values for the first and second arguments result in several possible format strings (for the specifiers used, see the table in the `DATE_FORMAT()` function description). ISO format refers to ISO 9075, not ISO 8601.

Function Call	**Result**
`GET_FORMAT(DATE,'USA')`	`'%m.%d.%Y'`

Function Call	Result
`GET_FORMAT(DATE,'JIS')`	`'%Y-%m-%d'`
`GET_FORMAT(DATE,'ISO')`	`'%Y-%m-%d'`
`GET_FORMAT(DATE,'EUR')`	`'%d.%m.%Y'`
`GET_FORMAT(DATE,'INTERNAL')`	`'%Y%m%d'`
`GET_FORMAT(DATETIME,'USA')`	`'%Y-%m-%d %H.%i.%s'`
`GET_FORMAT(DATETIME,'JIS')`	`'%Y-%m-%d %H:%i:%s'`
`GET_FORMAT(DATETIME,'ISO')`	`'%Y-%m-%d %H:%i:%s'`
`GET_FORMAT(DATETIME,'EUR')`	`'%Y-%m-%d %H.%i.%s'`
`GET_FORMAT(DATETIME,'INTERNAL')`	`'%Y%m%d%H%i%s'`
`GET_FORMAT(TIME,'USA')`	`'%h:%i:%s %p'`
`GET_FORMAT(TIME,'JIS')`	`'%H:%i:%s'`
`GET_FORMAT(TIME,'ISO')`	`'%H:%i:%s'`
`GET_FORMAT(TIME,'EUR')`	`'%H.%i.%s'`
`GET_FORMAT(TIME,'INTERNAL')`	`'%H%i%s'`

`TIMESTAMP` can also be used as the first argument to `GET_FORMAT()`, in which case the function returns the same values as for `DATETIME`.

```
mysql> SELECT DATE_FORMAT('2003-10-03',GET_FORMAT(DATE,'EUR'));
        -> '03.10.2003'
mysql> SELECT STR_TO_DATE('10.31.2003',GET_FORMAT(DATE,'USA'));
        -> '2003-10-31'
```

- `HOUR(`***`time`***`)`

  Returns the hour for ***`time`***. The range of the return value is `0` to `23` for time-of-day values. However, the range of `TIME` values actually is much larger, so `HOUR` can return values greater than `23`. Returns `NULL` if ***`time`*** is `NULL`.

```
mysql> SELECT HOUR('10:05:03');
        -> 10
mysql> SELECT HOUR('272:59:59');
        -> 272
```

- `LAST_DAY(`***`date`***`)`

Takes a date or datetime value and returns the corresponding value for the last day of the month. Returns `NULL` if the argument is invalid or `NULL`.

```
mysql> SELECT LAST_DAY('2003-02-05');
        -> '2003-02-28'
mysql> SELECT LAST_DAY('2004-02-05');
        -> '2004-02-29'
mysql> SELECT LAST_DAY('2004-01-01 01:01:01');
        -> '2004-01-31'
mysql> SELECT LAST_DAY('2003-03-32');
        -> NULL
```

- `LOCALTIME`, `LOCALTIME([fsp])`

  `LOCALTIME` and `LOCALTIME()` are synonyms for `NOW()`.

- `LOCALTIMESTAMP`, `LOCALTIMESTAMP([fsp])`

  `LOCALTIMESTAMP` and `LOCALTIMESTAMP()` are synonyms for `NOW()`.

- `MAKEDATE(year,dayofyear)`

  Returns a date, given year and day-of-year values. ***`dayofyear`*** must be greater than 0 or the result is `NULL`. The result is also `NULL` if either argument is `NULL`.

```
mysql> SELECT MAKEDATE(2011,31), MAKEDATE(2011,32);
        -> '2011-01-31', '2011-02-01'
mysql> SELECT MAKEDATE(2011,365), MAKEDATE(2014,365);
        -> '2011-12-31', '2014-12-31'
mysql> SELECT MAKEDATE(2011,0);
        -> NULL
```

- `MAKETIME(hour,minute,second)`

  Returns a time value calculated from the ***`hour`***, ***`minute`***, and ***`second`*** arguments. Returns `NULL` if any of its arguments are `NULL`.

  The ***`second`*** argument can have a fractional part.

```
mysql> SELECT MAKETIME(12,15,30);
        -> '12:15:30'
```

- `MICROSECOND(expr)`

Returns the microseconds from the time or datetime expression ***`expr`*** as a number in the range from `0` to `999999`. Returns `NULL` if ***`expr`*** is `NULL`.

```
mysql> SELECT MICROSECOND('12:00:00.123456');
        -> 123456
mysql> SELECT MICROSECOND('2019-12-31 23:59:59.000010');
        -> 10
```

- `MINUTE(`***`time`***`)`

  Returns the minute for ***`time`***, in the range `0` to `59`, or `NULL` if ***`time`*** is `NULL`.

```
mysql> SELECT MINUTE('2008-02-03 10:05:03');
        -> 5
```

- `MONTH(`***`date`***`)`

  Returns the month for ***`date`***, in the range `1` to `12` for January to December, or `0` for dates such as `'0000-00-00'` or `'2008-00-00'` that have a zero month part. Returns `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT MONTH('2008-02-03');
        -> 2
```

- `MONTHNAME(`***`date`***`)`

  Returns the full name of the month for ***`date`***. The language used for the name is controlled by the value of the `lc_time_names` system variable (Section 12.16, “MySQL Server Locale Support”). Returns `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT MONTHNAME('2008-02-03');
        -> 'February'
```

- `NOW([`***`fsp`***`])`

  Returns the current date and time as a value in `'`***`YYYY-MM-DD hh:mm:ss`***`'` or ***`YYYYMMDDhhmmss`*** format, depending on whether the function is used in string or numeric context. The value is expressed in the session time zone.

  If the ***`fsp`*** argument is given to specify a fractional seconds precision from 0 to 6, the return value includes a fractional seconds part of that many digits.

```
mysql> SELECT NOW();
        -> '2007-12-15 23:50:26'
mysql> SELECT NOW() + 0;
        -> 20071215235026.000000
```

`NOW()` returns a constant time that indicates the time at which the statement began to execute. (Within a stored function or trigger, `NOW()` returns the time at which the function or triggering statement began to execute.) This differs from the behavior for `SYSDATE()`, which returns the exact time at which it executes.

```
mysql> SELECT NOW(), SLEEP(2), NOW();
+---------------------+----------+---------------------+
| NOW()               | SLEEP(2) | NOW()               |
+---------------------+----------+---------------------+
| 2006-04-12 13:47:36 |        0 | 2006-04-12 13:47:36 |
+---------------------+----------+---------------------+

mysql> SELECT SYSDATE(), SLEEP(2), SYSDATE();
+---------------------+----------+---------------------+
| SYSDATE()           | SLEEP(2) | SYSDATE()           |
+---------------------+----------+---------------------+
| 2006-04-12 13:47:44 |        0 | 2006-04-12 13:47:46 |
+---------------------+----------+---------------------+
```

In addition, the `SET TIMESTAMP` statement affects the value returned by `NOW()` but not by `SYSDATE()`. This means that timestamp settings in the binary log have no effect on invocations of `SYSDATE()`. Setting the timestamp to a nonzero value causes each subsequent invocation of `NOW()` to return that value. Setting the timestamp to zero cancels this effect so that `NOW()` once again returns the current date and time.

See the description for `SYSDATE()` for additional information about the differences between the two functions.

- `PERIOD_ADD(`***`P`***`,`***`N`***`)`

  Adds ***`N`*** months to period ***`P`*** (in the format ***`YYMM`*** or ***`YYYYMM`***). Returns a value in the format ***`YYYYMM`***.

  > **Note**
  >
  > The period argument ***`P`*** is *not* a date value.

  This function returns `NULL` if ***`P`*** or ***`N`*** is `NULL`.

```
mysql> SELECT PERIOD_ADD(200801,2);
        -> 200803
```

- `PERIOD_DIFF(`***`P1`***`,`***`P2`***`)`

  Returns the number of months between periods ***`P1`*** and ***`P2`***. ***`P1`*** and ***`P2`*** should be in the format ***`YYMM`*** or ***`YYYYMM`***. Note that the period arguments ***`P1`*** and ***`P2`*** are *not* date values.

  This function returns `NULL` if ***`P1`*** or ***`P2`*** is `NULL`.

```
mysql> SELECT PERIOD_DIFF(200802,200703);
        -> 11
```

- `QUARTER(`***`date`***`)`

  Returns the quarter of the year for ***`date`***, in the range `1` to `4`, or `NULL` if ***`date`*** is `NULL`.

```
mysql> SELECT QUARTER('2008-04-01');
        -> 2
```

- `SECOND(`***`time`***`)`

  Returns the second for ***`time`***, in the range `0` to `59`, or `NULL` if ***`time`*** is `NULL`.

```
mysql> SELECT SECOND('10:05:03');
        -> 3
```

- `SEC_TO_TIME(`***`seconds`***`)`

  Returns the ***`seconds`*** argument, converted to hours, minutes, and seconds, as a `TIME` value. The range of the result is constrained to that of the `TIME` data type. A warning occurs if the argument corresponds to a value outside that range.

  The function returns `NULL` if ***`seconds`*** is `NULL`.

```
mysql> SELECT SEC_TO_TIME(2378);
        -> '00:39:38'
mysql> SELECT SEC_TO_TIME(2378) + 0;
        -> 3938
```

- `STR_TO_DATE(`***`str`***`,`***`format`***`)`

This is the inverse of the `DATE_FORMAT()` function. It takes a string ***str*** and a format string ***format***. `STR_TO_DATE()` returns a `DATETIME` value if the format string contains both date and time parts, or a `DATE` or `TIME` value if the string contains only date or time parts. If ***str*** or ***format*** is `NULL`, the function returns `NULL`. If the date, time, or datetime value extracted from ***str*** cannot be parsed according to the rules followed by the server, `STR_TO_DATE()` returns `NULL` and produces a warning.

The server scans ***str*** attempting to match ***format*** to it. The format string can contain literal characters and format specifiers beginning with `%`. Literal characters in ***format*** must match literally in ***str***. Format specifiers in ***format*** must match a date or time part in ***str***. For the specifiers that can be used in ***format***, see the `DATE_FORMAT()` function description.

```
mysql> SELECT STR_TO_DATE('01,5,2013','%d,%m,%Y');
        -> '2013-05-01'
mysql> SELECT STR_TO_DATE('May 1, 2013','%M %d,%Y');
        -> '2013-05-01'
```

Scanning starts at the beginning of ***str*** and fails if ***format*** is found not to match. Extra characters at the end of ***str*** are ignored.

```
mysql> SELECT STR_TO_DATE('a09:30:17','a%h:%i:%s');
        -> '09:30:17'
mysql> SELECT STR_TO_DATE('a09:30:17','%h:%i:%s');
        -> NULL
mysql> SELECT STR_TO_DATE('09:30:17a','%h:%i:%s');
        -> '09:30:17'
```

Unspecified date or time parts have a value of 0, so incompletely specified values in ***str*** produce a result with some or all parts set to 0:

```
mysql> SELECT STR_TO_DATE('abc','abc');
        -> '0000-00-00'
mysql> SELECT STR_TO_DATE('9','%m');
        -> '0000-09-00'
mysql> SELECT STR_TO_DATE('9','%s');
        -> '00:00:09'
```

Range checking on the parts of date values is as described in Section 13.2.2, “The DATE, DATETIME, and TIMESTAMP Types”. This means, for example, that “zero” dates or dates with part values of 0 are permitted unless the SQL mode is set to disallow such values.

```
mysql> SELECT STR_TO_DATE('00/00/0000', '%m/%d/%Y');
        -> '0000-00-00'
mysql> SELECT STR_TO_DATE('04/31/2004', '%m/%d/%Y');
        -> '2004-04-31'
```

If the `NO_ZERO_DATE` SQL mode is enabled, zero dates are disallowed. In that case, `STR_TO_DATE()` returns `NULL` and generates a warning:

```
mysql> SET sql_mode = '';
mysql> SELECT STR_TO_DATE('00/00/0000', '%m/%d/%Y');
+---------------------------------------+
| STR_TO_DATE('00/00/0000', '%m/%d/%Y') |
+---------------------------------------+
| 0000-00-00                            |
+---------------------------------------+
mysql> SET sql_mode = 'NO_ZERO_DATE';
mysql> SELECT STR_TO_DATE('00/00/0000', '%m/%d/%Y');
+---------------------------------------+
| STR_TO_DATE('00/00/0000', '%m/%d/%Y') |
+---------------------------------------+
| NULL                                  |
+---------------------------------------+
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 1411
Message: Incorrect datetime value: '00/00/0000' for function str_to_date
```

Prior to MySQL 8.0.35, it was possible to pass an invalid date string such as `'2021-11-31'` to this function. In MySQL 8.0.35 and later, `STR_TO_DATE()` performs complete range checking and raises an error if the date after conversion would be invalid.

