MariaDB starting with 10.4.3

Bitemporal tables are tables that use versioning both at the system and application-time period levels.

Contents

Using Bitemporal Tables

To create a bitemporal table, use:

```
CREATE TABLE test.t3 (
   date_1 DATE,
   date_2 DATE,
   row_start TIMESTAMP(6) AS ROW START INVISIBLE,
   row_end TIMESTAMP(6) AS ROW END INVISIBLE,
   PERIOD FOR application_time(date_1, date_2),
   PERIOD FOR system_time(row_start, row_end))
WITH SYSTEM VERSIONING;
```

Note that, while `system_time` here is also a time period, it cannot be used in `DELETE FOR PORTION` or `UPDATE FOR PORTION` statements.

```
DELETE FROM test.t3
FOR PORTION OF system_time
    FROM '2000-01-01' TO '2018-01-01';
ERROR 42000: You have an error in your SQL syntax; check the manual that corresponds
  to your MariaDB server version for the right syntax to use near
  'of system_time from '2000-01-01' to '2018-01-01'' at line 1
```

1.2 Built-in Functions

Functions and procedures in MariaDB

Function and Operator Reference

A complete list of MariaDB functions and operators in alphabetical order.

String Functions

Built-In functions for the handling of strings and columns containing string values.

Date & Time Functions

Built-In functions for the handling of dates and times.

Aggregate Functions

Aggregate functions used with GROUP BY clauses.

Numeric Functions

Numeric and arithmetic related functions in MariaDB.

Control Flow Functions

Built-In functions for assessing data to determine what results to return.

Pseudo Columns

MariaDB has pseudo columns that can be used for different purposes.

Secondary Functions

Bit Functions and Operators

Operators for comparison and setting of values, and related functions.

Encryption, Hashing and Compression Functions

Functions used for encryption, hashing and compression.

Information Functions

Functions which return information on the server, the user, or a given query.

Miscellaneous Functions

Functions for very singular and specific needs.

Special Functions

Dynamic Columns Functions

Functions for storing key/value pairs of data within a column.

Galera Functions

Built-in functions related to Galera.

Geographic Functions

Geographic, as well as geometric functions.

JSON Functions

Built-in functions related to JSON.

SEQUENCE Functions

Functions that can be used on SEQUENCEs.

Spider Functions

User-defined functions available with the Spider storage engine.

Window Functions

Window functions for performing calculations on a set of rows related to the current row.

There are 11 related questions.

1.2.1 Function and Operator Reference

Name	Description	Added
+	Addition operator	
/	Division operator	
*	Multiplication operator	
%	Modulo operator. Returns the remainder of N divided by M	
-	Subtraction operator	
!=	Not equals	
<	Less than	
<=	Less than or equal	
<=>	NULL-safe equal	
=	Equal	
>	Greater than	
>=	Greater than or equal	
&	Bitwise AND	
<<	Shift left	
>>	Shift right	
^	Bitwise XOR	
!	Logical NOT	
&&	Logical AND	

XOR	Logical XOR	
||	Logical OR	
|	Bitwise OR	
:=	Assignment operator	
=	Assignment and comparison operator	
~	Bitwise NOT	
ABS	Returns an absolute value	
ACOS	Returns an arc cosine	
ADD_MONTHS	Add months to a date	
ADDDATE	Add days or another interval to a date	
ADDTIME	Adds a time to a time or datetime	
AES_DECRYPT	Decryption data encrypted with AES_ENCRYPT	
AES_ENCRYPT	Encrypts a string with the AES algorithm	
AREA	Synonym for ST_AREA	
AsBinary	Synonym for ST_AsBinary	
ASCII	Numeric ASCII value of leftmost character	
ASIN	Returns the arc sine	
AsText	Synonym for ST_AsText	
AsWKB	Synonym for ST_AsBinary	
AsWKT	Synonym for ST_AsText	
ATAN	Returns the arc tangent	
ATAN2	Returns the arc tangent of two variables	
AVG	Returns the average value	
BENCHMARK	Executes an expression repeatedly	
BETWEEN AND	True if expression between two values	
BIN	Returns binary value	
BINARY OPERATOR	Casts to a binary string	
BINLOG_GTID_POS	Returns a string representation of the corresponding GTID position	
BIT_AND	Bitwise AND	
BIT_COUNT	Returns the number of set bits	
BIT_LENGTH	Returns the length of a string in bits	
BIT_OR	Bitwise OR	
BIT_XOR	Bitwise XOR	
BOUNDARY	Synonym for ST_BOUNDARY	
BUFFER	Synonym for ST_BUFFER	
CASE	Returns the result where value=compare_value or for the first condition that is true	
CAST	Casts a value of one type to another type	
CEIL	Synonym for CEILING()	

CEILING	Returns the smallest integer not less than X	
CENTROID	Synonym for ST_CENTROID	
CHAR Function	Returns string based on the integer values for the individual characters	
CHARACTER_LENGTH	Synonym for CHAR_LENGTH()	
CHAR_LENGTH	Length of the string in characters	
CHARSET	Returns the character set	
CHR	Returns a string consisting of the character given by the code values of the integer	
COALESCE	Returns the first non-NULL parameter	
COERCIBILITY	Returns the collation coercibility value	
COLLATION	Collation of the string argument	
COLUMN_ADD	Adds or updates dynamic columns	
COLUMN_CHECK	Checks if a dynamic column blob is valid	
COLUMN_CREATE	Returns a dynamic columns blob	
COLUMN_DELETE	Deletes a dynamic column	
COLUMN_EXISTS	Checks is a column exists	
COLUMN_GET	Gets a dynamic column value by name	
COLUMN_JSON	Returns a JSON representation of dynamic column blob data	
COLUMN_LIST	Returns comma-separated list	
COMPRESS	Returns a binary, compressed string	
CONCAT	Returns concatenated string	
CONCAT_WS	Concatenate with separator	
CONNECTION_ID	Connection thread ID	
CONTAINS	Whether one geometry contains another	
CONVERT	Convert a value from one type to another type	
CONV	Converts numbers between different number bases	
CONVERT_TZ	Converts a datetime from on time zone to another	
CONVEXHULL	Synonym for ST_CONVEXHULL	
COS	Returns the cosine	
COT	Returns the cotangent	
COUNT	Returns count of non-null values	
COUNT DISTINCT	Returns count of number of different non-NULL values	
CRC32	Computes a cyclic redundancy check value	
CRC32C	Computes a cyclic redundancy check value	MariaDB 10.8
CROSSES	Whether two geometries spatially cross	

CUME_DIST	Window function that returns the cumulative distribution of a given row	
CURDATE	Returns the current date	
CURRENT_DATE	Synonym for CURDATE()	
CURRENT_ROLE	Current role name	
CURRENT_TIME	Synonym for CURTIME()	
CURRENT_TIMESTAMP	Synonym for NOW()	
CURRENT_USER	Username/host that authenicated the current client	
CURTIME	Returns the current time	
DATABASE	Current default database	
DATE FUNCTION	Extracts the date portion of a datetime	
DATEDIFF	Difference in days between two date/time values	
DATE_ADD	Date arithmetic - addition	
DATE_FORMAT	Formats the date value according to the format string	
DATE_SUB	Date arithmetic - subtraction	
DAY	Synonym for DAYOFMONTH()	
DAYNAME	Return the name of the weekday	
DAYOFMONTH	Returns the day of the month	
DAYOFWEEK	Returns the day of the week index	
DAYOFYEAR	Returns the day of the year	
DECODE	Decrypts a string encoded with ENCODE()	
DECODE_HISTOGRAM	Returns comma separated numerics corresponding to a probability distribution represented by a histogram	
DEFAULT	Returns column default	
DEGREES	Converts from radians to degrees	
DENSE_RANK	Rank of a given row with identical values receiving the same result, no skipping	
DES_DECRYPT	Decrypts a string encrypted with DES_ENCRYPT()	
DES_ENCRYPT	Encrypts a string using the Triple-DES algorithm	
DIMENSION	Synonym for ST_DIMENSION	
DISJOINT	Whether the two elements do not intersect	
DIV	Integer division	
ELT	Returns the N'th element from a set of strings	
ENCODE	Encrypts a string	
ENCRYPT	Encrypts a string with Unix crypt()	
ENDPOINT	Synonym for ST_ENDPOINT	
ENVELOPE	Synonym for ST_ENVELOPE	
EQUALS	Indicates whether two geometries are spatially equal	

EXP	e raised to the power of the argument	
EXPORT_SET	Returns an on string for every bit set, an off string for every bit not set	
ExteriorRing	Synonym for ST_ExteriorRing	
EXTRACT	Extracts a portion of the date	
EXTRACTVALUE	Returns the text of the first text node matched by the XPath expression	
FIELD	Returns the index position of a string in a list	
FIND_IN_SET	Returns the position of a string in a set of strings	
FLOOR	Largest integer value not greater than the argument	
FORMAT	Formats a number	
FORMAT_PICO_TIME	Given a time in picoseconds, returns a human-readable time value and unit indicator	MariaDB 11.0.2
FOUND_ROWS	Number of (potentially) returned rows	
FROM_BASE64	Given a base-64 encoded string, returns the decoded result as a binary string	
FROM_DAYS	Returns a date given a day	
FROM_UNIXTIME	Returns a datetime from a Unix timestamp	
GeomCollFromText	Synonym for ST_GeomCollFromText	
GeomCollFromWKB	Synonym for ST_GeomCollFromWKB	
GeometryCollectionFromText	Synonym for ST_GeomCollFromText	
GeometryCollectionFromWKB	Synonym for ST_GeomCollFromWKB	
GeometryFromText	Synonym for ST_GeomFromText	
GeometryFromWKB	Synonym for ST_GeomFromWKB	
GeomFromText	Synonym for ST_GeomFromText	
GeomFromWKB	Synonym for ST_GeomFromWKB	
GeometryN	Synonym for ST_GeometryN	
GEOMETRYCOLLECTION	Constructs a WKB GeometryCollection	
GeometryType	Synonym for ST_GeometryType	
GET_FORMAT	Returns a format string	
GET_LOCK	Obtain LOCK	
GLENGTH	Length of a LineString value	
GREATEST	Returns the largest argument	
GROUP_CONCAT	Returns string with concatenated values from a group	
HEX	Returns hexadecimal value	
HOUR	Returns the hour	
IF	If expr1 is TRUE, returns expr2; otherwise returns expr3	

IFNULL	Check whether an expression is NULL	
IN	True if expression equals any of the values in the list	
INTERVAL	Index of the argument that is less than the first argument	
INET6_ATON	Given an IPv6 or IPv4 network address, returns a VARBINARY numeric value	
INET6_NTOA	Given an IPv6 or IPv4 network address, returns the address as a nonbinary string	
INET_ATON	Returns numeric value of IPv4 address	
INET_NTOA	Returns dotted-quad representation of IPv4 address	
INSERT Function	Replaces a part of a string with another string	
INSTR	Returns the position of a string withing a string	
InteriorRingN	Synonym for ST_InteriorRingN	
INTERSECTS	Indicates whether two geometries spatially intersect	
IS	Tests whether a boolean is TRUE, FALSE, or UNKNOWN	
IsClosed	Synonym for ST_IsClosed	
IsEmpty	Synonym for ST_IsEmpty	
IS_FREE_LOCK	Checks whether lock is free to use	
IS_IPV4	Whether or not an expression is a valid IPv4 address	
IS_IPV4_COMPAT	Whether or not an IPv6 address is IPv4-compatible	
IS_IPV4_MAPPED	Whether an IPv6 address is a valid IPv4-mapped address	
IS_IPV6	Whether or not an expression is a valid IPv6 address	
IS NOT	Tests whether a boolean value is not TRUE, FALSE, or UNKNOWN	
IS NOT NULL	Tests whether a value is not NULL	
IS NULL	Tests whether a value is NULL	
ISNULL	Checks if an expression is NULL	
IsRing	Synonym for ST_IsRing	
IsSimple	Synonym for ST_IsSimple	
IS_USED_LOCK	Check if lock is in use	
JSON_ARRAY	Returns a JSON array containing the listed values	
JSON_ARRAY_INTERSECT		MariaDB 11.2.0
JSON_ARRAY_APPEND	Appends values to the end of the given arrays within a JSON document	
JSON_ARRAY_INSERT	Inserts a value into a JSON document	
JSON_COMPACT	Removes all unnecessary spaces so the json document is as short as possible	

JSON_CONTAINS	Whether a value is found in a given JSON document or at a specified path within the document	
JSON_CONTAINS_PATH	Indicates whether the given JSON document contains data at the specified path or paths	
JSON_DEPTH	Maximum depth of a JSON document	
JSON_DETAILED	Represents JSON in the most understandable way emphasizing nested structures	
JSON_EQUALS	Check for equality between JSON objects.	
JSON_EXISTS	Determines whether a specified JSON value exists in the given data	
JSON_EXTRACT	Extracts data from a JSON document.	
JSON_INSERT	Inserts data into a JSON document	
JSON_KEYS	Returns keys from top-level value of a JSON object or top-level keys from the path	
JSON_LENGTH	Returns the length of a JSON document, or the length of a value within the document	
JSON_LOOSE	Adds spaces to a JSON document to make it look more readable	
JSON_MERGE	Merges the given JSON documents	
JSON_MERGE_PATCH	RFC 7396-compliant merge of the given JSON documents	
JSON_MERGE_PRESERVE	Synonym for JSON_MERGE_PATCH.	
JSON_NORMALIZE	Recursively sorts keys and removes spaces, allowing comparison of json documents for equality	
JSON_OBJECT	Returns a JSON object containing the given key/value pairs	
JSON_OBJECT_FILTER_KEYS		MariaDB 11.2.0
JSON_OBJECT_TO_ARRAY		MariaDB 11.2.0
JSON_OBJECTAGG	Returns a JSON object containing key-value pairs	
JSON_OVERLAPS	Compares two json documents for overlaps	
JSON_PRETTY	Alias for json_detailed	MariaDB 10.10.3, MariaDB 10.9.5, MariaDB 10.8.7, MariaDB 10.7.8, MariaDB 10.6.12, MariaDB 10.5.19 and MariaDB 10.4.28
JSON_QUERY	Given a JSON document, returns an object or array specified by the path	
JSON_QUOTE	Quotes a string as a JSON value	
JSON_REMOVE	Removes data from a JSON document	
JSON_REPLACE	Replaces existing values in a JSON document	
JSON_SCHEMA_VALID	Validates a JSON schema	MariaDB 11.1.0
JSON_SEARCH	Returns the path to the given string within a JSON document	

JSON_SET	Updates or inserts data into a JSON document	
JSON_TABLE	Returns a representation of a JSON document as a relational table	
JSON_TYPE	Returns the type of a JSON value	
JSON_UNQUOTE	Unquotes a JSON value, returning a string	
JSON_VALID	Whether a value is a valid JSON document or not	
JSON_VALUE	Given a JSON document, returns the specified scalar	
KDF	Key derivation function	MariaDB 11.3.0
LAST_DAY	Returns the last day of the month	
LAST_INSERT_ID	Last inserted autoinc value	
LAST_VALUE	Returns the last value in a list	
LASTVAL	Get last value generated from a sequence	
LCASE	Synonym for [LOWER()	
LEAST	Returns the smallest argument	
LEFT	Returns the leftmost characters from a string	
LENGTH	Length of the string in bytes	
LIKE	Whether expression matches a pattern	
LineFromText	Synonym for ST_LineFromText	
LineFromWKB	Synonym for ST_LineFromWKB	
LINESTRING	Constructs a WKB LineString value from a number of WKB Point arguments	
LineStringFromText	Synonym for ST_LineFromText	
LineStringFromWKB	Synonym for ST_LineFromWKB	
LN	Returns natural logarithm	
LOAD_FILE	Returns file contents as a string	
LOCALTIME	Synonym for NOW()	
LOCALTIMESTAMP	Synonym for NOW()	
LOCATE	Returns the position of a substring in a string	
LOG	Returns the natural logarithm	
LOG10	Returns the base-10 logarithm	
LOG2	Returns the base-2 logarithm	
LOWER	Returns a string with all characters changed to lowercase	
LPAD	Returns the string left-padded with another string to a given length	
LTRIM	Returns the string with leading space characters removed	
MAKE_SET	Make a set of strings that matches a bitmask	
MAKEDATE	Returns a date given a year and day	
MAKETIME	Returns a time	
MASTER_GTID_WAIT	Wait until slave reaches the GTID position	

MASTER_POS_WAIT	Blocks until the slave has applied all specified updates	
MATCH AGAINST	Perform a fulltext search on a fulltext index	
MAX	Returns the maximum value	
MBRContains	Indicates one Minimum Bounding Rectangle contains another	
MBRDisjoint	Indicates whether the Minimum Bounding Rectangles of two geometries are disjoint	
MBREqual	Whether the Minimum Bounding Rectangles of two geometries are the same.	
MBRIntersects	Indicates whether the Minimum Bounding Rectangles of the two geometries intersect	
MBROverlaps	Whether the Minimum Bounding Rectangles of two geometries overlap	
MBRTouches	Whether the Minimum Bounding Rectangles of two geometries touch.	
MBRWithin	Indicates whether one Minimum Bounding Rectangle is within another	
MD5	MD5 checksum	
MEDIAN	Window function that returns the median value of a range of values	
MICROSECOND	Returns microseconds from a date or datetime	
MID	Synonym for SUBSTRING(str,pos,len)	
MIN	Returns the minimum value	
MINUTE	Returns a minute from 0 to 59	
MLineFromText	Constructs MULTILINESTRING using its WKT representation and SRID	
MLineFromWKB	Constructs a MULTILINESTRING	
MOD	Modulo operation. Remainder of N divided by M	
MONTH	Returns a month from 1 to 12	
MONTHNAME	Returns the full name of the month	
MPointFromText	Constructs a MULTIPOINT value using its WKT and SRID	
MPointFromWKB	Constructs a MULTIPOINT value using its WKB representation and SRID	
MPolyFromText	Constructs a MULTIPOLYGON value	
MPolyFromWKB	Constructs a MULTIPOLYGON value using its WKB representation and SRID	
MultiLineStringFromText	Synonym for MLineFromText	
MultiLineStringFromWKB	A synonym for MLineFromWKB	
MULTIPOINT	Constructs a WKB MultiPoint value	
MultiPointFromText	Synonym for MPointFromText	
MultiPointFromWKB	Synonym for MPointFromWKB	

MULTIPOLYGON	Constructs a WKB MultiPolygon	
MultiPolygonFromText	Synonym for MPolyFromText	
MultiPolygonFromWKB	Synonym for MPolyFromWKB	
MULTILINESTRING	Constructs a MultiLineString value	
NAME_CONST	Returns the given value	
NATURAL_SORT_KEY	Sorting that is more more similar to natural human sorting	
NOT LIKE	Same as NOT(expr LIKE pat [ESCAPE 'escape_char'])	
NOT REGEXP	Same as NOT (expr REGEXP pat)	
NULLIF	Returns NULL if expr1 = expr2	
NEXTVAL	Generate next value for sequence	
NOT BETWEEN	Same as NOT (expr BETWEEN min AND max)	
NOT IN	Same as NOT (expr IN (value,...))	
NOW	Returns the current date and time	
NTILE	Returns an integer indicating which group a given row falls into	
NumGeometries	Synonym for ST_NumGeometries	
NumInteriorRings	Synonym for NumInteriorRings	
NumPoints	Synonym for ST_NumPoints	
OCT	Returns octal value	
OCTET_LENGTH	Synonym for LENGTH()	
OLD_PASSWORD	Pre MySQL 4.1 password implementation	
ORD	Return ASCII or character code	
OVERLAPS	Indicates whether two elements spatially overlap	
PASSWORD	Calculates a password string	
PERCENT_RANK	Window function that returns the relative percent rank of a given row	
PERCENTILE_CONT	Returns a value which corresponds to the given fraction in the sort order.	
PERCENTILE_DISC	Returns the first value in the set whose ordered position is the same or more than the specified fraction.	
PERIOD_ADD	Add months to a period	
PERIOD_DIFF	Number of months between two periods	
PI	Returns the value of π (pi)	
POINT	Constructs a WKB Point	
PointFromText	Synonym for ST_PointFromText	
PointFromWKB	Synonym for PointFromWKB	
PointN	Synonym for PointN	
PointOnSurface	Synonym for ST_PointOnSurface	
POLYGON	Constructs a WKB Polygon value from a number of WKB LineString arguments	
PolyFromText	Synonym for ST_PolyFromText	
PolyFromWKB	Synonym for ST_PolyFromWKB	

PolygonFromText	Synonym for ST_PolyFromText	
PolygonFromWKB	Synonym for ST_PolyFromWKB	
POSITION	Returns the position of a substring in a string	
POW	Returns X raised to the power of Y	
POWER	Synonym for POW()	
QUARTER	Returns year quarter from 1 to 4	
QUOTE	Returns quoted, properly escaped string	
RADIANS	Converts from degrees to radians	
RAND	Random floating-point value	
RANK	Rank of a given row with identical values receiving the same result	
REGEXP	Performs pattern matching	
REGEXP_INSTR	Position of the first appearance of a regex	
REGEXP_REPLACE	Replaces all occurrences of a pattern	
REGEXP_SUBSTR	Returns the matching part of a string	
RELEASE_LOCK	Releases lock obtained with GET_LOCK()	
REPEAT Function	Returns a string repeated a number of times	
REPLACE Function	Replace occurrences of a string	
REVERSE	Reverses the order of a string	
RIGHT	Returns the rightmost N characters from a string	
RLIKE	Synonym for REGEXP()	
RPAD	Returns the string right-padded with another string to a given length	
ROUND	Rounds a number	
ROW_COUNT	Number of rows affected by previous statement	
ROW_NUMBER	Row number of a given row with identical values receiving a different result	
RTRIM	Returns the string with trailing space characters removed	
SCHEMA	Synonym for DATABASE()	
SECOND	Returns the second of a time	
SEC_TO_TIME	Converts a second to a time	
SETVAL	Set the next value to be returned by a sequence	
SESSION_USER	Synonym for USER()	
SHA	Synonym for SHA1()	
SHA1	Calculates an SHA-1 checksum	
SHA2	Calculates an SHA-2 checksum	
SIGN	Returns 1, 0 or -1	
SIN	Returns the sine	
SLEEP	Pauses for the given number of seconds	

SOUNDEX	Returns a string based on how the string sounds	
SOUNDS LIKE	SOUNDEX(expr1) = SOUNDEX(expr2)	
SPACE	Returns a string of space characters	
SPIDER_BG_DIRECT_SQL	Background SQL execution	
SPIDER_COPY_TABLES	Copy table data	
SPIDER_DIRECT_SQL	Execute SQL on the remote server	
SPIDER_FLUSH_TABLE_MON_CACHE	Refreshing Spider monitoring server information	
SQRT	Square root	
SRID	Synonym for ST_SRID	
ST_AREA	Area of a Polygon	
ST_AsBinary	Converts a value to its WKB representation	
ST_AsText	Converts a value to its WKT-Definition	
ST_AsWKB	Synonym for ST_AsBinary	
ST_ASWKT	Synonym for ST_ASTEXT()	
ST_BOUNDARY	Returns a geometry that is the closure of a combinatorial boundary	
ST_BUFFER	A new geometry with a buffer added to the original geometry	
ST_CENTROID	The mathematical centroid (geometric center) for a MultiPolygon	
ST_CONTAINS	Whether one geometry is contained by another	
ST_CONVEXHULL	The minimum convex geometry enclosing all geometries within the set	
ST_CROSSES	Whether two geometries spatially cross	
ST_DIFFERENCE	Point set difference	
ST_DIMENSION	Inherent dimension of a geometry value	
ST_DISJOINT	Whether one geometry is spatially disjoint from another	
ST_DISTANCE	The distance between two geometries	
ST_DISTANCE_SPHERE	The spherical distance between two geometries	
ST_ENDPOINT	Returns the endpoint of a LineString	
ST_ENVELOPE	Returns the Minimum Bounding Rectangle for a geometry value	
ST_EQUALS	Whether two geometries are spatoially equal	
ST_ExteriorRing	Returns the exterior ring of a Polygon as a LineString	
ST_GeomCollFromText	Constructs a GEOMETRYCOLLECTION value	
ST_GeomCollFromWKB	Constructs a GEOMETRYCOLLECTION value from a WKB	

ST_GeometryCollectionFromText	Synonym for ST_GeomCollFromText	
ST_GeometryCollectionFromWKB	Synonym for ST_GeomCollFromWKB	
ST_GeometryFromText	Synonym for ST_GeomFromText	
ST_GeometryFromWKB	Synonym for ST_GeomFromWKB	
ST_GEOMETRYN	Returns the N-th geometry in a GeometryCollection	
ST_GEOMETRYTYPE	Returns name of the geometry type of which a given geometry instance is a member	
ST_GeomFromText	Constructs a geometry value using its WKT and SRID	
ST_GeomFromWKB	Constructs a geometry value using its WKB representation and SRID	
ST_InteriorRingN	Returns the N-th interior ring for a Polygon	
ST_INTERSECTION	The intersection, or shared portion, of two geometries	
ST_INTERSECTS	Whether two geometries spatially intersect	
ST_ISCLOSED	Returns true if a given LINESTRING's start and end points are the same	
ST_ISEMPTY	Indicated validity of geometry value	
ST_IsRing	Returns true if a given LINESTRING is both ST_IsClosed and ST_IsSimple	
ST_IsSimple	Returns true if the given Geometry has no anomalous geometric points	
ST_LENGTH	Length of a LineString value	
ST_LineFromText	Creates a linestring value	
ST_LineFromWKB	Constructs a LINESTRING using its WKB and SRID	
ST_LineStringFromText	Synonym for ST_LineFromText	
ST_LineStringFromWKB	Synonym for ST_LineFromWKB	
ST_NUMGEOMETRIES	Number of geometries in a GeometryCollection	
ST_NumInteriorRings	Number of interior rings in a Polygon	
ST_NUMPOINTS	Returns the number of Point objects in a LineString	
ST_OVERLAPS	Whether two geometries overlap	
ST_PointFromText	Constructs a POINT value	
ST_PointFromWKB	Constructs POINT using its WKB and SRID	
ST_POINTN	Returns the N-th Point in the LineString	
ST_POINTONSURFACE	Returns a POINT guaranteed to intersect a surface	
ST_PolyFromText	Constructs a POLYGON value	
ST_PolyFromWKB	Constructs POLYGON value using its WKB representation and SRID	
ST_PolygonFromText	Synonym for ST_PolyFromText	
ST_PolygonFromWKB	Synonym for ST_PolyFromWKB	

ST_RELATE	Returns true if two geometries are related	
ST_SRID	Returns a Spatial Reference System ID	
ST_STARTPOINT	Returns the start point of a LineString	
ST_SYMDIFFERENCE	Portions of two geometries that don't intersect	
ST_TOUCHES	Whether one geometry g1 spatially touches another	
ST_UNION	Union of two geometries	
ST_WITHIN	Whether one geometry is within another	
ST_X	X-coordinate value for a point	
ST_Y	Y-coordinate for a point	
STARTPOINT	Synonym for ST_StartPoint	
STD	Population standard deviation	
STDDEV	Population standard deviation	
STDDEV_POP	Returns the population standard deviation	
STDDEV_SAMP	Standard deviation	
STR_TO_DATE	Converts a string to date	
STRCMP	Compares two strings in sort order	
SUBDATE	Subtract a date unit or number of days	
SUBSTR	Returns a substring from string starting at a given position	
SUBSTRING	Returns a substring from string starting at a given position	
SUBSTRING_INDEX	Returns the substring from string before count occurrences of a delimiter	
SUBTIME	Subtracts a time from a date/time	
SUM	Sum total	
SYS.EXTRACT_SCHEMA_FROM_FILE_NAME	Given a file path, returns the schema (database) name	MariaDB 10.6
SYS.EXTRACT_TABLE_FROM_FILE_NAME	Given a file path, returns the table name	MariaDB 10.6
SYS.FORMAT_BYTES	Returns a string consisting of a value and the units in a human-readable format	MariaDB 10.6
SYS.FORMAT_PATH	Returns a modified path after replacing subpaths matching the values of various system variables with the variable name	MariaDB 10.6
SYS.FORMAT_STATEMENT	Returns a reduced length string	MariaDB 10.6
SYS.FORMAT_TIME	Returns a human-readable time value and unit indicator	MariaDB 10.6
SYS.LIST_ADD	Adds a value to a given list	MariaDB 10.6
SYS.LIST_DROP	Drops a value from a given list	MariaDB 10.6
SYS.PS_IS_ACCOUNT_ENABLED	Whether Performance Schema instrumentation for the given account is enabled	MariaDB 10.6

SYS.PS_IS_CONSUMER_ENABLED	Whether Performance Schema instrumentation for the given consumer is enabled	MariaDB 10.6
SYS.PS_IS_INSTRUMENT_DEFAULT_ENABLED	Whether a given Performance Schema instrument is enabled by default	MariaDB 10.6
SYS.PS_IS_INSTRUMENT_DEFAULT_TIMED	Returns whether a given Performance Schema instrument is timed by default	MariaDB 10.6
SYS.PS_IS_THREAD_INSTRUMENTED	Returns whether or not Performance Schema instrumentation for the given connection_id is enabled	MariaDB 10.6
SYS.PS_THREAD_ACCOUNT	Returns the account (username@hostname) associated with the given thread_id	MariaDB 10.6
SYS.PS_THREAD_ID	Returns the thread_id associated with the given connection_id	MariaDB 10.6
SYS.PS_THREAD_STACK	Returns all statements, stages, and events within the Performance Schema for a given thread_id	MariaDB 10.6
SYS.PS_THREAD_TRX_INFO	Returns a JSON object with information about the thread specified by the given thread_id	MariaDB 10.6
SYS.QUOTE_IDENTIFIER	Quotes a string to produce a result that can be used as an identifier in an SQL statement	MariaDB 10.6
SYS.SYS_GET_CONFIG	Returns a configuration option value from the sys_config table	MariaDB 10.6
SYS.VERSION_MAJOR	Returns the MariaDB Server major release version	MariaDB 10.6
SYS.VERSION_MINOR	Returns the MariaDB Server minor release version	MariaDB 10.6
SYS.VERSION_PATCH	Returns the MariaDB Server patch release version	MariaDB 10.6
SYS_GUID	Generates a globally unique identifier	
SYSDATE	Returns the current date and time	
SYSTEM_USER	Synonym for USER()	
TAN	Returns the tangent	
TIME function	Extracts the time	
TIMEDIFF	Returns the difference between two date/times	
TIMESTAMP FUNCTION	Return the datetime, or add a time to a date/time	
TIMESTAMPADD	Add interval to a date or datetime	
TIMESTAMPDIFF	Difference between two datetimes	
TIME_FORMAT	Formats the time value according to the format string	
TIME_TO_SEC	Returns the time argument, converted to seconds	
TO_BASE64	Converts a string to its base-64 encoded form	
TO_CHAR	Converts a date/time type to a char	
TO_DAYS	Number of days since year 0	
TO_SECONDS	Number of seconds since year 0	
TOUCHES	Whether two geometries spatially touch	

TRIM	Returns a string with all given prefixes or suffixes removed	
TRUNCATE	Truncates X to D decimal places	
UCASE	Synonym for UPPER]]()	
UNHEX	Interprets pairs of hex digits as a number and converts to the character represented by the number	
UNCOMPRESS	Uncompresses string compressed with COMPRESS()	
UNCOMPRESSED_LENGTH	Returns length of a string before being compressed with COMPRESS()	
UNIX_TIMESTAMP	Returns a Unix timestamp	
UPDATEXML	Replace XML	
UPPER	Changes string to uppercase	
USER	Current user/host	
UTC_DATE	Returns the current UTC date	
UTC_TIME	Returns the current UTC time	
UTC_TIMESTAMP	Returns the current UTC date and time	
UUID	Returns a Universal Unique Identifier	
UUID_SHORT	Return short universal identifier	
VALUES or VALUE	Refer to columns in INSERT ... ON DUPLICATE KEY UPDATE	
VAR_POP	Population standard variance	
VAR_SAMP	Returns the sample variance	
VARIANCE	Population standard variance	
VERSION	MariaDB server version	
WEEK	Returns the week number	
WEEKDAY	Returns the weekday index	
WEEKOFYEAR	Returns the calendar week of the date as a number in the range from 1 to 53	
WEIGHT_STRING	Weight of the input string	
WITHIN	Indicate whether a geographic element is spacially within another	
WSREP_LAST_SEEN_GTID	Returns the Global Transaction ID of the most recent write transaction observed by the client.	
WSREP_LAST_WRITTEN_GTID	Returns the Global Transaction ID of the most recent write transaction performed by the client.	
WSREP_SYNC_WAIT_UPTO_GTID	Blocks the client until the transaction specified by the given Global Transaction ID is applied and committed by the node	
X	Synonym for ST_X	
Y	Synonym for ST_Y	
YEAR	Returns the year for the given date	
YEARWEEK	Returns year and week for a date	

1.2.2 String Functions

Functions dealing with strings, such as CHAR, CONVERT, CONCAT, PAD, REGEXP, TRIM, etc.

Regular Expressions Functions

Functions for dealing with regular expressions

Dynamic Columns Functions

Functions for storing key/value pairs of data within a column.

ASCII

Numeric ASCII value of leftmost character.

BIN

Returns binary value.

BINARY Operator

Casts to a binary string.

BIT_LENGTH

Returns the length of a string in bits.

CAST

Casts a value of one type to another type.

CHAR Function

Returns string based on the integer values for the individual characters.

CHARACTER_LENGTH

Synonym for CHAR_LENGTH().

CHAR_LENGTH

Length of the string in characters.

CHR

Returns string based on integer values of the individual characters.

CONCAT

Returns concatenated string.

CONCAT_WS

Concatenate with separator.

CONVERT

Convert a value from one type to another type.

ELT

Returns the N'th element from a set of strings.

EXPORT_SET

Returns an on string for every bit set, an off string for every bit not set.

EXTRACTVALUE

Returns the text of the first text node matched by the XPath expression.

FIELD

Returns the index position of a string in a list.

FIND_IN_SET

Returns the position of a string in a set of strings.

FORMAT

Formats a number.

FROM_BASE64

Given a base-64 encoded string, returns the decoded result as a binary string.

HEX

Returns hexadecimal value.

INSERT Function

Replaces a part of a string with another string.

INSTR

Returns the position of a string within a string.

LCASE

Synonym for LOWER().

LEFT

Returns the leftmost characters from a string.

LENGTH

Length of the string in bytes.

LENGTHB

Length of the given string, in bytes.

LIKE

Whether expression matches a pattern.

LOAD_FILE

Returns file contents as a string.