> **Note**
>
> You cannot use format `"%X%V"` to convert a year-week string to a date because the combination of a year and week does not uniquely identify a year and month if the week crosses a month boundary. To convert a year-week to a date, you should also specify the weekday:
>
> ```
> mysql> SELECT STR_TO_DATE('200442 Monday', '%X%V %W');
>         -> '2004-10-18'
> ```

You should also be aware that, for dates and the date portions of datetime values, `STR_TO_DATE()` checks (only) the individual year, month, and day of month values for validity. More precisely, this means that the year is checked to be sure that it is in the range 0-9999 inclusive, the month is checked to ensure that it is in the range 1-12 inclusive, and the day of month is checked to make sure that it is in the range 1-31 inclusive, but the server does not check the values in combination. For example, `SELECT STR_TO_DATE('23-2-31', '%Y-%m-%d')` returns `2023-02-31`. Enabling or disabling the `ALLOW_INVALID_DATES` server SQL mode has no effect on this behavior. See Section 13.2.2, “The DATE, DATETIME, and TIMESTAMP Types”, for more information.

- `SUBDATE(`***`date`***`,INTERVAL `***`expr unit`***`)`, `SUBDATE(`***`expr`***`,`***`days`***`)`

  When invoked with the `INTERVAL` form of the second argument, `SUBDATE()` is a synonym for `DATE_SUB()`. For information on the `INTERVAL` ***`unit`*** argument, see the discussion for `DATE_ADD()`.

  ```
  mysql> SELECT DATE_SUB('2008-01-02', INTERVAL 31 DAY);
          -> '2007-12-02'
  mysql> SELECT SUBDATE('2008-01-02', INTERVAL 31 DAY);
          -> '2007-12-02'
  ```

  The second form enables the use of an integer value for ***`days`***. In such cases, it is interpreted as the number of days to be subtracted from the date or datetime expression ***`expr`***.

  ```
  mysql> SELECT SUBDATE('2008-01-02 12:00:00', 31);
          -> '2007-12-02 12:00:00'
  ```

  This function returns `NULL` if any of its arguments are `NULL`.

- `SUBTIME(`***`expr1`***`,`***`expr2`***`)`

  `SUBTIME()` returns ***`expr1`*** − ***`expr2`*** expressed as a value in the same format as ***`expr1`***. ***`expr1`*** is a time or datetime expression, and ***`expr2`*** is a time expression.

  Resolution of this function's return type is performed as it is for the `ADDTIME()` function; see the description of that function for more information.

  ```
  mysql> SELECT SUBTIME('2007-12-31 23:59:59.999999','1 1:1:1.000002');
          -> '2007-12-30 22:58:58.999997'
  mysql> SELECT SUBTIME('01:00:00.999999', '02:00:00.999998');
          -> '-00:59:59.999999'
  ```

This function returns `NULL` if ***`expr1`*** or ***`expr2`*** is `NULL`.

- `SYSDATE([`***`fsp`***`])`

  Returns the current date and time as a value in `'`***`YYYY-MM-DD hh:mm:ss`***`'` or ***`YYYYMMDDhhmmss`*** format, depending on whether the function is used in string or numeric context.

  If the ***`fsp`*** argument is given to specify a fractional seconds precision from 0 to 6, the return value includes a fractional seconds part of that many digits.

  `SYSDATE()` returns the time at which it executes. This differs from the behavior for `NOW()`, which returns a constant time that indicates the time at which the statement began to execute. (Within a stored function or trigger, `NOW()` returns the time at which the function or triggering statement began to execute.)

```
mysql> SELECT NOW(), SLEEP(2), NOW();
+---------------------+----------+---------------------+
| NOW()               | SLEEP(2) | NOW()               |
+---------------------+----------+---------------------+
| 2006-04-12 13:47:36 |        0 | 2006-04-12 13:47:36 |
+---------------------+----------+---------------------+

mysql> SELECT SYSDATE(), SLEEP(2), SYSDATE();
+---------------------+----------+---------------------+
| SYSDATE()           | SLEEP(2) | SYSDATE()           |
+---------------------+----------+---------------------+
| 2006-04-12 13:47:44 |        0 | 2006-04-12 13:47:46 |
+---------------------+----------+---------------------+
```

  In addition, the `SET TIMESTAMP` statement affects the value returned by `NOW()` but not by `SYSDATE()`. This means that timestamp settings in the binary log have no effect on invocations of `SYSDATE()`.

  Because `SYSDATE()` can return different values even within the same statement, and is not affected by `SET TIMESTAMP`, it is nondeterministic and therefore unsafe for replication if statement-based binary logging is used. If that is a problem, you can use row-based logging.

  Alternatively, you can use the `--sysdate-is-now` option to cause `SYSDATE()` to be an alias for `NOW()`. This works if the option is used on both the replication source server and the replica.

  The nondeterministic nature of `SYSDATE()` also means that indexes cannot be used for evaluating expressions that refer to it.

- `TIME(`***`expr`***`)`

Extracts the time part of the time or datetime expression ***expr*** and returns it as a string. Returns `NULL` if ***expr*** is `NULL`.

This function is unsafe for statement-based replication. A warning is logged if you use this function when `binlog_format` is set to `STATEMENT`.

```
mysql> SELECT TIME('2003-12-31 01:02:03');
        -> '01:02:03'
mysql> SELECT TIME('2003-12-31 01:02:03.000123');
        -> '01:02:03.000123'
```

- `TIMEDIFF(`***expr1***`,`***expr2***`)`

  `TIMEDIFF()` returns ***expr1*** − ***expr2*** expressed as a time value. ***expr1*** and ***expr2*** are strings which are converted to `TIME` or `DATETIME` expressions; these must be of the same type following conversion. Returns `NULL` if ***expr1*** or ***expr2*** is `NULL`.

  The result returned by `TIMEDIFF()` is limited to the range allowed for `TIME` values. Alternatively, you can use either of the functions `TIMESTAMPDIFF()` and `UNIX_TIMESTAMP()`, both of which return integers.

```
mysql> SELECT TIMEDIFF('2000-01-01 00:00:00',
    ->                 '2000-01-01 00:00:00.000001');
        -> '-00:00:00.000001'
mysql> SELECT TIMEDIFF('2008-12-31 23:59:59.000001',
    ->                 '2008-12-30 01:01:01.000002');
        -> '46:58:57.999999'
```

- `TIMESTAMP(`***expr***`)`, `TIMESTAMP(`***expr1***`,`***expr2***`)`

  With a single argument, this function returns the date or datetime expression ***expr*** as a datetime value. With two arguments, it adds the time expression ***expr2*** to the date or datetime expression ***expr1*** and returns the result as a datetime value. Returns `NULL` if ***expr***, ***expr1***, or ***expr2*** is `NULL`.

```
mysql> SELECT TIMESTAMP('2003-12-31');
        -> '2003-12-31 00:00:00'
mysql> SELECT TIMESTAMP('2003-12-31 12:00:00','12:00:00');
        -> '2004-01-01 00:00:00'
```

- `TIMESTAMPADD(`***unit***`,`***interval***`,`***datetime_expr***`)`

  Adds the integer expression ***interval*** to the date or datetime expression ***datetime_expr***. The unit for ***interval*** is given by the ***unit*** argument, which should be one of the following values:

`MICROSECOND` (microseconds), `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `QUARTER`, or `YEAR`.

The ***`unit`*** value may be specified using one of keywords as shown, or with a prefix of `SQL_TSI_`. For example, `DAY` and `SQL_TSI_DAY` both are legal.

This function returns `NULL` if ***`interval`*** or ***`datetime_expr`*** is `NULL`.

```
mysql> SELECT TIMESTAMPADD(MINUTE, 1, '2003-01-02');
        -> '2003-01-02 00:01:00'
mysql> SELECT TIMESTAMPADD(WEEK,1,'2003-01-02');
        -> '2003-01-09'
```

When adding a `MONTH` interval to a `DATE` or `DATETIME` value, and the resulting date includes a day that does not exist in the given month, the day is adjusted to the last day of the month, as shown here:

```
mysql> SELECT TIMESTAMPADD(MONTH, 1, DATE '2024-03-30') AS t1,
     >        TIMESTAMPADD(MONTH, 1, DATE '2024-03-31') AS t2;
+------------+------------+
| t1         | t2         |
+------------+------------+
| 2024-04-30 | 2024-04-30 |
+------------+------------+
1 row in set (0.00 sec)
```

- `TIMESTAMPDIFF(`***`unit`***`,`***`datetime_expr1`***`,`***`datetime_expr2`***`)`

  Returns ***`datetime_expr2`*** − ***`datetime_expr1`***, where ***`datetime_expr1`*** and ***`datetime_expr2`*** are date or datetime expressions. One expression may be a date and the other a datetime; a date value is treated as a datetime having the time part `'00:00:00'` where necessary. The unit for the result (an integer) is given by the ***`unit`*** argument. The legal values for ***`unit`*** are the same as those listed in the description of the `TIMESTAMPADD()` function.

  This function returns `NULL` if ***`datetime_expr1`*** or ***`datetime_expr2`*** is `NULL`.

```
mysql> SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');
        -> 3
mysql> SELECT TIMESTAMPDIFF(YEAR,'2002-05-01','2001-01-01');
        -> -1
mysql> SELECT TIMESTAMPDIFF(MINUTE,'2003-02-01','2003-05-01 12:05:55');
        -> 128885
```

Note

> The order of the date or datetime arguments for this function is the opposite of that used with the `TIMESTAMP()` function when invoked with 2 arguments.

- `TIME_FORMAT(`***`time`***`,`***`format`***`)`

  This is used like the `DATE_FORMAT()` function, but the ***`format`*** string may contain format specifiers only for hours, minutes, seconds, and microseconds. Other specifiers produce a `NULL` or `0`. `TIME_FORMAT()` returns `NULL` if ***`time`*** or ***`format`*** is `NULL`.

  If the ***`time`*** value contains an hour part that is greater than `23`, the `%H` and `%k` hour format specifiers produce a value larger than the usual range of `0..23`. The other hour format specifiers produce the hour value modulo 12.

  ```
  mysql> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l');
          -> '100 100 04 04 4'
  ```

- `TIME_TO_SEC(`***`time`***`)`

  Returns the ***`time`*** argument, converted to seconds. Returns `NULL` if ***`time`*** is `NULL`.

  ```
  mysql> SELECT TIME_TO_SEC('22:23:00');
          -> 80580
  mysql> SELECT TIME_TO_SEC('00:39:38');
          -> 2378
  ```

- `TO_DAYS(`***`date`***`)`

  Given a date ***`date`***, returns a day number (the number of days since year 0). Returns `NULL` if ***`date`*** is `NULL`.

  ```
  mysql> SELECT TO_DAYS(950501);
          -> 728779
  mysql> SELECT TO_DAYS('2007-10-07');
          -> 733321
  ```

  `TO_DAYS()` is not intended for use with values that precede the advent of the Gregorian calendar (1582), because it does not take into account the days that were lost when the calendar was changed. For dates before 1582 (and possibly a later year in other locales), results from this function are not reliable. See Section 13.2.7, “What Calendar Is Used By MySQL?”, for details.