LOCATE

Returns the position of a substring in a string.

LOWER

Returns a string with all characters changed to lowercase.

LPAD

Returns the string left-padded with another string to a given length.

LTRIM

Returns the string with leading space characters removed.

MAKE_SET

Make a set of strings that matches a bitmask.

MATCH AGAINST

Perform a fulltext search on a fulltext index.

Full-Text Index Stopwords

Default list of full-text stopwords used by MATCH...AGAINST.

MID

Synonym for SUBSTRING(str,pos,len).

NATURAL_SORT_KEY

Sorting that is closer to natural human sorting.

NOT LIKE

Same as NOT(expr LIKE pat [ESCAPE 'escape_char']).

NOT REGEXP

Same as NOT (expr REGEXP pat).

OCTET_LENGTH

Returns the length of the given string, in bytes.

ORD

Return ASCII or character code.

POSITION

Returns the position of a substring in a string.

QUOTE

Returns quoted, properly escaped string.

REPEAT Function

Returns a string repeated a number of times.

REPLACE Function

Replace occurrences of a string.

REVERSE

Reverses the order of a string.

RIGHT

Returns the rightmost N characters from a string.

RPAD

Returns the string right-padded with another string to a given length.

RTRIM

Returns the string with trailing space characters removed.

SFORMAT

Given a string and a formatting specification, returns a formatted string.

SOUNDEX

Returns a string based on how the string sounds.

SOUNDS LIKE

SOUNDEX(expr1) = SOUNDEX(expr2).

SPACE

Returns a string of space characters.

STRCMP

Compares two strings in sort order.

SUBSTR

Returns a substring from string starting at a given position.

SUBSTRING

Returns a substring from string starting at a given position.

SUBSTRING_INDEX

Returns the substring from string before count occurrences of a delimiter.

TO_BASE64

Converts a string to its base-64 encoded form.

TO_CHAR

Converts a date/time/timestamp type expression to a string.

TRIM

Returns a string with all given prefixes or suffixes removed.

TRIM_ORACLE

Synonym for the Oracle mode version of TRIM().

UCASE

Synonym for UPPER().

UNCOMPRESS

Uncompresses string compressed with COMPRESS().

UNCOMPRESSED_LENGTH

Returns length of a string before being compressed with COMPRESS().

UNHEX

Interprets pairs of hex digits as numbers and converts to the character represented by the number.

UPDATEXML

Replace XML.

UPPER

Changes string to uppercase.

WEIGHT_STRING

Weight of the input string.

Type Conversion

When implicit type conversion takes place.

There are 3 related questions.

1.2.2.1 Regular Expressions Functions

MariaDB includes a number of functions for dealing with regular expressions.

Regular Expressions Overview

Regular Expressions allow MariaDB to perform complex pattern matching on a string.

PCRE - Perl Compatible Regular Expressions

PCRE (Perl Compatible Regular Expressions) for enhanced regular expressions.

NOT REGEXP

Same as NOT (expr REGEXP pat).

REGEXP

Performs pattern matching

REGEXP_INSTR

Position of the first appearance of a regex.

REGEXP_REPLACE

Replaces all occurrences of a pattern.

REGEXP_SUBSTR

Returns the matching part of a string.

RLIKE

Synonym for REGEXP

There are 1 related questions.

1.2.2.1.1 Regular Expressions Overview

Contents

Regular Expressions allow MariaDB to perform complex pattern matching on a string. In many cases, the simple pattern matching provided by LIKE is sufficient. `LIKE` performs two kinds of matches:

- `_` - the underscore, matching a single character
- `%` - the percentage sign, matching any number of characters.

In other cases you may need more control over the returned matches, and will need to use regular expressions.

Until MariaDB 10.0.5, MariaDB used the POSIX 1003.2 compliant regular expression library. The current PCRE library is mostly backwards compatible with what is described below - see the PCRE Regular Expressions article for the enhancements made in 10.0.5.

Regular expression matches are performed with the REGEXP function. `RLIKE` is a synonym for `REGEXP`.

> Comparisons are performed on the byte value, so characters that are treated as equivalent by a collation, but do not have the same byte-value, such as accented characters, could evaluate as unequal.

Without any special characters, a regular expression match is true if the characters match. The match is case-insensitive, except in the case of BINARY strings.

```
SELECT 'Maria' REGEXP 'Maria';
+------------------------+
| 'Maria' REGEXP 'Maria' |
+------------------------+
|                      1 |
+------------------------+

SELECT 'Maria' REGEXP 'maria';
+------------------------+
| 'Maria' REGEXP 'maria' |
+------------------------+
|                      1 |
+------------------------+

SELECT BINARY 'Maria' REGEXP 'maria';
+-------------------------------+
| BINARY 'Maria' REGEXP 'maria' |
+-------------------------------+
|                             0 |
+-------------------------------+
```

Note that the word being matched must match the whole pattern:

```
SELECT 'Maria' REGEXP 'Mari';
+-----------------------+
| 'Maria' REGEXP 'Mari' |
+-----------------------+
|                     1 |
+-----------------------+

SELECT 'Mari' REGEXP 'Maria';
+-----------------------+
| 'Mari' REGEXP 'Maria' |
+-----------------------+
|                     0 |
+-----------------------+
```

The first returns true because the pattern "Mari" exists in the expression "Maria". When the order is reversed, the result is false, as the pattern "Maria" does not exist in the expression "Mari"

A match can be performed against more than one word with the `|` character. For example:

```
SELECT 'Maria' REGEXP 'Monty|Maria';
+------------------------------+
| 'Maria' REGEXP 'Monty|Maria' |
+------------------------------+
|                            1 |
+------------------------------+
```

Special Characters

The above examples introduce the syntax, but are not very useful on their own. It's the special characters that give regular expressions their power.

^

`^` matches the beginning of a string (inside square brackets it can also mean NOT - see below):

```
SELECT 'Maria' REGEXP '^Ma';
+----------------------+
| 'Maria' REGEXP '^Ma' |
+----------------------+
|                    1 |
+----------------------+
```

$

`$` matches the end of a string:

```
SELECT 'Maria' REGEXP 'ia$';
+----------------------+
| 'Maria' REGEXP 'ia$' |
+----------------------+
|                    1 |
+----------------------+
```

.

`.` matches any single character:

```
SELECT 'Maria' REGEXP 'Ma.ia';
+------------------------+
| 'Maria' REGEXP 'Ma.ia' |
+------------------------+
|                      1 |
+------------------------+

SELECT 'Maria' REGEXP 'Ma..ia';
+-------------------------+
| 'Maria' REGEXP 'Ma..ia' |
+-------------------------+
|                       0 |
+-------------------------+
```

*

`x*` matches zero or more of a character `x`. In the examples below, it's the `r` character.

```
SELECT 'Maria' REGEXP 'Mar*ia';
+-------------------------+
| 'Maria' REGEXP 'Mar*ia' |
+-------------------------+
|                       1 |
+-------------------------+

SELECT 'Maia' REGEXP 'Mar*ia';
+------------------------+
| 'Maia' REGEXP 'Mar*ia' |
+------------------------+
|                      1 |
+------------------------+

SELECT 'Marrria' REGEXP 'Mar*ia';
+---------------------------+
| 'Marrria' REGEXP 'Mar*ia' |
+---------------------------+
|                         1 |
+---------------------------+
```

+

`x+` matches one or more of a character `x`. In the examples below, it's the `r` character.

```
SELECT 'Maria' REGEXP 'Mar+ia';
+-------------------------+
| 'Maria' REGEXP 'Mar+ia' |
+-------------------------+
|                       1 |
+-------------------------+

SELECT 'Maia' REGEXP 'Mar+ia';
+------------------------+
| 'Maia' REGEXP 'Mar+ia' |
+------------------------+
|                      0 |
+------------------------+

SELECT 'Marrria' REGEXP 'Mar+ia';
+---------------------------+
| 'Marrria' REGEXP 'Mar+ia' |
+---------------------------+
|                         1 |
+---------------------------+
```

?

`x?` matches zero or one of a character `x`. In the examples below, it's the `r` character.

```
SELECT 'Maria' REGEXP 'Mar?ia';
+-------------------------+
| 'Maria' REGEXP 'Mar?ia' |
+-------------------------+
|                       1 |
+-------------------------+

SELECT 'Maia' REGEXP 'Mar?ia';
+------------------------+
| 'Maia' REGEXP 'Mar?ia' |
+------------------------+
|                      1 |
+------------------------+

SELECT 'Marrria' REGEXP 'Mar?ia';
+---------------------------+
| 'Marrria' REGEXP 'Mar?ia' |
+---------------------------+
|                         0 |
+---------------------------+
```

()

`(xyz)` - combine a sequence, for example `(xyz)+` or `(xyz)*`

```
SELECT 'Maria' REGEXP '(ari)+';
+-------------------------+
| 'Maria' REGEXP '(ari)+' |
+-------------------------+
|                       1 |
+-------------------------+
```

{}

`x{n}` and `x{m,n}` This notation is used to match many instances of the `x` . In the case of `x{n}` the match must be exactly that many times. In the case of `x{m,n}` , the match can occur from `m` to `n` times. For example, to match zero or one instance of the string `ari` (which is identical to `(ari)?`), the following can be used:

```
SELECT 'Maria' REGEXP '(ari){0,1}';
+-----------------------------+
| 'Maria' REGEXP '(ari){0,1}' |
+-----------------------------+
|                           1 |
+-----------------------------+
```

[]

`[xy]` groups characters for matching purposes. For example, to match either the `p` or the `r` character:

```
SELECT 'Maria' REGEXP 'Ma[pr]ia';
+---------------------------+
| 'Maria' REGEXP 'Ma[pr]ia' |
+---------------------------+
|                         1 |
+---------------------------+
```

The square brackets also permit a range match, for example, to match any character from a-z, `[a-z]` is used. Numeric ranges are also permitted.

```
SELECT 'Maria' REGEXP 'Ma[a-z]ia';
+---------------------------+
| 'Maria' REGEXP 'Ma[a-z]ia' |
+---------------------------+
|                          1 |
+---------------------------+
```

The following does not match, as `r` falls outside of the range `a-p` .

```
SELECT 'Maria' REGEXP 'Ma[a-p]ia';
+----------------------------+
| 'Maria' REGEXP 'Ma[a-p]ia' |
+----------------------------+
|                          0 |
+----------------------------+
```

^

The `^` character means does `NOT` match, for example:

```
SELECT 'Maria' REGEXP 'Ma[^p]ia';
+----------------------------+
| 'Maria' REGEXP 'Ma[^p]ia' |
+----------------------------+
|                          1 |
+----------------------------+

SELECT 'Maria' REGEXP 'Ma[^r]ia';
+----------------------------+
| 'Maria' REGEXP 'Ma[^r]ia' |
+----------------------------+
|                          0 |
+----------------------------+
```

The `[` and `]` characters on their own can be literally matched inside a `[]` block, without escaping, as long as they immediately match the opening bracket:

```
SELECT '[Maria' REGEXP '[[]';
+-----------------------+
| '[Maria' REGEXP '[[]' |
+-----------------------+
|                     1 |
+-----------------------+

SELECT '[Maria' REGEXP '[]]';
+-----------------------+
| '[Maria' REGEXP '[]]' |
+-----------------------+
|                     0 |
+-----------------------+

SELECT ']Maria' REGEXP '[]]';
+-----------------------+
| ']Maria' REGEXP '[]]' |
+-----------------------+
|                     1 |
+-----------------------+

SELECT ']Maria' REGEXP '[]a]';
+------------------------+
| ']Maria' REGEXP '[]a]' |
+------------------------+
|                      1 |
+------------------------+
```

Incorrect order, so no match:

```
SELECT ']Maria' REGEXP '[a]]';
+------------------------+
| ']Maria' REGEXP '[a]]' |
+------------------------+
|                      0 |
+------------------------+
```

The `-` character can also be matched in the same way:

```
SELECT '-Maria' REGEXP '[1-10]';
+--------------------------+
| '-Maria' REGEXP '[1-10]' |
+--------------------------+
|                        0 |
+--------------------------+

SELECT '-Maria' REGEXP '[-1-10]';
+---------------------------+
| '-Maria' REGEXP '[-1-10]' |
+---------------------------+
|                         1 |
+---------------------------+
```

Word boundaries

The `:<:` and `:>:` patterns match the beginning and the end of a word respectively. For example:

```
SELECT 'How do I upgrade MariaDB?' REGEXP '[[:<:]]MariaDB[[:>:]]';
+-----------------------------------------------------------+
| 'How do I upgrade MariaDB?' REGEXP '[[:<:]]MariaDB[[:>:]]' |
+-----------------------------------------------------------+
|                                                         1 |
+-----------------------------------------------------------+

SELECT 'How do I upgrade MariaDB?' REGEXP '[[:<:]]Maria[[:>:]]';
+-----------------------------------------------------------+
| 'How do I upgrade MariaDB?' REGEXP '[[:<:]]Maria[[:>:]]' |
+-----------------------------------------------------------+
|                                                         0 |
+-----------------------------------------------------------+
```

Character Classes

There are a number of shortcuts to match particular preset character classes. These are matched with the `[:character_class:]` pattern (inside a `[]` set). The following character classes exist:

Character Class	Description
alnum	Alphanumeric
alpha	Alphabetic
blank	Whitespace
cntrl	Control characters
digit	Digits
graph	Graphic characters
lower	Lowercase alphabetic
print	Graphic or space characters
punct	Punctuation
space	Space, tab, newline, and carriage return
upper	Uppercase alphabetic
xdigit	Hexadecimal digit

For example:

```
SELECT 'Maria' REGEXP 'Mar[[:alnum:]]*';
+---------------------------------+
| 'Maria' REGEXP 'Mar[:alnum:]*' |
+---------------------------------+
|                               1 |
+---------------------------------+
```

Remember that matches are by default case-insensitive, unless a binary string is used, so the following example, specifically looking for an uppercase, counter-intuitively matches a lowercase character:

```
SELECT 'Mari' REGEXP 'Mar[[:upper:]]+';
+---------------------------------+
| 'Mari' REGEXP 'Mar[[:upper:]]+' |
+---------------------------------+
|                               1 |
+---------------------------------+

SELECT BINARY 'Mari' REGEXP 'Mar[[:upper:]]+';
+----------------------------------------+
| BINARY 'Mari' REGEXP 'Mar[[:upper:]]+' |
+----------------------------------------+
|                                      0 |
+----------------------------------------+
```

Character Names

There are also number of shortcuts to match particular preset character names. These are matched with the `[.character.]` pattern (inside a `[]` set). The following character classes exist:

Name	Character
NUL	0
SOH	001
STX	002
ETX	003
EOT	004
ENQ	005
ACK	006
BEL	007
alert	007
BS	010
backspace	'\b'
HT	011
tab	'\t'
LF	012
newline	'\n'
VT	013
vertical-tab	'\v'
FF	014
form-feed	'\f'
CR	015
carriage-return	'\r'
SO	016
SI	017
DLE	020
DC1	021
DC2	022
DC3	023
DC4	024
NAK	025
SYN	026

ETB	027
CAN	030
EM	031
SUB	032
ESC	033
IS4	034
FS	034
IS3	035
GS	035
IS2	036
RS	036
IS1	037
US	037
space	' '
exclamation-mark	'!'
quotation-mark	'"'
number-sign	'#'
dollar-sign	'$'
percent-sign	'%'
ampersand	'&'
apostrophe	'\''
left-parenthesis	'('
right-parenthesis	')'
asterisk	'*'
plus-sign	'+'
comma	','
hyphen	'-'
hyphen-minus	'-'
period	'.'
full-stop	'.'
slash	'/'
solidus	'/'
zero	'0'
one	'1'
two	'2'
three	'3'
four	'4'
five	'5'
six	'6'
seven	'7'
eight	'8'
nine	'9'
colon	':'

semicolon	';'
less-than-sign	'<'
equals-sign	'='
greater-than-sign	'>'
question-mark	'?'
commercial-at	'@'
left-square-bracket	'['
backslash	' '
reverse-solidus	' '
right-square-bracket	']'
circumflex	'^'
circumflex-accent	'^'
underscore	'_'
low-line	'_'
grave-accent	'`'
left-brace	'{'
left-curly-bracket	'{'
vertical-line	'|'
right-brace	'}'
right-curly-bracket	'}'
tilde	''
DEL	177

For example:

```
SELECT '|' REGEXP '[[.vertical-line.]]';
+-----------------------------------+
| '|' REGEXP '[[.vertical-line.]]' |
+-----------------------------------+
|                                 1 |
+-----------------------------------+
```

Combining

The true power of regular expressions is unleashed when the above is combined, to form more complex examples. Regular expression's reputation for complexity stems from the seeming complexity of multiple combined regular expressions, when in reality, it's simply a matter of understanding the characters and how they apply:

The first example fails to match, as while the `Ma` matches, either `i` or `r` only matches once before the `ia` characters at the end.

```
SELECT 'Maria' REGEXP 'Ma[ir]{2}ia';
+------------------------------+
| 'Maria' REGEXP 'Ma[ir]{2}ia' |
+------------------------------+
|                            0 |
+------------------------------+
```

This example matches, as either `i` or `r` match exactly twice after the `Ma`, in this case one `r` and one `i`.

```
SELECT 'Maria' REGEXP 'Ma[ir]{2}';
+----------------------------+
| 'Maria' REGEXP 'Ma[ir]{2}' |
+----------------------------+
|                          1 |
+----------------------------+
```

Escaping

With the large number of special characters, care needs to be taken to properly escape characters. Two backslash characters,
(one for the MariaDB parser, one for the regex library), are required to properly escape a character. For example:

To match the literal `(Ma` :

```
SELECT '(Maria)' REGEXP '(Ma';
ERROR 1139 (42000): Got error 'parentheses not balanced' from regexp

SELECT '(Maria)' REGEXP '\(Ma';
ERROR 1139 (42000): Got error 'parentheses not balanced' from regexp

SELECT '(Maria)' REGEXP '\\(Ma';
+--------------------------+
| '(Maria)' REGEXP '\\(Ma' |
+--------------------------+
|                        1 |
+--------------------------+
```

To match `r+` : The first two examples are incorrect, as they match `r` one or more times, not `r+` :

```
SELECT 'Mar+ia' REGEXP 'r+';
+----------------------+
| 'Mar+ia' REGEXP 'r+' |
+----------------------+
|                    1 |
+----------------------+

SELECT 'Maria' REGEXP 'r+';
+--------------------+
| 'Maria' REGEXP 'r+' |
+--------------------+
|                  1 |
+--------------------+

SELECT 'Maria' REGEXP 'r\\+';
+----------------------+
| 'Maria' REGEXP 'r\\+' |
+----------------------+
|                    0 |
+----------------------+

SELECT 'Maria' REGEXP 'r+';
+--------------------+
| 'Maria' REGEXP 'r+' |
+--------------------+
|                  1 |
+--------------------+
```

1.2.2.1.2 Perl Compatible Regular Expressions (PCRE) Documentation

Contents

PCRE Versions

PCRE Version	Introduced	Maturity
PCRE2 10.34	MariaDB 10.5.1	Stable
PCRE 8.43	MariaDB 10.1.39	Stable
PCRE 8.42	MariaDB 10.2.15, MariaDB 10.1.33, MariaDB 10.0.35	Stable
PCRE 8.41	MariaDB 10.2.8, MariaDB 10.1.26, MariaDB 10.0.32	Stable
PCRE 8.40	MariaDB 10.2.5, MariaDB 10.1.22, MariaDB 10.0.30	Stable
PCRE 8.39	MariaDB 10.1.15, MariaDB 10.0.26	Stable
PCRE 8.38	MariaDB 10.1.10, MariaDB 10.0.23	Stable
PCRE 8.37	MariaDB 10.1.5, MariaDB 10.0.18	Stable
PCRE 8.36	MariaDB 10.1.2, MariaDB 10.0.15	Stable
PCRE 8.35	MariaDB 10.1.0, MariaDB 10.0.12	Stable
PCRE 8.34	MariaDB 10.0.8	Stable

PCRE Enhancements

MariaDB 10.0.5 switched to the PCRE library, which significantly improved the power of the REGEXP/RLIKE operator.

The switch to PCRE added a number of features, including recursive patterns, named capture, look-ahead and look-behind assertions, non-capturing groups, non-greedy quantifiers, Unicode character properties, extended syntax for characters and character classes, multi-line matching, and many other.

Additionally, MariaDB 10.0.5 introduced three new functions that work with regular expressions: REGEXP_REPLACE(), REGEXP_INSTR() and REGEXP_SUBSTR().

Also, REGEXP/RLIKE, and the new functions, now work correctly with all multi-byte character sets supported by MariaDB, including East-Asian character sets (big5, gb2313, gbk, eucjp, eucjpms, cp932, ujis, euckr), and Unicode character sets (utf8, utf8mb4, ucs2, utf16, utf16le, utf32). In earlier versions of MariaDB (and all MySQL versions) REGEXP/RLIKE works correctly only with 8-bit character sets.

New Regular Expression Functions

- REGEXP_REPLACE(subject, pattern, replace) - Replaces all occurrences of a pattern.
- REGEXP_INSTR(subject, pattern) - Position of the first appearance of a regex .
- REGEXP_SUBSTR(subject,pattern) - Returns the matching part of a string.

See the individual articles for more details and examples.

PCRE Syntax

In most cases PCRE is backward compatible with the old POSIX 1003.2 compliant regexp library (see Regular Expressions Overview), so you won't need to change your applications that use SQL queries with the REGEXP/RLIKE predicate.

MariaDB 10.0.11 introduced the default_regex_flags variable to address the remaining compatibilities between PCRE and the old regex library.

This section briefly describes the most important extended PCRE features. For more details please refer to the documentation on the PCRE site, or to the documentation which is bundled in the /pcre/doc/html/ directory of a MariaDB sources distribution. The pages pcresyntax.html and pcrepattern.html should be a good start. Regular-Expressions.Info is another good resource to learn about PCRE and regular expressions generally.

Special Characters

PCRE supports the following escape sequences to match special characters:

Sequence	Description
\a	0x07 (BEL)
\cx	"control-x", where x is any ASCII character
\e	0x1B (escape)
\f	0x0C (form feed)
\n	0x0A (newline)
\r	0x0D (carriage return)
\t	0x09 (TAB)
\ddd	character with octal code ddd
\xhh	character with hex code hh
\x{hhh..}	character with hex code hhh..

Note, the backslash characters (here, and in all examples in the sections below) must be escaped with another backslash, unless you're using the SQL_MODE `NO_BACKSLASH_ESCAPES` .

This example tests if a character has hex code 0x61:

```
SELECT 'a' RLIKE '\\x{61}';
-> 1
```

Character Classes

PCRE supports the standard POSIX character classes such as `alnum` , `alpha` , `blank` , `cntrl` , `digit` , `graph` , `lower` , `print` , `punct` , `space` , `upper` , `xdigit` , with the following additional classes:

Class	Description
ascii	any ASCII character (0x00..0x7F)

word	any "word" character (a letter, a digit, or an underscore)

This example checks if the string consists of ASCII characters only:

```
SELECT 'abc' RLIKE '^[[:ascii:]]+$';
-> 1
```

Generic Character Types

Generic character types complement the POSIX character classes and serve to simplify writing patterns:

Class	Description
\d	a decimal digit (same as [:digit:])
\D	a character that is not a decimal digit
\h	a horizontal white space character
\H	a character that is not a horizontal white space character
\N	a character that is not a new line
\R	a newline sequence
\s	a white space character
\S	a character that is not a white space character
\v	a vertical white space character
\V	a character that is not a vertical white space character
\w	a "word" character (same as [:word:])
\W	a "non-word" character

This example checks if the string consists of "word" characters only:

```
SELECT 'abc' RLIKE '^\\w+$';
-> 1
```

Unicode Character Properties

`\p{xx}` is a character with the `xx` property, and `\P{xx}` is a character without the `xx` property.

The property names represented by `xx` above are limited to the Unicode script names, the general category properties, and "Any", which matches any character (including newline). Those that are not part of an identified script are lumped together as "Common".

General Category Properties For \p and \P

Property	Description
C	Other
Cc	Control
Cf	Format
Cn	Unassigned
Co	Private use
Cs	Surrogate
L	Letter
Ll	Lower case letter
Lm	Modifier letter
Lo	Other letter
Lt	Title case letter

Lu	Upper case letter
L&	Ll, Lu, or Lt
M	Mark
Mc	Spacing mark
Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P	Punctuation
Pc	Connector punctuation
Pd	Dash punctuation
Pe	Close punctuation
Pf	Final punctuation
Pi	Initial punctuation
Po	Other punctuation
Ps	Open punctuation
S	Symbol
Sc	Currency symbol
Sk	Modifier symbol
Sm	Mathematical symbol
So	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator
Zs	Space separator

This example checks if the string consists only of characters with property N (number):

```
SELECT '1¼①' RLIKE '^\\p{N}+$';
-> 1
```

Special Category Properties For \p and \P

Property	Description
Xan	Alphanumeric: union of properties L and N
Xps	POSIX space: property Z or tab, NL, VT, FF, CR
Xsp	Perl space: property Z or tab, NL, FF, CR
Xuc	A character than can be represented by a Universal Character Name
Xwd	Perl word: property Xan or underscore

The property `Xuc` matches any character that can be represented by a Universal Character Name (in C++ and other programming languages). These include `$`, `@`, `` ` ``, and all characters with Unicode code points greater than `U+00A0`, excluding the surrogates `U+D800` .. `U+DFFF`.

Script Names For \p and \P

Arabic, Armenian, Avestan, Balinese, Bamum, Batak, Bengali, Bopomofo, Brahmi, Braille, Buginese, Buhid,

Canadian_Aboriginal, Carian, Chakma, Cham, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Egyptian_Hieroglyphs, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Imperial_Aramaic, Inherited, Inscriptional_Pahlavi, Inscriptional_Parthian, Javanese, Kaithi, Kannada, Katakana, Kayah_Li, Kharoshthi, Khmer, Lao, Latin, Lepcha, Limbu, Linear_B, Lisu, Lycian, Lydian, Malayalam, Mandaic, Meetei_Mayek, Meroitic_Cursive, Meroitic_Hieroglyphs, Miao, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Old_South_Arabian, Old_Turkic, Ol_Chiki, Oriya, Osmanya, Phags_Pa, Phoenician, Rejang, Runic, Samaritan, Saurashtra, Sharada, Shavian, Sinhala, Sora_Sompeng, Sundanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tai_Tham, Tai_Viet, Takri, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Vai, Yi.

This example checks if the string consists only of Greek characters:

```
SELECT 'ΣΦΩ' RLIKE '^\\p{Greek}+$';
-> 1
```

Extended Unicode Grapheme Sequence

The `\X` escape sequence matches a character sequence that makes an "extended grapheme cluster", i.e. a composite character that consists of multiple Unicode code points.

One of the examples of a composite character can be a letter followed by non-spacing accent marks. This example demonstrates that `U+0045 LATIN CAPITAL LETTER E` followed by `U+0302 COMBINING CIRCUMFLEX ACCENT` followed by `U+0323 COMBINING DOT BELOW` together form an extended grapheme cluster:

```
SELECT _ucs2 0x004503020323 RLIKE '^\\X$';
-> 1
```

See the PCRE documentation for the other types of extended grapheme clusters.

Simple Assertions

An assertion specifies a certain condition that must match at a particular point, but without consuming characters from the subject string. In addition to the standard POSIX simple assertions `^` (that matches at the beginning of a line) and `$` (that matches at the end of a line), PCRE supports a number of other assertions:

Assertion	Description
\b	matches at a word boundary
\B	matches when not at a word boundary
\A	matches at the start of the subject
\Z	matches at the end of the subject, also matches before a newline at the end of the subject
\z	matches only at the end of the subject
\G	matches at the first matching position in the subject

This example cuts a word that consists only of 3 characters from a string:

```
SELECT REGEXP_SUBSTR('---abcd---xyz---', '\\b\\w{3}\\b');
-> xyz
```

Notice that the two `\b` assertions checked the word boundaries but did not get into the matching pattern.

The `\b` assertions work well in the beginning and the end of the subject string:

```
SELECT REGEXP_SUBSTR('xyz', '\\b\\w{3}\\b');
-> xyz
```

By default, the `^` and `$` assertions have the same meaning with `\A`, `\Z`, and `\z`. However, the meanings of `^` and `$` can change in multiline mode (see below). By contrast, the meanings of `\A`, `\Z`, and `\z` are always the same; they are independent of the multiline mode.

Option Setting

A number of options that control the default match behavior can be changed within the pattern by a sequence of option letters enclosed between `(?` and `)`.

Option	Description
(?i)	case insensitive match
(?m)	multiline mode
(?s)	dotall mode (dot matches newline characters)
(?x)	extended (ignore white space)
(?U)	ungreedy (lazy) match
(?J)	allow named subpatterns with duplicate names
(?X)	extra PCRE functionality (e.g. force error on unknown escaped character)
(?-...)	unset option(s)

For example, `(?im)` sets case insensitive multiline matching.

A hyphen followed by the option letters unset the options. For example, `(?-im)` means case sensitive single line match.

A combined setting and unsetting is also possible, e.g. `(?im-sx)`.

If an option is set outside of subpattern parentheses, the option applies to the remainder of the pattern that follows the option. If an option is set inside a subpattern, it applies to the part of this subpattern that follows the option.

In this example the pattern `(?i)m((?-i)aria)db` matches the words `MariaDB`, `Mariadb`, `mariadb`, but not `MARIADB`:

```
SELECT 'MariaDB' RLIKE '(?i)m((?-i)aria)db';
-> 1

SELECT 'mariaDB' RLIKE '(?i)m((?-i)aria)db';
-> 1

SELECT 'Mariadb' RLIKE '(?i)m((?-i)aria)db';
-> 1

SELECT 'MARIADB' RLIKE '(?i)m((?-i)aria)db';
-> 0
```

This example demonstrates that the `(?x)` option makes the regexp engine ignore all white spaces in the pattern (other than in a class).

```
SELECT 'ab' RLIKE '(?x)a b';
-> 1
```

Note, putting spaces into a pattern in combination with the (?x) option can be useful to split different logical parts of a complex pattern, to make it more readable.

Multiline Matching

Multiline matching changes the meaning of `^` and `$` from "the beginning of the subject string" and "the end of the subject string" to "the beginning of any line in the subject string" and "the end of any line in the subject string" respectively.

This example checks if the subject string contains two consequent lines that fully consist of digits:

```
SELECT 'abc\n123\n456\nxyz\n' RLIKE '(?m)^\\d+\\R\\d+$';
-> 1
```

Notice the `(?m)` option in the beginning of the pattern, which switches to the multiline matching mode.

Newline Conventions

PCRE supports five line break conventions:

- `CR (\r)` - a single carriage return character
- `LF (\n)` - a single linefeed character
- `CRLF (\r\n)` - a carriage return followed by a linefeed
- any of the previous three

- any Unicode newline sequence

By default, the newline convention is set to any Unicode newline sequence, which includes:

Sequence	Description
LF	(U+000A, carriage return)
CR	(U+000D, carriage return)
CRLF	(a carriage return followed by a linefeed)
VT	(U+000B, vertical tab)
FF	(U+000C, form feed)
NEL	(U+0085, next line)
LS	(U+2028, line separator)
PS	(U+2029, paragraph separator)

The newline convention can be set by starting a pattern with one of the following sequences:

Sequence	Description
(*CR)	carriage return
(*LF)	linefeed
(*CRLF)	carriage return followed by linefeed
(*ANYCRLF)	any of the previous three
(*ANY)	all Unicode newline sequences

The newline conversion affects the `^` and `$` assertions, the interpretation of the dot metacharacter, and the behavior of `\N`.

Note, the new line convention does not affect the meaning of `\R`.

This example demonstrates that the dot metacharacter matches `\n`, because it is not a newline sequence anymore:

```
SELECT 'a\nb' RLIKE '(*CR)a.b';
-> 1
```

Newline Sequences

By default, the escape sequence `\R` matches any Unicode newline sequences.

The meaning of `\R` can be set by starting a pattern with one of the following sequences:

Sequence	Description
(*BSR_ANYCRLF)	any of CR, LF or CRLF
(*BSR_UNICODE)	any Unicode newline sequence

Comments

It's possible to include comments inside a pattern. Comments do not participate in the pattern matching. Comments start at the `(?#` sequence and continue up to the next closing parenthesis:

```
SELECT 'ab12' RLIKE 'ab(?#expect digits)12';
-> 1
```

Quoting

POSIX uses the backslash to remove a special meaning from a character. PCRE introduces a syntax to remove special meaning from a sequence of characters. The characters inside `\Q` ... `\E` are treated literally, without their special meaning.

This example checks if the string matches a dollar sign followed by a parenthesized name (a variable reference in some languages):

```
SELECT '$(abc)' RLIKE '^\\Q$(\\E\\w+\\Q)\\E$';
-> 1
```

Note that the leftmost dollar sign and the parentheses are used literally, while the rightmost dollar sign is still used to match the end of the string.

Resetting the Match Start

The escape sequence `\K` causes any previously matched characters to be excluded from the final matched sequence. For example, the pattern: `(foo)\Kbar` matches `foobar`, but reports that it has matched `bar`. This feature is similar to a look-behind assertion. However, in this case, the part of the subject before the real match does not have to be of fixed length:

```
SELECT REGEXP_SUBSTR('aaa123', '[a-z]*\\K[0-9]*');
-> 123
```

Non-Capturing Groups

The question mark and the colon after the opening parenthesis create a non-capturing group: `(?:...)`.

This example removes an optional article from a word, for example for better sorting of the results.

```
SELECT REGEXP_REPLACE('The King','(?:the|an|a)[^a-z]([a-z]+)','\\1');
-> King
```

Note that the articles are listed inside the left parentheses using the alternation operator `|` but they do not produce a captured subpattern, so the word followed by the article is referenced by `'1'` in the third argument to the function. Using non-capturing groups can be useful to save numbers on the sup-patterns that won't be used in the third argument of REGEXP_REPLACE(), as well as for performance purposes.

Non-Greedy Quantifiers

By default, the repetition quantifiers `?`, `*`, `+` and `{n,m}` are "greedy", that is, they try to match as much as possible. Adding a question mark after a repetition quantifier makes it "non-greedy", so the pattern matches the minimum number of times possible.

This example cuts C comments from a line:

```
SELECT REGEXP_REPLACE('/* Comment1 */ i+= 1; /* Comment2 */', '/[*].*?[*]/','');
->  i+= 1;
```

The pattern without the non-greedy flag to the quantifier `/[*].*[*]/` would match the entire string between the leftmost `/*` and the rightmost `*/`.