  Remember that MySQL converts two-digit year values in dates to four-digit form using the rules in Section 13.2, “Date and Time Data Types”. For example, `'2008-10-07'` and `'08-10-07'` are seen as identical dates:

```
mysql> SELECT TO_DAYS('2008-10-07'), TO_DAYS('08-10-07');
        -> 733687, 733687
```

In MySQL, the zero date is defined as `'0000-00-00'`, even though this date is itself considered invalid. This means that, for `'0000-00-00'` and `'0000-01-01'`, `TO_DAYS()` returns the values shown here:

```
mysql> SELECT TO_DAYS('0000-00-00');
+-----------------------+
| to_days('0000-00-00') |
+-----------------------+
|                  NULL |
+-----------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Incorrect datetime value: '0000-00-00' |
+---------+------+----------------------------------------+
1 row in set (0.00 sec)


mysql> SELECT TO_DAYS('0000-01-01');
+-----------------------+
| to_days('0000-01-01') |
+-----------------------+
|                     1 |
+-----------------------+
1 row in set (0.00 sec)
```

This is true whether or not the `ALLOW_INVALID_DATES` SQL server mode is enabled.

- `TO_SECONDS(`***`expr`***`)`

  Given a date or datetime ***`expr`***, returns the number of seconds since the year 0. If ***`expr`*** is not a valid date or datetime value (including `NULL`), it returns `NULL`.

```
mysql> SELECT TO_SECONDS(950501);
        -> 62966505600
mysql> SELECT TO_SECONDS('2009-11-29');
        -> 63426672000
mysql> SELECT TO_SECONDS('2009-11-29 13:43:32');
```

```
        -> 63426721412
mysql> SELECT TO_SECONDS( NOW() );
        -> 63426721458
```

Like `TO_DAYS()`, `TO_SECONDS()` is not intended for use with values that precede the advent of the Gregorian calendar (1582), because it does not take into account the days that were lost when the calendar was changed. For dates before 1582 (and possibly a later year in other locales), results from this function are not reliable. See Section 13.2.7, “What Calendar Is Used By MySQL?”, for details.

Like `TO_DAYS()`, `TO_SECONDS()`, converts two-digit year values in dates to four-digit form using the rules in Section 13.2, “Date and Time Data Types”.

In MySQL, the zero date is defined as `'0000-00-00'`, even though this date is itself considered invalid. This means that, for `'0000-00-00'` and `'0000-01-01'`, `TO_SECONDS()` returns the values shown here:

```
mysql> SELECT TO_SECONDS('0000-00-00');
+--------------------------+
| TO_SECONDS('0000-00-00') |
+--------------------------+
|                     NULL |
+--------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Incorrect datetime value: '0000-00-00' |
+---------+------+----------------------------------------+
1 row in set (0.00 sec)


mysql> SELECT TO_SECONDS('0000-01-01');
+--------------------------+
| TO_SECONDS('0000-01-01') |
+--------------------------+
|                    86400 |
+--------------------------+
1 row in set (0.00 sec)
```

This is true whether or not the `ALLOW_INVALID_DATES` SQL server mode is enabled.

- `UNIX_TIMESTAMP([`***`date`***`])`

  If `UNIX_TIMESTAMP()` is called with no ***`date`*** argument, it returns a Unix timestamp representing seconds since `'1970-01-01 00:00:00'` UTC.

If `UNIX_TIMESTAMP()` is called with a ***date*** argument, it returns the value of the argument as seconds since `'1970-01-01 00:00:00'` UTC. The server interprets ***date*** as a value in the session time zone and converts it to an internal Unix timestamp value in UTC. (Clients can set the session time zone as described in Section 7.1.15, "MySQL Server Time Zone Support".) The ***date*** argument may be a `DATE`, `DATETIME`, or `TIMESTAMP` string, or a number in ***YYMMDD***, ***YYMMDDhhmmss***, ***YYYYMMDD***, or ***YYYYMMDDhhmmss*** format. If the argument includes a time part, it may optionally include a fractional seconds part.

The return value is an integer if no argument is given or the argument does not include a fractional seconds part, or `DECIMAL` if an argument is given that includes a fractional seconds part.

When the ***date*** argument is a `TIMESTAMP` column, `UNIX_TIMESTAMP()` returns the internal timestamp value directly, with no implicit "string-to-Unix-timestamp" conversion.

Prior to MySQL 8.0.28, the valid range of argument values is the same as for the `TIMESTAMP` data type: `'1970-01-01 00:00:01.000000'` UTC to `'2038-01-19 03:14:07.999999'` UTC. This is also the case in MySQL 8.0.28 and later for 32-bit platforms. For MySQL 8.0.28 and later running on 64-bit platforms, the valid range of argument values for `UNIX_TIMESTAMP()` is `'1970-01-01 00:00:01.000000'` UTC to `'3001-01-19 03:14:07.999999'` UTC (corresponding to 32536771199.999999 seconds).

Regardless of MySQL version or platform architecture, if you pass an out-of-range date to `UNIX_TIMESTAMP()`, it returns `0`. If ***date*** is `NULL`, it returns `NULL`.

```
mysql> SELECT UNIX_TIMESTAMP();
        -> 1447431666
mysql> SELECT UNIX_TIMESTAMP('2015-11-13 10:20:19');
        -> 1447431619
mysql> SELECT UNIX_TIMESTAMP('2015-11-13 10:20:19.012');
        -> 1447431619.012
```

If you use `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` to convert between values in a non-UTC time zone and Unix timestamp values, the conversion is lossy because the mapping is not one-to-one in both directions. For example, due to conventions for local time zone changes such as Daylight Saving Time (DST), it is possible for `UNIX_TIMESTAMP()` to map two values that are distinct in a non-UTC time zone to the same Unix timestamp value. `FROM_UNIXTIME()` maps that value back to only one of the original values. Here is an example, using values that are distinct in the `MET` time zone:

```
mysql> SET time_zone = 'MET';
mysql> SELECT UNIX_TIMESTAMP('2005-03-27 03:00:00');
+---------------------------------------+
| UNIX_TIMESTAMP('2005-03-27 03:00:00') |
+---------------------------------------+
|                            1111885200 |
+---------------------------------------+
mysql> SELECT UNIX_TIMESTAMP('2005-03-27 02:00:00');
+---------------------------------------+
| UNIX_TIMESTAMP('2005-03-27 02:00:00') |
+---------------------------------------+
|                            1111885200 |
+---------------------------------------+
mysql> SELECT FROM_UNIXTIME(1111885200);
+---------------------------+
| FROM_UNIXTIME(1111885200) |
+---------------------------+
| 2005-03-27 03:00:00       |
+---------------------------+
```

> **Note**
>
> To use named time zones such as `'MET'` or `'Europe/Amsterdam'`, the time zone tables must be properly set up. For instructions, see Section 7.1.15, “MySQL Server Time Zone Support”.

If you want to subtract `UNIX_TIMESTAMP()` columns, you might want to cast them to signed integers. See Section 14.10, “Cast Functions and Operators”.

- `UTC_DATE`, `UTC_DATE()`

  Returns the current UTC date as a value in `'YYYY-MM-DD'` or `YYYYMMDD` format, depending on whether the function is used in string or numeric context.

```
mysql> SELECT UTC_DATE(), UTC_DATE() + 0;
        -> '2003-08-14', 20030814
```

- `UTC_TIME`, `UTC_TIME([fsp])`

  Returns the current UTC time as a value in `'hh:mm:ss'` or `hhmmss` format, depending on whether the function is used in string or numeric context.

  If the `fsp` argument is given to specify a fractional seconds precision from 0 to 6, the return value includes a fractional seconds part of that many digits.

```
mysql> SELECT UTC_TIME(), UTC_TIME() + 0;
        -> '18:07:53', 180753.000000
```

- `UTC_TIMESTAMP`, `UTC_TIMESTAMP([fsp])`

  Returns the current UTC date and time as a value in `'YYYY-MM-DD hh:mm:ss'` or `YYYYMMDDhhmmss` format, depending on whether the function is used in string or numeric context.

  If the ***`fsp`*** argument is given to specify a fractional seconds precision from 0 to 6, the return value includes a fractional seconds part of that many digits.

```
mysql> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0;
        -> '2003-08-14 18:08:04', 20030814180804.000000
```

- `WEEK(date[,mode])`

  This function returns the week number for ***`date`***. The two-argument form of `WEEK()` enables you to specify whether the week starts on Sunday or Monday and whether the return value should be in the range from `0` to `53` or from `1` to `53`. If the ***`mode`*** argument is omitted, the value of the `default_week_format` system variable is used. See Section 7.1.8, “Server System Variables”. For a `NULL` date value, the function returns `NULL`.

  The following table describes how the ***`mode`*** argument works.

Mode	First day of week	Range	Week 1 is the first week …
0	Sunday	0-53	with a Sunday in this year
1	Monday	0-53	with 4 or more days this year
2	Sunday	1-53	with a Sunday in this year
3	Monday	1-53	with 4 or more days this year
4	Sunday	0-53	with 4 or more days this year
5	Monday	0-53	with a Monday in this year
6	Sunday	1-53	with 4 or more days this year
7	Monday	1-53	with a Monday in this year

  For ***`mode`*** values with a meaning of “with 4 or more days this year,” weeks are numbered according to ISO 8601:1988:

  - If the week containing January 1 has 4 or more days in the new year, it is week 1.
  - Otherwise, it is the last week of the previous year, and the next week is week 1.

```
mysql> SELECT WEEK('2008-02-20');
        -> 7
mysql> SELECT WEEK('2008-02-20',0);
```

```
        -> 7
mysql> SELECT WEEK('2008-02-20',1);
        -> 8
mysql> SELECT WEEK('2008-12-31',1);
        -> 53
```

If a date falls in the last week of the previous year, MySQL returns `0` if you do not use `2`, `3`, `6`, or `7` as the optional ***mode*** argument:

```
mysql> SELECT YEAR('2000-01-01'), WEEK('2000-01-01',0);
        -> 2000, 0
```

One might argue that `WEEK()` should return `52` because the given date actually occurs in the 52nd week of 1999. `WEEK()` returns `0` instead so that the return value is "the week number in the given year." This makes use of the `WEEK()` function reliable when combined with other functions that extract a date part from a date.

If you prefer a result evaluated with respect to the year that contains the first day of the week for the given date, use `0`, `2`, `5`, or `7` as the optional ***mode*** argument.

```
mysql> SELECT WEEK('2000-01-01',2);
        -> 52
```

Alternatively, use the `YEARWEEK()` function:

```
mysql> SELECT YEARWEEK('2000-01-01');
        -> 199952
mysql> SELECT MID(YEARWEEK('2000-01-01'),5,2);
        -> '52'
```

- `WEEKDAY(`***`date`***`)`

  Returns the weekday index for ***date*** (`0` = Monday, `1` = Tuesday, ... `6` = Sunday). Returns `NULL` if ***date*** is `NULL`.

```
mysql> SELECT WEEKDAY('2008-02-03 22:23:00');
        -> 6
mysql> SELECT WEEKDAY('2007-11-06');
        -> 1
```

- `WEEKOFYEAR(`***`date`***`)`

Returns the calendar week of the date as a number in the range from `1` to `53`. Returns `NULL` if ***`date`*** is `NULL`.

`WEEKOFYEAR()` is a compatibility function that is equivalent to `WEEK(date,3)`.

```
mysql> SELECT WEEKOFYEAR('2008-02-20');
        -> 8
```

- `YEAR(date)`

  Returns the year for ***`date`***, in the range `1000` to `9999`, or `0` for the "zero" date. Returns `NULL` if ***`date`*** is `NULL`.

  ```
  mysql> SELECT YEAR('1987-01-01');
          -> 1987
  ```

- `YEARWEEK(date)`, `YEARWEEK(date,mode)`

  Returns year and week for a date. The year in the result may be different from the year in the date argument for the first and the last week of the year. Returns `NULL` if ***`date`*** is `NULL`.

  The ***`mode`*** argument works exactly like the ***`mode`*** argument to `WEEK()`. For the single-argument syntax, a ***`mode`*** value of 0 is used. Unlike `WEEK()`, the value of `default_week_format` does not influence `YEARWEEK()`.

  ```
  mysql> SELECT YEARWEEK('1987-01-01');
          -> 198652
  ```

  The week number is different from what the `WEEK()` function would return (`0`) for optional arguments `0` or `1`, as `WEEK()` then returns the week in the context of the given year.

14.8 String Functions and Operators

Table 14.12 String Functions and Operators

Name	Description
`ASCII()`	Return numeric value of left-most character
`BIN()`	Return a string containing binary representation of a number
`BIT_LENGTH()`	Return length of argument in bits
`CHAR()`	Return the character for each integer passed
`CHAR_LENGTH()`	Return number of characters in argument
`CHARACTER_LENGTH()`	Synonym for CHAR_LENGTH()
`CONCAT()`	Return concatenated string
`CONCAT_WS()`	Return concatenate with separator
`ELT()`	Return string at index number
`EXPORT_SET()`	Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
`FIELD()`	Index (position) of first argument in subsequent arguments
`FIND_IN_SET()`	Index (position) of first argument within second argument
`FORMAT()`	Return a number formatted to specified number of decimal places
`FROM_BASE64()`	Decode base64 encoded string and return result
`HEX()`	Hexadecimal representation of decimal or string value
`INSERT()`	Insert substring at specified position up to specified number of characters
`INSTR()`	Return the index of the first occurrence of substring
`LCASE()`	Synonym for LOWER()
`LEFT()`	Return the leftmost number of characters as specified
`LENGTH()`	Return the length of a string in bytes
`LIKE`	Simple pattern matching
`LOAD_FILE()`	Load the named file
`LOCATE()`	Return the position of the first occurrence of substring
`LOWER()`	Return the argument in lowercase

Name	Description
`LPAD()`	Return the string argument, left-padded with the specified string
`LTRIM()`	Remove leading spaces
`MAKE_SET()`	Return a set of comma-separated strings that have the corresponding bit in bits set
`MATCH()`	Perform full-text search
`MID()`	Return a substring starting from the specified position
`NOT LIKE`	Negation of simple pattern matching
`NOT REGEXP`	Negation of REGEXP
`OCT()`	Return a string containing octal representation of a number
`OCTET_LENGTH()`	Synonym for LENGTH()
`ORD()`	Return character code for leftmost character of the argument
`POSITION()`	Synonym for LOCATE()
`QUOTE()`	Escape the argument for use in an SQL statement
`REGEXP`	Whether string matches regular expression
`REGEXP_INSTR()`	Starting index of substring matching regular expression
`REGEXP_LIKE()`	Whether string matches regular expression
`REGEXP_REPLACE()`	Replace substrings matching regular expression
`REGEXP_SUBSTR()`	Return substring matching regular expression
`REPEAT()`	Repeat a string the specified number of times
`REPLACE()`	Replace occurrences of a specified string
`REVERSE()`	Reverse the characters in a string
`RIGHT()`	Return the specified rightmost number of characters
`RLIKE`	Whether string matches regular expression
`RPAD()`	Append string the specified number of times
`RTRIM()`	Remove trailing spaces
`SOUNDEX()`	Return a soundex string
`SOUNDS LIKE`	Compare sounds
`SPACE()`	Return a string of the specified number of spaces
`STRCMP()`	Compare two strings
`SUBSTR()`	Return the substring as specified
`SUBSTRING()`	Return the substring as specified
`SUBSTRING_INDEX()`	Return a substring from a string before the specified number of occurrences of the delimiter

Name	Description
`TO_BASE64()`	Return the argument converted to a base-64 string
`TRIM()`	Remove leading and trailing spaces
`UCASE()`	Synonym for UPPER()
`UNHEX()`	Return a string containing hex representation of a number
`UPPER()`	Convert to uppercase
`WEIGHT_STRING()`	Return the weight string for a string

String-valued functions return `NULL` if the length of the result would be greater than the value of the `max_allowed_packet` system variable. See Section 7.1.1, “Configuring the Server”.

For functions that operate on string positions, the first position is numbered 1.

For functions that take length arguments, noninteger arguments are rounded to the nearest integer.

- `ASCII(`***`str`***`)`

  Returns the numeric value of the leftmost character of the string ***`str`***. Returns `0` if ***`str`*** is the empty string. Returns `NULL` if ***`str`*** is `NULL`. `ASCII()` works for 8-bit characters.

  ```
  mysql> SELECT ASCII('2');
          -> 50
  mysql> SELECT ASCII(2);
          -> 50
  mysql> SELECT ASCII('dx');
          -> 100
  ```

  See also the `ORD()` function.

- `BIN(`***`N`***`)`

  Returns a string representation of the binary value of ***`N`***, where ***`N`*** is a longlong (`BIGINT`) number. This is equivalent to `CONV(`***`N`***`,10,2)`. Returns `NULL` if ***`N`*** is `NULL`.

  ```
  mysql> SELECT BIN(12);
          -> '1100'
  ```

- `BIT_LENGTH(`***`str`***`)`

  Returns the length of the string ***`str`*** in bits. Returns `NULL` if ***`str`*** is `NULL`.

```
mysql> SELECT BIT_LENGTH('text');
        -> 32
```

- CHAR(***N***,... [USING ***charset_name***])

  CHAR() interprets each argument ***N*** as an integer and returns a string consisting of the characters given by the code values of those integers. NULL values are skipped.

```
mysql> SELECT CHAR(77,121,83,81,'76');
+----------------------------------------------------+
| CHAR(77,121,83,81,'76')                            |
+----------------------------------------------------+
| 0x4D7953514C                                       |
+----------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CHAR(77,77.3,'77.3');
+--------------------------------------------+
| CHAR(77,77.3,'77.3')                       |
+--------------------------------------------+
| 0x4D4D4D                                   |
+--------------------------------------------+
1 row in set (0.00 sec)
```

  By default, CHAR() returns a binary string. To produce a string in a given character set, use the optional USING clause:

```
mysql> SELECT CHAR(77,121,83,81,'76' USING utf8mb4);
+----------------------------------------+
| CHAR(77,121,83,81,'76' USING utf8mb4) |
+----------------------------------------+
| MySQL                                  |
+----------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CHAR(77,77.3,'77.3' USING utf8mb4);
+-----------------------------------+
| CHAR(77,77.3,'77.3' USING utf8mb4) |
+-----------------------------------+
| MMM                               |
+-----------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------+
| Level   | Code | Message                                   |
+---------+------+-------------------------------------------+
| Warning | 1292 | Truncated incorrect INTEGER value: '77.3' |
+---------+------+-------------------------------------------+
1 row in set (0.00 sec)
```

If `USING` is given and the result string is illegal for the given character set, a warning is issued. Also, if strict SQL mode is enabled, the result from `CHAR()` becomes `NULL`.

If `CHAR()` is invoked from within the **mysql** client, binary strings display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

`CHAR()` arguments larger than 255 are converted into multiple result bytes. For example, `CHAR(256)` is equivalent to `CHAR(1,0)`, and `CHAR(256*256)` is equivalent to `CHAR(1,0,0)`:

```
mysql> SELECT HEX(CHAR(1,0)), HEX(CHAR(256));
+----------------+----------------+
| HEX(CHAR(1,0)) | HEX(CHAR(256)) |
+----------------+----------------+
| 0100           | 0100           |
+----------------+----------------+
1 row in set (0.00 sec)

mysql> SELECT HEX(CHAR(1,0,0)), HEX(CHAR(256*256));
+------------------+--------------------+
| HEX(CHAR(1,0,0)) | HEX(CHAR(256*256)) |
+------------------+--------------------+
| 010000           | 010000             |
+------------------+--------------------+
1 row in set (0.00 sec)
```

- `CHAR_LENGTH(`***`str`***`)`

Returns the length of the string ***`str`***, measured in code points. A multibyte character counts as a single code point. This means that, for a string containing two 3-byte characters, `LENGTH()` returns `6`, whereas `CHAR_LENGTH()` returns `2`, as shown here:

```
mysql> SET @dolphin:='海豚';
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT LENGTH(@dolphin), CHAR_LENGTH(@dolphin);
+------------------+-----------------------+
| LENGTH(@dolphin) | CHAR_LENGTH(@dolphin) |
+------------------+-----------------------+
|                6 |                     2 |
+------------------+-----------------------+
1 row in set (0.00 sec)
```

`CHAR_LENGTH()` returns `NULL` if ***str*** is `NULL`.

- `CHARACTER_LENGTH(`***str***`)`

  `CHARACTER_LENGTH()` is a synonym for `CHAR_LENGTH()`.

- `CONCAT(`***str1***`,`***str2***`,...)`

  Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are nonbinary strings, the result is a nonbinary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent nonbinary string form.

  `CONCAT()` returns `NULL` if any argument is `NULL`.

```
mysql> SELECT CONCAT('My', 'S', 'QL');
        -> 'MySQL'
mysql> SELECT CONCAT('My', NULL, 'QL');
        -> NULL
mysql> SELECT CONCAT(14.3);
        -> '14.3'
```

  For quoted strings, concatenation can be performed by placing the strings next to each other:

```
mysql> SELECT 'My' 'S' 'QL';
        -> 'MySQL'
```

  If `CONCAT()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

- `CONCAT_WS(`***separator***`,`***str1***`,`***str2***`,...)`

  `CONCAT_WS()` stands for Concatenate With Separator and is a special form of `CONCAT()`. The first argument is the separator for the rest of the arguments. The separator is added between the strings to be concatenated. The separator can be a string, as can the rest of the arguments. If the separator is `NULL`, the result is `NULL`.

```
mysql> SELECT CONCAT_WS(',','First name','Second name','Last Name');
        -> 'First name,Second name,Last Name'
mysql> SELECT CONCAT_WS(',','First name',NULL,'Last Name');
        -> 'First name,Last Name'
```

`CONCAT_WS()` does not skip empty strings. However, it does skip any `NULL` values after the separator argument.

- `ELT(`***`N`***`,`***`str1`***`,`***`str2`***`,`***`str3`***`,...)`

  `ELT()` returns the ***`N`***th element of the list of strings: ***`str1`*** if ***`N`*** = `1`, ***`str2`*** if ***`N`*** = `2`, and so on. Returns `NULL` if ***`N`*** is less than `1`, greater than the number of arguments, or `NULL`. `ELT()` is the complement of `FIELD()`.

```
mysql> SELECT ELT(1, 'Aa', 'Bb', 'Cc', 'Dd');
        -> 'Aa'
mysql> SELECT ELT(4, 'Aa', 'Bb', 'Cc', 'Dd');
        -> 'Dd'
```

- `EXPORT_SET(`***`bits`***`,`***`on`***`,`***`off`***`[,`***`separator`***`[,`***`number_of_bits`***`]])`

  Returns a string such that for every bit set in the value ***`bits`***, you get an ***`on`*** string and for every bit not set in the value, you get an ***`off`*** string. Bits in ***`bits`*** are examined from right to left (from low-order to high-order bits). Strings are added to the result from left to right, separated by the ***`separator`*** string (the default being the comma character `,`). The number of bits examined is given by ***`number_of_bits`***, which has a default of 64 if not specified. ***`number_of_bits`*** is silently clipped to 64 if larger than 64. It is treated as an unsigned integer, so a value of −1 is effectively the same as 64.

```
mysql> SELECT EXPORT_SET(5,'Y','N',',',4);
        -> 'Y,N,Y,N'
mysql> SELECT EXPORT_SET(6,'1','0',',',10);
        -> '0,1,1,0,0,0,0,0,0,0'
```

- `FIELD(`***`str`***`,`***`str1`***`,`***`str2`***`,`***`str3`***`,...)`

  Returns the index (position) of ***`str`*** in the ***`str1`***, ***`str2`***, ***`str3`***, `...` list. Returns `0` if ***`str`*** is not found.

  If all arguments to `FIELD()` are strings, all arguments are compared as strings. If all arguments are numbers, they are compared as numbers. Otherwise, the arguments are compared as double.