Atomic Groups

A sequence inside `(?>...)` makes an atomic group. Backtracking inside an atomic group is prevented once it has matched; however, backtracking past to the previous items works normally.

Consider the pattern `\d+foo` applied to the subject string `123bar`. Once the engine scans `123` and fails on the letter `b`, it would normally backtrack to `2` and try to match again, then fail and backtrack to `1` and try to match and fail again, and finally fail the entire pattern. In case of an atomic group `(?>\d+)foo` with the same subject string `123bar`, the engine gives up immediately after the first failure to match `foo`. An atomic group with a quantifier can match all or nothing.

Atomic groups produce faster false results (i.e. in case when a long subject string does not match the pattern), because the regexp engine saves performance on backtracking. However, don't hurry to put everything into atomic groups. This example demonstrates the difference between atomic and non-atomic match:

```
SELECT 'abcc' RLIKE 'a(?>bc|b)c' AS atomic1;
-> 1

SELECT 'abc' RLIKE 'a(?>bc|b)c' AS atomic2;
-> 0

SELECT 'abcc' RLIKE 'a(bc|b)c' AS non_atomic1;
-> 1

SELECT 'abc' RLIKE 'a(bc|b)c' AS non_atomic2;
-> 1
```

The non-atomic pattern matches both `abbc` and `abc`, while the atomic pattern matches `abbc` only.

The atomic group `(?>bc|b)` in the above example can be "translated" as "if there is `bc`, then don't try to match as `b`". So `b` can match only if `bc` is not found.

Atomic groups are not capturing. To make an atomic group capturing, put it into parentheses:

```
SELECT REGEXP_REPLACE('abcc','a((?>bc|b))c','\\1');
-> bc
```

Possessive quantifiers

An atomic group which ends with a quantifier can be rewritten using a so called "possessive quantifier" syntax by putting an additional `+` sign following the quantifier.

The pattern `(?>\d+)foo` from the previous section's example can be rewritten as `\d++foo`.

Absolute and Relative Numeric Backreferences

Backreferences match the same text as previously matched by a capturing group. Backreferences can be written using:

- a backslash followed by a digit
- the `\g` escape sequence followed by a positive or negative number
- the `\g` escape sequence followed by a positive or negative number enclosed in braces

The following backreferences are identical and refer to the first capturing group:

- `\1`
- `\g1`
- `\g{1}`

This example demonstrates a pattern that matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility":

```
SELECT 'sense and sensibility' RLIKE '(sens|respons)e and \\1ibility';
-> 1
```

This example removes doubled words that can unintentionally creep in when you edit a text in a text editor:

```
SELECT REGEXP_REPLACE('using using the the regexp regexp',
 '\\b(\\w+)\\s+\\1\\b','\\1');
-> using the regexp
```

Note that all double words were removed, in the beginning, in the middle and in the end of the subject string.

A negative number in a `\g` sequence means a relative reference. Relative references can be helpful in long patterns, and also in patterns that are created by joining fragments together that contain references within themselves. The sequence `\g{-1}` is a reference to the most recently started capturing subpattern before `\g`.

In this example `\g{-1}` is equivalent to `\2`:

```
SELECT 'abc123def123' RLIKE '(abc(123)def)\\g{-1}';
-> 1

SELECT 'abc123def123' RLIKE '(abc(123)def)\\2';
-> 1
```

Named Subpatterns and Backreferences

Using numeric backreferences for capturing groups can be hard to track in a complicated regular expression. Also, the numbers can change if an expression is modified. To overcome these difficulties, PCRE supports named subpatterns.

A subpattern can be named in one of three ways: `(?<name> ... )` or `(?'name' ... )` as in Perl, or `(?P<name> ... )` as in Python. References to capturing subpatterns from other parts of the pattern, can be made by name as well as by number.

Backreferences to a named subpattern can be written using the .NET syntax `\k{name}`, the Perl syntax `\k<name>` or `\k'name'` or `\g{name}`, or the Python syntax `(?P=name)`.

This example tests if the string is a correct HTML tag:

```
SELECT '<a href="../">Up</a>' RLIKE '<(?<tag>[a-z][a-z0-9]*)[^>]*>[^<]*</(?P=tag)>';
-> 1
```

Positive and Negative Look-Ahead and Look-Behind Assertions

Look-ahead and look-behind assertions serve to specify the context for the searched regular expression pattern. Note that the assertions only check the context, they do not capture anything themselves!

This example finds the letter which is not followed by another letter (negative look-ahead):

```
SELECT REGEXP_SUBSTR('ab1','[a-z](?![a-z])');
-> b
```

This example finds the letter which is followed by a digit (positive look-ahead):

```
SELECT REGEXP_SUBSTR('ab1','[a-z](?=[0-9])');
-> b
```

This example finds the letter which does not follow a digit character (negative look-behind):

```
SELECT REGEXP_SUBSTR('1ab','(?<![0-9])[a-z]');
-> b
```

This example finds the letter which follows another letter character (positive look-behind):

```
SELECT REGEXP_SUBSTR('1ab','(?<=[a-z])[a-z]');
-> b
```

Note that look-behind assertions can only be of fixed length; you cannot have repetition operators or alternations with different lengths:

```
SELECT 'aaa' RLIKE '(?<=(a|bc))a';
ERROR 1139 (42000): Got error 'lookbehind assertion is not fixed length at offset 10' from regexp
```

Subroutine Reference and Recursive Patterns

PCRE supports a special syntax to recourse the entire pattern or its individual subpatterns:

Syntax	Description
(?R)	Recourse the entire pattern
(?n)	call subpattern by absolute number
(?+n)	call subpattern by relative number
(?-n)	call subpattern by relative number
(?&name)	call subpattern by name (Perl)
(?P>name)	call subpattern by name (Python)
\g<name>	call subpattern by name (Oniguruma)
\g'name'	call subpattern by name (Oniguruma)

\g<n>	call subpattern by absolute number (Oniguruma)
\g'n'	call subpattern by absolute number (Oniguruma)
\g<+n>	call subpattern by relative number
\g<-n>	call subpattern by relative number
\g'+n'	call subpattern by relative number
\g'-n'	call subpattern by relative number

This example checks for a correct additive arithmetic expression consisting of numbers, unary plus and minus, binary plus and minus, and parentheses:

```
SELECT '1+2-3+(+(4-1)+(-2)+(+1))' RLIKE  '^(([+-]?(\\d+|[(](?1)[)]))(([+-](?1))*))$';
-> 1
```

The recursion is done using `(?1)` to call for the first parenthesized subpattern, which includes everything except the leading `^` and the trailing `$`.

The regular expression in the above example implements the following BNF grammar:

1. `<expression> ::= <term> [(<sign> <term>)...]`
2. `<term> ::= [ <sign> ] <primary>`
3. `<primary> ::= <number> | <left paren> <expression> <right paren>`
4. `<sign> ::= <plus sign> | <minus sign>`

Defining Subpatterns For Use By Reference

Use the `(?(DEFINE) ... )` syntax to define subpatterns that can be referenced from elsewhere.

This example defines a subpattern with the name `letters` that matches one or more letters, which is further reused two times:

```
SELECT 'abc123xyz' RLIKE '^(?(DEFINE)(?<letters>[a-z]+))(?&letters)[0-9]+(?&letters)$';
-> 1
```

The above example can also be rewritten to define the digit part as a subpattern as well:

```
SELECT 'abc123xyz' RLIKE
 '^(?(DEFINE)(?<letters>[a-z]+)(?<digits>[0-9]+))(?&letters)(?&digits)(?&letters)$';
-> 1
```

Conditional Subpatterns

There are two forms of conditional subpatterns:

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

The `yes-pattern` is used if the condition is satisfied, otherwise the `no-pattern` (if any) is used.

Conditions With Subpattern References

If a condition consists of a number, it makes a condition with a subpattern reference. Such a condition is true if a capturing subpattern corresponding to the number has previously matched.

This example finds an optionally parenthesized number in a string:

```
SELECT REGEXP_SUBSTR('a(123)b', '([(])?[0-9]+(?(1)[)])');
-> (123)
```

The `([(])?` part makes a capturing subpattern that matches an optional opening parenthesis; the `[0-9]+` part matches a number, and the `(?(1)[)])` part matches a closing parenthesis, but only if the opening parenthesis has been previously found.

Other Kinds of Conditions

The other possible condition kinds are: recursion references and assertions. See the PCRE documentation for details.

Matching Zero Bytes (0x00)

PCRE correctly works with zero bytes in the subject strings:

```
SELECT 'a\0b' RLIKE '^a.b$';
-> 1
```

Zero bytes, however, are not supported literally in the pattern strings and should be escaped using the `\xhh` or `\x{hh}` syntax:

```
SELECT 'a\0b' RLIKE '^a\\x{00}b$';
-> 1
```

Other PCRE Features

PCRE provides other extended features that were not covered in this document, such as duplicate subpattern numbers, backtracking control, breaking utf-8 sequences into individual bytes, setting the match limit, setting the recursion limit, optimization control, recursion conditions, assertion conditions and more types of extended grapheme clusters. Please refer to the PCRE documentation for details.

Enhanced regex was implemented as a GSoC 2013 project by Sudheera Palihakkara.

default_regex_flags Examples

The default_regex_flags variable was introduced to address the remaining incompatibilities between PCRE and the old regex library. Here are some examples of its usage:

The default behaviour (multiline match is off)

```
SELECT 'a\nb\nc' RLIKE '^b$';
+----------------------------+
| '(?m)a\nb\nc' RLIKE '^b$' |
+----------------------------+
|                          0 |
+----------------------------+
```

Enabling the multiline option using the PCRE option syntax:

```
SELECT 'a\nb\nc' RLIKE '(?m)^b$';
+---------------------------+
| 'a\nb\nc' RLIKE '(?m)^b$' |
+---------------------------+
|                         1 |
+---------------------------+
```

Enabling the miltiline option using default_regex_flags

```
SET default_regex_flags='MULTILINE';
SELECT 'a\nb\nc' RLIKE '^b$';
+-----------------------+
| 'a\nb\nc' RLIKE '^b$' |
+-----------------------+
|                     1 |
+-----------------------+
```

1.2.2.1.3 NOT REGEXP

Syntax

```
expr NOT REGEXP pat, expr NOT RLIKE pat
```

Description

This is the same as NOT (expr REGEXP pat).

1.2.2.1.4 REGEXP

Syntax

```
expr REGEXP pat, expr RLIKE pat
```

Description

Performs a pattern match of a string expression `expr` against a pattern `pat`. The pattern can be an extended regular expression. See Regular Expressions Overview for details on the syntax for regular expressions (see also PCRE Regular Expressions).

Returns `1` if `expr` matches `pat` or `0` if it doesn't match. If either `expr` or `pat` are NULL, the result is NULL.

The negative form NOT REGEXP also exists, as an alias for `NOT (string REGEXP pattern)`. RLIKE and NOT RLIKE are synonyms for REGEXP and NOT REGEXP, originally provided for mSQL compatibility.

The pattern need not be a literal string. For example, it can be specified as a string expression or table column.

Note: Because MariaDB uses the C escape syntax in strings (for example, "\n" to represent the newline character), you must double any "\" that you use in your REGEXP strings.

REGEXP is not case sensitive, except when used with binary strings.

MariaDB 10.0.5 moved to the PCRE regex library - see PCRE Regular Expressions for enhancements to REGEXP introduced in MariaDB 10.0.5.

The default_regex_flags variable addresses the remaining compatibilities between PCRE and the old regex library.

Examples

```
SELECT 'Monty!' REGEXP 'm%y%%';
+-------------------------+
| 'Monty!' REGEXP 'm%y%%' |
+-------------------------+
|                       0 |
+-------------------------+

SELECT 'Monty!' REGEXP '.*';
+----------------------+
| 'Monty!' REGEXP '.*' |
+----------------------+
|                    1 |
+----------------------+

SELECT 'new*\n*line' REGEXP 'new\\*.\\*line';
+---------------------------------------+
| 'new*\n*line' REGEXP 'new\\*.\\*line' |
+---------------------------------------+
|                                     1 |
+---------------------------------------+

SELECT 'a' REGEXP 'A', 'a' REGEXP BINARY 'A';
+----------------+-----------------------+
| 'a' REGEXP 'A' | 'a' REGEXP BINARY 'A' |
+----------------+-----------------------+
|              1 |                     0 |
+----------------+-----------------------+

SELECT 'a' REGEXP '^[a-d]';
+---------------------+
| 'a' REGEXP '^[a-d]' |
+---------------------+
|                   1 |
+---------------------+
```

default_regex_flags examples

MariaDB 10.0.11 introduced the default_regex_flags variable to address the remaining compatibilities between PCRE and the old regex library.

The default behaviour (multiline match is off)

```
SELECT 'a\nb\nc' RLIKE '^b$';
+---------------------------+
| '(?m)a\nb\nc' RLIKE '^b$' |
+---------------------------+
|                         0 |
+---------------------------+
```

Enabling the multiline option using the PCRE option syntax:

```
SELECT 'a\nb\nc' RLIKE '(?m)^b$';
+---------------------------+
| 'a\nb\nc' RLIKE '(?m)^b$' |
+---------------------------+
|                         1 |
+---------------------------+
```

Enabling the multiline option using default_regex_flags

```
SET default_regex_flags='MULTILINE';
SELECT 'a\nb\nc' RLIKE '^b$';
+-----------------------+
| 'a\nb\nc' RLIKE '^b$' |
+-----------------------+
|                     1 |
+-----------------------+
```

1.2.2.1.5 REGEXP_INSTR

Syntax

```
REGEXP_INSTR(subject, pattern)
```

Returns the position of the first occurrence of the regular expression `pattern` in the string `subject`, or 0 if pattern was not found.

The positions start with 1 and are measured in characters (i.e. not in bytes), which is important for multi-byte character sets. You can cast a multi-byte character set to BINARY to get offsets in bytes.

The function follows the case sensitivity rules of the effective collation. Matching is performed case insensitively for case insensitive collations, and case sensitively for case sensitive collations and for binary data.

The collation case sensitivity can be overwritten using the (?i) and (?-i) PCRE flags.

MariaDB uses the PCRE regular expression library for enhanced regular expression performance, and REGEXP_INSTR was introduced as part of this enhancement.

Examples

```
SELECT REGEXP_INSTR('abc','b');
-> 2

SELECT REGEXP_INSTR('abc','x');
-> 0

SELECT REGEXP_INSTR('BJÖRN','N');
-> 5
```

Casting a multi-byte character set as BINARY to get offsets in bytes:

```
SELECT REGEXP_INSTR(BINARY 'BJÖRN','N') AS cast_utf8_to_binary;
-> 6
```

Case sensitivity:

```
SELECT REGEXP_INSTR('ABC','b');
-> 2

SELECT REGEXP_INSTR('ABC' COLLATE utf8_bin,'b');
-> 0

SELECT REGEXP_INSTR(BINARY'ABC','b');
-> 0

SELECT REGEXP_INSTR('ABC','(?-i)b');
-> 0

SELECT REGEXP_INSTR('ABC' COLLATE utf8_bin,'(?i)b');
-> 2
```

1.2.2.1.6 REGEXP_REPLACE

Syntax

```
REGEXP_REPLACE(subject, pattern, replace)
```

Description

`REGEXP_REPLACE` returns the string `subject` with all occurrences of the regular expression `pattern` replaced by the string `replace`. If no occurrences are found, then `subject` is returned as is.

The replace string can have backreferences to the subexpressions in the form \N, where N is a number from 1 to 9.

The function follows the case sensitivity rules of the effective collation. Matching is performed case insensitively for case insensitive collations, and case sensitively for case sensitive collations and for binary data.

The collation case sensitivity can be overwritten using the (?i) and (?-i) PCRE flags.

MariaDB uses the PCRE regular expression library for enhanced regular expression performance, and `REGEXP_REPLACE` was introduced as part of this enhancement.

The default_regex_flags variable addresses the remaining compatibilities between PCRE and the old regex library.

Examples

```
SELECT REGEXP_REPLACE('ab12cd','[0-9]','') AS remove_digits;
-> abcd

SELECT REGEXP_REPLACE('<html><head><title>title</title><body>body</body></htm>', '<.+?>',' ')
AS strip_html;
-> title  body
```

Backreferences to the subexpressions in the form `\N`, where N is a number from 1 to 9:

```
SELECT REGEXP_REPLACE('James Bond','^(.*) (.*)$','\\2, \\1') AS reorder_name;
-> Bond, James
```

Case insensitive and case sensitive matches:

```
SELECT REGEXP_REPLACE('ABC','b','-') AS case_insensitive;
-> A-C

SELECT REGEXP_REPLACE('ABC' COLLATE utf8_bin,'b','-') AS case_sensitive;
-> ABC

SELECT REGEXP_REPLACE(BINARY 'ABC','b','-') AS binary_data;
-> ABC
```

Overwriting the collation case sensitivity using the (?i) and (?-i) PCRE flags.

```
SELECT REGEXP_REPLACE('ABC','(?-i)b','-') AS force_case_sensitive;
-> ABC

SELECT REGEXP_REPLACE(BINARY 'ABC','(?i)b','-') AS force_case_insensitive;
-> A-C
```

1.2.2.1.7 REGEXP_SUBSTR

Syntax

```
REGEXP_SUBSTR(subject,pattern)
```

Description

Returns the part of the string `subject` that matches the regular expression `pattern`, or an empty string if `pattern` was not found.

The function follows the case sensitivity rules of the effective collation. Matching is performed case insensitively for case insensitive collations, and case sensitively for case sensitive collations and for binary data.

The collation case sensitivity can be overwritten using the (?i) and (?-i) PCRE flags.

MariaDB uses the PCRE regular expression library for enhanced regular expression performance, and `REGEXP_SUBSTR`

was introduced as part of this enhancement.

The default_regex_flags variable addresses the remaining compatibilities between PCRE and the old regex library.

Examples

```
SELECT REGEXP_SUBSTR('ab12cd','[0-9]+');
-> 12

SELECT REGEXP_SUBSTR(
  'See https://mariadb.org/en/foundation/ for details',
  'https?://[^/]*');
-> https://mariadb.org
```

```
SELECT REGEXP_SUBSTR('ABC','b');
-> B

SELECT REGEXP_SUBSTR('ABC' COLLATE utf8_bin,'b');
->

SELECT REGEXP_SUBSTR(BINARY'ABC','b');
->

SELECT REGEXP_SUBSTR('ABC','(?i)b');
-> B

SELECT REGEXP_SUBSTR('ABC' COLLATE utf8_bin,'(?+i)b');
-> B
```

1.2.2.1.8 RLIKE

Syntax

```
expr REGEXP pat, expr RLIKE pat
```

Description

`RLIKE` is a synonym for REGEXP.

1.2.2.2 Dynamic Columns Functions

Dynamic columns is a feature that allows one to store different sets of columns for each row in a table. It works by storing a set of columns in a blob and having a small set of functions to manipulate it.

COLUMN_ADD

Adds or updates dynamic columns.

COLUMN_CHECK

Checks if a dynamic column blob is valid

COLUMN_CREATE

Returns a dynamic columns blob.

COLUMN_DELETE

Deletes a dynamic column.

COLUMN_EXISTS

Checks is a column exists.

COLUMN_GET

Gets a dynamic column value by name.

COLUMN_JSON

Returns a JSON representation of dynamic column blob data

COLUMN_LIST

Returns comma-separated list of columns names.

1.2.2.2.1 COLUMN_ADD

Syntax

```
COLUMN_ADD(dyncol_blob, column_nr, value [as type], [column_nr, value [as type]]...);
COLUMN_ADD(dyncol_blob, column_name, value [as type], [column_name, value [as type]]...);
```

Description

Adds or updates dynamic columns.

- **`dyncol_blob`** must be either a valid dynamic columns blob (for example, `COLUMN_CREATE` returns such blob), or an empty string.
- **`column_name`** specifies the name of the column to be added. If `dyncol_blob` already has a column with this name, it will be overwritten.
- **`value`** specifies the new value for the column. Passing a NULL value will cause the column to be deleted.
- **`as type`** is optional. See #datatypes section for a discussion about types.

The return value is a dynamic column blob after the modifications.

Examples

```
UPDATE t1 SET dyncol_blob=COLUMN_ADD(dyncol_blob, "column_name", "value") WHERE id=1;
```

Note: `COLUMN_ADD()` is a regular function (just like `CONCAT()`), hence, in order to update the value in the table you have to use the `UPDATE ... SET dynamic_col=COLUMN_ADD(dynamic_col, ....)` pattern.

1.2.2.2.2 COLUMN_CHECK

Syntax

```
COLUMN_CHECK(dyncol_blob);
```

Description

Check if `dyncol_blob` is a valid packed dynamic columns blob. Return value of 1 means the blob is valid, return value of 0 means it is not.

Rationale: Normally, one works with valid dynamic column blobs. Functions like COLUMN_CREATE, COLUMN_ADD, COLUMN_DELETE always return valid dynamic column blobs. However, if a dynamic column blob is accidentally truncated, or transcoded from one character set to another, it will be corrupted. This function can be used to check if a value in a blob field is a valid dynamic column blob.

1.2.2.2.3 COLUMN_CREATE

Syntax

```
COLUMN_CREATE(column_nr, value [as type], [column_nr, value [as type]]...);
COLUMN_CREATE(column_name, value [as type], [column_name, value [as type]]...);
```

Description

Returns a dynamic columns blob that stores the specified columns with values.

The return value is suitable for

- storing in a table
- further modification with other dynamic columns functions

The `as type` part allows one to specify the value type. In most cases, this is redundant because MariaDB will be able to deduce the type of the value. Explicit type specification may be needed when the type of the value is not apparent. For example, a literal `'2012-12-01'` has a CHAR type by default, one will need to specify `'2012-12-01' AS DATE` to have it stored as a date. See Dynamic Columns:Datatypes for further details.

Examples

```
INSERT INTO tbl SET dyncol_blob=COLUMN_CREATE("column_name", "value");
```

1.2.2.2.4 COLUMN_DELETE

Syntax

```
COLUMN_DELETE(dyncol_blob, column_nr, column_nr...);
COLUMN_DELETE(dyncol_blob, column_name, column_name...);
```

Description

Deletes a dynamic column with the specified name. Multiple names can be given. The return value is a dynamic column blob after the modification.

1.2.2.2.5 COLUMN_EXISTS

Syntax

```
COLUMN_EXISTS(dyncol_blob, column_nr);
COLUMN_EXISTS(dyncol_blob, column_name);
```

Description

Checks if a column with name `column_name` exists in `dyncol_blob`. If yes, return `1`, otherwise return `0`. See dynamic columns for more information.

1.2.2.2.6 COLUMN_GET

Syntax

```
COLUMN_GET(dyncol_blob, column_nr as type);
COLUMN_GET(dyncol_blob, column_name as type);
```

Description

Gets the value of a dynamic column by its name. If no column with the given name exists, `NULL` will be returned.

`column_name as type` requires that one specify the datatype of the dynamic column they are reading.

This may seem counter-intuitive: why would one need to specify which datatype they're retrieving? Can't the dynamic columns system figure the datatype from the data being stored?

The answer is: SQL is a statically-typed language. The SQL interpreter needs to know the datatypes of all expressions before the query is run (for example, when one is using prepared statements and runs `"select COLUMN_GET(...)"`, the prepared statement API requires the server to inform the client about the datatype of the column being read before the query is executed and the server can see what datatype the column actually has).

Lengths

If you're running queries like:

```
SELECT COLUMN_GET(blob, 'colname' as CHAR) ...
```

without specifying a maximum length (i.e. using `as CHAR`, not `as CHAR(n)`), MariaDB will report the maximum length of the resultset column to be 16,777,216. This may cause excessive memory usage in some client libraries, because they try to pre-allocate a buffer of maximum resultset width. To avoid this problem, use CHAR(n) whenever you're using COLUMN_GET in the select list.

See Dynamic Columns:Datatypes for more information about datatypes.

1.2.2.2.7 COLUMN_JSON

Syntax

```
COLUMN_JSON(dyncol_blob)
```

Description

Returns a JSON representation of data in `dyncol_blob`. Can also be used to display nested columns. See dynamic columns for more information.

Example

```
select item_name, COLUMN_JSON(dynamic_cols) from assets;
+----------------+---------------------------------------+
| item_name      | COLUMN_JSON(dynamic_cols)             |
+----------------+---------------------------------------+
| MariaDB T-shirt | {"size":"XL","color":"blue"}          |
| Thinkpad Laptop | {"color":"black","warranty":"3 years"} |
+----------------+---------------------------------------+
```

Limitation: `COLUMN_JSON` will decode nested dynamic columns at a nesting level of not more than 10 levels deep. Dynamic columns that are nested deeper than 10 levels will be shown as BINARY string, without encoding.

1.2.2.2.8 COLUMN_LIST

Syntax

```
COLUMN_LIST(dyncol_blob);
```

Description

Returns a comma-separated list of column names. The names are quoted with backticks.

See dynamic columns for more information.

1.2.2.3 ASCII

Syntax

```
ASCII(str)
```

Description

Returns the numeric ASCII value of the leftmost character of the string argument. Returns `0` if the given string is empty and `NULL` if it is `NULL`.

`ASCII()` works for 8-bit characters.

Examples

```
SELECT ASCII(9);
+----------+
| ASCII(9) |
+----------+
|       57 |
+----------+

SELECT ASCII('9');
+------------+
| ASCII('9') |
+------------+
|         57 |
+------------+

SELECT ASCII('abc');
+--------------+
| ASCII('abc') |
+--------------+
|           97 |
+--------------+
```

1.2.2.4 BIN

Syntax

```
BIN(N)
```

Description

Returns a string representation of the binary value of the given longlong (that is, `BIGINT`) number. This is equivalent to `CONV(N,10,2)`. The argument should be positive. If it is a `FLOAT`, it will be truncated. Returns `NULL` if the argument is `NULL`.

Examples

```
SELECT BIN(12);
+---------+
| BIN(12) |
+---------+
| 1100    |
+---------+
```

1.2.2.5 BINARY Operator

This page describes the BINARY operator. For details about the data type, see Binary Data Type.

Syntax

```
BINARY
```

Description

The `BINARY` operator casts the string following it to a binary string. This is an easy way to force a column comparison to be done byte by byte rather than character by character. This causes the comparison to be case sensitive even if the column isn't defined as `BINARY` or `BLOB`.

`BINARY` also causes trailing spaces to be significant.

Examples

```
SELECT 'a' = 'A';
+-----------+
| 'a' = 'A' |
+-----------+
|         1 |
+-----------+

SELECT BINARY 'a' = 'A';
+------------------+
| BINARY 'a' = 'A' |
+------------------+
|                0 |
+------------------+

SELECT 'a' = 'a ';
+------------+
| 'a' = 'a ' |
+------------+
|          1 |
+------------+

SELECT BINARY 'a' = 'a ';
+-------------------+
| BINARY 'a' = 'a ' |
+-------------------+
|                 0 |
+-------------------+
```

1.2.2.6 BIT_LENGTH

Syntax

```
BIT_LENGTH(str)
```

Contents

Description

Returns the length of the given string argument in bits. If the argument is not a string, it will be converted to string. If the argument is `NULL`, it returns `NULL`.

Examples

```
SELECT BIT_LENGTH('text');
+--------------------+
| BIT_LENGTH('text') |
+--------------------+
|                 32 |
+--------------------+
```

```
SELECT BIT_LENGTH('');
+----------------+
| BIT_LENGTH('') |
+----------------+
|              0 |
+----------------+
```

Compatibility

PostgreSQL and Sybase support BIT_LENGTH().

1.2.2.7 CAST

Syntax

```
CAST(expr AS type)
```

Contents

Description

The `CAST()` function takes a value of one type and produces a value of another type, similar to the CONVERT() function.

The type can be one of the following values:

- BINARY
- CHAR
- DATE
- DATETIME
- DECIMAL[(M[,D])]
- DOUBLE
- FLOAT (from MariaDB 10.4.5)
- INTEGER
  - Short for `SIGNED INTEGER`
- SIGNED [INTEGER]
- UNSIGNED [INTEGER]
- TIME
- VARCHAR (in Oracle mode, from MariaDB 10.3)

The main difference between `CAST` and CONVERT() is that `CONVERT(expr,type)` is ODBC syntax while `CAST(expr as type)` and `CONVERT(... USING ...)` are SQL92 syntax.

In MariaDB 10.4 and later, you can use the `CAST()` function with the `INTERVAL` keyword.

Until MariaDB 5.5.31, `X'HHHH'`, the standard SQL syntax for binary string literals, erroneously worked in the same way as `0xHHHH`. In 5.5.31 it was intentionally changed to behave as a string in all contexts (and never as a number).

This introduced an incompatibility with previous versions of MariaDB, and all versions of MySQL (see the example below).

Examples

Simple casts:

```
SELECT CAST("abc" AS BINARY);
SELECT CAST("1" AS UNSIGNED INTEGER);
SELECT CAST(123 AS CHAR CHARACTER SET utf8)
```

Note that when one casts to CHAR without specifying the character set, the collation_connection character set collation will be used. When used with `CHAR CHARACTER SET`, the default collation for that character set will be used.

```
SELECT COLLATION(CAST(123 AS CHAR));
+-----------------------------+
| COLLATION(CAST(123 AS CHAR)) |
+-----------------------------+
| latin1_swedish_ci           |
+-----------------------------+

SELECT COLLATION(CAST(123 AS CHAR CHARACTER SET utf8));
+-----------------------------------------------+
| COLLATION(CAST(123 AS CHAR CHARACTER SET utf8)) |
+-----------------------------------------------+
| utf8_general_ci                               |
+-----------------------------------------------+
```

If you also want to change the collation, you have to use the `COLLATE` operator:

```
SELECT COLLATION(CAST(123 AS CHAR CHARACTER SET utf8)
  COLLATE utf8_unicode_ci);
+-----------------------------------------------------------------------------+
| COLLATION(CAST(123 AS CHAR CHARACTER SET utf8) COLLATE utf8_unicode_ci) |
+-----------------------------------------------------------------------------+
| utf8_unicode_ci                                                             |
+-----------------------------------------------------------------------------+
```

Using `CAST()` to order an `ENUM` field as a `CHAR` rather than the internal numerical value:

```
CREATE TABLE enum_list (enum_field enum('c','a','b'));

INSERT INTO enum_list (enum_field)
VALUES('c'),('a'),('c'),('b');

SELECT * FROM enum_list
ORDER BY enum_field;
+------------+
| enum_field |
+------------+
| c          |
| c          |
| a          |
| b          |
+------------+

SELECT * FROM enum_list
ORDER BY CAST(enum_field AS CHAR);
+------------+
| enum_field |
+------------+
| a          |
| b          |
| c          |
| c          |
+------------+
```

From MariaDB 5.5.31, the following will trigger warnings, since `x'aa'` and `'X'aa'` no longer behave as a number. Previously, and in all versions of MySQL, no warnings are triggered since they did erroneously behave as a number:

```
SELECT CAST(0xAA AS UNSIGNED), CAST(x'aa' AS UNSIGNED), CAST(X'aa' AS UNSIGNED);
+------------------------+-------------------------+-------------------------+
| CAST(0xAA AS UNSIGNED) | CAST(x'aa' AS UNSIGNED) | CAST(X'aa' AS UNSIGNED) |
+------------------------+-------------------------+-------------------------+
|                    170 |                       0 |                       0 |
+------------------------+-------------------------+-------------------------+
1 row in set, 2 warnings (0.00 sec)

Warning (Code 1292): Truncated incorrect INTEGER value: '\xAA'
Warning (Code 1292): Truncated incorrect INTEGER value: '\xAA'
```

Casting to intervals:

```
SELECT CAST(2019-01-04 INTERVAL AS DAY_SECOND(2)) AS "Cast";

+-------------+
| Cast        |
+-------------+
| 00:20:17.00 |
+-------------+
```

1.2.2.8 CHAR Function

Syntax

```
CHAR(N,... [USING charset_name])
```

Description

`CHAR()` interprets each argument as an `INT` and returns a string consisting of the characters given by the code values of those integers. `NULL` values are skipped. By default, `CHAR()` returns a binary string. To produce a string in a given character set, use the optional `USING` clause:

```
SELECT CHARSET(CHAR(0x65)), CHARSET(CHAR(0x65 USING utf8));
+---------------------+-------------------------------+
| CHARSET(CHAR(0x65)) | CHARSET(CHAR(0x65 USING utf8)) |
+---------------------+-------------------------------+
| binary              | utf8                          |
+---------------------+-------------------------------+
```

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`.

Examples

```
SELECT CHAR(77,97,114,'105',97,'68',66);
+----------------------------------+
| CHAR(77,97,114,'105',97,'68',66) |
+----------------------------------+
| MariaDB                          |
+----------------------------------+

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

Warning (Code 1292): Truncated incorrect INTEGER value: '77.3'
```

1.2.2.9 CHAR_LENGTH

Syntax

```
CHAR_LENGTH(str)
CHARACTER_LENGTH(str)
```

Contents

Description

Returns the length of the given string argument, measured in characters. A multi-byte character counts as a single character. This means that for a string containing five two-byte characters, LENGTH() (or OCTET_LENGTH() in Oracle mode) returns 10, whereas `CHAR_LENGTH()` returns 5. If the argument is `NULL`, it returns `NULL`.

If the argument is not a string value, it is converted into a string.

It is synonymous with the `CHARACTER_LENGTH()` function.

Examples

```
SELECT CHAR_LENGTH('MariaDB');
+------------------------+
| CHAR_LENGTH('MariaDB') |
+------------------------+
|                      7 |
+------------------------+
```

When Oracle mode from MariaDB 10.3 is not set:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+--------------------+--------------+---------------+--------------------+
| CHAR_LENGTH('π')   | LENGTH('π')  | LENGTHB('π')  | OCTET_LENGTH('π')  |
+--------------------+--------------+---------------+--------------------+
|                  1 |            2 |             2 |                  2 |
+--------------------+--------------+---------------+--------------------+
```

In Oracle mode from MariaDB 10.3:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+--------------------+--------------+---------------+--------------------+
| CHAR_LENGTH('π')   | LENGTH('π')  | LENGTHB('π')  | OCTET_LENGTH('π')  |
+--------------------+--------------+---------------+--------------------+
|                  1 |            1 |             2 |                  2 |
+--------------------+--------------+---------------+--------------------+
```

1.2.2.10 CHARACTER_LENGTH

Syntax

```
CHARACTER_LENGTH(str)
```

Description

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

1.2.2.11 CHR

Syntax

```
CHR(N)
```

Description

`CHR()` interprets each argument N as an integer and returns a `VARCHAR(1)` string consisting of the character given by the code values of the integer. The character set and collation of the string are set according to the values of the `character_set_database` and `collation_database` system variables.

`CHR()` is similar to the `CHAR()` function, but only accepts a single argument.