  If ***`str`*** is `NULL`, the return value is `0` because `NULL` fails equality comparison with any value. `FIELD()` is the complement of `ELT()`.

```
mysql> SELECT FIELD('Bb', 'Aa', 'Bb', 'Cc', 'Dd', 'Ff');
        -> 2
```

```
mysql> SELECT FIELD('Gg', 'Aa', 'Bb', 'Cc', 'Dd', 'Ff');
        -> 0
```

- `FIND_IN_SET(`***`str`***`,`***`strlist`***`)`

  Returns a value in the range of 1 to ***`N`*** if the string ***`str`*** is in the string list ***`strlist`*** consisting of ***`N`*** substrings. A string list is a string composed of substrings separated by `,` characters. If the first argument is a constant string and the second is a column of type `SET`, the `FIND_IN_SET()` function is optimized to use bit arithmetic. Returns `0` if ***`str`*** is not in ***`strlist`*** or if ***`strlist`*** is the empty string. Returns `NULL` if either argument is `NULL`. This function does not work properly if the first argument contains a comma (`,`) character.

```
mysql> SELECT FIND_IN_SET('b','a,b,c,d');
        -> 2
```

- `FORMAT(`***`X`***`,`***`D`***`[,`***`locale`***`])`

  Formats the number ***`X`*** to a format like `'#,###,###.##'`, rounded to ***`D`*** decimal places, and returns the result as a string. If ***`D`*** is `0`, the result has no decimal point or fractional part. If ***`X`*** or ***`D`*** is `NULL`, the function returns `NULL`.

  The optional third parameter 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 (see Section 12.16, “MySQL Server Locale Support”). If the locale is `NULL` or not specified, the default locale is `'en_US'`.

```
mysql> SELECT FORMAT(12332.123456, 4);
        -> '12,332.1235'
mysql> SELECT FORMAT(12332.1,4);
        -> '12,332.1000'
mysql> SELECT FORMAT(12332.2,0);
        -> '12,332'
mysql> SELECT FORMAT(12332.2,2,'de_DE');
        -> '12.332,20'
```

- `FROM_BASE64(`***`str`***`)`

  Takes a string encoded with the base-64 encoded rules used by `TO_BASE64()` and returns the decoded result as a binary string. The result is `NULL` if the argument is `NULL` or not a valid base-64 string. See the description of `TO_BASE64()` for details about the encoding and decoding rules.

```
mysql> SELECT TO_BASE64('abc'), FROM_BASE64(TO_BASE64('abc'));
        -> 'JWJj', 'abc'
```

If `FROM_BASE64()` is invoked from within the **mysql** client, binary strings display using hexadecimal notation. You can disable this behavior by setting the value of the `--binary-as-hex` to `0` when starting the **mysql** client. For more information about that option, see Section 6.5.1, "mysql — The MySQL Command-Line Client".

- `HEX(`***`str`***`)`, `HEX(`***`N`***`)`

  For a string argument ***`str`***, `HEX()` returns a hexadecimal string representation of ***`str`*** where each byte of each character in ***`str`*** is converted to two hexadecimal digits. (Multibyte characters therefore become more than two digits.) The inverse of this operation is performed by the `UNHEX()` function.

  For a numeric argument ***`N`***, `HEX()` returns a hexadecimal string representation of the value of ***`N`*** treated as a longlong (`BIGINT`) number. This is equivalent to `CONV(`***`N`***`,10,16)`. The inverse of this operation is performed by `CONV(HEX(`***`N`***`),16,10)`.

  For a `NULL` argument, this function returns `NULL`.

```
mysql> SELECT X'616263', HEX('abc'), UNHEX(HEX('abc'));
        -> 'abc', 616263, 'abc'
mysql> SELECT HEX(255), CONV(HEX(255),16,10);
        -> 'FF', 255
```

- `INSERT(`***`str`***`,`***`pos`***`,`***`len`***`,`***`newstr`***`)`

  Returns the string ***`str`***, with the substring beginning at position ***`pos`*** and ***`len`*** characters long replaced by the string ***`newstr`***. Returns the original string if ***`pos`*** is not within the length of the string. Replaces the rest of the string from position ***`pos`*** if ***`len`*** is not within the length of the rest of the string. Returns `NULL` if any argument is `NULL`.

```
mysql> SELECT INSERT('Quadratic', 3, 4, 'What');
        -> 'QuWhattic'
mysql> SELECT INSERT('Quadratic', -1, 4, 'What');
        -> 'Quadratic'
mysql> SELECT INSERT('Quadratic', 3, 100, 'What');
        -> 'QuWhat'
```

  This function is multibyte safe.

- `INSTR(`***`str`***`,`***`substr`***`)`

  Returns the position of the first occurrence of substring ***`substr`*** in string ***`str`***. This is the same as the two-argument form of `LOCATE()`, except that the order of the arguments is reversed.

  ```
  mysql> SELECT INSTR('foobarbar', 'bar');
          -> 4
  mysql> SELECT INSTR('xbar', 'foobar');
          -> 0
  ```

  This function is multibyte safe, and is case-sensitive only if at least one argument is a binary string. If either argument is `NULL`, this functions returns `NULL`.

- `LCASE(`***`str`***`)`

  `LCASE()` is a synonym for `LOWER()`.

  `LCASE()` used in a view is rewritten as `LOWER()` when storing the view's definition. (Bug #12844279)

- `LEFT(`***`str`***`,`***`len`***`)`

  Returns the leftmost ***`len`*** characters from the string ***`str`***, or `NULL` if any argument is `NULL`.

  ```
  mysql> SELECT LEFT('foobarbar', 5);
          -> 'fooba'
  ```

  This function is multibyte safe.

- `LENGTH(`***`str`***`)`

  Returns the length of the string ***`str`***, measured in bytes. A multibyte character counts as multiple bytes. This means that for a string containing five 2-byte characters, `LENGTH()` returns `10`, whereas `CHAR_LENGTH()` returns `5`. Returns `NULL` if ***`str`*** is `NULL`.

  ```
  mysql> SELECT LENGTH('text');
          -> 4
  ```

  > **Note**
  >
  > The `Length()` OpenGIS spatial function is named `ST_Length()` in MySQL.

- `LOAD_FILE(`***`file_name`***`)`

  Reads the file and returns the file contents as a string. To use this function, the file must be located on the server host, you must specify the full path name to the file, and you must have the `FILE` privilege. The file must be readable by the server and its size less than `max_allowed_packet` bytes. If the `secure_file_priv` system variable is set to a nonempty directory name, the file to be loaded must be located in that directory. (Prior to MySQL 8.0.17, the file must be readable by all, not just readable by the server.)

  If the file does not exist or cannot be read because one of the preceding conditions is not satisfied, the function returns `NULL`.

  The `character_set_filesystem` system variable controls interpretation of file names that are given as literal strings.

```
mysql> UPDATE t
            SET blob_col=LOAD_FILE('/tmp/picture')
            WHERE id=1;
```

- `LOCATE(`***`substr`***`,`***`str`***`)`, `LOCATE(`***`substr`***`,`***`str`***`,`***`pos`***`)`

  The first syntax returns the position of the first occurrence of substring ***`substr`*** in string ***`str`***. The second syntax returns the position of the first occurrence of substring ***`substr`*** in string ***`str`***, starting at position ***`pos`***. Returns `0` if ***`substr`*** is not in ***`str`***. Returns `NULL` if any argument is `NULL`.

```
mysql> SELECT LOCATE('bar', 'foobarbar');
        -> 4
mysql> SELECT LOCATE('xbar', 'foobar');
        -> 0
mysql> SELECT LOCATE('bar', 'foobarbar', 5);
        -> 7
```

  This function is multibyte safe, and is case-sensitive only if at least one argument is a binary string.

- `LOWER(`***`str`***`)`

  Returns the string ***`str`*** with all characters changed to lowercase according to the current character set mapping, or `NULL` if ***`str`*** is `NULL`. The default character set is `utf8mb4`.

```
mysql> SELECT LOWER('QUADRATICALLY');
        -> 'quadratically'
```

`LOWER()` (and `UPPER()`) are ineffective when applied to binary strings (`BINARY`, `VARBINARY`, `BLOB`). To perform lettercase conversion of a binary string, first convert it to a nonbinary string using a character set appropriate for the data stored in the string:

```
mysql> SET @str = BINARY 'New York';
mysql> SELECT LOWER(@str), LOWER(CONVERT(@str USING utf8mb4));
+-------------+------------------------------------+
| LOWER(@str) | LOWER(CONVERT(@str USING utf8mb4)) |
+-------------+------------------------------------+
| New York    | new york                           |
+-------------+------------------------------------+
```

For collations of Unicode character sets, `LOWER()` and `UPPER()` work according to the Unicode Collation Algorithm (UCA) version in the collation name, if there is one, and UCA 4.0.0 if no version is specified. For example, `utf8mb4_0900_ai_ci` and `utf8mb3_unicode_520_ci` work according to UCA 9.0.0 and 5.2.0, respectively, whereas `utf8mb3_unicode_ci` works according to UCA 4.0.0. See Section 12.10.1, “Unicode Character Sets”.

This function is multibyte safe.

`LCASE()` used within views is rewritten as `LOWER()`.

- `LPAD(`***`str`***`,`***`len`***`,`***`padstr`***`)`

  Returns the string ***`str`***, left-padded with the string ***`padstr`*** to a length of ***`len`*** characters. If ***`str`*** is longer than ***`len`***, the return value is shortened to ***`len`*** characters.

  ```
  mysql> SELECT LPAD('hi',4,'??');
          -> '??hi'
  mysql> SELECT LPAD('hi',1,'??');
          -> 'h'
  ```

  Returns `NULL` if any of its arguments are `NULL`.

- `LTRIM(`***`str`***`)`

  Returns the string ***`str`*** with leading space characters removed. Returns `NULL` if ***`str`*** is `NULL`.

  ```
  mysql> SELECT LTRIM('  barbar');
          -> 'barbar'
  ```

  This function is multibyte safe.

- `MAKE_SET(`***`bits`***`,`***`str1`***`,`***`str2`***`,...)`

  Returns a set value (a string containing substrings separated by `,` characters) consisting of the strings that have the corresponding bit in ***`bits`*** set. ***`str1`*** corresponds to bit 0, ***`str2`*** to bit 1, and so on. `NULL` values in ***`str1`***, ***`str2`***, `...` are not appended to the result.

```
mysql> SELECT MAKE_SET(1,'a','b','c');
        -> 'a'
mysql> SELECT MAKE_SET(1 | 4,'hello','nice','world');
        -> 'hello,world'
mysql> SELECT MAKE_SET(1 | 4,'hello','nice',NULL,'world');
        -> 'hello'
mysql> SELECT MAKE_SET(0,'a','b','c');
        -> ''
```

- `MID(`***`str`***`,`***`pos`***`,`***`len`***`)`

  `MID(`***`str`***`,`***`pos`***`,`***`len`***`)` is a synonym for `SUBSTRING(`***`str`***`,`***`pos`***`,`***`len`***`)`.

- `OCT(`***`N`***`)`

  Returns a string representation of the octal value of ***`N`***, where ***`N`*** is a longlong (`BIGINT`) number. This is equivalent to `CONV(`***`N`***`,10,8)`. Returns `NULL` if ***`N`*** is `NULL`.

```
mysql> SELECT OCT(12);
        -> '14'
```

- `OCTET_LENGTH(`***`str`***`)`

  `OCTET_LENGTH()` is a synonym for `LENGTH()`.

- `ORD(`***`str`***`)`

  If the leftmost character of the string ***`str`*** is a multibyte character, returns the code for that character, calculated from the numeric values of its constituent bytes using this formula:

```
  (1st byte code)
+ (2nd byte code * 256)
+ (3rd byte code * 256^2) ...
```

  If the leftmost character is not a multibyte character, `ORD()` returns the same value as the `ASCII()` function. The function returns `NULL` if ***`str`*** is `NULL`.

```
mysql> SELECT ORD('2');
        -> 50
```

- `POSITION(`***`substr`***` IN `***`str`***`)`

  `POSITION(`***`substr`***` IN `***`str`***`)` is a synonym for `LOCATE(`***`substr`***`,`***`str`***`)`.

- `QUOTE(`***`str`***`)`

  Quotes a string to produce a result that can be used as a properly escaped data value in an SQL statement. The string is returned enclosed by single quotation marks and with each instance of backslash (`\`), single quote (`'`), ASCII `NUL`, and Control+Z preceded by a backslash. If the argument is `NULL`, the return value is the word “NULL” without enclosing single quotation marks.

```
mysql> SELECT QUOTE('Don\'t!');
        -> 'Don\'t!'
mysql> SELECT QUOTE(NULL);
        -> NULL
```

  For comparison, see the quoting rules for literal strings and within the C API in Section 11.1.1, “String Literals”, and mysql_real_escape_string_quote().

- `REPEAT(`***`str`***`,`***`count`***`)`

  Returns a string consisting of the string ***`str`*** repeated ***`count`*** times. If ***`count`*** is less than 1, returns an empty string. Returns `NULL` if ***`str`*** or ***`count`*** is `NULL`.

```
mysql> SELECT REPEAT('MySQL', 3);
        -> 'MySQLMySQLMySQL'
```

- `REPLACE(`***`str`***`,`***`from_str`***`,`***`to_str`***`)`

  Returns the string ***`str`*** with all occurrences of the string ***`from_str`*** replaced by the string ***`to_str`***. `REPLACE()` performs a case-sensitive match when searching for ***`from_str`***.

```
mysql> SELECT REPLACE('www.mysql.com', 'w', 'Ww');
        -> 'WwWwWw.mysql.com'
```

  This function is multibyte safe. It returns `NULL` if any of its arguments are `NULL`.

- `REVERSE(`***`str`***`)`

Returns the string ***str*** with the order of the characters reversed, or `NULL` if ***str*** is `NULL`.

```
mysql> SELECT REVERSE('abc');
        -> 'cba'
```

This function is multibyte safe.

- `RIGHT(`***str*****,** ***len***`)`

  Returns the rightmost ***len*** characters from the string ***str***, or `NULL` if any argument is `NULL`.

  ```
  mysql> SELECT RIGHT('foobarbar', 4);
          -> 'rbar'
  ```

  This function is multibyte safe.

- `RPAD(`***str*****,** ***len*****,** ***padstr***`)`

  Returns the string ***str***, right-padded with the string ***padstr*** to a length of ***len*** characters. If ***str*** is longer than ***len***, the return value is shortened to ***len*** characters. If ***str***, ***padstr***, or ***len*** is `NULL`, the function returns `NULL`.

  ```
  mysql> SELECT RPAD('hi',5,'?');
          -> 'hi???'
  mysql> SELECT RPAD('hi',1,'?');
          -> 'h'
  ```

  This function is multibyte safe.

- `RTRIM(`***str***`)`

  Returns the string ***str*** with trailing space characters removed.

  ```
  mysql> SELECT RTRIM('barbar   ');
          -> 'barbar'
  ```

  This function is multibyte safe, and returns `NULL` if ***str*** is `NULL`.

- `SOUNDEX(`***str***`)`

  Returns a soundex string from ***str***, or `NULL` if ***str*** is `NULL`. Two strings that sound almost the same should have identical soundex strings. A standard soundex string is four characters long, but the `SOUNDEX()` function returns an arbitrarily long string. You can use `SUBSTRING()` on the result to get a standard soundex string. All nonalphabetic characters in ***`str`*** are ignored. All international alphabetic characters outside the A-Z range are treated as vowels.

> **Important**
>
> When using `SOUNDEX()`, you should be aware of the following limitations:

- This function, as currently implemented, is intended to work well with strings that are in the English language only. Strings in other languages may not produce reliable results.
- This function is not guaranteed to provide consistent results with strings that use multibyte character sets, including `utf-8`. See Bug #22638 for more information.

```
mysql> SELECT SOUNDEX('Hello');
        -> 'H400'
mysql> SELECT SOUNDEX('Quadratically');
        -> 'Q36324'
```

> **Note**
>
> This function implements the original Soundex algorithm, not the more popular enhanced version (also described by D. Knuth). The difference is that original version discards vowels first and duplicates second, whereas the enhanced version discards duplicates first and vowels second.

- ***`expr1`*** `SOUNDS LIKE` ***`expr2`***

  This is the same as `SOUNDEX(`***`expr1`***`) = SOUNDEX(`***`expr2`***`)`.

- `SPACE(`***`N`***`)`

  Returns a string consisting of ***`N`*** space characters, or `NULL` if ***`N`*** is `NULL`.

```
mysql> SELECT SPACE(6);
        -> '      '
```

- `SUBSTR(`***`str`***`,`***`pos`***`)`, `SUBSTR(`***`str`*** `FROM` ***`pos`***`)`, `SUBSTR(`***`str`***`,`***`pos`***`,`***`len`***`)`, `SUBSTR(`***`str`*** `FROM` ***`pos`*** `FOR` ***`len`***`)`

  `SUBSTR()` is a synonym for `SUBSTRING()`.

- `SUBSTRING(`***`str`***`,`***`pos`***`)`, `SUBSTRING(`***`str`***` FROM `***`pos`***`)`, `SUBSTRING(`***`str`***`,`***`pos`***`,`***`len`***`)`, `SUBSTRING(`***`str`***` FROM `***`pos`***` FOR `***`len`***`)`

  The forms without a ***`len`*** argument return a substring from string ***`str`*** starting at position ***`pos`***. The forms with a ***`len`*** argument return a substring ***`len`*** characters long from string ***`str`***, starting at position ***`pos`***. The forms that use `FROM` are standard SQL syntax. It is also possible to use a negative value for ***`pos`***. In this case, the beginning of the substring is ***`pos`*** characters from the end of the string, rather than the beginning. A negative value may be used for ***`pos`*** in any of the forms of this function. A value of 0 for ***`pos`*** returns an empty string.

  For all forms of `SUBSTRING()`, the position of the first character in the string from which the substring is to be extracted is reckoned as `1`.

```
mysql> SELECT SUBSTRING('Quadratically',5);
        -> 'ratically'
mysql> SELECT SUBSTRING('foobarbar' FROM 4);
        -> 'barbar'
mysql> SELECT SUBSTRING('Quadratically',5,6);
        -> 'ratica'
mysql> SELECT SUBSTRING('Sakila', -3);
        -> 'ila'
mysql> SELECT SUBSTRING('Sakila', -5, 3);
        -> 'aki'
mysql> SELECT SUBSTRING('Sakila' FROM -4 FOR 2);
        -> 'ki'
```

  This function is multibyte safe. It returns `NULL` if any of its arguments are `NULL`.

  If ***`len`*** is less than 1, the result is the empty string.

- `SUBSTRING_INDEX(`***`str`***`,`***`delim`***`,`***`count`***`)`

  Returns the substring from string ***`str`*** before ***`count`*** occurrences of the delimiter ***`delim`***. If ***`count`*** is positive, everything to the left of the final delimiter (counting from the left) is returned. If ***`count`*** is negative, everything to the right of the final delimiter (counting from the right) is returned. `SUBSTRING_INDEX()` performs a case-sensitive match when searching for ***`delim`***.

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
        -> 'www.mysql'
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
        -> 'mysql.com'
```

  This function is multibyte safe.

`SUBSTRING_INDEX()` returns `NULL` if any of its arguments are `NULL`.

- `TO_BASE64(`***`str`***`)`

  Converts the string argument to base-64 encoded form and returns the result as a character string with the connection character set and collation. If the argument is not a string, it is converted to a string before conversion takes place. The result is `NULL` if the argument is `NULL`. Base-64 encoded strings can be decoded using the `FROM_BASE64()` function.

```
mysql> SELECT TO_BASE64('abc'), FROM_BASE64(TO_BASE64('abc'));
        -> 'JWJj', 'abc'
```

  Different base-64 encoding schemes exist. These are the encoding and decoding rules used by `TO_BASE64()` and `FROM_BASE64()`:

  - The encoding for alphabet value 62 is `'+'`.
  - The encoding for alphabet value 63 is `'/'`.
  - Encoded output consists of groups of 4 printable characters. Each 3 bytes of the input data are encoded using 4 characters. If the last group is incomplete, it is padded with `'='` characters to a length of 4.
  - A newline is added after each 76 characters of encoded output to divide long output into multiple lines.
  - Decoding recognizes and ignores newline, carriage return, tab, and space.

- `TRIM([{BOTH | LEADING | TRAILING} [`***`remstr`***`] FROM] `***`str`***`)`, `TRIM([`***`remstr`***` FROM] `***`str`***`)`

  Returns the string ***`str`*** with all ***`remstr`*** prefixes or suffixes removed. If none of the specifiers `BOTH`, `LEADING`, or `TRAILING` is given, `BOTH` is assumed. ***`remstr`*** is optional and, if not specified, spaces are removed.

```
mysql> SELECT TRIM('  bar   ');
        -> 'bar'
mysql> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');
        -> 'barxxx'
mysql> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');
        -> 'bar'
mysql> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');
        -> 'barx'
```

  This function is multibyte safe. It returns `NULL` if any of its arguments are `NULL`.

- `UCASE(`***`str`***`)`

  `UCASE()` is a synonym for `UPPER()`.

  `UCASE()` used within views is rewritten as `UPPER()`.

- `UNHEX(`***`str`***`)`

  For a string argument ***`str`***, `UNHEX(`***`str`***`)` interprets each pair of characters in the argument as a hexadecimal number and converts it to the byte represented by the number. The return value is a binary string.

```
mysql> SELECT UNHEX('4D7953514C');
        -> 'MySQL'
mysql> SELECT X'4D7953514C';
        -> 'MySQL'
mysql> SELECT UNHEX(HEX('string'));
        -> 'string'
mysql> SELECT HEX(UNHEX('1267'));
        -> '1267'
```

  The characters in the argument string must be legal hexadecimal digits: `'0'` .. `'9'`, `'A'` .. `'F'`, `'a'` .. `'f'`. If the argument contains any nonhexadecimal digits, or is itself `NULL`, the result is `NULL`:

```
mysql> SELECT UNHEX('GG');
+-------------+
| UNHEX('GG') |
+-------------+
| NULL        |
+-------------+

mysql> SELECT UNHEX(NULL);
+-------------+
| UNHEX(NULL) |
+-------------+
| NULL        |
+-------------+
```

  A `NULL` result can also occur if the argument to `UNHEX()` is a `BINARY` column, because values are padded with `0x00` bytes when stored but those bytes are not stripped on retrieval. For example, `'41'` is stored into a `CHAR(3)` column as `'41 '` and retrieved as `'41'` (with the trailing pad space stripped), so `UNHEX()` for the column value returns `X'41'`. By contrast, `'41'` is stored into a `BINARY(3)` column as `'41\0'` and retrieved as `'41\0'` (with the trailing pad `0x00` byte not stripped). `'\0'` is not a legal hexadecimal digit, so `UNHEX()` for the column value returns `NULL`.

For a numeric argument ***N***, the inverse of `HEX(N)` is not performed by `UNHEX()`. Use `CONV(HEX(N),16,10)` instead. See the description of `HEX()`.

If `UNHEX()` is invoked from within the **mysql** client, binary strings display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

- `UPPER(str)`

  Returns the string ***str*** with all characters changed to uppercase according to the current character set mapping, or `NULL` if ***str*** is `NULL`. The default character set is `utf8mb4`.