`CHR()` is available in all sql_modes.

Examples

```
SELECT CHR(67);
+---------+
| CHR(67) |
+---------+
| C       |
+---------+

SELECT CHR('67');
+-----------+
| CHR('67') |
+-----------+
| C         |
+-----------+

SELECT CHR('C');
+----------+
| CHR('C') |
+----------+
|          |
+----------+
1 row in set, 1 warning (0.000 sec)

SHOW WARNINGS;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Truncated incorrect INTEGER value: 'C' |
+---------+------+----------------------------------------+
```

1.2.2.12 CONCAT

Syntax

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

Contents

Description

Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are non-binary strings, the result is a non-binary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent binary string form; if you want to avoid that, you can use an explicit type cast, as in this example:

```
SELECT CONCAT(CAST(int_col AS CHAR), char_col);
```

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

A `NULL` parameter hides all information contained in other parameters from the result. Sometimes this is not desirable; to avoid this, you can:

- Use the `CONCAT_WS()` function with an empty separator, because that function is `NULL`-safe.
- Use `IFNULL()` to turn NULLs into empty strings.

Oracle Mode

In Oracle mode, `CONCAT` ignores NULL.

Examples

```
SELECT CONCAT('Ma', 'ria', 'DB');
+--------------------------+
| CONCAT('Ma', 'ria', 'DB') |
+--------------------------+
| MariaDB                  |
+--------------------------+

SELECT CONCAT('Ma', 'ria', NULL, 'DB');
+---------------------------------+
| CONCAT('Ma', 'ria', NULL, 'DB') |
+---------------------------------+
| NULL                            |
+---------------------------------+

SELECT CONCAT(42.0);
+-------------+
| CONCAT(42.0) |
+-------------+
| 42.0         |
+-------------+
```

Using `IFNULL()` to handle NULLs:

```
SELECT CONCAT('The value of @v is: ', IFNULL(@v, ''));
+-----------------------------------------------+
| CONCAT('The value of @v is: ', IFNULL(@v, '')) |
+-----------------------------------------------+
| The value of @v is:                           |
+-----------------------------------------------+
```

In Oracle mode, from MariaDB 10.3:

```
SELECT CONCAT('Ma', 'ria', NULL, 'DB');
+---------------------------------+
| CONCAT('Ma', 'ria', NULL, 'DB') |
+---------------------------------+
| MariaDB                         |
+---------------------------------+
```

1.2.2.13 CONCAT_WS

Syntax

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

Description

`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`; all other `NULL` values are skipped. This makes `CONCAT_WS()` suitable when you want to concatenate some values and avoid losing all information if one of them is `NULL`.

Examples

```
SELECT CONCAT_WS(',','First name','Second name','Last Name');
+----------------------------------------------------+
| CONCAT_WS(',','First name','Second name','Last Name') |
+----------------------------------------------------+
| First name,Second name,Last Name                   |
+----------------------------------------------------+

SELECT CONCAT_WS('-','Floor',NULL,'Room');
+-----------------------------------+
| CONCAT_WS('-','Floor',NULL,'Room') |
+-----------------------------------+
| Floor-Room                        |
+-----------------------------------+
```

In some cases, remember to include a space in the separator string:

```
SET @a = 'gnu', @b = 'penguin', @c = 'sea lion';
Query OK, 0 rows affected (0.00 sec)

SELECT CONCAT_WS(', ', @a, @b, @c);
+-----------------------------+
| CONCAT_WS(', ', @a, @b, @c) |
+-----------------------------+
| gnu, penguin, sea lion      |
+-----------------------------+
```

Using `CONCAT_WS()` to handle `NULL`s:

```
SET @a = 'a', @b = NULL, @c = 'c';

SELECT CONCAT_WS('', @a, @b, @c);
+--------------------------+
| CONCAT_WS('', @a, @b, @c) |
+--------------------------+
| ac                       |
+--------------------------+
```

1.2.2.14 CONVERT

Syntax

```
CONVERT(expr,type), CONVERT(expr USING transcoding_name)
```

Contents

Description

The `CONVERT()` and CAST() functions take a value of one type and produce a value of another type.

The type can be one of the following values:

- BINARY
- CHAR
- DATE
- DATETIME
- DECIMAL[(M[,D])]
- DOUBLE
- FLOAT (from MariaDB 10.4.5)
- INTEGER
  - Short for `SIGNED INTEGER`
- SIGNED [INTEGER]
- UNSIGNED [INTEGER]
- TIME
- VARCHAR (in Oracle mode, from MariaDB 10.3)

Note that in MariaDB, `INT` and `INTEGER` are the same thing.

`BINARY` produces a string with the BINARY data type. If the optional length is given, `BINARY(N)` causes the cast to use no more than `N` bytes of the argument. Values shorter than the given number in bytes are padded with 0x00 bytes to make them equal the length value.

`CHAR(N)` causes the cast to use no more than the number of characters given in the argument.

The main difference between the CAST() and `CONVERT()` is that `CONVERT(expr,type)` is ODBC syntax while CAST(expr as type) and `CONVERT(... USING ...)` are SQL92 syntax.

`CONVERT()` with `USING` is used to convert data between different character sets. In MariaDB, transcoding names are the same as the corresponding character set names. For example, this statement converts the string 'abc' in the default character set to the corresponding string in the utf8 character set:

```
SELECT CONVERT('abc' USING utf8);
```

Examples

```
SELECT enum_col FROM tbl_name
ORDER BY CAST(enum_col AS CHAR);
```

Converting a BINARY to string to permit the LOWER function to work:

```
SET @x = 'AardVark';

SET @x = BINARY 'AardVark';

SELECT LOWER(@x), LOWER(CONVERT (@x USING latin1));
+-----------+-----------------------------------+
| LOWER(@x) | LOWER(CONVERT (@x USING latin1)) |
+-----------+-----------------------------------+
| AardVark  | aardvark                          |
+-----------+-----------------------------------+
```

1.2.2.15 ELT

Syntax

```
ELT(N, str1[, str2, str3,...])
```

Description

Takes a numeric argument and a series of string arguments. Returns the string that corresponds to the given numeric position. For instance, it returns `str1` if `N` is 1, `str2` if `N` is 2, and so on. If the numeric argument is a `FLOAT`, MariaDB rounds it to the nearest `INTEGER`. If the numeric argument is less than 1, greater than the total number of arguments, or not a number, `ELT()` returns `NULL`. It must have at least two arguments.

It is complementary to the `FIELD()` function.

Examples

```
SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo');
+----------------------------------+
| ELT(1, 'ej', 'Heja', 'hej', 'foo') |
+----------------------------------+
| ej                               |
+----------------------------------+

SELECT ELT(4, 'ej', 'Heja', 'hej', 'foo');
+----------------------------------+
| ELT(4, 'ej', 'Heja', 'hej', 'foo') |
+----------------------------------+
| foo                              |
+----------------------------------+
```

1.2.2.16 EXPORT_SET

Syntax

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

Description

Takes a minimum of three arguments. Returns a string where each bit in the given `bits` argument is returned, with the string values given for `on` and `off`.

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 a separator string (defaults as ' , '). You can optionally limit the number of bits the `EXPORT_SET()` function examines using the `number_of_bits` option.

If any of the arguments are set as `NULL`, the function returns `NULL`.

Examples

```
SELECT EXPORT_SET(5,'Y','N',',',4);
+-----------------------------+
| EXPORT_SET(5,'Y','N',',',4) |
+-----------------------------+
| Y,N,Y,N                     |
+-----------------------------+

SELECT EXPORT_SET(6,'1','0',',',10);
+-----------------------------+
| EXPORT_SET(6,'1','0',',',10) |
+-----------------------------+
| 0,1,1,0,0,0,0,0,0,0         |
+-----------------------------+
```

1.2.2.17 EXTRACTVALUE

Syntax

```
EXTRACTVALUE(xml_frag, xpath_expr)
```

Contents

Description

The `EXTRACTVALUE()` function takes two string arguments: a fragment of XML markup and an XPath expression, (also known as a locator). It returns the text (That is, CDDATA), of the first text node which is a child of the element or elements matching the XPath expression.

In cases where a valid XPath expression does not match any text nodes in a valid XML fragment, (including the implicit `/text()` expression), the `EXTRACTVALUE()` function returns an empty string.

Invalid Arguments

When either the XML fragment or the XPath expression is `NULL`, the `EXTRACTVALUE()` function returns `NULL`. When the XML fragment is invalid, it raises a warning Code 1525:

```
Warning (Code 1525): Incorrect XML value: 'parse error at line 1 pos 11: unexpected END-OF-
INPUT'
```

When the XPath value is invalid, it generates an Error 1105:

```
ERROR 1105 (HY000): XPATH syntax error: ')'
```

Explicit text() Expressions

This function is the equivalent of performing a match using the XPath expression after appending `/text()`. In other words:

```
SELECT
   EXTRACTVALUE('<cases><case>example</case></cases>', '/cases/case')
    AS 'Base Example',
   EXTRACTVALUE('<cases><case>example</case></cases>', '/cases/case/text()')
    AS 'text() Example';
+--------------+----------------+
| Base Example | text() Example |
+--------------+----------------+
| example      | example        |
+--------------+----------------+
```

Count Matches

When `EXTRACTVALUE()` returns multiple matches, it returns the content of the first child text node of each matching element, in the matched order, as a single, space-delimited string.

By design, the `EXTRACTVALUE()` function makes no distinction between a match on an empty element and no match at all. If you need to determine whether no matching element was found in the XML fragment or if an element was found that contained no child text nodes, use the XPath `count()` function.

For instance, when looking for a value that exists, but contains no child text nodes, you would get a count of the number of matching instances:

```
SELECT
   EXTRACTVALUE('<cases><case/></cases>', '/cases/case')
    AS 'Empty Example',
   EXTRACTVALUE('<cases><case/></cases>', 'count(/cases/case)')
    AS 'count() Example';
+---------------+-----------------+
| Empty Example | count() Example |
+---------------+-----------------+
|               |               1 |
+---------------+-----------------+
```

Alternatively, when looking for a value that doesn't exist, `count()` returns 0.

```
SELECT
   EXTRACTVALUE('<cases><case/></cases>', '/cases/person')
    AS 'No Match Example',
   EXTRACTVALUE('<cases><case/></cases>', 'count(/cases/person)')
    AS 'count() Example';
+------------------+-----------------+
| No Match Example | count() Example |
+------------------+-----------------+
|                  |               0|
+------------------+-----------------+
```

Matches

Important: The `EXTRACTVALUE()` function only returns CDDATA. It does not return tags that the element might contain or the text that these child elements contain.

```
SELECT
  EXTRACTVALUE('<cases><case>Person<email>x@example.com</email></case></cases>', '/cases')
   AS Case;
+--------+
| Case   |
+--------+
| Person |
+--------+
```

Note, in the above example, while the XPath expression matches to the parent `<case>` instance, it does not return the contained `<email>` tag or its content.

Examples

```
SELECT
    ExtractValue('<a>ccc<b>ddd</b></a>', '/a')            AS val1,
    ExtractValue('<a>ccc<b>ddd</b></a>', '/a/b')          AS val2,
    ExtractValue('<a>ccc<b>ddd</b></a>', '//b')           AS val3,
    ExtractValue('<a>ccc<b>ddd</b></a>', '/b')            AS val4,
    ExtractValue('<a>ccc<b>ddd</b><b>eee</b></a>', '//b') AS val5;
+------+------+------+------+---------+
| val1 | val2 | val3 | val4 | val5    |
+------+------+------+------+---------+
| ccc  | ddd  | ddd  |      | ddd eee |
+------+------+------+------+---------+
```

1.2.2.18 FIELD

Syntax

```
FIELD(pattern, str1[,str2,...])
```

Description

Returns the index position of the string or number matching the given pattern. Returns `0` in the event that none of the arguments match the pattern. Raises an Error 1582 if not given at least two arguments.

When all arguments given to the `FIELD()` function are strings, they are treated as case-insensitive. When all the arguments are numbers, they are treated as numbers. Otherwise, they are treated as doubles.

If the given pattern occurs more than once, the `FIELD()` function only returns the index of the first instance. If the given pattern is `NULL`, the function returns `0`, as a `NULL` pattern always fails to match.

This function is complementary to the `ELT()` function.

Examples

```
SELECT FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo')
   AS 'Field Results';
+---------------+
| Field Results |
+---------------+
|             2 |
+---------------+

SELECT FIELD('fo', 'Hej', 'ej', 'Heja', 'hej', 'foo')
   AS 'Field Results';
+---------------+
| Field Results |
+---------------+
|             0 |
+---------------+

SELECT FIELD(1, 2, 3, 4, 5, 1) AS 'Field Results';
+---------------+
| Field Results |
+---------------+
|             5 |
+---------------+

SELECT FIELD(NULL, 2, 3) AS 'Field Results';
+---------------+
| Field Results |
+---------------+
|             0 |
+---------------+

SELECT FIELD('fail') AS 'Field Results';
Error 1582 (42000): Incorrect parameter count in call
to native function 'field'
```

1.2.2.19 FIND_IN_SET

Syntax

```
FIND_IN_SET(pattern, strlist)
```

Description

Returns the index position where the given pattern occurs in a string list. The first argument is the pattern you want to search for. The second argument is a string containing comma-separated variables. If the second argument is of the `SET` data-type, the function is optimized to use bit arithmetic.

If the pattern does not occur in the string list or if the string list is an empty string, the function returns `0`. If either argument is `NULL`, the function returns `NULL`. The function does not return the correct result if the pattern contains a comma (`,`) character.

Examples

```
SELECT FIND_IN_SET('b','a,b,c,d') AS "Found Results";
+---------------+
| Found Results |
+---------------+
|             2 |
+---------------+
```

1.2.2.20 FORMAT

Syntax

```
FORMAT(num, decimal_position[, locale])
```

Description

Formats the given number for display as a string, adding separators to appropriate position and rounding the results to the given decimal position. For instance, it would format `15233.345` to `15,233.35`.

If the given decimal position is `0`, it rounds to return no decimal point or fractional part. You can optionally specify a locale value to format numbers to the pattern appropriate for the given region.

Examples

```
SELECT FORMAT(1234567890.09876543210, 4) AS 'Format';
+--------------------+
| Format             |
+--------------------+
| 1,234,567,890.0988 |
+--------------------+

SELECT FORMAT(1234567.89, 4) AS 'Format';
+----------------+
| Format         |
+----------------+
| 1,234,567.8900 |
+----------------+

SELECT FORMAT(1234567.89, 0) AS 'Format';
+-----------+
| Format    |
+-----------+
| 1,234,568 |
+-----------+

SELECT FORMAT(123456789,2,'rm_CH') AS 'Format';
+----------------+
| Format         |
+----------------+
| 123'456'789,00 |
+----------------+
```

1.2.2.21 FROM_BASE64

Syntax

```
FROM_BASE64(str)
```

Description

Decodes the given base-64 encode string, returning the result as a binary string. Returns `NULL` if the given string is `NULL` or if it's invalid.

It is the reverse of the `TO_BASE64` function.

There are numerous methods to base-64 encode a string. MariaDB uses the following:

- It encodes alphabet value 64 as '`+`'.
- It encodes alphabet value 63 as '`/`'.
- It codes output in groups of four printable characters. Each three byte of data encoded uses four characters. If the final group is incomplete, it pads the difference with the '`=`' character.
- It divides long output, adding a new line very 76 characters.
- In decoding, it recognizes and ignores newlines, carriage returns, tabs and space whitespace characters.

```
SELECT TO_BASE64('Maria') AS 'Input';
+-----------+
| Input     |
+-----------+
| TWFyaWE=  |
+-----------+

SELECT FROM_BASE64('TWFyaWE=') AS 'Output';
+--------+
| Output |
+--------+
| Maria  |
+--------+
```

1.2.2.22 HEX

Syntax

```
HEX(N_or_S)
```

Description

If `N_or_S` is a number, returns a string representation of the hexadecimal value of `N`, where `N` is a `longlong` (`BIGINT`) number. This is equivalent to `CONV(N,10,16)`.

If `N_or_S` is a string, returns a hexadecimal string representation of `N_or_S` where each byte of each character in `N_or_S` is converted to two hexadecimal digits. If `N_or_S` is NULL, returns NULL. The inverse of this operation is performed by the UNHEX() function.

MariaDB starting with 10.5.0

HEX() with an INET6 argument returns a hexadecimal representation of the underlying 16-byte binary string.

Examples

```
SELECT HEX(255);
+----------+
| HEX(255) |
+----------+
| FF       |
+----------+

SELECT 0x4D617269614442;
+------------------+
| 0x4D617269614442 |
+------------------+
| MariaDB          |
+------------------+

SELECT HEX('MariaDB');
+----------------+
| HEX('MariaDB') |
+----------------+
| 4D617269614442 |
+----------------+
```

From MariaDB 10.5.0:

```
SELECT HEX(CAST('2001:db8::ff00:42:8329' AS INET6));
+---------------------------------------------+
| HEX(CAST('2001:db8::ff00:42:8329' AS INET6)) |
+---------------------------------------------+
| 20010DB8000000000000FF0000428329             |
+---------------------------------------------+
```

1.2.2.23 INSTR

Syntax

```
INSTR(str,substr)
```

Contents

Description

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.

`INSTR()` performs a case-insensitive search.

If any argument is `NULL`, returns `NULL`.

Examples

```
SELECT INSTR('foobarbar', 'bar');
+--------------------------+
| INSTR('foobarbar', 'bar') |
+--------------------------+
|                        4 |
+--------------------------+

SELECT INSTR('My', 'Maria');
+----------------------+
| INSTR('My', 'Maria') |
+----------------------+
|                    0 |
+----------------------+
```

1.2.2.24 LCASE

Syntax

```
LCASE(str)
```

Description

LCASE() is a synonym for LOWER().

1.2.2.25 LEFT

Syntax

```
LEFT(str,len)
```

Description

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

Examples

```
SELECT LEFT('MariaDB', 5);
+--------------------+
| LEFT('MariaDB', 5) |
+--------------------+
| Maria              |
+--------------------+
```

1.2.2.26 INSERT Function

Syntax

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

Description

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.

Examples

```
SELECT INSERT('Quadratic', 3, 4, 'What');
+-----------------------------------+
| INSERT('Quadratic', 3, 4, 'What') |
+-----------------------------------+
| QuWhattic                         |
+-----------------------------------+

SELECT INSERT('Quadratic', -1, 4, 'What');
+------------------------------------+
| INSERT('Quadratic', -1, 4, 'What') |
+------------------------------------+
| Quadratic                          |
+------------------------------------+

SELECT INSERT('Quadratic', 3, 100, 'What');
+-------------------------------------+
| INSERT('Quadratic', 3, 100, 'What') |
+-------------------------------------+
| QuWhat                              |
+-------------------------------------+
```

1.2.2.27 LENGTH

Syntax

```
LENGTH(str)
```

Contents

Description

Returns the length of the string `str`.

In the default mode, when Oracle mode from MariaDB 10.3 is not set, the length is measured in bytes. In this case, a multi-byte character counts as multiple bytes. This means that for a string containing five two-byte characters, `LENGTH()` returns 10, whereas CHAR_LENGTH() returns 5.

When running Oracle mode from MariaDB 10.3, the length is measured in characters, and `LENGTH` is a synonym for CHAR_LENGTH().

If `str` is not a string value, it is converted into a string. If `str` is `NULL`, the function returns `NULL`.

Examples

```
SELECT LENGTH('MariaDB');
+-------------------+
| LENGTH('MariaDB') |
+-------------------+
|                 7 |
+-------------------+
```

When Oracle mode from MariaDB 10.3 is not set:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+------------------+-------------+--------------+--------------------+
| CHAR_LENGTH('π') | LENGTH('π') | LENGTHB('π') | OCTET_LENGTH('π') |
+------------------+-------------+--------------+--------------------+
|                1 |           2 |            2 |                  2 |
+------------------+-------------+--------------+--------------------+
```

In Oracle mode from MariaDB 10.3:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+------------------+-------------+--------------+--------------------+
| CHAR_LENGTH('π') | LENGTH('π') | LENGTHB('π') | OCTET_LENGTH('π') |
+------------------+-------------+--------------+--------------------+
|                1 |           1 |            2 |                  2 |
+------------------+-------------+--------------+--------------------+
```

1.2.2.28 LENGTHB

MariaDB starting with 10.3.1

Introduced in MariaDB 10.3.1 as part of the Oracle compatibility enhancements.

Syntax

```
LENGTHB(str)
```

Description

`LENGTHB()` returns the length of the given string, in bytes. When Oracle mode is not set, this is a synonym for LENGTH.

A multi-byte character counts as multiple bytes. This means that for a string containing five two-byte characters, `LENGTHB()` returns 10, whereas CHAR_LENGTH() returns 5.

If `str` is not a string value, it is converted into a string. If `str` is `NULL`, the function returns `NULL`.

Examples

When Oracle mode from MariaDB 10.3 is not set:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+------------------+-------------+--------------+--------------------+
| CHAR_LENGTH('π') | LENGTH('π') | LENGTHB('π') | OCTET_LENGTH('π') |
+------------------+-------------+--------------+--------------------+
|                1 |           2 |            2 |                  2 |
+------------------+-------------+--------------+--------------------+
```

In Oracle mode from MariaDB 10.3:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+------------------+-------------+--------------+--------------------+
| CHAR_LENGTH('π') | LENGTH('π') | LENGTHB('π') | OCTET_LENGTH('π') |
+------------------+-------------+--------------+--------------------+
|                1 |           1 |            2 |                  2 |
+------------------+-------------+--------------+--------------------+
```

1.2.2.29 LIKE

Syntax

```
expr LIKE pat [ESCAPE 'escape_char']
expr NOT LIKE pat [ESCAPE 'escape_char']
```

Contents

Description

Tests whether *expr* matches the pattern *pat*. Returns either 1 (`TRUE`) or 0 (`FALSE`). Both *expr* and *pat* may be any valid expression and are evaluated to strings. Patterns may use the following wildcard characters:

- `%` matches any number of characters, including zero.
- `_` matches any single character.

Use `NOT LIKE` to test if a string does not match a pattern. This is equivalent to using the `NOT` operator on the entire `LIKE` expression.

If either the expression or the pattern is `NULL` , the result is `NULL` .

`LIKE` performs case-insensitive substring matches if the collation for the expression and pattern is case-insensitive. For case-sensitive matches, declare either argument to use a binary collation using `COLLATE` , or coerce either of them to a `BINARY` string using `CAST` . Use `SHOW COLLATION` to get a list of available collations. Collations ending in `_bin` are case-sensitive.

Numeric arguments are coerced to binary strings.

The `_` wildcard matches a single character, not byte. It will only match a multi-byte character if it is valid in the expression's character set. For example, `_` will match `_utf8"€"` , but it will not match `_latin1"€"` because the Euro sign is not a valid latin1 character. If necessary, use `CONVERT` to use the expression in a different character set.

If you need to match the characters `_` or `%` , you must escape them. By default, you can prefix the wildcard characters the backslash character `\` to escape them. The backslash is used both to encode special characters like newlines when a string is parsed as well as to escape wildcards in a pattern after parsing. Thus, to match an actual backslash, you sometimes need to double-escape it as `"\ \ \ \"` .

To avoid difficulties with the backslash character, you can change the wildcard escape character using `ESCAPE` in a `LIKE` expression. The argument to `ESCAPE` must be a single-character string.

Examples

Select the days that begin with "T":

```
CREATE TABLE t1 (d VARCHAR(16));
INSERT INTO t1 VALUES
  ("Monday"), ("Tuesday"), ("Wednesday"),
  ("Thursday"), ("Friday"), ("Saturday"), ("Sunday");
SELECT * FROM t1 WHERE d LIKE "T%";
```

```
SELECT * FROM t1 WHERE d LIKE "T%";
+-----------+
| d         |
+-----------+
| Tuesday   |
| Thursday  |
+-----------+
```

Select the days that contain the substring "es":

```
SELECT * FROM t1 WHERE d LIKE "%es%";
```

```
SELECT * FROM t1 WHERE d LIKE "%es%";
+-----------+
| d         |
+-----------+
| Tuesday   |
| Wednesday |
+-----------+
```

Select the six-character day names:

```
SELECT * FROM t1 WHERE d like "___day";
```

```
SELECT * FROM t1 WHERE d like "___day";
+---------+
| d       |
+---------+
| Monday  |
| Friday  |
| Sunday  |
+---------+
```

With the default collations, `LIKE` is case-insensitive:

```
SELECT * FROM t1 where d like "t%";
```

```
SELECT * FROM t1 where d like "t%";
+----------+
| d        |
+----------+
| Tuesday  |
| Thursday |
+----------+
```

Use `COLLATE` to specify a binary collation, forcing case-sensitive matches:

```
SELECT * FROM t1 WHERE d like "t%" COLLATE latin1_bin;
```

```
SELECT * FROM t1 WHERE d like "t%" COLLATE latin1_bin;
Empty set (0.00 sec)
```

You can include functions and operators in the expression to match. Select dates based on their day name:

```
CREATE TABLE t2 (d DATETIME);
INSERT INTO t2 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
SELECT * FROM t2 WHERE DAYNAME(d) LIKE "T%";
```

```
SELECT * FROM t2 WHERE DAYNAME(d) LIKE "T%";
+------------------+
| d                |
+------------------+
| 2007-01-30 21:31 |
| 2011-04-21 12:34 |
| 2004-10-07 11:19 |
+------------------+
3 rows in set, 7 warnings (0.00 sec)
```

Optimizing LIKE

- MariaDB can use indexes for LIKE on string columns in the case where the LIKE doesn't start with `%` or `_`.
- Starting from MariaDB 10.0, one can set the optimizer_use_condition_selectivity variable to 5. If this is done, then the optimizer will read optimizer_selectivity_sampling_limit rows to calculate the selectivity of the LIKE expression before starting to calculate the query plan. This can help speed up some LIKE queries by providing the optimizer with more information about your data.

1.1.1.4.2.4.4 LOAD_FILE

1.2.2.31 LOCATE

Syntax

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

Description

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`.

`LOCATE()` performs a case-insensitive search.

If any argument is `NULL`, returns `NULL`.

INSTR() is the same as the two-argument form of `LOCATE()`, except that the order of the arguments is reversed.

Examples

```
SELECT LOCATE('bar', 'foobarbar');
+----------------------------+
| LOCATE('bar', 'foobarbar') |
+----------------------------+
|                          4 |
+----------------------------+

SELECT LOCATE('My', 'Maria');
+-----------------------+
| LOCATE('My', 'Maria') |
+-----------------------+
|                     0 |
+-----------------------+

SELECT LOCATE('bar', 'foobarbar', 5);
+-------------------------------+
| LOCATE('bar', 'foobarbar', 5) |
+-------------------------------+
|                             7 |
+-------------------------------+
```

1.2.2.32 LOWER

Syntax

```
LOWER(str)
LCASE(str)
```

Contents

Description

Returns the string `str` with all characters changed to lowercase according to the current character set mapping. The default is latin1 (cp1252 West European).

`LCASE` is a synonym for `LOWER`

Examples

```
 SELECT LOWER('QUADRATICALLY');
+------------------------+
| LOWER('QUADRATICALLY') |
+------------------------+
| quadratically          |
+------------------------+
```

`LOWER()` (and `UPPER()`) are ineffective when applied to binary strings (`BINARY`, `VARBINARY`, `BLOB`). To perform lettercase conversion, CONVERT the string to a non-binary string:

```
SET @str = BINARY 'North Carolina';

SELECT LOWER(@str), LOWER(CONVERT(@str USING latin1));
+----------------+-----------------------------------+
| LOWER(@str)    | LOWER(CONVERT(@str USING latin1)) |
+----------------+-----------------------------------+
| North Carolina | north carolina                    |
+----------------+-----------------------------------+
```

1.2.2.33 LPAD

Syntax

```
LPAD(str, len [,padstr])
```

Contents

Description

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. If `padstr` is omitted, the LPAD function pads spaces.

Prior to MariaDB 10.3.1, the `padstr` parameter was mandatory.

Returns NULL if given a NULL argument. If the result is empty (zero length), returns either an empty string or, from MariaDB 10.3.6 with SQL_MODE=Oracle, NULL.

The Oracle mode version of the function can be accessed outside of Oracle mode by using `LPAD_ORACLE` as the function name.

Examples

```
SELECT LPAD('hello',10,'.');
+----------------------+
| LPAD('hello',10,'.') |
+----------------------+
| .....hello           |
+----------------------+

SELECT LPAD('hello',2,'.');
+--------------------+
| LPAD('hello',2,'.') |
+--------------------+
| he                 |
+--------------------+
```

From MariaDB 10.3.1, with the pad string defaulting to space.

```
SELECT LPAD('hello',10);
+------------------+
| LPAD('hello',10) |
+------------------+
|      hello       |
+------------------+
```

Oracle mode version from MariaDB 10.3.6:

```
SELECT LPAD('',0),LPAD_ORACLE('',0);
+-----------+-------------------+
| LPAD('',0) | LPAD_ORACLE('',0) |
+-----------+-------------------+
|           | NULL              |
+-----------+-------------------+
```

1.2.2.34 LTRIM

Syntax

```
LTRIM(str)
```

Contents

Description

Returns the string `str` with leading space characters removed.

Returns NULL if given a NULL argument. If the result is empty, returns either an empty string, or, from MariaDB 10.3.6 with SQL_MODE=Oracle, NULL.

The Oracle mode version of the function can be accessed outside of Oracle mode by using `LTRIM_ORACLE` as the function name.

Examples

```
SELECT QUOTE(LTRIM('   MariaDB   '));
+-------------------------------+
| QUOTE(LTRIM('   MariaDB   ')) |
+-------------------------------+
| 'MariaDB   '                  |
+-------------------------------+
```

Oracle mode version from MariaDB 10.3.6:

```
SELECT LTRIM(''),LTRIM_ORACLE('');
+-----------+-----------------+
| LTRIM('') | LTRIM_ORACLE('') |
+-----------+-----------------+
|           | NULL            |
+-----------+-----------------+
```

1.2.2.35 MAKE_SET

Syntax

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

Description

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.

Examples

```
SELECT MAKE_SET(1,'a','b','c');
+------------------------+
| MAKE_SET(1,'a','b','c') |
+------------------------+
| a                      |
+------------------------+

SELECT MAKE_SET(1 | 4,'hello','nice','world');
+-------------------------------------+
| MAKE_SET(1 | 4,'hello','nice','world') |
+-------------------------------------+
| hello,world                         |
+-------------------------------------+

SELECT MAKE_SET(1 | 4,'hello','nice',NULL,'world');
+-----------------------------------------+
| MAKE_SET(1 | 4,'hello','nice',NULL,'world') |
+-----------------------------------------+
| hello                                   |
+-----------------------------------------+

SELECT QUOTE(MAKE_SET(0,'a','b','c'));
+--------------------------------+
| QUOTE(MAKE_SET(0,'a','b','c')) |
+--------------------------------+
| ''                             |
+--------------------------------+
```

1.2.2.36 MATCH AGAINST

Syntax

```
MATCH (col1,col2,...) AGAINST (expr [search_modifier])
```

Description

A special construct used to perform a fulltext search on a fulltext index.

See Fulltext Index Overview for a full description, and Full-text Indexes for more articles on the topic.

Examples

```
CREATE TABLE ft_myisam(copy TEXT,FULLTEXT(copy)) ENGINE=MyISAM;

INSERT INTO ft_myisam(copy) VALUES ('Once upon a time'), ('There was a wicked witch'),
 ('Who ate everybody up');

SELECT * FROM ft_myisam WHERE MATCH(copy) AGAINST('wicked');
+--------------------------+
| copy                     |
+--------------------------+
| There was a wicked witch |
+--------------------------+
```

```
SELECT id, body, MATCH (title,body) AGAINST
    ('Security implications of running MySQL as root'
    IN NATURAL LANGUAGE MODE) AS score
    FROM articles WHERE MATCH (title,body) AGAINST
    ('Security implications of running MySQL as root'
    IN NATURAL LANGUAGE MODE);
+----+-------------------------------------+-----------------+
| id | body                                | score           |
+----+-------------------------------------+-----------------+
|  4 | 1. Never run mysqld as root. 2. ... | 1.5219271183014 |
|  6 | When configured properly, MySQL ... | 1.3114095926285 |
+----+-------------------------------------+-----------------+
```

3.3.3.3.2 Full-Text Index Stopwords

1.2.2.38 MID

Syntax

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

Description

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

Examples

```
SELECT MID('abcd',4,1);
+-----------------+
| MID('abcd',4,1) |
+-----------------+
| d               |
+-----------------+

SELECT MID('abcd',2,2);
+-----------------+
| MID('abcd',2,2) |
+-----------------+
| bc              |
+-----------------+
```

A negative starting position:

```
SELECT MID('abcd',-2,4);
+------------------+
| MID('abcd',-2,4) |
+------------------+
| cd               |
+------------------+
```

1.2.2.39 NATURAL_SORT_KEY

MariaDB starting with 10.7.0

NATURAL_SORT_KEY was added in MariaDB 10.7.0.

Syntax

```
NATURAL_SORT_KEY(str)
```

Contents

Description

The `NATURAL_SORT_KEY` function is used for sorting that is closer to natural sorting. Strings are sorted in alphabetical order, while numbers are treated in a way such that, for example, `10` is greater than `2`, whereas in other forms of sorting, `2` would be greater than `10`, just like `z` is greater than `ya`.

There are multiple natural sort implementations, differing in the way they handle leading zeroes, fractions, i18n, negatives, decimals and so on.

MariaDB's implementation ignores leading zeroes when performing the sort.

You can use also use `NATURAL_SORT_KEY` with generated columns. The value is not stored permanently in the table. When using a generated column, the virtual column must be longer than the base column to cater for embedded numbers in the string and MDEV-24582.