```
mysql> SELECT UPPER('Hej');
        -> 'HEJ'
```

  See the description of `LOWER()` for information that also applies to `UPPER()`. This included information about how to perform lettercase conversion of binary strings (`BINARY`, `VARBINARY`, `BLOB`) for which these functions are ineffective, and information about case folding for Unicode character sets.

  This function is multibyte safe.

  `UCASE()` used within views is rewritten as `UPPER()`.

- `WEIGHT_STRING(str [AS {CHAR|BINARY}(N)] [flags])`

  This function returns the weight string for the input string. The return value is a binary string that represents the comparison and sorting value of the string, or `NULL` if the argument is `NULL`. It has these properties:

  - If `WEIGHT_STRING(str1)` = `WEIGHT_STRING(str2)`, then `str1 = str2` (***str1*** and ***str2*** are considered equal)

  - If `WEIGHT_STRING(str1)` < `WEIGHT_STRING(str2)`, then `str1 < str2` (***str1*** sorts before ***str2***)

  `WEIGHT_STRING()` is a debugging function intended for internal use. Its behavior can change without notice between MySQL versions. It can be used for testing and debugging of collations, especially if you are adding a new collation. See Section 12.14, “Adding a Collation to a Character Set”.

  This list briefly summarizes the arguments. More details are given in the discussion following the list.

- ***`str`***: The input string expression.
- `AS` clause: Optional; cast the input string to a given type and length.
- ***`flags`***: Optional; unused.

The input string, ***`str`***, is a string expression. If the input is a nonbinary (character) string such as a `CHAR`, `VARCHAR`, or `TEXT` value, the return value contains the collation weights for the string. If the input is a binary (byte) string such as a `BINARY`, `VARBINARY`, or `BLOB` value, the return value is the same as the input (the weight for each byte in a binary string is the byte value). If the input is `NULL`, `WEIGHT_STRING()` returns `NULL`.

Examples:

```
mysql> SET @s = _utf8mb4 'AB' COLLATE utf8mb4_0900_ai_ci;
mysql> SELECT @s, HEX(@s), HEX(WEIGHT_STRING(@s));
+------+---------+------------------------+
| @s   | HEX(@s) | HEX(WEIGHT_STRING(@s)) |
+------+---------+------------------------+
| AB   | 4142    | 1C471C60               |
+------+---------+------------------------+
```

```
mysql> SET @s = _utf8mb4 'ab' COLLATE utf8mb4_0900_ai_ci;
mysql> SELECT @s, HEX(@s), HEX(WEIGHT_STRING(@s));
+------+---------+------------------------+
| @s   | HEX(@s) | HEX(WEIGHT_STRING(@s)) |
+------+---------+------------------------+
| ab   | 6162    | 1C471C60               |
+------+---------+------------------------+
```

```
mysql> SET @s = CAST('AB' AS BINARY);
mysql> SELECT @s, HEX(@s), HEX(WEIGHT_STRING(@s));
+------+---------+------------------------+
| @s   | HEX(@s) | HEX(WEIGHT_STRING(@s)) |
+------+---------+------------------------+
| AB   | 4142    | 4142                   |
+------+---------+------------------------+
```

```
mysql> SET @s = CAST('ab' AS BINARY);
mysql> SELECT @s, HEX(@s), HEX(WEIGHT_STRING(@s));
+------+---------+------------------------+
| @s   | HEX(@s) | HEX(WEIGHT_STRING(@s)) |
+------+---------+------------------------+
| ab   | 6162    | 6162                  |
+------+---------+-----------------------+
```

The preceding examples use `HEX()` to display the `WEIGHT_STRING()` result. Because the result is a binary value, `HEX()` can be especially useful when the result contains nonprinting values, to display it in printable form:

```
mysql> SET @s = CONVERT(X'C39F' USING utf8mb4) COLLATE utf8mb4_czech_ci;
mysql> SELECT HEX(WEIGHT_STRING(@s));
+------------------------+
| HEX(WEIGHT_STRING(@s)) |
+------------------------+
| 0FEA0FEA               |
+------------------------+
```

For non-`NULL` return values, the data type of the value is `VARBINARY` if its length is within the maximum length for `VARBINARY`, otherwise the data type is `BLOB`.

The `AS` clause may be given to cast the input string to a nonbinary or binary string and to force it to a given length:

- `AS CHAR(`***`N`***`)` casts the string to a nonbinary string and pads it on the right with spaces to a length of ***`N`*** characters. ***`N`*** must be at least 1. If ***`N`*** is less than the length of the input string, the string is truncated to ***`N`*** characters. No warning occurs for truncation.

- `AS BINARY(`***`N`***`)` is similar but casts the string to a binary string, ***`N`*** is measured in bytes (not characters), and padding uses `0x00` bytes (not spaces).

```
mysql> SET NAMES 'latin1';
mysql> SELECT HEX(WEIGHT_STRING('ab' AS CHAR(4)));
+-------------------------------------+
| HEX(WEIGHT_STRING('ab' AS CHAR(4))) |
+-------------------------------------+
| 41422020                            |
+-------------------------------------+
mysql> SET NAMES 'utf8mb4';
mysql> SELECT HEX(WEIGHT_STRING('ab' AS CHAR(4)));
+-------------------------------------+
| HEX(WEIGHT_STRING('ab' AS CHAR(4))) |
+-------------------------------------+
| 1C471C60                            |
+-------------------------------------+
```

```
mysql> SELECT HEX(WEIGHT_STRING('ab' AS BINARY(4)));
+---------------------------------------+
| HEX(WEIGHT_STRING('ab' AS BINARY(4))) |
+---------------------------------------+
| 61620000                              |
+---------------------------------------+
```

The ***`flags`*** clause currently is unused.

If `WEIGHT_STRING()` is invoked from within the **mysql** client, binary strings display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

14.19.1 Aggregate Function Descriptions

This section describes aggregate functions that operate on sets of values. They are often used with a `GROUP BY` clause to group values into subsets.

Table 14.29 Aggregate Functions

Name	Description
`AVG()`	Return the average value of the argument
`BIT_AND()`	Return bitwise AND
`BIT_OR()`	Return bitwise OR
`BIT_XOR()`	Return bitwise XOR
`COUNT()`	Return a count of the number of rows returned
`COUNT(DISTINCT)`	Return the count of a number of different values
`GROUP_CONCAT()`	Return a concatenated string
`JSON_ARRAYAGG()`	Return result set as a single JSON array
`JSON_OBJECTAGG()`	Return result set as a single JSON object
`MAX()`	Return the maximum value
`MIN()`	Return the minimum value
`STD()`	Return the population standard deviation
`STDDEV()`	Return the population standard deviation
`STDDEV_POP()`	Return the population standard deviation
`STDDEV_SAMP()`	Return the sample standard deviation
`SUM()`	Return the sum
`VAR_POP()`	Return the population standard variance
`VAR_SAMP()`	Return the sample variance
`VARIANCE()`	Return the population standard variance

Unless otherwise stated, aggregate functions ignore `NULL` values.

If you use an aggregate function in a statement containing no `GROUP BY` clause, it is equivalent to grouping on all rows. For more information, see Section 14.19.3, “MySQL Handling of GROUP BY”.

Most aggregate functions can be used as window functions. Those that can be used this way are signified in their syntax description by `[over_clause]`, representing an optional `OVER` clause.

over_clause is described in Section 14.20.2, “Window Function Concepts and Syntax”, which also includes other information about window function usage.

For numeric arguments, the variance and standard deviation functions return a `DOUBLE` value. The `SUM()` and `AVG()` functions return a `DECIMAL` value for exact-value arguments (integer or `DECIMAL`), and a `DOUBLE` value for approximate-value arguments (`FLOAT` or `DOUBLE`).

The `SUM()` and `AVG()` aggregate functions do not work with temporal values. (They convert the values to numbers, losing everything after the first nonnumeric character.) To work around this problem, convert to numeric units, perform the aggregate operation, and convert back to a temporal value. Examples:

```
SELECT SEC_TO_TIME(SUM(TIME_TO_SEC(time_col))) FROM tbl_name;
SELECT FROM_DAYS(SUM(TO_DAYS(date_col))) FROM tbl_name;
```

Functions such as `SUM()` or `AVG()` that expect a numeric argument cast the argument to a number if necessary. For `SET` or `ENUM` values, the cast operation causes the underlying numeric value to be used.

The `BIT_AND()`, `BIT_OR()`, and `BIT_XOR()` aggregate functions perform bit operations. Prior to MySQL 8.0, bit functions and operators required `BIGINT` (64-bit integer) arguments and returned `BIGINT` values, so they had a maximum range of 64 bits. Non-`BIGINT` arguments were converted to `BIGINT` prior to performing the operation and truncation could occur.

In MySQL 8.0, bit functions and operators permit binary string type arguments (`BINARY`, `VARBINARY`, and the `BLOB` types) and return a value of like type, which enables them to take arguments and produce return values larger than 64 bits. For discussion about argument evaluation and result types for bit operations, see the introductory discussion in Section 14.12, “Bit Functions and Operators”.

- `AVG([DISTINCT] `***`expr`***`) [`***`over_clause`***`]`

  Returns the average value of ***expr***. The `DISTINCT` option can be used to return the average of the distinct values of ***expr***.

  If there are no matching rows, `AVG()` returns `NULL`. The function also returns `NULL` if ***expr*** is `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

```
mysql> SELECT student_name, AVG(test_score)
       FROM student
       GROUP BY student_name;
```

- `BIT_AND(`***`expr`***`) [`***`over_clause`***`]`

  Returns the bitwise `AND` of all bits in ***`expr`***.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.

  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes, an `ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE` error occurs. Numeric evaluation produces an unsigned 64-bit integer.

  If there are no matching rows, `BIT_AND()` returns a neutral value (all bits set to 1) having the same length as the argument values.

  `NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

  For more information discussion about argument evaluation and result types, see the introductory discussion in Section 14.12, “Bit Functions and Operators”.

  If `BIT_AND()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

  As of MySQL 8.0.12, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

- `BIT_OR(`***`expr`***`) [`***`over_clause`***`]`

  Returns the bitwise `OR` of all bits in ***`expr`***.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.

  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes, an `ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE` error occurs. Numeric evaluation produces an unsigned 64-bit integer.

  If there are no matching rows, `BIT_OR()` returns a neutral value (all bits set to 0) having the same length as the argument values.

  `NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

  For more information discussion about argument evaluation and result types, see the introductory discussion in Section 14.12, “Bit Functions and Operators”.

  If `BIT_OR()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

  As of MySQL 8.0.12, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

- `BIT_XOR(`***`expr`***`) [`***`over_clause`***`]`

  Returns the bitwise `XOR` of all bits in ***`expr`***.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.

  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes, an `ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE` error occurs. Numeric evaluation produces an unsigned 64-bit integer.

  If there are no matching rows, `BIT_XOR()` returns a neutral value (all bits set to 0) having the same length as the argument values.

  `NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

For more information discussion about argument evaluation and result types, see the introductory discussion in Section 14.12, “Bit Functions and Operators”.

If `BIT_XOR()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

As of MySQL 8.0.12, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

- `COUNT(`***`expr`***`) [`***`over_clause`***`]`

  Returns a count of the number of non-`NULL` values of ***`expr`*** in the rows retrieved by a `SELECT` statement. The result is a `BIGINT` value.

  If there are no matching rows, `COUNT()` returns `0`. `COUNT(NULL)` returns 0.

  This function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT student.student_name,COUNT(*)
       FROM student,course
       WHERE student.student_id=course.student_id
       GROUP BY student_name;
```

  `COUNT(*)` is somewhat different in that it returns a count of the number of rows retrieved, whether or not they contain `NULL` values.

  For transactional storage engines such as `InnoDB`, storing an exact row count is problematic. Multiple transactions may be occurring at the same time, each of which may affect the count.

  `InnoDB` does not keep an internal count of rows in a table because concurrent transactions might “see” different numbers of rows at the same time. Consequently, `SELECT COUNT(*)` statements only count rows visible to the current transaction.

  As of MySQL 8.0.13, `SELECT COUNT(*) FROM` ***`tbl_name`*** query performance for `InnoDB` tables is optimized for single-threaded workloads if there are no extra clauses such as `WHERE` or `GROUP BY`.

  `InnoDB` processes `SELECT COUNT(*)` statements by traversing the smallest available secondary index unless an index or optimizer hint directs the optimizer to use a different index. If a secondary index is not present, `InnoDB` processes `SELECT COUNT(*)` statements by scanning the clustered index.

Processing `SELECT COUNT(*)` statements takes some time if index records are not entirely in the buffer pool. For a faster count, create a counter table and let your application update it according to the inserts and deletes it does. However, this method may not scale well in situations where thousands of concurrent transactions are initiating updates to the same counter table. If an approximate row count is sufficient, use `SHOW TABLE STATUS`.

`InnoDB` handles `SELECT COUNT(*)` and `SELECT COUNT(1)` operations in the same way. There is no performance difference.

For `MyISAM` tables, `COUNT(*)` is optimized to return very quickly if the `SELECT` retrieves from one table, no other columns are retrieved, and there is no `WHERE` clause. For example:

```
mysql> SELECT COUNT(*) FROM student;
```

This optimization only applies to `MyISAM` tables, because an exact row count is stored for this storage engine and can be accessed very quickly. `COUNT(1)` is only subject to the same optimization if the first column is defined as `NOT NULL`.

- `COUNT(DISTINCT` ***`expr`*****`,`**`[`***`expr`***`...])`

  Returns a count of the number of rows with different non-`NULL` ***`expr`*** values.

  If there are no matching rows, `COUNT(DISTINCT)` returns `0`.

```
mysql> SELECT COUNT(DISTINCT results) FROM student;
```

  In MySQL, you can obtain the number of distinct expression combinations that do not contain `NULL` by giving a list of expressions. In standard SQL, you would have to do a concatenation of all expressions inside `COUNT(DISTINCT ...)`.

- `GROUP_CONCAT(`***`expr`***`)`

  This function returns a string result with the concatenated non-`NULL` values from a group. It returns `NULL` if there are no non-`NULL` values. The full syntax is as follows:

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
             [ORDER BY {unsigned_integer | col_name | expr}
                 [ASC | DESC] [,col_name ...]]
             [SEPARATOR str_val])
```

```
mysql> SELECT student_name,
         GROUP_CONCAT(test_score)
       FROM student
       GROUP BY student_name;
```

Or:

```
mysql> SELECT student_name,
         GROUP_CONCAT(DISTINCT test_score
                      ORDER BY test_score DESC SEPARATOR ' ')
       FROM student
       GROUP BY student_name;
```

In MySQL, you can get the concatenated values of expression combinations. To eliminate duplicate values, use the `DISTINCT` clause. To sort values in the result, use the `ORDER BY` clause. To sort in reverse order, add the `DESC` (descending) keyword to the name of the column you are sorting by in the `ORDER BY` clause. The default is ascending order; this may be specified explicitly using the `ASC` keyword. The default separator between values in a group is comma (`,`). To specify a separator explicitly, use `SEPARATOR` followed by the string literal value that should be inserted between group values. To eliminate the separator altogether, specify `SEPARATOR ''`.

The result is truncated to the maximum length that is given by the `group_concat_max_len` system variable, which has a default value of 1024. The value can be set higher, although the effective maximum length of the return value is constrained by the value of `max_allowed_packet`. The syntax to change the value of `group_concat_max_len` at runtime is as follows, where ***`val`*** is an unsigned integer:

```
SET [GLOBAL | SESSION] group_concat_max_len = val;
```

The return value is a nonbinary or binary string, depending on whether the arguments are nonbinary or binary strings. The result type is `TEXT` or `BLOB` unless `group_concat_max_len` is less than or equal to 512, in which case the result type is `VARCHAR` or `VARBINARY`.

If `GROUP_CONCAT()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 6.5.1, “mysql — The MySQL Command-Line Client”.

See also `CONCAT()` and `CONCAT_WS()`: Section 14.8, “String Functions and Operators”.

- `JSON_ARRAYAGG(`***`col_or_expr`***`) [`***`over_clause`***`]`

Aggregates a result set as a single `JSON` array whose elements consist of the rows. The order of elements in this array is undefined. The function acts on a column or an expression that evaluates to a single value. Returns `NULL` if the result contains no rows, or in the event of an error. If ***`col_or_expr`*** is `NULL`, the function returns an array of JSON `[null]` elements.

As of MySQL 8.0.14, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT o_id, attribute, value FROM t3;
+------+-----------+-------+
| o_id | attribute | value |
+------+-----------+-------+
|    2 | color     | red   |
|    2 | fabric    | silk  |
|    3 | color     | green |
|    3 | shape     | square|
+------+-----------+-------+
4 rows in set (0.00 sec)

mysql> SELECT o_id, JSON_ARRAYAGG(attribute) AS attributes
    -> FROM t3 GROUP BY o_id;
+------+---------------------+
| o_id | attributes          |
+------+---------------------+
|    2 | ["color", "fabric"] |
|    3 | ["color", "shape"]  |
+------+---------------------+
2 rows in set (0.00 sec)
```

- `JSON_OBJECTAGG(`***`key, value`***`) [`***`over_clause`***`]`

Takes two column names or expressions as arguments, the first of these being used as a key and the second as a value, and returns a JSON object containing key-value pairs. Returns `NULL` if the result contains no rows, or in the event of an error. An error occurs if any key name is `NULL` or the number of arguments is not equal to 2.

As of MySQL 8.0.14, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT o_id, attribute, value FROM t3;
+------+-----------+-------+
| o_id | attribute | value |
+------+-----------+-------+
|    2 | color     | red   |
|    2 | fabric    | silk  |
|    3 | color     | green |
|    3 | shape     | square|
+------+----------+-------+
4 rows in set (0.00 sec)

mysql> SELECT o_id, JSON_OBJECTAGG(attribute, value)
    -> FROM t3 GROUP BY o_id;
+------+---------------------------------------+
| o_id | JSON_OBJECTAGG(attribute, value)      |
+------+---------------------------------------+
|    2 | {"color": "red", "fabric": "silk"}    |
|    3 | {"color": "green", "shape": "square"} |
+------+---------------------------------------+
2 rows in set (0.00 sec)
```

Duplicate key handling. When the result of this function is normalized, values having duplicate keys are discarded. In keeping with the MySQL `JSON` data type specification that does not permit duplicate keys, only the last value encountered is used with that key in the returned object (“last duplicate key wins”). This means that the result of using this function on columns from a `SELECT` can depend on the order in which the rows are returned, which is not guaranteed.

When used as a window function, if there are duplicate keys within a frame, only the last value for the key is present in the result. The value for the key from the last row in the frame is deterministic if the `ORDER BY` specification guarantees that the values have a specific order. If not, the resulting value of the key is nondeterministic.

Consider the following:

```
mysql> CREATE TABLE t(c VARCHAR(10), i INT);
Query OK, 0 rows affected (0.33 sec)

mysql> INSERT INTO t VALUES ('key', 3), ('key', 4), ('key', 5);
Query OK, 3 rows affected (0.10 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT c, i FROM t;
+------+------+
| c    | i    |
+------+------+
| key  |    3 |
| key  |    4 |
| key  |    5 |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT JSON_OBJECTAGG(c, i) FROM t;
+----------------------+
| JSON_OBJECTAGG(c, i) |
+----------------------+
| {"key": 5}           |
+----------------------+
1 row in set (0.00 sec)

mysql> DELETE FROM t;
Query OK, 3 rows affected (0.08 sec)

mysql> INSERT INTO t VALUES ('key', 3), ('key', 5), ('key', 4);
Query OK, 3 rows affected (0.06 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT c, i FROM t;
+------+------+
| c    | i    |
+------+------+
| key  |    3 |
| key  |    5 |
| key  |    4 |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT JSON_OBJECTAGG(c, i) FROM t;
+----------------------+
| JSON_OBJECTAGG(c, i) |
+----------------------+
| {"key": 4}           |
+----------------------+
1 row in set (0.00 sec)
```

The key chosen from the last query is nondeterministic. If the query does not use `GROUP BY` (which usually imposes its own ordering regardless) and you prefer a particular key ordering, you can invoke `JSON_OBJECTAGG()` as a window function by including an `OVER` clause with an `ORDER BY` specification to impose a particular order on frame rows. The following examples show what happens with and without `ORDER BY` for a few different frame specifications.

Without `ORDER BY`, the frame is the entire partition:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER () AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 4}  |
| {"key": 4}  |
| {"key": 4}  |
+-------------+
```

With `ORDER BY`, where the frame is the default of `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` (in both ascending and descending order):

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i) AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 3}  |
| {"key": 4}  |
| {"key": 5}  |
+-------------+
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i DESC) AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
| {"key": 4}  |
| {"key": 3}  |
+-------------+
```

With `ORDER BY` and an explicit frame of the entire partition:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
        AS json_object
       FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
| {"key": 5}  |
| {"key": 5}  |
+-------------+
```

To return a particular key value (such as the smallest or largest), include a `LIMIT` clause in the appropriate query. For example:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i) AS json_object FROM t LIMIT 1;
+-------------+
| json_object |
+-------------+
| {"key": 3}  |
+-------------+
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i DESC) AS json_object FROM t LIMIT 1;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
+-------------+
```

See Normalization, Merging, and Autowrapping of JSON Values, for additional information and examples.

- `MAX([DISTINCT] expr) [over_clause]`

  Returns the maximum value of ***expr***. `MAX()` may take a string argument; in such cases, it returns the maximum string value. See Section 10.3.1, “How MySQL Uses Indexes”. The `DISTINCT` keyword can be used to find the maximum of the distinct values of ***expr***, however, this produces the same result as omitting `DISTINCT`.

  If there are no matching rows, or if ***expr*** is `NULL`, `MAX()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

```
mysql> SELECT student_name, MIN(test_score), MAX(test_score)
       FROM student
       GROUP BY student_name;
```

  For `MAX()`, MySQL currently compares `ENUM` and `SET` columns by their string value rather than by the string's relative position in the set. This differs from how `ORDER BY` compares them.

- `MIN([DISTINCT] expr) [over_clause]`

  Returns the minimum value of ***expr***. `MIN()` may take a string argument; in such cases, it returns the minimum string value. See Section 10.3.1, “How MySQL Uses Indexes”. The `DISTINCT` keyword can be used to find the minimum of the distinct values of ***expr***, however, this produces the same result as omitting `DISTINCT`.

  If there are no matching rows, or if ***expr*** is `NULL`, `MIN()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

```
mysql> SELECT student_name, MIN(test_score), MAX(test_score)
       FROM student
       GROUP BY student_name;
```

For `MIN()`, MySQL currently compares `ENUM` and `SET` columns by their string value rather than by the string's relative position in the set. This differs from how `ORDER BY` compares them.

- `STD(`***`expr`***`) [`***`over_clause`***`]`

  Returns the population standard deviation of ***`expr`***. `STD()` is a synonym for the standard SQL function `STDDEV_POP()`, provided as a MySQL extension.

  If there are no matching rows, or if ***`expr`*** is `NULL`, `STD()` returns `NULL`.

  This function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, "Window Function Concepts and Syntax".

- `STDDEV(`***`expr`***`) [`***`over_clause`***`]`

  Returns the population standard deviation of ***`expr`***. `STDDEV()` is a synonym for the standard SQL function `STDDEV_POP()`, provided for compatibility with Oracle.

  If there are no matching rows, or if ***`expr`*** is `NULL`, `STDDEV()` returns `NULL`.

  This function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, "Window Function Concepts and Syntax".

- `STDDEV_POP(`***`expr`***`) [`***`over_clause`***`]`

  Returns the population standard deviation of ***`expr`*** (the square root of `VAR_POP()`). You can also use `STD()` or `STDDEV()`, which are equivalent but not standard SQL.

  If there are no matching rows, or if ***`expr`*** is `NULL`, `STDDEV_POP()` returns `NULL`.

  This function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, "Window Function Concepts and Syntax".

- `STDDEV_SAMP(`***`expr`***`) [`***`over_clause`***`]`

  Returns the sample standard deviation of ***`expr`*** (the square root of `VAR_SAMP()`.

  If there are no matching rows, or if ***`expr`*** is `NULL`, `STDDEV_SAMP()` returns `NULL`.

  This function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 14.20.2, "Window Function Concepts and Syntax".

- `SUM([DISTINCT] `***`expr`***`) [`***`over_clause`***`]`

  Returns the sum of ***`expr`***. If the return set has no rows, `SUM()` returns `NULL`. The `DISTINCT` keyword can be used to sum only the distinct values of ***`expr`***.

If there are no matching rows, or if ***expr*** is `NULL`, `SUM()` returns `NULL`.

This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

- `VAR_POP(`***expr***`) [`***over_clause***`]`

  Returns the population standard variance of ***expr***. It considers rows as the whole population, not as a sample, so it has the number of rows as the denominator. You can also use `VARIANCE()`, which is equivalent but is not standard SQL.

  If there are no matching rows, or if ***expr*** is `NULL`, `VAR_POP()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

- `VAR_SAMP(`***expr***`) [`***over_clause***`]`

  Returns the sample variance of ***expr***. That is, the denominator is the number of rows minus one.

  If there are no matching rows, or if ***expr*** is `NULL`, `VAR_SAMP()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.

- `VARIANCE(`***expr***`) [`***over_clause***`]`

  Returns the population standard variance of ***expr***. `VARIANCE()` is a synonym for the standard SQL function `VAR_POP()`, provided as a MySQL extension.

  If there are no matching rows, or if ***expr*** is `NULL`, `VARIANCE()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 14.20.2, “Window Function Concepts and Syntax”.