Examples

Strings and Numbers

```
CREATE TABLE t1 (c TEXT);

INSERT INTO t1 VALUES ('b1'),('a2'),('a11'),('a1');

SELECT c FROM t1;
+------+
| c    |
+------+
| b1   |
| a2   |
| a11  |
| a1   |
+------+

SELECT c FROM t1 ORDER BY c;
+------+
| c    |
+------+
| a1   |
| a11  |
| a2   |
| b1   |
+------+
```

Unsorted, regular sort and natural sort:

```
TRUNCATE t1;

INSERT INTO t1 VALUES
  ('5.5.31'),('10.7.0'),('10.2.1'),
  ('10.1.22'),('10.3.32'),('10.2.12');

SELECT c FROM t1;
+---------+
| c       |
+---------+
| 5.5.31  |
| 10.7.0  |
| 10.2.1  |
| 10.1.22 |
| 10.3.32 |
| 10.2.12 |
+---------+

SELECT c FROM t1 ORDER BY c;
+---------+
| c       |
+---------+
| 10.1.22 |
| 10.2.1  |
| 10.2.12 |
| 10.3.32 |
| 10.7.0  |
| 5.5.31  |
+---------+

SELECT c FROM t1 ORDER BY NATURAL_SORT_KEY(c);
+---------+
| c       |
+---------+
| 5.5.31  |
| 10.1.22 |
| 10.2.1  |
| 10.2.12 |
| 10.3.32 |
| 10.7.0  |
+---------+
```

IPs

Sorting IPs, unsorted, regular sort and natural sort::

```
TRUNCATE t1;

INSERT INTO t1 VALUES
  ('192.167.3.1'),('192.167.1.12'),('100.200.300.400'),
  ('100.50.60.70'),('100.8.9.9'),('127.0.0.1'),('0.0.0.0');

SELECT c FROM t1;
+-----------------+
| c               |
+-----------------+
| 192.167.3.1     |
| 192.167.1.12    |
| 100.200.300.400 |
| 100.50.60.70    |
| 100.8.9.9       |
| 127.0.0.1       |
| 0.0.0.0         |
+-----------------+

SELECT c FROM t1 ORDER BY c;
+-----------------+
| c               |
+-----------------+
| 0.0.0.0         |
| 100.200.300.400 |
| 100.50.60.70    |
| 100.8.9.9       |
| 127.0.0.1       |
| 192.167.1.12    |
| 192.167.3.1     |
+-----------------+

SELECT c FROM t1 ORDER BY NATURAL_SORT_KEY(c);
+-----------------+
| c               |
+-----------------+
| 0.0.0.0         |
| 100.8.9.9       |
| 100.50.60.70    |
| 100.200.300.400 |
| 127.0.0.1       |
| 192.167.1.12    |
| 192.167.3.1     |
+-----------------+
```

Generated Columns

Using with a generated column:

```
CREATE TABLE t(c VARCHAR(3), k VARCHAR(4) AS (NATURAL_SORT_KEY(c)) INVISIBLE);

INSERT INTO t(c) VALUES ('b1'),('a2'),('a11'),('a10');

SELECT * FROM t ORDER by k;
+------+
| c    |
+------+
| a2   |
| a10  |
| a11  |
| b1   |
+------+
```

Note that if the virtual column is not longer, results may not be as expected:

```
CREATE TABLE t2(c VARCHAR(3), k VARCHAR(3) AS (NATURAL_SORT_KEY(c)) INVISIBLE);

INSERT INTO t2(c) VALUES ('b1'),('a2'),('a11'),('a10');

SELECT * FROM t2 ORDER by k;
+------+
| c    |
+------+
| a2   |
| a11  |
| a10  |
| b1   |
+------+
```

Leading Zeroes

Ignoring leading zeroes can lead to undesirable results in certain contexts. For example:

```
CREATE TABLE t3 (a VARCHAR(4));

INSERT INTO t3 VALUES
  ('a1'), ('a001'), ('a10'), ('a001'), ('a10'),
  ('a01'), ('a01'), ('a01b'), ('a01b'), ('a1');

SELECT a FROM t3 ORDER BY a;
+------+
| a    |
+------+
| a001 |
| a001 |
| a01  |
| a01  |
| a01b |
| a01b |
| a1   |
| a1   |
| a10  |
| a10  |
+------+
10 rows in set (0.000 sec)

SELECT a FROM t3 ORDER BY NATURAL_SORT_KEY(a);
+------+
| a    |
+------+
| a1   |
| a01  |
| a01  |
| a001 |
| a001 |
| a1   |
| a01b |
| a01b |
| a10  |
| a10  |
+------+
```

This may not be what we were hoping for in a 'natural' sort. A workaround is to sort by both NATURAL_SORT_KEY and regular sort.

```
SELECT a FROM t3 ORDER BY NATURAL_SORT_KEY(a), a;
+------+
| a    |
+------+
| a001 |
| a001 |
| a01  |
| a01  |
| a1   |
| a1   |
| a01b |
| a01b |
| a10  |
| a10  |
+------+
```

1.2.2.40 NOT LIKE

Syntax

```
expr NOT LIKE pat [ESCAPE 'escape_char']
```

Description

This is the same as NOT (expr LIKE pat [ESCAPE 'escape_char']).

1.2.2.1.3 NOT REGEXP

1.2.2.42 OCTET_LENGTH

Syntax

```
OCTET_LENGTH(str)
```

Description

`OCTET_LENGTH()` returns the length of the given string, in octets (bytes). This is a synonym for LENGTHB(), and, when Oracle mode from MariaDB 10.3 is not set, a synonym for LENGTH().

A multi-byte character counts as multiple bytes. This means that for a string containing five two-byte characters, `OCTET_LENGTH()` returns 10, whereas CHAR_LENGTH() returns 5.

If `str` is not a string value, it is converted into a string. If `str` is `NULL`, the function returns `NULL`.

Examples

When Oracle mode from MariaDB 10.3 is not set:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+-------------------+--------------+---------------+--------------------+
| CHAR_LENGTH('π')  | LENGTH('π')  | LENGTHB('π')  | OCTET_LENGTH('π')  |
+-------------------+--------------+---------------+--------------------+
|                 1 |            2 |             2 |                  2 |
+-------------------+--------------+---------------+--------------------+
```

In Oracle mode from MariaDB 10.3:

```
SELECT CHAR_LENGTH('π'), LENGTH('π'), LENGTHB('π'), OCTET_LENGTH('π');
+------------------+--------------+---------------+--------------------+
| CHAR_LENGTH('π') | LENGTH('π')  | LENGTHB('π')  | OCTET_LENGTH('π')  |
+------------------+--------------+---------------+--------------------+
|                1 |            1 |             2 |                  2 |
+------------------+--------------+---------------+--------------------+
```

1.2.2.43 ORD

Syntax

```
ORD(str)
```

Description

If the leftmost character of the string `str` is a multi-byte 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 x 256)
+ (3rd byte code x 256 x 256) ...
```

If the leftmost character is not a multi-byte character, ORD() returns the same value as the ASCII() function.

Examples

```
SELECT ORD('2');
+----------+
| ORD('2') |
+----------+
|       50 |
+----------+
```

1.2.2.44 POSITION

Syntax

```
POSITION(substr IN str)
```

Description

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

It's part of ODBC 3.0.

1.2.2.45 QUOTE

Syntax

```
QUOTE(str)
```

Description

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 quotes and with each instance of single quote (" ' "), backslash (" \ "), `ASCII NUL`, and Control-Z preceded by a backslash. If the argument is `NULL`, the return value is the word " `NULL` " without enclosing single quotes.

Examples

```
SELECT QUOTE("Don't!");
+-----------------+
| QUOTE("Don't!") |
+-----------------+
| 'Don\'t!'       |
+-----------------+

SELECT QUOTE(NULL);
+-------------+
| QUOTE(NULL) |
+-------------+
| NULL        |
+-------------+
```

1.2.2.46 REPEAT Function

Syntax

```
REPEAT(str,count)
```

Description

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` are NULL.

Examples

```
SELECT QUOTE(REPEAT('MariaDB ',4));
+-----------------------------------+
| QUOTE(REPEAT('MariaDB ',4))       |
+-----------------------------------+
| 'MariaDB MariaDB MariaDB MariaDB ' |
+-----------------------------------+
```

1.2.2.47 REPLACE Function

Syntax

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

Description

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`.

Examples

```
SELECT REPLACE('www.mariadb.org', 'w', 'Ww');
+--------------------------------------+
| REPLACE('www.mariadb.org', 'w', 'Ww') |
+--------------------------------------+
| WwWwWw.mariadb.org                   |
+--------------------------------------+
```

1.2.2.48 REVERSE

Syntax

```
REVERSE(str)
```

Description

Returns the string `str` with the order of the characters reversed.

Examples

```
SELECT REVERSE('desserts');
+---------------------+
| REVERSE('desserts') |
+---------------------+
| stressed            |
+---------------------+
```

1.2.2.49 RIGHT

Syntax

```
RIGHT(str,len)
```

Description

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

Examples

```
SELECT RIGHT('MariaDB', 2);
+---------------------+
| RIGHT('MariaDB', 2) |
+---------------------+
| DB                  |
+---------------------+
```

1.2.2.50 RPAD

Syntax

```
RPAD(str, len [, padstr])
```

Contents

Description

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 `padstr` is omitted, the RPAD function pads spaces.

Prior to MariaDB 10.3.1, the `padstr` parameter was mandatory.

Returns NULL if given a NULL argument. If the result is empty (a length of zero), returns either an empty string, or, from MariaDB 10.3.6 with SQL_MODE=Oracle, NULL.

The Oracle mode version of the function can be accessed outside of Oracle mode by using `RPAD_ORACLE` as the function name.

Examples

```
SELECT RPAD('hello',10,'.');
+----------------------+
| RPAD('hello',10,'.') |
+----------------------+
| hello.....           |
+----------------------+

SELECT RPAD('hello',2,'.');
+---------------------+
| RPAD('hello',2,'.') |
+---------------------+
| he                  |
+---------------------+
```

From MariaDB 10.3.1, with the pad string defaulting to space.

```
SELECT RPAD('hello',30);
+--------------------------------+
| RPAD('hello',30)               |
+--------------------------------+
| hello                          |
+--------------------------------+
```

Oracle mode version from MariaDB 10.3.6:

```
SELECT RPAD('',0),RPAD_ORACLE('',0);
+------------+-------------------+
| RPAD('',0) | RPAD_ORACLE('',0) |
+------------+-------------------+
|            | NULL              |
+------------+-------------------+
```

1.2.2.51 RTRIM

Syntax

```
RTRIM(str)
```

Contents

Description

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

Returns NULL if given a NULL argument. If the result is empty, returns either an empty string, or, from MariaDB 10.3.6 with SQL_MODE=Oracle, NULL.

The Oracle mode version of the function can be accessed outside of Oracle mode by using `RTRIM_ORACLE` as the function name.

Examples

```
SELECT QUOTE(RTRIM('MariaDB    '));
+-----------------------------+
| QUOTE(RTRIM('MariaDB    ')) |
+-----------------------------+
| 'MariaDB'                   |
+-----------------------------+
```

Oracle mode version from MariaDB 10.3.6:

```
SELECT RTRIM(''),RTRIM_ORACLE('');
+-----------+------------------+
| RTRIM('') | RTRIM_ORACLE('') |
+-----------+------------------+
|           | NULL             |
+-----------+------------------+
```

1.2.2.52 SFORMAT

MariaDB starting with 10.7.0
SFORMAT was added in MariaDB 10.7.0.

Description

The `SFORMAT` function takes an input string and a formatting specification and returns the string formatted using the rules the user passed in the specification.

It uses the fmtlib library for Python-like (as well as Rust, C++20, etc) string formatting.

Only fmtlib 7.0.0+ is supported.

There is no native support for temporal and decimal values:

- TIME_RESULT is handled as STRING_RESULT
- DECIMAL_RESULT as REAL_RESULT

Examples

```
SELECT SFORMAT("The answer is {}.", 42);
+---------------------------------+
| SFORMAT("The answer is {}.", 42) |
+---------------------------------+
| The answer is 42.               |
+---------------------------------+

CREATE TABLE test_sformat(mdb_release char(6), mdev int, feature char(20));

INSERT INTO test_sformat VALUES('10.7.0', 25015, 'Python style sformat'),
  ('10.7.0', 4958, 'UUID');

SELECT * FROM test_sformat;
+-------------+-------+----------------------+
| mdb_release | mdev  | feature              |
+-------------+-------+----------------------+
| 10.7.0      | 25015 | Python style sformat |
| 10.7.0      |  4958 | UUID                 |
+-------------+-------+----------------------+

SELECT SFORMAT('MariaDB Server {} has a preview for MDEV-{} which is about {}',
  mdb_release, mdev, feature) AS 'Preview Release Examples'
  FROM test_sformat;
+-----------------------------------------------------------------------------------------+
| Preview Release Examples                                                                |
+-----------------------------------------------------------------------------------------+
| MariaDB Server 10.7.0 has a preview for MDEV-25015 which is about Python style sformat |
| MariaDB Server 10.7.0 has a preview for MDEV-4958 which is about UUID                   |
+-----------------------------------------------------------------------------------------+
```

1.2.2.53 SOUNDEX

Syntax

```
SOUNDEX(str)
```

Description

Returns a soundex string from *str*. 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 non-alphabetic 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 details:

- 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 reasonable results.
- 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.

Examples

```
SOUNDEX('Hello');
+------------------+
| SOUNDEX('Hello') |
+------------------+
| H400             |
+------------------+
```

```
SELECT SOUNDEX('MariaDB');
+--------------------+
| SOUNDEX('MariaDB') |
+--------------------+
| M631               |
+--------------------+
```

```
SELECT SOUNDEX('Knowledgebase');
+--------------------------+
| SOUNDEX('Knowledgebase') |
+--------------------------+
| K543212                  |
+--------------------------+
```

```
SELECT givenname, surname FROM users WHERE SOUNDEX(givenname) = SOUNDEX("robert");
+-----------+---------+
| givenname | surname |
+-----------+---------+
| Roberto   | Castro  |
+-----------+---------+
```

1.2.2.54 SOUNDS LIKE

Syntax

```
expr1 SOUNDS LIKE expr2
```

Description

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

Example

```
SELECT givenname, surname FROM users WHERE givenname SOUNDS LIKE "robert";
+-----------+---------+
| givenname | surname |
+-----------+---------+
| Roberto   | Castro  |
+-----------+---------+
```

1.2.2.55 SPACE

Syntax

```
SPACE(N)
```

Description

Returns a string consisting of *N* space characters. If `N` is NULL, returns NULL.

Examples

```
SELECT QUOTE(SPACE(6));
+-----------------+
| QUOTE(SPACE(6)) |
+-----------------+
| '      '        |
+-----------------+
```

1.2.2.56 STRCMP

Syntax

```
STRCMP(expr1,expr2)
```

Description

`STRCMP()` returns `0` if the strings are the same, `-1` if the first argument is smaller than the second according to the current sort order, and `1` if the strings are otherwise not the same. Returns `NULL` is either argument is `NULL`.

Examples

```
SELECT STRCMP('text', 'text2');
+-------------------------+
| STRCMP('text', 'text2') |
+-------------------------+
|                      -1 |
+-------------------------+

SELECT STRCMP('text2', 'text');
+-------------------------+
| STRCMP('text2', 'text') |
+-------------------------+
|                       1 |
+-------------------------+

SELECT STRCMP('text', 'text');
+-----------------------+
| STRCMP('text', 'text') |
+-----------------------+
|                     0 |
+-----------------------+
```

1.2.2.57 SUBSTR

Description

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

1.2.2.58 SUBSTRING

Syntax

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

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

Contents

Description

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.

By default, the position of the first character in the string from which the substring is to be extracted is reckoned as 1. For Oracle-compatibility, from MariaDB 10.3.3, when sql_mode is set to 'oracle', position zero is treated as position 1 (although the first character is still reckoned as 1).

If any argument is `NULL`, returns `NULL`.

Examples

```
SELECT SUBSTRING('Knowledgebase',5);
+-----------------------------+
| SUBSTRING('Knowledgebase',5) |
+-----------------------------+
| ledgebase                   |
+-----------------------------+

SELECT SUBSTRING('MariaDB' FROM 6);
+----------------------------+
| SUBSTRING('MariaDB' FROM 6) |
+----------------------------+
| DB                         |
+----------------------------+

SELECT SUBSTRING('Knowledgebase',3,7);
+-------------------------------+
| SUBSTRING('Knowledgebase',3,7) |
+-------------------------------+
| owledge                       |
+-------------------------------+

SELECT SUBSTRING('Knowledgebase', -4);
+--------------------------------+
| SUBSTRING('Knowledgebase', -4) |
+--------------------------------+
| base                           |
+--------------------------------+

SELECT SUBSTRING('Knowledgebase', -8, 4);
+-----------------------------------+
| SUBSTRING('Knowledgebase', -8, 4) |
+-----------------------------------+
| edge                              |
+-----------------------------------+

SELECT SUBSTRING('Knowledgebase' FROM -8 FOR 4);
+-----------------------------------------+
| SUBSTRING('Knowledgebase' FROM -8 FOR 4) |
+-----------------------------------------+
| edge                                    |
+-----------------------------------------+
```

Oracle mode from MariaDB 10.3.3:

```
SELECT SUBSTR('abc',0,3);
+-------------------+
| SUBSTR('abc',0,3) |
+-------------------+
|                   |
+-------------------+

SELECT SUBSTR('abc',1,2);
+-------------------+
| SUBSTR('abc',1,2) |
+-------------------+
| ab                |
+-------------------+

SET sql_mode='oracle';

SELECT SUBSTR('abc',0,3);
+-------------------+
| SUBSTR('abc',0,3) |
+-------------------+
| abc               |
+-------------------+

SELECT SUBSTR('abc',1,2);
+-------------------+
| SUBSTR('abc',1,2) |
+-------------------+
| ab                |
+-------------------+
```

1.2.2.59 SUBSTRING_INDEX

Syntax

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

Contents

Description

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*.

If any argument is NULL, returns NULL.

For example

```
SUBSTRING_INDEX('www.mariadb.org', '.', 2)
```

means "Return all of the characters up to the 2nd occurrence of ."

Examples

```
SELECT SUBSTRING_INDEX('www.mariadb.org', '.', 2);
+--------------------------------------------+
| SUBSTRING_INDEX('www.mariadb.org', '.', 2) |
+--------------------------------------------+
| www.mariadb                                |
+--------------------------------------------+

SELECT SUBSTRING_INDEX('www.mariadb.org', '.', -2);
+---------------------------------------------+
| SUBSTRING_INDEX('www.mariadb.org', '.', -2) |
+---------------------------------------------+
| mariadb.org                                 |
+---------------------------------------------+
```

1.2.2.60 TO_BASE64

Syntax

```
TO_BASE64(str)
```

Description

Converts the string argument `str` to its base-64 encoded form, returning the result as a character string in the connection character set and collation.

The argument `str` will be converted to string first if it is not a string. A NULL argument will return a NULL result.

The reverse function, FROM_BASE64(), decodes an encoded base-64 string.

There are a numerous different methods to base-64 encode a string. The following are used by MariaDB and MySQL:

- Alphabet value 64 is encoded as '+'.
- Alphabet value 63 is encoded as '/'.
- Encoding output is made up of groups of four printable characters, with each three bytes of data encoded using four characters. If the final group is not complete, it is padded with '=' characters to make up a length of four.
- To divide long output, a newline is added after every 76 characters.
- Decoding will recognize and ignore newlines, carriage returns, tabs, and spaces.

Examples

```
SELECT TO_BASE64('Maria');
+--------------------+
| TO_BASE64('Maria') |
+--------------------+
| TWFyaWE=           |
+--------------------+
```

1.2.2.61 TO_CHAR

MariaDB starting with 10.6.1

The TO_CHAR function was introduced in MariaDB 10.6.1 to enhance Oracle compatibility.

Syntax

```
TO_CHAR(expr[, fmt])
```

Contents

Description

The `TO_CHAR` function converts an *expr* of type date, datetime, time or timestamp to a string. The optional *fmt* argument supports YYYY/YYY/YY/RRRR/RR/MM/MON/MONTH/MI/DD/DY/HH/HH12/HH24/SS and special characters. The default value is "YYYY-MM-DD HH24:MI:SS".

In Oracle, TO_CHAR can also be used to convert numbers to strings, but this is not supported in MariaDB and will give an error.

Examples

```
SELECT TO_CHAR('1980-01-11 04:50:39', 'YYYY-MM-DD');
+-------------------------------------------+
| TO_CHAR('1980-01-11 04:50:39', 'YYYY-MM-DD') |
+-------------------------------------------+
| 1980-01-11                                |
+-------------------------------------------+

SELECT TO_CHAR('1980-01-11 04:50:39', 'HH24-MI-SS');
+-------------------------------------------+
| TO_CHAR('1980-01-11 04:50:39', 'HH24-MI-SS') |
+-------------------------------------------+
| 04-50-39                                  |
+-------------------------------------------+

SELECT TO_CHAR('00-01-01 00:00:00', 'YY-MM-DD HH24:MI:SS');
+-----------------------------------------------------+
| TO_CHAR('00-01-01 00:00:00', 'YY-MM-DD HH24:MI:SS') |
+-----------------------------------------------------+
| 00-01-01 00:00:00                                   |
+-----------------------------------------------------+

SELECT TO_CHAR('99-12-31 23:59:59', 'YY-MM-DD HH24:MI:SS');
+-----------------------------------------------------+
| TO_CHAR('99-12-31 23:59:59', 'YY-MM-DD HH24:MI:SS') |
+-----------------------------------------------------+
| 99-12-31 23:59:59                                   |
+-----------------------------------------------------+

SELECT TO_CHAR('9999-12-31 23:59:59', 'YY-MM-DD HH24:MI:SS');
+-----------------------------------------------------+
| TO_CHAR('9999-12-31 23:59:59', 'YY-MM-DD HH24:MI:SS') |
+-----------------------------------------------------+
| 99-12-31 23:59:59                                    |
+-----------------------------------------------------+

SELECT TO_CHAR('21-01-03 08:30:00', 'Y-MONTH-DY HH:MI:SS');
+-----------------------------------------------------+
| TO_CHAR('21-01-03 08:30:00', 'Y-MONTH-DY HH:MI:SS') |
+-----------------------------------------------------+
| 1-January  -Sun 08:30:00                            |
+-----------------------------------------------------+
```

1.2.2.62 TRIM

Syntax

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

From MariaDB 10.3.6

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

Contents

Description

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.

Returns NULL if given a NULL argument. If the result is empty, returns either an empty string, or, from MariaDB 10.3.6 with SQL_MODE=Oracle, NULL. `SQL_MODE=Oracle` is not set by default.

The Oracle mode version of the function can be accessed in any mode by using `TRIM_ORACLE` as the function name.

Examples

```
SELECT TRIM('  bar   ')\G
*************************** 1. row ***************************
TRIM('  bar   '): bar

SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx')\G
*************************** 1. row ***************************
TRIM(LEADING 'x' FROM 'xxxbarxxx'): barxxx

SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx')\G
*************************** 1. row ***************************
TRIM(BOTH 'x' FROM 'xxxbarxxx'): bar

SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz')\G
*************************** 1. row ***************************
TRIM(TRAILING 'xyz' FROM 'barxxyz'): barx
```

From MariaDB 10.3.6, with SQL_MODE=Oracle not set:

```
SELECT TRIM(''),TRIM_ORACLE('');
+----------+-----------------+
| TRIM('') | TRIM_ORACLE('') |
+----------+-----------------+
|          | NULL            |
+----------+-----------------+
```

From MariaDB 10.3.6, with SQL_MODE=Oracle set:

```
SELECT TRIM(''),TRIM_ORACLE('');
+----------+-----------------+
| TRIM('') | TRIM_ORACLE('') |
+----------+-----------------+
| NULL     | NULL            |
+----------+-----------------+
```

1.2.2.63 TRIM_ORACLE

MariaDB starting with 10.3.6

`TRIM_ORACLE` is a synonym for the Oracle mode version of the TRIM function, and is available in all modes.

1.2.2.64 UCASE

Syntax

```
UCASE(str)
```

Description

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

1.2.2.65 UNCOMPRESS

Syntax

```
UNCOMPRESS(string_to_uncompress)
```

Description

Uncompresses a string compressed by the `COMPRESS()` function. If the argument is not a compressed value, the result is `NULL`. This function requires MariaDB to have been compiled with a compression library such as zlib. Otherwise, the return value is always `NULL`. The have_compress server system variable indicates whether a compression library is present.

Examples

```
SELECT UNCOMPRESS(COMPRESS('a string'));
+----------------------------------+
| UNCOMPRESS(COMPRESS('a string')) |
+----------------------------------+
| a string                         |
+----------------------------------+

SELECT UNCOMPRESS('a string');
+------------------------+
| UNCOMPRESS('a string') |
+------------------------+
| NULL                   |
+------------------------+
```

1.2.2.66 UNCOMPRESSED_LENGTH

Syntax

```
UNCOMPRESSED_LENGTH(compressed_string)
```

Description

Returns the length that the compressed string had before being compressed with `COMPRESS()`.

`UNCOMPRESSED_LENGTH()` returns `NULL` or an incorrect result if the string is not compressed.

Until MariaDB 10.3.1, returns `MYSQL_TYPE_LONGLONG`, or bigint(10), in all cases. From MariaDB 10.3.1, returns `MYSQL_TYPE_LONG`, or int(10), when the result would fit within 32-bits.

Examples

```
SELECT UNCOMPRESSED_LENGTH(COMPRESS(REPEAT('a',30)));
+-----------------------------------------------+
| UNCOMPRESSED_LENGTH(COMPRESS(REPEAT('a',30))) |
+-----------------------------------------------+
|                                            30 |
+-----------------------------------------------+
```

1.2.2.67 UNHEX

Syntax

```
UNHEX(str)
```

Description

Performs the inverse operation of HEX(str). That is, it interprets each pair of hexadecimal digits in the argument as a number and converts it to the character represented by the number. The resulting characters are returned as a binary string.

If `str` is `NULL`, `UNHEX()` returns `NULL`.

Examples

```
SELECT HEX('MariaDB');
+----------------+
| HEX('MariaDB') |
+----------------+
| 4D617269614442 |
+----------------+

SELECT UNHEX('4D617269614442');
+-------------------------+
| UNHEX('4D617269614442') |
+-------------------------+
| MariaDB                 |
+-------------------------+

SELECT 0x4D617269614442;
+------------------+
| 0x4D617269614442 |
+------------------+
| MariaDB          |
+------------------+

SELECT UNHEX(HEX('string'));
+----------------------+
| UNHEX(HEX('string')) |
+----------------------+
| string               |
+----------------------+

SELECT HEX(UNHEX('1267'));
+--------------------+
| HEX(UNHEX('1267')) |
+--------------------+
| 1267               |
+--------------------+
```

1.2.2.68 UPDATEXML

Syntax

```
UpdateXML(xml_target, xpath_expr, new_xml)
```

Description

This function replaces a single portion of a given fragment of XML markup `xml_target` with a new XML fragment `new_xml`, and then returns the changed XML. The portion of `xml_target` that is replaced matches an XPath expression `xpath_expr` supplied by the user. If no expression matching `xpath_expr` is found, or if multiple matches are found, the function returns the original `xml_target` XML fragment. All three arguments should be strings.

Examples

```
SELECT
    UpdateXML('<a><b>ccc</b><d></d></a>', '/a', '<e>fff</e>') AS val1,
    UpdateXML('<a><b>ccc</b><d></d></a>', '/b', '<e>fff</e>') AS val2,
    UpdateXML('<a><b>ccc</b><d></d></a>', '//b', '<e>fff</e>') AS val3,
    UpdateXML('<a><b>ccc</b><d></d></a>', '/a/d', '<e>fff</e>') AS val4,
    UpdateXML('<a><d></d><b>ccc</b><d></d></a>', '/a/d', '<e>fff</e>') AS val5
    \G
*************************** 1. row ***************************
val1: <e>fff</e>
val2: <a><b>ccc</b><d></d></a>
val3: <a><e>fff</e><d></d></a>
val4: <a><b>ccc</b><e>fff</e></a>
val5: <a><d></d><b>ccc</b><d></d></a>
1 row in set (0.00 sec)
```

1.2.2.69 UPPER

Syntax

```
UPPER(str)
UCASE(str)
```

Description

Returns the string `str` with all characters changed to uppercase according to the current character set mapping. The default is latin1 (cp1252 West European).

`UCASE` is a synonym.

```
SELECT UPPER(surname), givenname FROM users ORDER BY surname;
+----------------+------------+
| UPPER(surname) | givenname  |
+----------------+------------+
| ABEL           | Jacinto    |
| CASTRO         | Robert     |
| COSTA          | Phestos    |
| MOSCHELLA      | Hippolytos |
+----------------+------------+
```

`UPPER()` is ineffective when applied to binary strings (BINARY, VARBINARY, BLOB). The description of LOWER() shows how to perform lettercase conversion of binary strings.

Prior to MariaDB 11.3, the query optimizer did not handle queries of the format `UCASE(varchar_col)=...`. An optimizer_switch option, `sargable_casefold=ON`, was added in MariaDB 11.3.0 to handle this case. (MDEV-31496)

1.2.2.70 WEIGHT_STRING

Syntax

```
WEIGHT_STRING(str [AS {CHAR|BINARY}(N)] [LEVEL levels] [flags])
  levels: N [ASC|DESC|REVERSE] [, N [ASC|DESC|REVERSE]] ...
```

Description

Returns a binary string representing the string's sorting and comparison value. A string with a lower result means that for sorting purposes the string appears before a string with a higher result.

WEIGHT_STRING() is particularly useful when adding new collations, for testing purposes.

If `str` is a non-binary string (CHAR, VARCHAR or TEXT), WEIGHT_STRING returns the string's collation weight. If `str` is a binary string (BINARY, VARBINARY or BLOB), the return value is simply the input value, since the weight for each byte in a binary string is the byte value.

WEIGHT_STRING() returns NULL if given a NULL input.

The optional AS clause permits casting the input string to a binary or non-binary string, as well as to a particular length.

AS BINARY(N) measures the length in bytes rather than characters, and right pads with 0x00 bytes to the desired length.

AS CHAR(N) measures the length in characters, and right pads with spaces to the desired length.

N has a minimum value of 1, and if it is less than the length of the input string, the string is truncated without warning.

The optional LEVEL clause specifies that the return value should contain weights for specific collation levels. The `levels` specifier can either be a single integer, a comma-separated list of integers, or a range of integers separated by a dash (whitespace is ignored). Integers can range from 1 to a maximum of 6, dependent on the collation, and need to be listed in ascending order.

If the LEVEL clause is no provided, a default of 1 to the maximum for the collation is assumed.

If the LEVEL is specified without using a range, an optional modifier is permitted.

`ASC`, the default, returns the weights without any modification.

`DESC` returns bitwise-inverted weights.

`REVERSE` returns the weights in reverse order.

Examples

The examples below use the HEX() function to represent non-printable results in hexadecimal format.

```
SELECT HEX(WEIGHT_STRING('x'));
+-------------------------+
| HEX(WEIGHT_STRING('x')) |
+-------------------------+
| 0058                    |
+-------------------------+

SELECT HEX(WEIGHT_STRING('x' AS BINARY(4)));
+--------------------------------------+
| HEX(WEIGHT_STRING('x' AS BINARY(4))) |
+--------------------------------------+
| 78000000                             |
+--------------------------------------+

SELECT HEX(WEIGHT_STRING('x' AS CHAR(4)));
+-----------------------------------+
| HEX(WEIGHT_STRING('x' AS CHAR(4))) |
+-----------------------------------+
| 0058002000200020                  |
+-----------------------------------+

SELECT HEX(WEIGHT_STRING(0xaa22ee LEVEL 1));
+--------------------------------------+
| HEX(WEIGHT_STRING(0xaa22ee LEVEL 1)) |
+--------------------------------------+
| AA22EE                               |
+--------------------------------------+

SELECT HEX(WEIGHT_STRING(0xaa22ee LEVEL 1 DESC));
+-----------------------------------------+
| HEX(WEIGHT_STRING(0xaa22ee LEVEL 1 DESC)) |
+-----------------------------------------+
| 55DD11                                  |
+-----------------------------------------+

SELECT HEX(WEIGHT_STRING(0xaa22ee LEVEL 1 REVERSE));
+-------------------------------------------+
| HEX(WEIGHT_STRING(0xaa22ee LEVEL 1 REVERSE)) |
+-------------------------------------------+
| EE22AA                                    |
+-------------------------------------------+
```

1.2.2.71 Type Conversion

Contents

Implicit type conversion takes place when MariaDB is using operands or different types, in order to make the operands compatible.

It is best practice not to rely upon implicit conversion; rather use CAST to explicitly convert types.

Rules for Conversion on Comparison

- If either argument is NULL, the result of the comparison is NULL unless the NULL-safe <=> equality comparison operator is used.
- If both arguments are integers, they are compared as integers.
- If both arguments are strings, they are compared as strings.
- If one argument is decimal and the other argument is decimal or integer, they are compared as decimals.
- If one argument is decimal and the other argument is a floating point, they are compared as floating point values.
- If one argument is string and the other argument is integer, they are compared as decimals. This conversion was added in MariaDB 10.3.36. Prior to 10.3.36, this combination was compared as floating point values, which did not always work well for huge 64-bit integers because of a possible precision loss on conversion to double.
- If a hexadecimal argument is not compared to a number, it is treated as a binary string.
- If a constant is compared to a TIMESTAMP or DATETIME, the constant is converted to a timestamp, unless used as an argument to the IN function.

- In other cases, arguments are compared as floating point, or real, numbers.

Note that if a string column is being compared with a numeric value, MariaDB will not use the index on the column, as there are numerous alternatives that may evaluate as equal (see examples below).

Comparison Examples

Converting a string to a number:

```
SELECT 15+'15';
+---------+
| 15+'15' |
+---------+
|      30 |
+---------+
```

Converting a number to a string:

```
SELECT CONCAT(15,'15');
+-----------------+
| CONCAT(15,'15') |
+-----------------+
| 1515            |
+-----------------+
```

Floating point number errors:

```
SELECT '9746718491924563214' = 9746718491924563213;
+-------------------------------------------+
| '9746718491924563214' = 9746718491924563213 |
+-------------------------------------------+
|                                         1 |
+-------------------------------------------+
```

Numeric equivalence with strings:

```
SELECT '5' = 5;
+---------+
| '5' = 5 |
+---------+
|       1 |
+---------+

SELECT '   5' = 5;
+------------+
| '   5' = 5 |
+------------+
|          1 |
+------------+

SELECT '   5  ' = 5;
+--------------+
| '   5  ' = 5 |
+--------------+
|            1 |
+--------------+
1 row in set, 1 warning (0.000 sec)

SHOW WARNINGS;
+-------+------+-------------------------------------------+
| Level | Code | Message                                   |
+-------+------+-------------------------------------------+
| Note  | 1292 | Truncated incorrect DOUBLE value: '   5  ' |
+-------+------+-------------------------------------------+
```

As a result of the above, MariaDB cannot use the index when comparing a string with a numeric value in the example below:

```
CREATE TABLE t (a VARCHAR(10), b VARCHAR(10), INDEX idx_a (a));

INSERT INTO t VALUES
  ('1', '1'), ('2', '2'), ('3', '3'),
  ('4', '4'), ('5', '5'), ('1', '5');

EXPLAIN SELECT * FROM t WHERE a = '3' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: ref
possible_keys: idx_a
          key: idx_a
      key_len: 13
          ref: const
         rows: 1
        Extra: Using index condition

EXPLAIN SELECT * FROM t WHERE a = 3 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: ALL
possible_keys: idx_a
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
        Extra: Using where
```

Rules for Conversion on Dyadic Arithmetic Operations

Implicit type conversion also takes place on dyadic arithmetic operations (+,-,*,/). MariaDB chooses the minimum data type that is guaranteed to fit the result and converts both arguments to the result data type.

For addition (+), subtraction (-) and multiplication (*), the result data type is chosen as follows:

- If either of the arguments is an approximate number (float, double), the result is double.
- If either of the arguments is a string (char, varchar, text), the result is double.
- If either of the arguments is a decimal number, the result is decimal.
- If either of the arguments is of a temporal type with a non-zero fractional second precision (time(N), datetime(N), timestamp(N)), the result is decimal.
- If either of the arguments is of a temporal type with a zero fractional second precision (time(0), date, datetime(0), timestamp(0)), the result may vary between int, int unsigned, bigint or bigint unsigned, depending on the exact data type combination.
- If both arguments are integer numbers (tinyint, smallint, mediumint, bigint), the result may vary between int, int unsigned, bigint or bigint unsigned, depending of the exact data types and their signs.

For division (/), the result data type is chosen as follows:

- If either of the arguments is an approximate number (float, double), the result is double.
- If either of the arguments is a string (char, varchar, text), the result is double.
- Otherwise, the result is decimal.

Arithmetic Examples

Note, the above rules mean that when an argument of a temporal data type appears in addition or subtraction, it's treated as a number by default.

```
SELECT TIME'10:20:30' + 1;
+--------------------+
| TIME'10:20:30' + 1 |
+--------------------+
|             102031 |
+--------------------+
```

In order to do temporal addition or subtraction instead, use the DATE_ADD() or DATE_SUB() functions, or an INTERVAL expression as the second argument:

```
SELECT TIME'10:20:30' + INTERVAL 1 SECOND;
+-----------------------------------+
| TIME'10:20:30' + INTERVAL 1 SECOND |
+-----------------------------------+
| 10:20:31                          |
+-----------------------------------+
```

```
SELECT "2.2" + 3;
+-----------+
| "2.2" + 3 |
+-----------+
|       5.2 |
+-----------+

SELECT 2.2 + 3;
+---------+
| 2.2 + 3 |
+---------+
| 5.2     |
+---------+

SELECT 2.2 / 3;
+---------+
| 2.2 / 3 |
+---------+
| 0.73333 |
+---------+

SELECT "2.2" / 3;
+--------------------+
| "2.2" / 3          |
+--------------------+
| 0.7333333333333334 |
+--------------------+
```

1.2.3 Date & Time Functions

Functions for handling date and time, e.g. TIME, DATE, DAYNAME etc.

Microseconds in MariaDB

Microseconds have been supported since MariaDB 5.3.

Date and Time Units

Date or time units

ADD_MONTHS

Adds a number of months to a date.

ADDDATE

Add days or another interval to a date.

ADDTIME

Adds a time to a time or datetime.

CONVERT_TZ

Converts a datetime from one time zone to another.

CURDATE

Returns the current date.

CURRENT_DATE

Synonym for CURDATE().

CURRENT_TIME

Synonym for CURTIME().

CURRENT_TIMESTAMP
Synonym for NOW().

CURTIME
Returns the current time.

DATE FUNCTION
Extracts the date portion of a datetime.

DATEDIFF
Difference in days between two date/time values.

DATE_ADD
Date arithmetic - addition.

DATE_FORMAT
Formats the date value according to the format string.

DATE_SUB
Date arithmetic - subtraction.

DAY
Synonym for DAYOFMONTH().

DAYNAME
Return the name of the weekday.

DAYOFMONTH
Returns the day of the month.

DAYOFWEEK
Returns the day of the week index.

DAYOFYEAR
Returns the day of the year.

EXTRACT
Extracts a portion of the date.

FORMAT_PICO_TIME
Given a time in picoseconds, returns a human-readable time value and unit indicator.

FROM_DAYS
Returns a date given a day.

FROM_UNIXTIME
Returns a datetime from a Unix timestamp.

GET_FORMAT
Returns a format string.

HOUR
Returns the hour.

LAST_DAY
Returns the last day of the month.

LOCALTIME
Synonym for NOW().

LOCALTIMESTAMP
Synonym for NOW().

MAKEDATE

Returns a date given a year and day.

MAKETIME

Returns a time.

MICROSECOND

Returns microseconds from a date or datetime.

MINUTE

Returns a minute from 0 to 59.

MONTH

Returns a month from 1 to 12.

MONTHNAME

Returns the full name of the month.

NOW

Returns the current date and time.

PERIOD_ADD

Add months to a period.

PERIOD_DIFF

Number of months between two periods.

QUARTER

Returns year quarter from 1 to 4.

SECOND

Returns the second of a time.

SEC_TO_TIME

Converts a second to a time.

STR_TO_DATE

Converts a string to date.

SUBDATE

Subtract a date unit or number of days.

SUBTIME

Subtracts a time from a date/time.

SYSDATE

Returns the current date and time.

TIME Function

Extracts the time.

TIMEDIFF

Returns the difference between two date/times.

TIMESTAMP FUNCTION

Return the datetime, or add a time to a date/time.

TIMESTAMPADD

Add interval to a date or datetime.

TIMESTAMPDIFF

Difference between two datetimes.

TIME_FORMAT
Formats the time value according to the format string.

TIME_TO_SEC
Returns the time argument, converted to seconds.

TO_DAYS
Number of days since year 0.

TO_SECONDS
Number of seconds since year 0.

UNIX_TIMESTAMP
Returns a Unix timestamp.

UTC_DATE
Returns the current UTC date.

UTC_TIME
Returns the current UTC time.

UTC_TIMESTAMP
Returns the current UTC date and time.

WEEK
Returns the week number.

WEEKDAY
Returns the weekday index.

WEEKOFYEAR
Returns the calendar week of the date as a number in the range from 1 to 53.

YEAR
Returns the year for the given date.

YEARWEEK
Returns year and week for a date.

There are 6 related questions.

1.2.3.1 Microseconds in MariaDB

Contents

The TIME, DATETIME, and TIMESTAMP types, along with the temporal functions, CAST and dynamic columns, support microseconds. The datetime precision of a column can be specified when creating the table with CREATE TABLE, for example:

```
CREATE TABLE example(
  col_microsec DATETIME(6),
  col_millisec TIME(3)
);
```

Generally, the precision can be specified for any `TIME`, `DATETIME`, or `TIMESTAMP` column, in parentheses, after the type name. The datetime precision specifies number of digits after the decimal dot and can be any integer number from 0 to 6. If no precision is specified it is assumed to be 0, for backward compatibility reasons.

A datetime precision can be specified wherever a type name is used. For example:

- when declaring arguments of stored routines.

- when specifying a return type of a stored function.
- when declaring variables.
- in a `CAST` function:

```
create function example(x datetime(5)) returns time(4)
begin
  declare y timestamp(6);
  return cast(x as time(2));
end;
```

`%f` is used as the formatting option for microseconds in the STR_TO_DATE, DATE_FORMAT and FROM_UNIXTIME functions, for example:

```
SELECT STR_TO_DATE('20200809 020917076','%Y%m%d %H%i%s%f');
+-------------------------------------------------------+
| STR_TO_DATE('20200809 020917076','%Y%m%d %H%i%s%f') |
+-------------------------------------------------------+
| 2020-08-09 02:09:17.076000                            |
+-------------------------------------------------------+
```

Additional Information

- when comparing anything to a temporal value (`DATETIME` , `TIME` , `DATE` , or `TIMESTAMP`), both values are compared as temporal values, not as strings.
- The INFORMATION_SCHEMA.COLUMNS table has a new column `DATETIME_PRECISION`
- NOW(), CURTIME(), UTC_TIMESTAMP(), UTC_TIME(), CURRENT_TIME(), CURRENT_TIMESTAMP(), LOCALTIME() and LOCALTIMESTAMP() now accept datetime precision as an optional argument. For example:

```
SELECT CURTIME(4);
--> 10:11:12.3456
```

- TIME_TO_SEC() and UNIX_TIMESTAMP() preserve microseconds of the argument. These functions will return a decimal number if the result non-zero datetime precision and an integer otherwise (for backward compatibility).

```
SELECT TIME_TO_SEC('10:10:10.12345');
--> 36610.12345
```

- Current versions of this patch fix a bug in the following optimization: in certain queries with `DISTINCT` MariaDB can ignore this clause if it can prove that all result rows are unique anyway, for example, when a primary key is compared with a constant. Sometimes this optimization was applied incorrectly, though — for example, when comparing a string with a date constant. This is now fixed.
- `DATE_ADD()` and `DATE_SUB()` functions can now take a `TIME` expression as an argument (not just `DATETIME` as before).

```
SELECT TIME('10:10:10') + INTERVAL 100 MICROSECOND;
--> 10:10:10.000100
```

- The `event_time` field in the mysql.general_log table and the `start_time` , `query_time` , and `lock_time` fields in the mysql.slow_log table now store values with microsecond precision.
- This patch fixed a bug when comparing a temporal value using the `BETWEEN` operator and one of the operands is `NULL` .
- The old syntax `TIMESTAMP(N)` , where `N` is the display width, is no longer supported. It was deprecated in MySQL 4.1.0 (released on 2003-04-03).
- when a `DATETIME` value is compared to a `TIME` value, the latter is treated as a full datetime with a zero date part, similar to comparing `DATE` to a `DATETIME` , or to comparing `DECIMAL` numbers. Earlier versions of MariaDB used to compare only the time part of both operands in such a case.
- In MariaDB, an extra column `TIME_MS` has been added to the `INFORMATION_SCHEMA.PROCESSLIST` table, as well as to the output of `SHOW FULL PROCESSLIST` .

Note: When you convert a temporal value to a value with a smaller precision, it will be truncated, not rounded. This is done to guarantee that the date part is not changed. For example:

```
SELECT CAST('2009-12-31 23:59:59.998877' as DATETIME(3));
-> 2009-12-31 23:59:59.998
```

MySQL 5.6 Microseconds

MySQL 5.6 introduced microseconds using a slightly different implementation to MariaDB 5.3. Since MariaDB 10.1, MariaDB has defaulted to the MySQL format, by means of the --mysql56-temporal-format variable. The MySQL version requires slightly more storage but has some advantages in permitting the eventual support of negative dates, and in replication.

1.2.3.2 Date and Time Units

The `INTERVAL` keyword can be used to add or subtract a time interval of time to a `DATETIME`, `DATE` or `TIME` value.

The syntax is:

```
INTERVAL time_quantity time_unit
```

For example, the `SECOND` unit is used below by the `DATE_ADD()` function:

```
SELECT '2008-12-31 23:59:59' + INTERVAL 1 SECOND;
+---------------------------------------+
| '2008-12-31 23:59:59' + INTERVAL 1 SECOND |
+---------------------------------------+
| 2009-01-01 00:00:00                   |
+---------------------------------------+
```

The following units are valid:

Unit	Description
`MICROSECOND`	Microseconds
`SECOND`	Seconds
`MINUTE`	Minutes
`HOUR`	Hours
`DAY`	Days
`WEEK`	Weeks
`MONTH`	Months
`QUARTER`	Quarters
`YEAR`	Years
`SECOND_MICROSECOND`	Seconds.Microseconds
`MINUTE_MICROSECOND`	Minutes.Seconds.Microseconds
`MINUTE_SECOND`	Minutes.Seconds
`HOUR_MICROSECOND`	Hours.Minutes.Seconds.Microseconds
`HOUR_SECOND`	Hours.Minutes.Seconds
`HOUR_MINUTE`	Hours.Minutes
`DAY_MICROSECOND`	Days Hours.Minutes.Seconds.Microseconds
`DAY_SECOND`	Days Hours.Minutes.Seconds
`DAY_MINUTE`	Days Hours.Minutes
`DAY_HOUR`	Days Hours
`YEAR_MONTH`	Years-Months

The time units containing an underscore are composite; that is, they consist of multiple base time units. For base time units, `time_quantity` is an integer number. For composite units, the quantity must be expressed as a string with multiple integer numbers separated by any punctuation character.

Example of composite units:

```
INTERVAL '2:2' YEAR_MONTH
INTERVAL '1:30:30' HOUR_SECOND
INTERVAL '1!30!30' HOUR_SECOND -- same as above
```

Time units can be used in the following contexts:

- after a `+` or a `-` operator;
- with the following `DATE` or `TIME` functions: `ADDDATE()`, `SUBDATE()`, `DATE_ADD()`, `DATE_SUB()`, `TIMESTAMPADD()`, `TIMESTAMPDIFF()`, `EXTRACT()`;
- in the `ON SCHEDULE` clause of `CREATE EVENT` and `ALTER EVENT`.
- when defining a partitioning `BY SYSTEM_TIME`

1.2.3.3 ADD_MONTHS

MariaDB starting with 10.6.1

The ADD_MONTHS function was introduced in MariaDB 10.6.1 to enhance Oracle compatibility. Similar functionality can be achieved with the DATE_ADD function.

Syntax

```
ADD_MONTHS(date, months)
```

Contents

Description

`ADD_MONTHS` adds an integer *months* to a given *date* (DATE, DATETIME or TIMESTAMP), returning the resulting date.

months can be positive or negative. If months is not a whole number, then it will be rounded to the nearest whole number (not truncated).

The resulting day component will remain the same as that specified in *date*, unless the resulting month has fewer days than the day component of the given date, in which case the day will be the last day of the resulting month.

Returns NULL if given an invalid date, or a NULL argument.

Examples

```
SELECT ADD_MONTHS('2012-01-31', 2);
+-----------------------------+
| ADD_MONTHS('2012-01-31', 2) |
+-----------------------------+
| 2012-03-31                  |
+-----------------------------+

SELECT ADD_MONTHS('2012-01-31', -5);
+-----------------------------+
| ADD_MONTHS('2012-01-31', -5) |
+-----------------------------+
| 2011-08-31                   |
+-----------------------------+

SELECT ADD_MONTHS('2011-01-31', 1);
+-----------------------------+
| ADD_MONTHS('2011-01-31', 1) |
+-----------------------------+
| 2011-02-28                  |
+-----------------------------+

SELECT ADD_MONTHS('2012-01-31', 1);
+-----------------------------+
| ADD_MONTHS('2012-01-31', 1) |
+-----------------------------+
| 2012-02-29                  |
+-----------------------------+

SELECT ADD_MONTHS('2012-01-31', 2);
+-----------------------------+
| ADD_MONTHS('2012-01-31', 2) |
+-----------------------------+
| 2012-03-31                  |
+-----------------------------+

SELECT ADD_MONTHS('2012-01-31', 3);
+-----------------------------+
| ADD_MONTHS('2012-01-31', 3) |
+-----------------------------+
| 2012-04-30                  |
+-----------------------------+

SELECT ADD_MONTHS('2011-01-15', 2.5);
+-------------------------------+
| ADD_MONTHS('2011-01-15', 2.5) |
+-------------------------------+
| 2011-04-15                    |
+-------------------------------+
1 row in set (0.001 sec)

SELECT ADD_MONTHS('2011-01-15', 2.6);
+-------------------------------+
| ADD_MONTHS('2011-01-15', 2.6) |
+-------------------------------+
| 2011-04-15                    |
+-------------------------------+
1 row in set (0.001 sec)

SELECT ADD_MONTHS('2011-01-15', 2.1);
+-------------------------------+
| ADD_MONTHS('2011-01-15', 2.1) |
+-------------------------------+
| 2011-03-15                    |
+-------------------------------+
1 row in set (0.004 sec)
```

1.2.3.4 ADDDATE

Syntax

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

Description

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 the discussion for `DATE_ADD()`.

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

Examples

```
SELECT DATE_ADD('2008-01-02', INTERVAL 31 DAY);
+----------------------------------------+
| DATE_ADD('2008-01-02', INTERVAL 31 DAY) |
+----------------------------------------+
| 2008-02-02                             |
+----------------------------------------+

SELECT ADDDATE('2008-01-02', INTERVAL 31 DAY);
+----------------------------------------+
| ADDDATE('2008-01-02', INTERVAL 31 DAY) |
+----------------------------------------+
| 2008-02-02                             |
+----------------------------------------+
```

```
SELECT ADDDATE('2008-01-02', 31);
+---------------------------+
| ADDDATE('2008-01-02', 31) |
+---------------------------+
| 2008-02-02                |
+---------------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT d, ADDDATE(d, 10) from t1;
+---------------------+---------------------+
| d                   | ADDDATE(d, 10)      |
+---------------------+---------------------+
| 2007-01-30 21:31:07 | 2007-02-09 21:31:07 |
| 1983-10-15 06:42:51 | 1983-10-25 06:42:51 |
| 2011-04-21 12:34:56 | 2011-05-01 12:34:56 |
| 2011-10-30 06:31:41 | 2011-11-09 06:31:41 |
| 2011-01-30 14:03:25 | 2011-02-09 14:03:25 |
| 2004-10-07 11:19:34 | 2004-10-17 11:19:34 |
+---------------------+---------------------+

SELECT d, ADDDATE(d, INTERVAL 10 HOUR) from t1;
+---------------------+-----------------------------+
| d                   | ADDDATE(d, INTERVAL 10 HOUR) |
+---------------------+-----------------------------+
| 2007-01-30 21:31:07 | 2007-01-31 07:31:07         |
| 1983-10-15 06:42:51 | 1983-10-15 16:42:51         |
| 2011-04-21 12:34:56 | 2011-04-21 22:34:56         |
| 2011-10-30 06:31:41 | 2011-10-30 16:31:41         |
| 2011-01-30 14:03:25 | 2011-01-31 00:03:25         |
| 2004-10-07 11:19:34 | 2004-10-07 21:19:34         |
+---------------------+-----------------------------+
```

1.2.3.5 ADDTIME

Syntax

```
ADDTIME(expr1,expr2)
```

Description

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

Examples

```
SELECT ADDTIME('2007-12-31 23:59:59.999999', '1 1:1:1.000002');
+----------------------------------------------------------+
| ADDTIME('2007-12-31 23:59:59.999999', '1 1:1:1.000002') |
+----------------------------------------------------------+
| 2008-01-02 01:01:01.000001                               |
+----------------------------------------------------------+

SELECT ADDTIME('01:00:00.999999', '02:00:00.999998');
+-----------------------------------------------+
| ADDTIME('01:00:00.999999', '02:00:00.999998') |
+-----------------------------------------------+
| 03:00:01.999997                               |
+-----------------------------------------------+
```

1.2.3.6 CONVERT_TZ

Syntax

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

Description

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.

In order to use named time zones, such as GMT, MET or Africa/Johannesburg, the time_zone tables must be loaded (see mysql_tzinfo_to_sql).

No conversion will take place if the value falls outside of the supported TIMESTAMP range ('1970-01-01 00:00:01' to '2038-01-19 05:14:07' UTC) when converted from *from_tz* to UTC.

This function returns NULL if the arguments are invalid (or named time zones have not been loaded).

See time zones for more information.

Examples

```
SELECT CONVERT_TZ('2016-01-01 12:00:00','+00:00','+10:00');
+-------------------------------------------------------+
| CONVERT_TZ('2016-01-01 12:00:00','+00:00','+10:00') |
+-------------------------------------------------------+
| 2016-01-01 22:00:00                                   |
+-------------------------------------------------------+
```

Using named time zones (with the time zone tables loaded):

```
SELECT CONVERT_TZ('2016-01-01 12:00:00','GMT','Africa/Johannesburg');
+-----------------------------------------------------------+
| CONVERT_TZ('2016-01-01 12:00:00','GMT','Africa/Johannesburg') |
+-----------------------------------------------------------+
| 2016-01-01 14:00:00                                       |
+-----------------------------------------------------------+
```

The value is out of the TIMESTAMP range, so no conversion takes place:

```
SELECT CONVERT_TZ('1969-12-31 22:00:00','+00:00','+10:00');
+-----------------------------------------------------+
| CONVERT_TZ('1969-12-31 22:00:00','+00:00','+10:00') |
+-----------------------------------------------------+
| 1969-12-31 22:00:00                                 |
+-----------------------------------------------------+
```

1.2.3.7 CURDATE

Syntax

```
CURDATE()
CURRENT_DATE
CURRENT_DATE()
```

Description

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

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

Examples

```
SELECT CURDATE();
+------------+
| CURDATE()  |
+------------+
| 2019-03-05 |
+------------+
```

In a numeric context (note this is not performing date calculations):

```
SELECT CURDATE() +0;
+--------------+
| CURDATE() +0 |
+--------------+
|     20190305 |
+--------------+
```

Data calculation:

```
SELECT CURDATE() - INTERVAL 5 DAY;
+----------------------------+
| CURDATE() - INTERVAL 5 DAY |
+----------------------------+
| 2019-02-28                 |
+----------------------------+
```

1.2.3.8 CURRENT_DATE

Syntax

```
CURRENT_DATE, CURRENT_DATE()
```

Description

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

1.2.3.9 CURRENT_TIME

Syntax

```
CURRENT_TIME
CURRENT_TIME([precision])
```

Description

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

1.2.3.10 CURRENT_TIMESTAMP

Syntax

```
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP([precision])
```

Description

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

1.2.3.11 CURTIME

Syntax

```
CURTIME([precision])
```

Description

Returns the current time as a value in 'HH:MM:SS' or HHMMSS.uuuuuu format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone.

The optional *precision* determines the microsecond precision. See Microseconds in MariaDB.

Examples

```
SELECT CURTIME();
+-----------+
| CURTIME() |
+-----------+
| 12:45:39  |
+-----------+

SELECT CURTIME() + 0;
+---------------+
| CURTIME() + 0 |
+---------------+
| 124545.000000 |
+---------------+
```

With precision:

```
SELECT CURTIME(2);
+-------------+
| CURTIME(2)  |
+-------------+
| 09:49:08.09 |
+-------------+
```

1.2.3.12 DATE FUNCTION

Syntax

```
DATE(expr)
```

Description

Extracts the date part of the date or datetime expression *expr*. Returns NULL and throws a warning when passed an invalid date.

Examples

```
SELECT DATE('2013-07-18 12:21:32');
+----------------------------+
| DATE('2013-07-18 12:21:32') |
+----------------------------+
| 2013-07-18                 |
+----------------------------+
```

1.2.3.13 DATEDIFF

Syntax

```
DATEDIFF(expr1,expr2)
```

Description

`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.

Examples

```
SELECT DATEDIFF('2007-12-31 23:59:59','2007-12-30');
+----------------------------------------------+
| DATEDIFF('2007-12-31 23:59:59','2007-12-30') |
+----------------------------------------------+
|                                            1 |
+----------------------------------------------+

SELECT DATEDIFF('2010-11-30 23:59:59','2010-12-31');
+----------------------------------------------+
| DATEDIFF('2010-11-30 23:59:59','2010-12-31') |
+----------------------------------------------+
|                                          -31 |
+----------------------------------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2011-05-23 10:56:05 |
+---------------------+

SELECT d, DATEDIFF(NOW(),d) FROM t1;
+---------------------+--------------------+
| d                   | DATEDIFF(NOW(),d)  |
+---------------------+--------------------+
| 2007-01-30 21:31:07 |               1574 |
| 1983-10-15 06:42:51 |              10082 |
| 2011-04-21 12:34:56 |                 32 |
| 2011-10-30 06:31:41 |               -160 |
| 2011-01-30 14:03:25 |                113 |
| 2004-10-07 11:19:34 |               2419 |
+---------------------+--------------------+
```

1.2.3.14 DATE_ADD

Syntax

```
DATE_ADD(date,INTERVAL expr unit)
```

Contents

Description

Performs date arithmetic. The *date* argument specifies the starting date or datetime value. *expr* is an expression specifying the interval value to be added to the starting date. *expr* is a string; it may start with a " - " for negative intervals. *unit* is a keyword indicating the units in which the expression should be interpreted. See Date and Time Units for a complete list of permitted units.

Examples

```
SELECT '2008-12-31 23:59:59' + INTERVAL 1 SECOND;
+-----------------------------------------+
| '2008-12-31 23:59:59' + INTERVAL 1 SECOND |
+-----------------------------------------+
| 2009-01-01 00:00:00                     |
+-----------------------------------------+
```

```
SELECT INTERVAL 1 DAY + '2008-12-31';
+-------------------------------+
| INTERVAL 1 DAY + '2008-12-31' |
+-------------------------------+
| 2009-01-01                    |
+-------------------------------+
```

```
SELECT '2005-01-01' - INTERVAL 1 SECOND;
+----------------------------------+
| '2005-01-01' - INTERVAL 1 SECOND |
+----------------------------------+
| 2004-12-31 23:59:59              |
+----------------------------------+
```

```
SELECT DATE_ADD('2000-12-31 23:59:59', INTERVAL 1 SECOND);
+----------------------------------------------------+
| DATE_ADD('2000-12-31 23:59:59', INTERVAL 1 SECOND) |
+----------------------------------------------------+
| 2001-01-01 00:00:00                                |
+----------------------------------------------------+
```

```
SELECT DATE_ADD('2010-12-31 23:59:59', INTERVAL 1 DAY);
+-------------------------------------------------+
| DATE_ADD('2010-12-31 23:59:59', INTERVAL 1 DAY) |
+-------------------------------------------------+
| 2011-01-01 23:59:59                             |
+-------------------------------------------------+
```

```
SELECT DATE_ADD('2100-12-31 23:59:59', INTERVAL '1:1' MINUTE_SECOND);
+-------------------------------------------------------------+
| DATE_ADD('2100-12-31 23:59:59', INTERVAL '1:1' MINUTE_SECOND) |
+-------------------------------------------------------------+
| 2101-01-01 00:01:00                                         |
+-------------------------------------------------------------+
```

```
SELECT DATE_ADD('1900-01-01 00:00:00', INTERVAL '-1 10' DAY_HOUR);
+-----------------------------------------------------------+
| DATE_ADD('1900-01-01 00:00:00', INTERVAL '-1 10' DAY_HOUR) |
+-----------------------------------------------------------+
| 1899-12-30 14:00:00                                       |
+-----------------------------------------------------------+
```

```
SELECT DATE_ADD('1992-12-31 23:59:59.000002', INTERVAL '1.999999' SECOND_MICROSECOND);
+--------------------------------------------------------------------------+
| DATE_ADD('1992-12-31 23:59:59.000002', INTERVAL '1.999999' SECOND_MICROSECOND) |
+--------------------------------------------------------------------------+
| 1993-01-01 00:00:01.000001                                               |
+--------------------------------------------------------------------------+
```

1.2.3.15 DATE_FORMAT

Syntax

```
DATE_FORMAT(date, format[, locale])
```

Contents

Description

Formats the date value according to the format string.

The language used for the names is controlled by the value of the lc_time_names system variable. See server locale for more on the supported locales.

The options that can be used by DATE_FORMAT(), as well as its inverse STR_TO_DATE() and the FROM_UNIXTIME() function, are:

Option	Description
%a	Short weekday name in current locale (Variable lc_time_names).
%b	Short form month name in current locale. For locale en_US this is one of: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov or Dec.
%c	Month with 1 or 2 digits.
%D	Day with English suffix 'th', 'nd', 'st' or 'rd''. (1st, 2nd, 3rd...).
%d	Day with 2 digits.
%e	Day with 1 or 2 digits.
%f	Microseconds 6 digits.
%H	Hour with 2 digits between 00-23.
%h	Hour with 2 digits between 01-12.
%I	Hour with 2 digits between 01-12.
%i	Minute with 2 digits.
%j	Day of the year (001-366)
%k	Hour with 1 digits between 0-23.
%l	Hour with 1 digits between 1-12.
%M	Full month name in current locale (Variable lc_time_names).
%m	Month with 2 digits.
%p	AM/PM according to current locale (Variable lc_time_names).
%r	Time in 12 hour format, followed by AM/PM. Short for '%I:%i:%S %p'.
%S	Seconds with 2 digits.
%s	Seconds with 2 digits.
%T	Time in 24 hour format. Short for '%H:%i:%S'.
%U	Week number (00-53), when first day of the week is Sunday.
%u	Week number (00-53), when first day of the week is Monday.
%V	Week number (01-53), when first day of the week is Sunday. Used with %X.
%v	Week number (01-53), when first day of the week is Monday. Used with %x.
%W	Full weekday name in current locale (Variable lc_time_names).
%w	Day of the week. 0 = Sunday, 6 = Saturday.
%X	Year with 4 digits when first day of the week is Sunday. Used with %V.
%x	Year with 4 digits when first day of the week is Monday. Used with %v.

`%Y`	Year with 4 digits.
`%y`	Year with 2 digits.
`%Z`	Timezone abbreviation. From MariaDB 11.3.0.
`%z`	Numeric timezone +hhmm or -hhmm presenting the hour and minute offset from UTC. From MariaDB 11.3.0.
`%#`	For str_to_date(), skip all numbers.
`%.`	For str_to_date(), skip all punctation characters.
`%@`	For str_to_date(), skip all alpha characters.
`%%`	A literal `%` character.

To get a date in one of the standard formats, `GET_FORMAT()` can be used.

Examples

```
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y');
+-----------------------------------------+
| DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y') |
+-----------------------------------------+
| Sunday October 2009                     |
+-----------------------------------------+

SELECT DATE_FORMAT('2007-10-04 22:23:00', '%H:%i:%s');
+-----------------------------------------+
| DATE_FORMAT('2007-10-04 22:23:00', '%H:%i:%s') |
+-----------------------------------------+
| 22:23:00                                |
+-----------------------------------------+

SELECT DATE_FORMAT('1900-10-04 22:23:00', '%D %y %a %d %m %b %j');
+-----------------------------------------------------------+
| DATE_FORMAT('1900-10-04 22:23:00', '%D %y %a %d %m %b %j') |
+-----------------------------------------------------------+
| 4th 00 Thu 04 10 Oct 277                                  |
+-----------------------------------------------------------+

SELECT DATE_FORMAT('1997-10-04 22:23:00', '%H %k %I %r %T %S %w');
+-----------------------------------------------------------+
| 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                        |
+-----------------------------------------------------------+

SELECT DATE_FORMAT('1999-01-01', '%X %V');
+-----------------------------------+
| DATE_FORMAT('1999-01-01', '%X %V') |
+-----------------------------------+
| 1998 52                           |
+-----------------------------------+

SELECT DATE_FORMAT('2006-06-00', '%d');
+---------------------------------+
| DATE_FORMAT('2006-06-00', '%d') |
+---------------------------------+
| 00                              |
+---------------------------------+
```

Optionally, the locale can be explicitly specified as the third DATE_FORMAT() argument. Doing so makes the function independent from the session settings, and the three argument version of DATE_FORMAT() can be used in virtual indexed and persistent generated-columns:

```
SELECT DATE_FORMAT('2006-01-01', '%W', 'el_GR');
+---------------------------------------+
| DATE_FORMAT('2006-01-01', '%W', 'el_GR') |
+---------------------------------------+
| Κυριακή                               |
+---------------------------------------+
```

From MariaDB 11.3, the timezone information:

```
SELECT DATE_FORMAT(NOW(), '%W %d %M %Y %H:%i:%s %Z %z');
+-------------------------------------------------+
| DATE_FORMAT(NOW(), '%W %d %M %Y %H:%i:%s %Z %z') |
+-------------------------------------------------+
| Wednesday 20 September 2023 15:00:23 SAST +0200  |
+-------------------------------------------------+
```

1.2.3.16 DATE_SUB

Syntax

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

Description

Performs date arithmetic. The *date* argument specifies the starting date or datetime value. *expr* is an expression specifying the interval value to be subtracted from the starting date. *expr* is a string; it may start with a " - " for negative intervals. *unit* is a keyword indicating the units in which the expression should be interpreted. See Date and Time Units for a complete list of permitted units.

See also `DATE_ADD()`.

Examples

```
SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
+----------------------------------------+
| DATE_SUB('1998-01-02', INTERVAL 31 DAY) |
+----------------------------------------+
| 1997-12-02                             |
+----------------------------------------+
```

```
SELECT DATE_SUB('2005-01-01 00:00:00', INTERVAL '1 1:1:1' DAY_SECOND);
+-----------------------------------------------------------------+
| DATE_SUB('2005-01-01 00:00:00', INTERVAL '1 1:1:1' DAY_SECOND) |
+-----------------------------------------------------------------+
| 2004-12-30 22:58:59                                             |
+-----------------------------------------------------------------+
```

1.2.3.17 DAY

Syntax

```
DAY(date)
```

Description

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

1.2.3.18 DAYNAME

Syntax

```
DAYNAME(date)
```

Description

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 server locale for more on the supported locales.

Examples

```
SELECT DAYNAME('2007-02-03');
+----------------------+
| DAYNAME('2007-02-03') |
+----------------------+
| Saturday             |
+----------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT d, DAYNAME(d) FROM t1;
+---------------------+------------+
| d                   | DAYNAME(d) |
+---------------------+------------+
| 2007-01-30 21:31:07 | Tuesday    |
| 1983-10-15 06:42:51 | Saturday   |
| 2011-04-21 12:34:56 | Thursday   |
| 2011-10-30 06:31:41 | Sunday     |
| 2011-01-30 14:03:25 | Sunday     |
| 2004-10-07 11:19:34 | Thursday   |
+---------------------+------------+
```

Changing the locale:

```
SET lc_time_names = 'fr_CA';

SELECT DAYNAME('2013-04-01');
+----------------------+
| DAYNAME('2013-04-01') |
+----------------------+
| lundi                |
+----------------------+
```

1.2.3.19 DAYOFMONTH

Syntax

```
DAYOFMONTH(date)
```

Description

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'` which have a zero day part.

DAY() is a synonym.

Examples

```
SELECT DAYOFMONTH('2007-02-03');
+--------------------------+
| DAYOFMONTH('2007-02-03') |
+--------------------------+
|                        3 |
+--------------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT d FROM t1 where DAYOFMONTH(d) = 30;
+---------------------+
| d                   |
+---------------------+
| 2007-01-30 21:31:07 |
| 2011-10-30 06:31:41 |
| 2011-01-30 14:03:25 |
+---------------------+
```

1.2.3.20 DAYOFWEEK

Syntax

```
DAYOFWEEK(date)
```

Description

Returns the day of the week index for the date (1 = Sunday, 2 = Monday, ..., 7 = Saturday). These index values correspond to the ODBC standard.

This contrasts with `WEEKDAY()` which follows a different index numbering (`0` = Monday, `1` = Tuesday, ... `6` = Sunday).

Examples

```
SELECT DAYOFWEEK('2007-02-03');
+--------------------------+
| DAYOFWEEK('2007-02-03') |
+--------------------------+
|                        7 |
+--------------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT d, DAYNAME(d), DAYOFWEEK(d), WEEKDAY(d) from t1;
+---------------------+------------+--------------+------------+
| d                   | DAYNAME(d) | DAYOFWEEK(d) | WEEKDAY(d) |
+---------------------+------------+--------------+------------+
| 2007-01-30 21:31:07 | Tuesday    |            3 |          1 |
| 1983-10-15 06:42:51 | Saturday   |            7 |          5 |
| 2011-04-21 12:34:56 | Thursday   |            5 |          3 |
| 2011-10-30 06:31:41 | Sunday     |            1 |          6 |
| 2011-01-30 14:03:25 | Sunday     |            1 |          6 |
| 2004-10-07 11:19:34 | Thursday   |            5 |          3 |
+---------------------+------------+--------------+------------+
```

1.2.3.21 DAYOFYEAR

Syntax

```
DAYOFYEAR(date)
```

Description

Returns the day of the year for date, in the range 1 to 366.

Examples

```
SELECT DAYOFYEAR('2018-02-16');
+-------------------------+
| DAYOFYEAR('2018-02-16') |
+-------------------------+
|                      47 |
+-------------------------+
```

1.2.3.22 EXTRACT

Syntax

```
EXTRACT(unit FROM date)
```

Contents

Description

The EXTRACT() function extracts the required unit from the date. See Date and Time Units for a complete list of permitted units.

In MariaDB 10.0.7 and MariaDB 5.5.35, `EXTRACT (HOUR FROM ...)` was changed to return a value from 0 to 23, adhering to the SQL standard. Until MariaDB 10.0.6 and MariaDB 5.5.34, and in all versions of MySQL at least as of MySQL 5.7, it could return a value > 23. HOUR() is not a standard function, so continues to adhere to the old behaviour inherited from MySQL.

Examples

```
SELECT EXTRACT(YEAR FROM '2009-07-02');
+---------------------------------+
| EXTRACT(YEAR FROM '2009-07-02') |
+---------------------------------+
|                            2009 |
+---------------------------------+

SELECT EXTRACT(YEAR_MONTH FROM '2009-07-02 01:02:03');
+-----------------------------------------------+
| EXTRACT(YEAR_MONTH FROM '2009-07-02 01:02:03') |
+-----------------------------------------------+
|                                        200907 |
+-----------------------------------------------+

SELECT EXTRACT(DAY_MINUTE FROM '2009-07-02 01:02:03');
+-----------------------------------------------+
| EXTRACT(DAY_MINUTE FROM '2009-07-02 01:02:03') |
+-----------------------------------------------+
|                                         20102 |
+-----------------------------------------------+

SELECT EXTRACT(MICROSECOND FROM '2003-01-02 10:30:00.000123');
+-------------------------------------------------------+
| EXTRACT(MICROSECOND FROM '2003-01-02 10:30:00.000123') |
+-------------------------------------------------------+
|                                                   123 |
+-------------------------------------------------------+
```

From MariaDB 10.0.7 and MariaDB 5.5.35, `EXTRACT (HOUR FROM...)` returns a value from 0 to 23, as per the SQL standard. `HOUR` is not a standard function, so continues to adhere to the old behaviour inherited from MySQL.

```
SELECT EXTRACT(HOUR FROM '26:30:00'), HOUR('26:30:00');
+--------------------------------+------------------+
| EXTRACT(HOUR FROM '26:30:00') | HOUR('26:30:00') |
+--------------------------------+------------------+
|                              2 |               26 |
+--------------------------------+------------------+
```

1.2.3.23 FORMAT_PICO_TIME

MariaDB starting with 11.0.2
Introduced in MariaDB 11.0.2

Syntax

```
FORMAT_PICO_TIME(time_val)
```

Contents

Description

Given a time in picoseconds, returns a human-readable time value and unit indicator. Resulting unit is dependent on the length of the argument, and can be:

- ps - picoseconds
- ns - nanoseconds
- us - microseconds
- ms - milliseconds
- s - seconds
- min - minutes
- h - hours

- d - days

With the exception of results under one nanosecond, which are not rounded and are represented as whole numbers, the result is rounded to 2 decimal places, with a minimum of 3 significant digits.

Returns NULL if the argument is NULL.

This function is very similar to the Sys Schema FORMAT_TIME function, but with the following differences:

- Represents minutes as `min` rather than `m`.
- Does not represent weeks.

Examples

```
SELECT
    FORMAT_PICO_TIME(43) AS ps,
    FORMAT_PICO_TIME(4321) AS ns,
    FORMAT_PICO_TIME(43211234) AS us,
    FORMAT_PICO_TIME(432112344321) AS ms,
    FORMAT_PICO_TIME(43211234432123) AS s,
    FORMAT_PICO_TIME(432112344321234) AS m,
    FORMAT_PICO_TIME(4321123443212345) AS h,
    FORMAT_PICO_TIME(432112344321234545) AS d;
+--------+---------+----------+-----------+---------+----------+--------+--------+
| ps     | ns      | us       | ms        | s       | m        | h      | d      |
+--------+---------+----------+-----------+---------+----------+--------+--------+
|  43 ps | 4.32 ns | 43.21 us | 432.11 ms | 43.21 s | 7.20 min | 1.20 h | 5.00 d |
+--------+---------+----------+-----------+---------+----------+--------+--------+
```

1.2.3.24 FROM_DAYS

Syntax

```
FROM_DAYS(N)
```

Description

Given a day number N, returns a DATE value. The day count is based on the number of days from the start of the standard calendar (0000-00-00).

The function is not designed for use with dates before the advent of the Gregorian calendar in October 1582. Results will not be reliable since it doesn't account for the lost days when the calendar changed from the Julian calendar.

This is the converse of the TO_DAYS() function.

Examples

```
SELECT FROM_DAYS(730669);
+-------------------+
| FROM_DAYS(730669) |
+-------------------+
| 2000-07-03        |
+-------------------+
```

1.2.3.25 FROM_UNIXTIME

Syntax

```
FROM_UNIXTIME(unix_timestamp), FROM_UNIXTIME(unix_timestamp,format)
```

Contents

Description

Returns a representation of the unix_timestamp argument as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS.uuuuuu format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone. unix_timestamp is an internal timestamp value such as is produced by the UNIX_TIMESTAMP() function.

If format is given, the result is formatted according to the format string, which is used the same way as listed in the entry for the DATE_FORMAT() function.

Timestamps in MariaDB have a maximum value of 2147483647, equivalent to 2038-01-19 05:14:07. This is due to the underlying 32-bit limitation. Using the function on a timestamp beyond this will result in NULL being returned. Use DATETIME as a storage type if you require dates beyond this.

The options that can be used by FROM_UNIXTIME(), as well as DATE_FORMAT() and STR_TO_DATE(), are:

Option	Description
%a	Short weekday name in current locale (Variable lc_time_names).
%b	Short form month name in current locale. For locale en_US this is one of: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov or Dec.
%c	Month with 1 or 2 digits.
%D	Day with English suffix 'th', 'nd', 'st' or 'rd''. (1st, 2nd, 3rd...).
%d	Day with 2 digits.
%e	Day with 1 or 2 digits.
%f	Microseconds 6 digits.
%H	Hour with 2 digits between 00-23.
%h	Hour with 2 digits between 01-12.
%I	Hour with 2 digits between 01-12.
%i	Minute with 2 digits.
%j	Day of the year (001-366)
%k	Hour with 1 digits between 0-23.
%l	Hour with 1 digits between 1-12.
%M	Full month name in current locale (Variable lc_time_names).
%m	Month with 2 digits.
%p	AM/PM according to current locale (Variable lc_time_names).
%r	Time in 12 hour format, followed by AM/PM. Short for '%I:%i:%S %p'.
%S	Seconds with 2 digits.
%s	Seconds with 2 digits.
%T	Time in 24 hour format. Short for '%H:%i:%S'.
%U	Week number (00-53), when first day of the week is Sunday.
%u	Week number (00-53), when first day of the week is Monday.
%V	Week number (01-53), when first day of the week is Sunday. Used with %X.
%v	Week number (01-53), when first day of the week is Monday. Used with %x.
%W	Full weekday name in current locale (Variable lc_time_names).

`%w`	Day of the week. 0 = Sunday, 6 = Saturday.
`%X`	Year with 4 digits when first day of the week is Sunday. Used with %V.
`%x`	Year with 4 digits when first day of the week is Sunday. Used with %v.
`%Y`	Year with 4 digits.
`%y`	Year with 2 digits.
`%#`	For str_to_date(), skip all numbers.
`%.`	For str_to_date(), skip all punctation characters.
`%@`	For str_to_date(), skip all alpha characters.
`%%`	A literal `%` character.

Performance Considerations

If your session time zone is set to `SYSTEM` (the default), `FROM_UNIXTIME()` will call the OS function to convert the data using the system time zone. At least on Linux, the corresponding function (`localtime_r`) uses a global mutex inside glibc that can cause contention under high concurrent load.

Set your time zone to a named time zone to avoid this issue. See mysql time zone tables for details on how to do this.

Examples

```
SELECT FROM_UNIXTIME(1196440219);
+---------------------------+
| FROM_UNIXTIME(1196440219) |
+---------------------------+
| 2007-11-30 11:30:19       |
+---------------------------+

SELECT FROM_UNIXTIME(1196440219) + 0;
+------------------------------+
| FROM_UNIXTIME(1196440219) + 0 |
+------------------------------+
|         20071130113019.000000 |
+------------------------------+

SELECT FROM_UNIXTIME(UNIX_TIMESTAMP(), '%Y %D %M %h:%i:%s %x');
+-----------------------------------------------------------+
| FROM_UNIXTIME(UNIX_TIMESTAMP(), '%Y %D %M %h:%i:%s %x') |
+-----------------------------------------------------------+
| 2010 27th March 01:03:47 2010                             |
+-----------------------------------------------------------+
```

1.2.3.26 GET_FORMAT

Syntax

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

Description

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

Possible result formats are:

Function Call	Result Format
GET_FORMAT(DATE,'EUR')	'%d.%m.%Y'

GET_FORMAT(DATE,'USA')	'%m.%d.%Y'
GET_FORMAT(DATE,'JIS')	'%Y-%m-%d'
GET_FORMAT(DATE,'ISO')	'%Y-%m-%d'
GET_FORMAT(DATE,'INTERNAL')	'%Y%m%d'
GET_FORMAT(DATETIME,'EUR')	'%Y-%m-%d %H.%i.%s'
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,'INTERNAL')	'%Y%m%d%H%i%s'
GET_FORMAT(TIME,'EUR')	'%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,'INTERNAL')	'%H%i%s'

Examples

Obtaining the string matching to the standard European date format:

```
SELECT GET_FORMAT(DATE, 'EUR');
+-------------------------+
| GET_FORMAT(DATE, 'EUR') |
+-------------------------+
| %d.%m.%Y                |
+-------------------------+
```

Using the same string to format a date:

```
SELECT DATE_FORMAT('2003-10-03',GET_FORMAT(DATE,'EUR'));
+------------------------------------------------+
| DATE_FORMAT('2003-10-03',GET_FORMAT(DATE,'EUR')) |
+------------------------------------------------+
| 03.10.2003                                     |
+------------------------------------------------+

SELECT STR_TO_DATE('10.31.2003',GET_FORMAT(DATE,'USA'));
+------------------------------------------------+
| STR_TO_DATE('10.31.2003',GET_FORMAT(DATE,'USA')) |
+------------------------------------------------+
| 2003-10-31                                     |
+------------------------------------------------+
```

1.2.3.27 HOUR

Syntax

```
HOUR(time)
```

Description

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.

The return value is always positive, even if a negative `TIME` value is provided.

Examples

```
SELECT HOUR('10:05:03');
+------------------+
| HOUR('10:05:03') |
+------------------+
|               10 |
+------------------+

SELECT HOUR('272:59:59');
+--------------------+
| HOUR('272:59:59') |
+--------------------+
|                272 |
+--------------------+
```

Difference between `EXTRACT (HOUR FROM ...)` (>= MariaDB 10.0.7 and MariaDB 5.5.35) and `HOUR`:

```
SELECT EXTRACT(HOUR FROM '26:30:00'), HOUR('26:30:00');
+--------------------------------+------------------+
| EXTRACT(HOUR FROM '26:30:00') | HOUR('26:30:00') |
+--------------------------------+------------------+
|                              2 |               26 |
+--------------------------------+------------------+
```

1.2.3.28 LAST_DAY

Syntax

```
LAST_DAY(date)
```

Description

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.

Examples

```
SELECT LAST_DAY('2003-02-05');
+-----------------------+
| LAST_DAY('2003-02-05') |
+-----------------------+
| 2003-02-28            |
+-----------------------+

SELECT LAST_DAY('2004-02-05');
+-----------------------+
| LAST_DAY('2004-02-05') |
+-----------------------+
| 2004-02-29            |
+-----------------------+

SELECT LAST_DAY('2004-01-01 01:01:01');
+--------------------------------+
| LAST_DAY('2004-01-01 01:01:01') |
+--------------------------------+
| 2004-01-31                     |
+--------------------------------+

SELECT LAST_DAY('2003-03-32');
+-----------------------+
| LAST_DAY('2003-03-32') |
+-----------------------+
| NULL                  |
+-----------------------+
1 row in set, 1 warning (0.00 sec)

Warning (Code 1292): Incorrect datetime value: '2003-03-32'
```

1.2.3.29 LOCALTIME

Syntax

```
LOCALTIME
LOCALTIME([precision])
```

Description

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

1.2.3.30 LOCALTIMESTAMP

Syntax

```
LOCALTIMESTAMP
LOCALTIMESTAMP([precision])
```

Description

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

1.2.3.31 MAKEDATE

Syntax

```
MAKEDATE(year,dayofyear)
```

Description

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

Examples

```
SELECT MAKEDATE(2011,31), MAKEDATE(2011,32);
+-------------------+-------------------+
| MAKEDATE(2011,31) | MAKEDATE(2011,32) |
+-------------------+-------------------+
| 2011-01-31        | 2011-02-01        |
+-------------------+-------------------+

SELECT MAKEDATE(2011,365), MAKEDATE(2014,365);
+--------------------+--------------------+
| MAKEDATE(2011,365) | MAKEDATE(2014,365) |
+--------------------+--------------------+
| 2011-12-31         | 2014-12-31         |
+--------------------+--------------------+

SELECT MAKEDATE(2011,0);
+------------------+
| MAKEDATE(2011,0) |
+------------------+
| NULL             |
+------------------+
```

1.2.3.32 MAKETIME

Syntax

```
MAKETIME(hour,minute,second)
```

Description

Returns a time value calculated from the `hour`, `minute`, and `second` arguments.

If `minute` or `second` are out of the range 0 to 60, NULL is returned. The `hour` can be in the range -838 to 838, outside of which the value is truncated with a warning.

Examples

```
SELECT MAKETIME(13,57,33);
+--------------------+
| MAKETIME(13,57,33) |
+--------------------+
| 13:57:33           |
+--------------------+

SELECT MAKETIME(-13,57,33);
+---------------------+
| MAKETIME(-13,57,33) |
+---------------------+
| -13:57:33           |
+---------------------+

SELECT MAKETIME(13,67,33);
+--------------------+
| MAKETIME(13,67,33) |
+--------------------+
| NULL               |
+--------------------+

SELECT MAKETIME(-1000,57,33);
+-----------------------+
| MAKETIME(-1000,57,33) |
+-----------------------+
| -838:59:59            |
+-----------------------+
1 row in set, 1 warning (0.00 sec)

SHOW WARNINGS;
+---------+------+---------------------------------------------+
| Level   | Code | Message                                     |
+---------+------+---------------------------------------------+
| Warning | 1292 | Truncated incorrect time value: '-1000:57:33' |
+---------+------+---------------------------------------------+
```

1.2.3.33 MICROSECOND

Syntax

```
MICROSECOND(expr)
```

Contents

Description

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

If *expr* is a time with no microseconds, zero is returned, while if *expr* is a date with no time, zero with a warning is returned.

Examples

```
SELECT MICROSECOND('12:00:00.123456');
+-------------------------------+
| MICROSECOND('12:00:00.123456') |
+-------------------------------+
|                         123456 |
+-------------------------------+

SELECT MICROSECOND('2009-12-31 23:59:59.000010');
+----------------------------------------+
| MICROSECOND('2009-12-31 23:59:59.000010') |
+----------------------------------------+
|                                     10 |
+----------------------------------------+

SELECT MICROSECOND('2013-08-07 12:13:14');
+-----------------------------------+
| MICROSECOND('2013-08-07 12:13:14') |
+-----------------------------------+
|                                 0 |
+-----------------------------------+

SELECT MICROSECOND('2013-08-07');
+---------------------------+
| MICROSECOND('2013-08-07') |
+---------------------------+
|                         0 |
+---------------------------+
1 row in set, 1 warning (0.00 sec)

SHOW WARNINGS;
+---------+------+-----------------------------------------------+
| Level   | Code | Message                                       |
+---------+------+-----------------------------------------------+
| Warning | 1292 | Truncated incorrect time value: '2013-08-07' |
+---------+------+-----------------------------------------------+
```

1.2.3.34 MINUTE

Syntax

```
MINUTE(time)
```

Description

Returns the minute for *time*, in the range 0 to 59.

Examples

```
SELECT MINUTE('2013-08-03 11:04:03');
+-------------------------------+
| MINUTE('2013-08-03 11:04:03') |
+-------------------------------+
|                             4 |
+-------------------------------+

 SELECT MINUTE ('23:12:50');
+---------------------+
| MINUTE ('23:12:50') |
+---------------------+
|                  12 |
+---------------------+
```

1.2.3.35 MONTH

Syntax

```
MONTH(date)
```

Description

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.

Examples

```
SELECT MONTH('2019-01-03');
+---------------------+
| MONTH('2019-01-03') |
+---------------------+
|                   1 |
+---------------------+

SELECT MONTH('2019-00-03');
+---------------------+
| MONTH('2019-00-03') |
+---------------------+
|                   0 |
+---------------------+
```

1.2.3.36 MONTHNAME

Syntax

```
MONTHNAME(date)
```

Description

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. See server locale for more on the supported locales.

Examples

```
SELECT MONTHNAME('2019-02-03');
+-------------------------+
| MONTHNAME('2019-02-03') |
+-------------------------+
| February                |
+-------------------------+
```

Changing the locale:

```
SET lc_time_names = 'fr_CA';

SELECT MONTHNAME('2019-05-21');
+-------------------------+
| MONTHNAME('2019-05-21') |
+-------------------------+
| mai                     |
+-------------------------+
```

1.2.3.37 NOW

Syntax

```
NOW([precision])
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP([precision])
LOCALTIME, LOCALTIME([precision])
LOCALTIMESTAMP
LOCALTIMESTAMP([precision])
```

Contents

Description

Returns the current date and time as a value in `'YYYY-MM-DD HH:MM:SS'` or `YYYYMMDDHHMMSS.uuuuuu` format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone.

The optional *precision* determines the microsecond precision. See Microseconds in MariaDB.

`NOW()` (or its synonyms) can be used as the default value for TIMESTAMP columns as well as, since MariaDB 10.0.1, DATETIME columns. Before MariaDB 10.0.1, it was only possible for a single TIMESTAMP column per table to contain the CURRENT_TIMESTAMP as its default.

When displayed in the INFORMATION_SCHEMA.COLUMNS table, a default CURRENT TIMESTAMP is displayed as `CURRENT_TIMESTAMP` up until MariaDB 10.2.2, and as `current_timestamp()` from MariaDB 10.2.3, due to to MariaDB 10.2 accepting expressions in the DEFAULT clause.

Changing the timestamp system variable with a SET `timestamp` statement affects the value returned by NOW(), but not by SYSDATE().

Examples

```
SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2010-03-27 13:13:25 |
+---------------------+

SELECT NOW() + 0;
+-----------------------+
| NOW() + 0             |
+-----------------------+
| 20100327131329.000000 |
+-----------------------+
```

With precision:

```
SELECT CURRENT_TIMESTAMP(2);
+------------------------+
| CURRENT_TIMESTAMP(2)   |
+------------------------+
| 2018-07-10 09:47:26.24 |
+------------------------+
```

Used as a default TIMESTAMP:

```
CREATE TABLE t (createdTS TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP);
```

From MariaDB 10.2.2:

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA='test'
  AND COLUMN_NAME LIKE '%ts%'\G
*************************** 1. row ***************************
           TABLE_CATALOG: def
            TABLE_SCHEMA: test
              TABLE_NAME: t
             COLUMN_NAME: ts
        ORDINAL_POSITION: 1
          COLUMN_DEFAULT: current_timestamp()
...
```

<= MariaDB 10.2.1

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA='test'
  AND COLUMN_NAME LIKE '%ts%'\G
*************************** 1. row ***************************
           TABLE_CATALOG: def
            TABLE_SCHEMA: test
              TABLE_NAME: t
             COLUMN_NAME: createdTS
        ORDINAL_POSITION: 1
          COLUMN_DEFAULT: CURRENT_TIMESTAMP
...
```

1.2.3.38 PERIOD_ADD

Syntax

```
PERIOD_ADD(P,N)
```

Description

Adds `N` months to period `P`. `P` is in the format YYMM or YYYYMM, and is not a date value. If `P` contains a two-digit year, values from 00 to 69 are converted to from 2000 to 2069, while values from 70 are converted to 1970 upwards.

Returns a value in the format YYYYMM.

Examples

```
SELECT PERIOD_ADD(200801,2);
+----------------------+
| PERIOD_ADD(200801,2) |
+----------------------+
|               200803 |
+----------------------+

SELECT PERIOD_ADD(6910,2);
+--------------------+
| PERIOD_ADD(6910,2) |
+--------------------+
|             206912 |
+--------------------+

SELECT PERIOD_ADD(7010,2);
+--------------------+
| PERIOD_ADD(7010,2) |
+--------------------+
|             197012 |
+--------------------+
```

1.2.3.39 PERIOD_DIFF

Syntax

```
PERIOD_DIFF(P1,P2)
```

Description

Returns the number of months between periods P1 and P2. P1 and P2 can be in the format `YYMM` or `YYYYMM`, and are not date values.

If P1 or P2 contains a two-digit year, values from 00 to 69 are converted to from 2000 to 2069, while values from 70 are converted to 1970 upwards.

Examples

```
SELECT PERIOD_DIFF(200802,200703);
+----------------------------+
| PERIOD_DIFF(200802,200703) |
+----------------------------+
|                         11 |
+----------------------------+

SELECT PERIOD_DIFF(6902,6803);
+------------------------+
| PERIOD_DIFF(6902,6803) |
+------------------------+
|                     11 |
+------------------------+

SELECT PERIOD_DIFF(7002,6803);
+------------------------+
| PERIOD_DIFF(7002,6803) |
+------------------------+
|                  -1177 |
+------------------------+
```

1.2.3.40 QUARTER

Syntax

```
QUARTER(date)
```

Description

Returns the quarter of the year for `date`, in the range 1 to 4. Returns 0 if month contains a zero value, or `NULL` if the given value is not otherwise a valid date (zero values are accepted).

Examples

```
SELECT QUARTER('2008-04-01');
+-----------------------+
| QUARTER('2008-04-01') |
+-----------------------+
|                     2 |
+-----------------------+

SELECT QUARTER('2019-00-01');
+-----------------------+
| QUARTER('2019-00-01') |
+-----------------------+
|                     0 |
+-----------------------+
```

1.2.3.41 SECOND

Syntax

```
SECOND(time)
```

Description

Returns the second for a given `time` (which can include microseconds), in the range 0 to 59, or `NULL` if not given a valid time value.

Examples

```
SELECT SECOND('10:05:03');
+--------------------+
| SECOND('10:05:03') |
+--------------------+
|                  3 |
+--------------------+

SELECT SECOND('10:05:01.999999');
+---------------------------+
| SECOND('10:05:01.999999') |
+---------------------------+
|                         1 |
+---------------------------+
```

1.2.3.42 SEC_TO_TIME

Syntax

```
SEC_TO_TIME(seconds)
```

Description

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 time will be returned in the format `hh:mm:ss`, or `hhmmss` if used in a numeric calculation.

Examples

```
SELECT SEC_TO_TIME(12414);
+--------------------+
| SEC_TO_TIME(12414) |
+--------------------+
| 03:26:54           |
+--------------------+

SELECT SEC_TO_TIME(12414)+0;
+----------------------+
| SEC_TO_TIME(12414)+0 |
+----------------------+
|                32654 |
+----------------------+

SELECT SEC_TO_TIME(9999999);
+----------------------+
| SEC_TO_TIME(9999999) |
+----------------------+
| 838:59:59            |
+----------------------+
1 row in set, 1 warning (0.00 sec)

SHOW WARNINGS;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Truncated incorrect time value: '9999999' |
+---------+------+----------------------------------------+
```

1.2.3.43 STR_TO_DATE

Syntax

```
STR_TO_DATE(str,format)
```

Contents

Description

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.

The date, time, or datetime values contained in `str` should be given in the format indicated by format. If str contains an illegal date, time, or datetime value, `STR_TO_DATE()` returns `NULL`. An illegal value also produces a warning.

Under specific SQL_MODE settings an error may also be generated if the `str` isn't a valid date:

- ALLOW_INVALID_DATES
- NO_ZERO_DATE
- NO_ZERO_IN_DATE

The options that can be used by STR_TO_DATE(), as well as its inverse DATE_FORMAT() and the FROM_UNIXTIME() function, are:

Option	Description
%a	Short weekday name in current locale (Variable lc_time_names).
%b	Short form month name in current locale. For locale en_US this is one of: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov or Dec.
%c	Month with 1 or 2 digits.

%D	Day with English suffix 'th', 'nd', 'st' or 'rd''. (1st, 2nd, 3rd...).
%d	Day with 2 digits.
%e	Day with 1 or 2 digits.
%f	Microseconds 6 digits.
%H	Hour with 2 digits between 00-23.
%h	Hour with 2 digits between 01-12.
%I	Hour with 2 digits between 01-12.
%i	Minute with 2 digits.
%j	Day of the year (001-366)
%k	Hour with 1 digits between 0-23.
%l	Hour with 1 digits between 1-12.
%M	Full month name in current locale (Variable lc_time_names).
%m	Month with 2 digits.
%p	AM/PM according to current locale (Variable lc_time_names).
%r	Time in 12 hour format, followed by AM/PM. Short for '%I:%i:%S %p'.
%S	Seconds with 2 digits.
%s	Seconds with 2 digits.
%T	Time in 24 hour format. Short for '%H:%i:%S'.
%U	Week number (00-53), when first day of the week is Sunday.
%u	Week number (00-53), when first day of the week is Monday.
%V	Week number (01-53), when first day of the week is Sunday. Used with %X.
%v	Week number (01-53), when first day of the week is Monday. Used with %x.
%W	Full weekday name in current locale (Variable lc_time_names).
%w	Day of the week. 0 = Sunday, 6 = Saturday.
%X	Year with 4 digits when first day of the week is Sunday. Used with %V.
%x	Year with 4 digits when first day of the week is Monday. Used with %v.
%Y	Year with 4 digits.
%y	Year with 2 digits.
%#	For str_to_date(), skip all numbers.
%.	For str_to_date(), skip all punctation characters.
%@	For str_to_date(), skip all alpha characters.
%%	A literal % character.

Examples

```
SELECT STR_TO_DATE('Wednesday, June 2, 2014', '%W, %M %e, %Y');
+-----------------------------------------------------+
| STR_TO_DATE('Wednesday, June 2, 2014', '%W, %M %e, %Y') |
+-----------------------------------------------------+
| 2014-06-02                                          |
+-----------------------------------------------------+


SELECT STR_TO_DATE('Wednesday23423, June 2, 2014', '%W, %M %e, %Y');
+-------------------------------------------------------------+
| STR_TO_DATE('Wednesday23423, June 2, 2014', '%W, %M %e, %Y') |
+-------------------------------------------------------------+
| NULL                                                        |
+-------------------------------------------------------------+
1 row in set, 1 warning (0.00 sec)

SHOW WARNINGS;
+---------+------+-----------------------------------------------------------------------------------------
| Level   | Code | Message
+---------+------+-----------------------------------------------------------------------------------------
| Warning | 1411 | Incorrect datetime value: 'Wednesday23423, June 2, 2014' for function str_to_d
+---------+------+-----------------------------------------------------------------------------------------

SELECT STR_TO_DATE('Wednesday23423, June 2, 2014', '%W%#, %M %e, %Y');
+-------------------------------------------------------------+
| STR_TO_DATE('Wednesday23423, June 2, 2014', '%W%#, %M %e, %Y') |
+-------------------------------------------------------------+
| 2014-06-02                                                  |
+-------------------------------------------------------------+
```

1.2.3.44 SUBDATE

Syntax

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

Description

When invoked with the `INTERVAL` form of the second argument, `SUBDATE()` is a synonym for `DATE_SUB()`. See Date and Time Units for a complete list of permitted units.

The second form allows 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.

Examples

```
SELECT DATE_SUB('2008-01-02', INTERVAL 31 DAY);
+----------------------------------------+
| DATE_SUB('2008-01-02', INTERVAL 31 DAY) |
+----------------------------------------+
| 2007-12-02                             |
+----------------------------------------+

SELECT SUBDATE('2008-01-02', INTERVAL 31 DAY);
+----------------------------------------+
| SUBDATE('2008-01-02', INTERVAL 31 DAY) |
+----------------------------------------+
| 2007-12-02                             |
+----------------------------------------+
```

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

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT d, SUBDATE(d, 10) from t1;
+---------------------+---------------------+
| d                   | SUBDATE(d, 10)      |
+---------------------+---------------------+
| 2007-01-30 21:31:07 | 2007-01-20 21:31:07 |
| 1983-10-15 06:42:51 | 1983-10-05 06:42:51 |
| 2011-04-21 12:34:56 | 2011-04-11 12:34:56 |
| 2011-10-30 06:31:41 | 2011-10-20 06:31:41 |
| 2011-01-30 14:03:25 | 2011-01-20 14:03:25 |
| 2004-10-07 11:19:34 | 2004-09-27 11:19:34 |
+---------------------+---------------------+

SELECT d, SUBDATE(d, INTERVAL 10 MINUTE) from t1;
+---------------------+-------------------------------+
| d                   | SUBDATE(d, INTERVAL 10 MINUTE) |
+---------------------+-------------------------------+
| 2007-01-30 21:31:07 | 2007-01-30 21:21:07            |
| 1983-10-15 06:42:51 | 1983-10-15 06:32:51            |
| 2011-04-21 12:34:56 | 2011-04-21 12:24:56            |
| 2011-10-30 06:31:41 | 2011-10-30 06:21:41            |
| 2011-01-30 14:03:25 | 2011-01-30 13:53:25            |
| 2004-10-07 11:19:34 | 2004-10-07 11:09:34            |
+---------------------+-------------------------------+
```

1.2.3.45 SUBTIME

Syntax

```
SUBTIME(expr1,expr2)
```

Description

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.

Examples

```
SELECT SUBTIME('2007-12-31 23:59:59.999999','1 1:1:1.000002');
+-----------------------------------------------------+
| SUBTIME('2007-12-31 23:59:59.999999','1 1:1:1.000002') |
+-----------------------------------------------------+
| 2007-12-30 22:58:58.999997                          |
+-----------------------------------------------------+

SELECT SUBTIME('01:00:00.999999', '02:00:00.999998');
+-----------------------------------------------+
| SUBTIME('01:00:00.999999', '02:00:00.999998') |
+-----------------------------------------------+
| -00:59:59.999999                              |
+-----------------------------------------------+
```

1.2.3.46 SYSDATE

Syntax

```
SYSDATE([precision])
```

Contents

Description

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS.uuuuuu format, depending on whether the function is used in a string or numeric context.

The optional *precision* determines the microsecond precision. See Microseconds in MariaDB.

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 routine or trigger, NOW() returns the time at which the routine or triggering statement began to execute.)

In addition, changing the timestamp system variable with a 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 non-deterministic and therefore unsafe for replication if statement-based binary logging is used. If that is a problem, you can use row-based logging, or start the server with the mysqld option --sysdate-is-now to cause SYSDATE() to be an alias for NOW(). The non-deterministic nature of SYSDATE() also means that indexes cannot be used for evaluating expressions that refer to it, and that statements using the SYSDATE() function are unsafe for statement-based replication.

Examples

Difference between NOW() and SYSDATE():

```
SELECT NOW(), SLEEP(2), NOW();
+---------------------+----------+---------------------+
| NOW()               | SLEEP(2) | NOW()               |
+---------------------+----------+---------------------+
| 2010-03-27 13:23:40 |        0 | 2010-03-27 13:23:40 |
+---------------------+----------+---------------------+

SELECT SYSDATE(), SLEEP(2), SYSDATE();
+---------------------+----------+---------------------+
| SYSDATE()           | SLEEP(2) | SYSDATE()           |
+---------------------+----------+---------------------+
| 2010-03-27 13:23:52 |        0 | 2010-03-27 13:23:54 |
+---------------------+----------+---------------------+
```

With precision:

```
SELECT SYSDATE(4);
+--------------------------+
| SYSDATE(4)               |
+--------------------------+
| 2018-07-10 10:17:13.1689 |
+--------------------------+
```

1.2.3.47 TIME Function

Syntax

```
TIME(expr)
```

Description

Extracts the time part of the time or datetime expression `expr` and returns it as a string.

Examples

```
SELECT TIME('2003-12-31 01:02:03');
+-----------------------------+
| TIME('2003-12-31 01:02:03') |
+-----------------------------+
| 01:02:03                    |
+-----------------------------+

SELECT TIME('2003-12-31 01:02:03.000123');
+--------------------------------------+
| TIME('2003-12-31 01:02:03.000123') |
+--------------------------------------+
| 01:02:03.000123                      |
+--------------------------------------+
```

1.2.3.48 TIMEDIFF

Syntax

```
TIMEDIFF(expr1,expr2)
```

Description

TIMEDIFF() returns `expr1 - expr2` expressed as a time value. `expr1` and `expr2` are time or date-and-time expressions, but both must be of the same type.

Examples

```
SELECT TIMEDIFF('2000:01:01 00:00:00', '2000:01:01 00:00:00.000001');
+----------------------------------------------------------------+
| TIMEDIFF('2000:01:01 00:00:00', '2000:01:01 00:00:00.000001') |
+----------------------------------------------------------------+
| -00:00:00.000001                                               |
+----------------------------------------------------------------+

SELECT TIMEDIFF('2008-12-31 23:59:59.000001', '2008-12-30 01:01:01.000002');
+----------------------------------------------------------------------+
| TIMEDIFF('2008-12-31 23:59:59.000001', '2008-12-30 01:01:01.000002') |
+----------------------------------------------------------------------+
| 46:58:57.999999                                                      |
+----------------------------------------------------------------------+
```

1.2.3.49 TIMESTAMP FUNCTION

Syntax

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

Description

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.

Examples

```
SELECT TIMESTAMP('2003-12-31');
+-------------------------+
| TIMESTAMP('2003-12-31') |
+-------------------------+
| 2003-12-31 00:00:00     |
+-------------------------+

SELECT TIMESTAMP('2003-12-31 12:00:00','6:30:00');
+-----------------------------------------+
| TIMESTAMP('2003-12-31 12:00:00','6:30:00') |
+-----------------------------------------+
| 2003-12-31 18:30:00                     |
+-----------------------------------------+
```

1.2.3.50 TIMESTAMPADD

Syntax

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

Description

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, 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.

Before MariaDB 5.5, FRAC_SECOND was permitted as a synonym for MICROSECOND.

Examples

```
SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');
+-------------------------------------+
| TIMESTAMPADD(MINUTE,1,'2003-01-02') |
+-------------------------------------+
| 2003-01-02 00:01:00                 |
+-------------------------------------+

SELECT TIMESTAMPADD(WEEK,1,'2003-01-02');
+----------------------------------+
| TIMESTAMPADD(WEEK,1,'2003-01-02') |
+----------------------------------+
| 2003-01-09                       |
+----------------------------------+
```

1.2.3.51 TIMESTAMPDIFF

Syntax

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

Description

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, i.e MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, or YEAR.

`TIMESTAMPDIFF` can also be used to calculate age.

Examples

```
SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');
+----------------------------------------------+
| TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01') |
+----------------------------------------------+
|                                            3 |
+----------------------------------------------+

SELECT TIMESTAMPDIFF(YEAR,'2002-05-01','2001-01-01');
+---------------------------------------------+
| TIMESTAMPDIFF(YEAR,'2002-05-01','2001-01-01') |
+---------------------------------------------+
|                                          -1 |
+---------------------------------------------+

SELECT TIMESTAMPDIFF(MINUTE,'2003-02-01','2003-05-01 12:05:55');
+-----------------------------------------------------------+
| TIMESTAMPDIFF(MINUTE,'2003-02-01','2003-05-01 12:05:55') |
+-----------------------------------------------------------+
|                                                    128885 |
+-----------------------------------------------------------+
```

Calculating age:

```
SELECT CURDATE();
+------------+
| CURDATE()  |
+------------+
| 2019-05-27 |
+------------+

SELECT TIMESTAMPDIFF(YEAR, '1971-06-06', CURDATE()) AS age;
+------+
| age  |
+------+
|   47 |
+------+

SELECT TIMESTAMPDIFF(YEAR, '1971-05-06', CURDATE()) AS age;
+------+
| age  |
+------+
|   48 |
+------+
```

Age as of 2014-08-02:

```
SELECT name, date_of_birth, TIMESTAMPDIFF(YEAR,date_of_birth,'2014-08-02') AS age
  FROM student_details;
+---------+---------------+------+
| name    | date_of_birth | age  |
+---------+---------------+------+
| Chun    | 1993-12-31    |   20 |
| Esben   | 1946-01-01    |   68 |
| Kaolin  | 1996-07-16    |   18 |
| Tatiana | 1988-04-13    |   26 |
+---------+---------------+------+
```

1.2.3.52 TIME_FORMAT

Syntax

```
TIME_FORMAT(time,format)
```

Description

This is used like the DATE_FORMAT() function, but the format string may contain format specifiers only for hours, minutes, and seconds. Other specifiers produce a NULL value or 0.

Examples

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

1.2.3.53 TIME_TO_SEC

Syntax

```
TIME_TO_SEC(time)
```

Description

Returns the time argument, converted to seconds.

The value returned by `TIME_TO_SEC` is of type `DOUBLE`. Before MariaDB 5.3 (and MySQL 5.6), the type was `INT`. The returned value preserves microseconds of the argument. See also Microseconds in MariaDB.

Examples

```
SELECT TIME_TO_SEC('22:23:00');
+-------------------------+
| TIME_TO_SEC('22:23:00') |
+-------------------------+
|                   80580 |
+-------------------------+
```

```
SELECT TIME_TO_SEC('00:39:38');
+-------------------------+
| TIME_TO_SEC('00:39:38') |
+-------------------------+
|                    2378 |
+-------------------------+
```

```
SELECT TIME_TO_SEC('09:12:55.2355');
+-------------------------------+
| TIME_TO_SEC('09:12:55.2355') |
+-------------------------------+
|                   33175.2355 |
+-------------------------------+
1 row in set (0.000 sec)
```

1.2.3.54 TO_DAYS

Syntax

```
TO_DAYS(date)
```

Description

Given a date `date`, returns the number of days since the start of the current calendar (0000-00-00).

The function is not designed for use with dates before the advent of the Gregorian calendar in October 1582. Results will not be reliable since it doesn't account for the lost days when the calendar changed from the Julian calendar.

This is the converse of the FROM_DAYS() function.

Examples

```
SELECT TO_DAYS('2007-10-07');
+-----------------------+
| TO_DAYS('2007-10-07') |
+-----------------------+
|                733321 |
+-----------------------+

SELECT TO_DAYS('0000-01-01');
+-----------------------+
| TO_DAYS('0000-01-01') |
+-----------------------+
|                     1 |
+-----------------------+

SELECT TO_DAYS(950501);
+-----------------+
| TO_DAYS(950501) |
+-----------------+
|          728779 |
+-----------------+
```

1.2.3.55 TO_SECONDS

Syntax

```
TO_SECONDS(expr)
```

Description

Returns the number of seconds from year 0 till `expr`, or NULL if `expr` is not a valid date or datetime.

Examples

```
SELECT TO_SECONDS('2013-06-13');
+--------------------------+
| TO_SECONDS('2013-06-13') |
+--------------------------+
|              63538300800 |
+--------------------------+

SELECT TO_SECONDS('2013-06-13 21:45:13');
+-----------------------------------+
| TO_SECONDS('2013-06-13 21:45:13') |
+-----------------------------------+
|                       63538379113 |
+-----------------------------------+

SELECT TO_SECONDS(NOW());
+-------------------+
| TO_SECONDS(NOW()) |
+-------------------+
|       63543530875 |
+-------------------+

SELECT TO_SECONDS(20130513);
+----------------------+
| TO_SECONDS(20130513) |
+----------------------+
|          63535622400 |
+----------------------+
1 row in set (0.00 sec)

SELECT TO_SECONDS(130513);
+--------------------+
| TO_SECONDS(130513) |
+--------------------+
|        63535622400 |
+--------------------+
```

1.2.3.56 UNIX_TIMESTAMP

Contents

Syntax

```
UNIX_TIMESTAMP()
UNIX_TIMESTAMP(date)
```

Description

If called with no argument, returns a Unix timestamp (seconds since '1970-01-01 00:00:00' UTC) as an unsigned integer. 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. date may be a `DATE` string, a `DATETIME` string, a `TIMESTAMP`, or a number in the format YYMMDD or YYYYMMDD. The server interprets date as a value in the current time zone and converts it to an internal value in UTC. Clients can set their time zone as described in time zones.

The inverse function of `UNIX_TIMESTAMP()` is `FROM_UNIXTIME()`

`UNIX_TIMESTAMP()` supports microseconds.

Timestamps in MariaDB have a maximum value of 2147483647, equivalent to 2038-01-19 05:14:07. This is due to the underlying 32-bit limitation. Using the function on a date beyond this will result in NULL being returned. Use DATETIME as a storage type if you require dates beyond this.

Error Handling

Returns NULL for wrong arguments to `UNIX_TIMESTAMP()`. In MySQL and MariaDB before 5.3 wrong arguments to `UNIX_TIMESTAMP()` returned 0.

Compatibility

As you can see in the examples above, UNIX_TIMESTAMP(constant-date-string) returns a timestamp with 6 decimals while MariaDB 5.2 and before returns it without decimals. This can cause a problem if you are using UNIX_TIMESTAMP() as a partitioning function. You can fix this by using FLOOR(UNIX_TIMESTAMP(..)) or changing the date string to a date number, like 20080101000000.

Examples

```
SELECT UNIX_TIMESTAMP();
+------------------+
| UNIX_TIMESTAMP() |
+------------------+
|       1269711082 |
+------------------+

SELECT UNIX_TIMESTAMP('2007-11-30 10:30:19');
+---------------------------------------+
| UNIX_TIMESTAMP('2007-11-30 10:30:19') |
+---------------------------------------+
|                   1196436619.000000 |
+---------------------------------------+

SELECT UNIX_TIMESTAMP("2007-11-30 10:30:19.123456");
+-----------------------------------------------+
| unix_timestamp("2007-11-30 10:30:19.123456") |
+-----------------------------------------------+
|                           1196411419.123456 |
+-----------------------------------------------+

SELECT FROM_UNIXTIME(UNIX_TIMESTAMP('2007-11-30 10:30:19'));
+-------------------------------------------------------+
| FROM_UNIXTIME(UNIX_TIMESTAMP('2007-11-30 10:30:19')) |
+-------------------------------------------------------+
| 2007-11-30 10:30:19.000000                            |
+-------------------------------------------------------+

SELECT FROM_UNIXTIME(FLOOR(UNIX_TIMESTAMP('2007-11-30 10:30:19')));
+-------------------------------------------------------------+
| FROM_UNIXTIME(FLOOR(UNIX_TIMESTAMP('2007-11-30 10:30:19'))) |
+-------------------------------------------------------------+
| 2007-11-30 10:30:19                                         |
+-------------------------------------------------------------+
```

1.2.3.57 UTC_DATE

Syntax

```
UTC_DATE, UTC_DATE()
```

Description

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

Examples

```
SELECT UTC_DATE(), UTC_DATE() + 0;
+------------+----------------+
| UTC_DATE() | UTC_DATE() + 0 |
+------------+----------------+
| 2010-03-27 |       20100327 |
+------------+----------------+
```

1.2.3.58 UTC_TIME

Syntax

```
UTC_TIME
UTC_TIME([precision])
```

Description

Returns the current UTC time as a value in 'HH:MM:SS' or HHMMSS.uuuuuu format, depending on whether the function is used in a string or numeric context.

The optional *precision* determines the microsecond precision. See Microseconds in MariaDB.

Examples

```
SELECT UTC_TIME(), UTC_TIME() + 0;
+------------+----------------+
| UTC_TIME() | UTC_TIME() + 0 |
+------------+----------------+
| 17:32:34   |  173234.000000 |
+------------+----------------+
```

With precision:

```
SELECT UTC_TIME(5);
+----------------+
| UTC_TIME(5)    |
+----------------+
| 07:52:50.78369 |
+----------------+
```

1.2.3.59 UTC_TIMESTAMP

Syntax

```
UTC_TIMESTAMP
UTC_TIMESTAMP([precision])
```

Description

Returns the current UTC date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS.uuuuuu format, depending on whether the function is used in a string or numeric context.

The optional *precision* determines the microsecond precision. See Microseconds in MariaDB.

Examples

```
SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0;
+---------------------+----------------------+
| UTC_TIMESTAMP()     | UTC_TIMESTAMP() + 0  |
+---------------------+----------------------+
| 2010-03-27 17:33:16 | 20100327173316.000000 |
+---------------------+----------------------+
```

With precision:

```
SELECT UTC_TIMESTAMP(4);
+--------------------------+
| UTC_TIMESTAMP(4)         |
+--------------------------+
| 2018-07-10 07:51:09.1019 |
+--------------------------+
```

1.2.3.60 WEEK

Syntax

```
WEEK(date[,mode])
```

Description

This function returns the week number for `date`. The two-argument form of `WEEK()` allows 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.

Modes

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

With the mode value of 3, which means 'more than 3 days this year', weeks are numbered according to ISO 8601:1988.

Examples

```
SELECT WEEK('2008-02-20');
+--------------------+
| WEEK('2008-02-20') |
+--------------------+
|                  7 |
+--------------------+

SELECT WEEK('2008-02-20',0);
+----------------------+
| WEEK('2008-02-20',0) |
+----------------------+
|                    7 |
+----------------------+

SELECT WEEK('2008-02-20',1);
+----------------------+
| WEEK('2008-02-20',1) |
+----------------------+
|                    8 |
+----------------------+

SELECT WEEK('2008-12-31',0);
+----------------------+
| WEEK('2008-12-31',0) |
+----------------------+
|                   52 |
+----------------------+

SELECT WEEK('2008-12-31',1);
+----------------------+
| WEEK('2008-12-31',1) |
+----------------------+
|                   53 |
+----------------------+

 SELECT WEEK('2019-12-30',3);
+----------------------+
| WEEK('2019-12-30',3) |
+----------------------+
|                    1 |
+----------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT d, WEEK(d,0), WEEK(d,1) from t1;
+---------------------+-----------+-----------+
| d                   | WEEK(d,0) | WEEK(d,1) |
+---------------------+-----------+-----------+
| 2007-01-30 21:31:07 |         4 |         5 |
| 1983-10-15 06:42:51 |        41 |        41 |
| 2011-04-21 12:34:56 |        16 |        16 |
| 2011-10-30 06:31:41 |        44 |        43 |
| 2011-01-30 14:03:25 |         5 |         4 |
| 2004-10-07 11:19:34 |        40 |        41 |
+---------------------+-----------+-----------+
```

1.2.3.61 WEEKDAY

Syntax

```
WEEKDAY(date)
```

Description

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

This contrasts with `DAYOFWEEK()` which follows the ODBC standard (`1` = Sunday, `2` = Monday, ..., `7` = Saturday).

Examples

```
SELECT WEEKDAY('2008-02-03 22:23:00');
+--------------------------------+
| WEEKDAY('2008-02-03 22:23:00') |
+--------------------------------+
|                              6 |
+--------------------------------+

SELECT WEEKDAY('2007-11-06');
+-----------------------+
| WEEKDAY('2007-11-06') |
+-----------------------+
|                     1 |
+-----------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT d FROM t1 where WEEKDAY(d) = 6;
+---------------------+
| d                   |
+---------------------+
| 2011-10-30 06:31:41 |
| 2011-01-30 14:03:25 |
+---------------------+
```

1.2.3.62 WEEKOFYEAR

Syntax

```
WEEKOFYEAR(date)
```

Description

Returns the calendar week of the date as a number in the range from 1 to 53. `WEEKOFYEAR()` is a compatibility function that is equivalent to `WEEK(date,3)` .

Examples

```
SELECT WEEKOFYEAR('2008-02-20');
+--------------------------+
| WEEKOFYEAR('2008-02-20') |
+--------------------------+
|                        8 |
+--------------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
 select * from t1;
+---------------------+
| d                   |
+---------------------+
| 2007-01-30 21:31:07 |
| 1983-10-15 06:42:51 |
| 2011-04-21 12:34:56 |
| 2011-10-30 06:31:41 |
| 2011-01-30 14:03:25 |
| 2004-10-07 11:19:34 |
+---------------------+
```

```
SELECT d, WEEKOFYEAR(d), WEEK(d,3) from t1;
+---------------------+---------------+-----------+
| d                   | WEEKOFYEAR(d) | WEEK(d,3) |
+---------------------+---------------+-----------+
| 2007-01-30 21:31:07 |             5 |         5 |
| 1983-10-15 06:42:51 |            41 |        41 |
| 2011-04-21 12:34:56 |            16 |        16 |
| 2011-10-30 06:31:41 |            43 |        43 |
| 2011-01-30 14:03:25 |             4 |         4 |
| 2004-10-07 11:19:34 |            41 |        41 |
+---------------------+---------------+-----------+
```

1.2.3.63 YEAR

Syntax

```
YEAR(date)
```

Description

Returns the year for the given date, in the range 1000 to 9999, or 0 for the "zero" date.

Examples

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT * FROM t1;
+---------------------+
| d                   |
+---------------------+
| 2007-01-30 21:31:07 |
| 1983-10-15 06:42:51 |
| 2011-04-21 12:34:56 |
| 2011-10-30 06:31:41 |
| 2011-01-30 14:03:25 |
| 2004-10-07 11:19:34 |
+---------------------+

SELECT * FROM t1 WHERE YEAR(d) = 2011;
+---------------------+
| d                   |
+---------------------+
| 2011-04-21 12:34:56 |
| 2011-10-30 06:31:41 |
| 2011-01-30 14:03:25 |
+---------------------+
```

```
SELECT YEAR('1987-01-01');
+--------------------+
| YEAR('1987-01-01') |
+--------------------+
|               1987 |
+--------------------+
```

1.2.3.64 YEARWEEK

Syntax

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

Description

Returns year and week for a date. The mode argument works exactly like the mode argument to WEEK(). 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.

Examples

```
SELECT YEARWEEK('1987-01-01');
+------------------------+
| YEARWEEK('1987-01-01') |
+------------------------+
|                 198652 |
+------------------------+
```

```
CREATE TABLE t1 (d DATETIME);
INSERT INTO t1 VALUES
    ("2007-01-30 21:31:07"),
    ("1983-10-15 06:42:51"),
    ("2011-04-21 12:34:56"),
    ("2011-10-30 06:31:41"),
    ("2011-01-30 14:03:25"),
    ("2004-10-07 11:19:34");
```

```
SELECT * FROM t1;
+---------------------+
| d                   |
+---------------------+
| 2007-01-30 21:31:07 |
| 1983-10-15 06:42:51 |
| 2011-04-21 12:34:56 |
| 2011-10-30 06:31:41 |
| 2011-01-30 14:03:25 |
| 2004-10-07 11:19:34 |
+---------------------+
6 rows in set (0.02 sec)
```

```
SELECT YEARWEEK(d) FROM t1 WHERE YEAR(d) = 2011;
+-------------+
| YEARWEEK(d) |
+-------------+
|      201116 |
|      201144 |
|      201105 |
+-------------+
3 rows in set (0.03 sec)
```

1.2.4 Aggregate Functions

The following functions (also called aggregate functions) can be used with the GROUP BY clause:

Stored Aggregate Functions

Custom aggregate functions.

AVG

Returns the average value.

BIT_AND

Bitwise AND.

BIT_OR

Bitwise OR.

BIT_XOR

Bitwise XOR.

COUNT

Returns count of non-null values.

COUNT DISTINCT

Returns count of number of different non-NULL values.

GROUP_CONCAT

Returns string with concatenated values from a group.

JSON_ARRAYAGG

Returns a JSON array containing an element for each value in a given set of JSON or SQL values.

JSON_OBJECTAGG

Returns a JSON object containing key-value pairs.

MAX

Returns the maximum value.

MIN

Returns the minimum value.

STD
Population standard deviation.

STDDEV
Population standard deviation.

STDDEV_POP
Returns the population standard deviation.

STDDEV_SAMP
Standard deviation.

SUM
Sum total.

VARIANCE
Population standard variance.

VAR_POP
Population standard variance.

VAR_SAMP
Returns the sample variance.

1.2.4.1 Stored Aggregate Functions

MariaDB starting with 10.3.3
The ability to create stored aggregate functions was added in MariaDB 10.3.3.

Contents

Aggregate functions are functions that are computed over a sequence of rows and return one result for the sequence of rows.

Creating a custom aggregate function is done using the CREATE FUNCTION statement with two main differences:

- The addition of the AGGREGATE keyword, so `CREATE AGGREGATE FUNCTION`
- The `FETCH GROUP NEXT ROW` instruction inside the loop
- Oracle PL/SQL compatibility using SQL/PL is provided

Standard Syntax

```
CREATE AGGREGATE FUNCTION function_name (parameters) RETURNS return_type
BEGIN
      All types of declarations
      DECLARE CONTINUE HANDLER FOR NOT FOUND RETURN return_val;
      LOOP
           FETCH GROUP NEXT ROW; // fetches next row from table
           other instructions
      END LOOP;
END
```

Stored aggregate functions were a 2016 Google Summer of Code project by Varun Gupta.

Using SQL/PL

```
SET sql_mode=Oracle;
DELIMITER //

CREATE AGGREGATE FUNCTION function_name (parameters) RETURN return_type
   declarations
BEGIN
   LOOP
      FETCH GROUP NEXT ROW; -- fetches next row from table
      -- other instructions

   END LOOP;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN return_val;
END //

DELIMITER ;
```

Examples

First a simplified example:

```
CREATE TABLE marks(stud_id INT, grade_count INT);

INSERT INTO marks VALUES (1,6), (2,4), (3,7), (4,5), (5,8);

SELECT * FROM marks;
+---------+-------------+
| stud_id | grade_count |
+---------+-------------+
|       1 |           6 |
|       2 |           4 |
|       3 |           7 |
|       4 |           5 |
|       5 |           8 |
+---------+-------------+

DELIMITER //
CREATE AGGREGATE FUNCTION IF NOT EXISTS aggregate_count(x INT) RETURNS INT
BEGIN
 DECLARE count_students INT DEFAULT 0;
 DECLARE CONTINUE HANDLER FOR NOT FOUND
 RETURN count_students;
      LOOP
          FETCH GROUP NEXT ROW;
          IF x  THEN
            SET count_students = count_students+1;
          END IF;
      END LOOP;
END //
DELIMITER ;
```

A non-trivial example that cannot easily be rewritten using existing functions:

```
DELIMITER //
CREATE AGGREGATE FUNCTION medi_int(x INT) RETURNS DOUBLE
BEGIN
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    BEGIN
      DECLARE res DOUBLE;
      DECLARE cnt INT DEFAULT (SELECT COUNT(*) FROM tt);
      DECLARE lim INT DEFAULT (cnt-1) DIV 2;
      IF cnt % 2 = 0 THEN
        SET res = (SELECT AVG(a) FROM (SELECT a FROM tt ORDER BY a LIMIT lim,2) ttt);
      ELSE
        SET res = (SELECT a FROM tt ORDER BY a LIMIT lim,1);
      END IF;
      DROP TEMPORARY TABLE tt;
      RETURN res;
    END;
  CREATE TEMPORARY TABLE tt (a INT);
  LOOP
    FETCH GROUP NEXT ROW;
    INSERT INTO tt VALUES (x);
  END LOOP;
END //
DELIMITER ;
```

SQL/PL Example

This uses the same marks table as created above.

```
SET sql_mode=Oracle;
DELIMITER //

CREATE AGGREGATE FUNCTION aggregate_count(x INT) RETURN INT AS count_students INT DEFAULT 0;
BEGIN
   LOOP
      FETCH GROUP NEXT ROW;
      IF x  THEN
        SET count_students := count_students+1;
      END IF;
   END LOOP;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN count_students;
END aggregate_count //
DELIMITER ;

SELECT aggregate_count(stud_id) FROM marks;
```

1.2.4.2 AVG

Syntax

```
AVG([DISTINCT] expr)
```

Contents

Description

Returns the average value of expr. The DISTINCT option can be used to return the average of the distinct values of expr. NULL values are ignored. It is an aggregate function, and so can be used with the GROUP BY clause.

AVG() returns NULL if there were no matching rows.

AVG() can be used as a window function.

Examples

```
CREATE TABLE sales (sales_value INT);

INSERT INTO sales VALUES(10),(20),(20),(40);

SELECT AVG(sales_value) FROM sales;
+------------------+
| AVG(sales_value) |
+------------------+
|          22.5000 |
+------------------+

SELECT AVG(DISTINCT(sales_value)) FROM sales;
+----------------------------+
| AVG(DISTINCT(sales_value)) |
+----------------------------+
|                    23.3333 |
+----------------------------+
```

Commonly, AVG() is used with a GROUP BY clause:

```
CREATE TABLE student (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student VALUES
  ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
  ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
  ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
  ('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);

SELECT name, AVG(score) FROM student GROUP BY name;
+---------+------------+
| name    | AVG(score) |
+---------+------------+
| Chun    |    74.0000 |
| Esben   |    37.0000 |
| Kaolin  |    72.0000 |
| Tatiana |    85.0000 |
+---------+------------+
```

Be careful to avoid this common mistake, not grouping correctly and returning mismatched data:

```
SELECT name,test,AVG(score) FROM student;
+------+------+------------+
| name | test | MIN(score) |
+------+------+------------+
| Chun | SQL  |         31 |
+------+------+------------+
```

As a window function:

```
CREATE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);

SELECT name, test, score, AVG(score) OVER (PARTITION BY test)
    AS average_by_test FROM student_test;
+---------+--------+-------+-----------------+
| name    | test   | score | average_by_test |
+---------+--------+-------+-----------------+
| Chun    | SQL    |    75 |         65.2500 |
| Chun    | Tuning |    73 |         68.7500 |
| Esben   | SQL    |    43 |         65.2500 |
| Esben   | Tuning |    31 |         68.7500 |
| Kaolin  | SQL    |    56 |         65.2500 |
| Kaolin  | Tuning |    88 |         68.7500 |
| Tatiana | SQL    |    87 |         65.2500 |
| Tatiana | Tuning |    83 |         68.7500 |
+---------+--------+-------+-----------------+
```

1.2.4.3 BIT_AND

Syntax

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

Description

Returns the bitwise AND of all bits in *expr*. The calculation is performed with 64-bit (BIGINT) precision. It is an aggregate function, and so can be used with the GROUP BY clause.

If no rows match, `BIT_AND` will return a value with all bits set to 1. NULL values have no effect on the result unless all results are NULL, which is treated as no match.

`BIT_AND` can be used as a window function with the addition of the *over_clause*.

Examples

```
CREATE TABLE vals (x INT);

INSERT INTO vals VALUES(111),(110),(100);

SELECT BIT_AND(x), BIT_OR(x), BIT_XOR(x) FROM vals;
+------------+-----------+------------+
| BIT_AND(x) | BIT_OR(x) | BIT_XOR(x) |
+------------+-----------+------------+
|        100 |       111 |        101 |
+------------+-----------+------------+
```

As an aggregate function:

```
CREATE TABLE vals2 (category VARCHAR(1), x INT);

INSERT INTO vals2 VALUES
  ('a',111),('a',110),('a',100),
  ('b','000'),('b',001),('b',011);

SELECT category, BIT_AND(x), BIT_OR(x), BIT_XOR(x)
  FROM vals GROUP BY category;
+----------+------------+-----------+------------+
| category | BIT_AND(x) | BIT_OR(x) | BIT_XOR(x) |
+----------+------------+-----------+------------+
| a        |        100 |       111 |        101 |
| b        |          0 |        11 |         10 |
+----------+------------+-----------+------------+
```

No match:

```
SELECT BIT_AND(NULL);
+----------------------+
| BIT_AND(NULL)        |
+----------------------+
| 18446744073709551615 |
+----------------------+
```

1.2.4.4 BIT_OR

Syntax

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

Contents

Description

Returns the bitwise OR of all bits in `expr`. The calculation is performed with 64-bit (BIGINT) precision. It is an aggregate function, and so can be used with the GROUP BY clause.

If no rows match, `BIT_OR` will return a value with all bits set to `0`. NULL values have no effect on the result unless all results are NULL, which is treated as no match.

`BIT_OR` can be used as a window function with the addition of the *over_clause*.

Examples

```
CREATE TABLE vals (x INT);

INSERT INTO vals VALUES(111),(110),(100);

SELECT BIT_AND(x), BIT_OR(x), BIT_XOR(x) FROM vals;
+------------+-----------+------------+
| BIT_AND(x) | BIT_OR(x) | BIT_XOR(x) |
+------------+-----------+------------+
|        100 |       111 |        101 |
+------------+-----------+------------+
```

As an aggregate function:

```
CREATE TABLE vals2 (category VARCHAR(1), x INT);

INSERT INTO vals2 VALUES
  ('a',111),('a',110),('a',100),
  ('b','000'),('b',001),('b',011);

SELECT category, BIT_AND(x), BIT_OR(x), BIT_XOR(x)
  FROM vals GROUP BY category;
+----------+------------+-----------+------------+
| category | BIT_AND(x) | BIT_OR(x) | BIT_XOR(x) |
+----------+------------+-----------+------------+
| a        |        100 |       111 |        101 |
| b        |          0 |        11 |         10 |
+----------+------------+-----------+------------+
```

No match:

```
SELECT BIT_OR(NULL);
+--------------+
| BIT_OR(NULL) |
+--------------+
|            0 |
+--------------+
```

1.2.4.5 BIT_XOR

Syntax

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

Contents

Description

Returns the bitwise XOR of all bits in `expr`. The calculation is performed with 64-bit (BIGINT) precision. It is an aggregate function, and so can be used with the GROUP BY clause.

If no rows match, `BIT_XOR` will return a value with all bits set to `0`. NULL values have no effect on the result unless all results are NULL, which is treated as no match.

`BIT_XOR` can be used as a window function with the addition of the *over_clause*.

Examples

```
CREATE TABLE vals (x INT);

INSERT INTO vals VALUES(111),(110),(100);

SELECT BIT_AND(x), BIT_OR(x), BIT_XOR(x) FROM vals;
+------------+-----------+------------+
| BIT_AND(x) | BIT_OR(x) | BIT_XOR(x) |
+------------+-----------+------------+
|        100 |       111 |        101 |
+------------+-----------+------------+
```

As an aggregate function:

```
CREATE TABLE vals2 (category VARCHAR(1), x INT);

INSERT INTO vals2 VALUES
  ('a',111),('a',110),('a',100),
  ('b','000'),('b',001),('b',011);

SELECT category, BIT_AND(x), BIT_OR(x), BIT_XOR(x)
  FROM vals GROUP BY category;
+----------+------------+-----------+------------+
| category | BIT_AND(x) | BIT_OR(x) | BIT_XOR(x) |
+----------+------------+-----------+------------+
| a        |        100 |       111 |        101 |
| b        |          0 |        11 |         10 |
+----------+------------+-----------+------------+
```

No match:

```
SELECT BIT_XOR(NULL);
+---------------+
| BIT_XOR(NULL) |
+---------------+
|             0 |
+---------------+
```

1.2.4.6 COUNT

Syntax

```
COUNT(expr)
```

Contents

Description

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. It is an aggregate function, and so can be used with the GROUP BY clause.

COUNT(*) counts the total number of rows in a table.

COUNT() returns 0 if there were no matching rows.

COUNT() can be used as a window function.

Examples

```
CREATE TABLE student (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student VALUES
  ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
  ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
  ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
  ('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);

SELECT COUNT(*) FROM student;
+----------+
| COUNT(*) |
+----------+
|        8 |
+----------+
```

COUNT(DISTINCT) example:

```
SELECT COUNT(DISTINCT (name)) FROM student;
+------------------------+
| COUNT(DISTINCT (name)) |
+------------------------+
|                      4 |
+------------------------+
```

As a window function

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, COUNT(score) OVER (PARTITION BY name)
    AS tests_written FROM student_test;
+---------+--------+-------+---------------+
| name    | test   | score | tests_written |
+---------+--------+-------+---------------+
| Chun    | SQL    |    75 |             2 |
| Chun    | Tuning |    73 |             2 |
| Esben   | SQL    |    43 |             2 |
| Esben   | Tuning |    31 |             2 |
| Kaolin  | SQL    |    56 |             2 |
| Kaolin  | Tuning |    88 |             2 |
| Tatiana | SQL    |    87 |             1 |
+---------+--------+-------+---------------+
```

1.2.4.7 COUNT DISTINCT

Syntax

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

Contents

Description

Returns a count of the number of different non-NULL values.

COUNT(DISTINCT) returns 0 if there were no matching rows.

Although, from MariaDB 10.2.0, COUNT can be used as a window function, COUNT DISTINCT cannot be.

Examples

```
CREATE TABLE student (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student VALUES
  ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
  ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
  ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
  ('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);

SELECT COUNT(*) FROM student;
+----------+
| COUNT(*) |
+----------+
|        8 |
+----------+

SELECT COUNT(DISTINCT (name)) FROM student;
+------------------------+
| COUNT(DISTINCT (name)) |
+------------------------+
|                      4 |
+------------------------+
```

1.2.4.8 GROUP_CONCAT

Syntax

```
GROUP_CONCAT(expr)
```

Contents

Description

This function returns a string result with the concatenated non-NULL values from a group. If any expr in GROUP_CONCAT evaluates to NULL, that tuple is not present in the list returned by GROUP_CONCAT.

It returns NULL if all arguments are NULL, or there are no matching rows.

The maximum returned length in bytes is determined by the group_concat_max_len server system variable, which defaults to 1M.

If group_concat_max_len <= 512, the return type is VARBINARY or VARCHAR; otherwise, the return type is BLOB or TEXT. The choice between binary or non-binary types depends from the input.

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]
             [LIMIT {[offset,] row_count | row_count OFFSET offset}])
```

`DISTINCT` eliminates duplicate values from the output string.

ORDER BY determines the order of returned values.

`SEPARATOR` specifies a separator between the values. The default separator is a comma (`,`). It is possible to avoid using a separator by specifying an empty string.

LIMIT

The LIMIT clause can be used with `GROUP_CONCAT` . This was not possible prior to MariaDB 10.3.3.

Examples

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

Get a readable list of MariaDB users from the mysql.user table:

```
SELECT GROUP_CONCAT(DISTINCT User ORDER BY User SEPARATOR '\n')
   FROM mysql.user;
```

In the former example, `DISTINCT` is used because the same user may occur more than once. The new line (`\n`) used as a `SEPARATOR` makes the results easier to read.

Get a readable list of hosts from which each user can connect:

```
SELECT User, GROUP_CONCAT(Host ORDER BY Host SEPARATOR ', ')
   FROM mysql.user GROUP BY User ORDER BY User;
```

The former example shows the difference between the `GROUP_CONCAT` 's ORDER BY (which sorts the concatenated hosts), and the `SELECT` 's ORDER BY (which sorts the rows).

From MariaDB 10.3.3, LIMIT can be used with `GROUP_CONCAT` , so, for example, given the following table:

```
CREATE TABLE d (dd DATE, cc INT);

INSERT INTO d VALUES ('2017-01-01',1);
INSERT INTO d VALUES ('2017-01-02',2);
INSERT INTO d VALUES ('2017-01-04',3);
```

the following query:

```
SELECT SUBSTRING_INDEX(GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC),",",1) FROM d;
+-----------------------------------------------------------------------------+
| SUBSTRING_INDEX(GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC),",",1) |
+-----------------------------------------------------------------------------+
| 2017-01-04:3                                                                |
+-----------------------------------------------------------------------------+
```

can be more simply rewritten as:

```
SELECT GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC LIMIT 1) FROM d;
+-----------------------------------------------------------+
| GROUP_CONCAT(CONCAT_WS(":",dd,cc) ORDER BY cc DESC LIMIT 1) |
+-----------------------------------------------------------+
| 2017-01-04:3                                              |
+-----------------------------------------------------------+
```

NULLS:

```
CREATE OR REPLACE TABLE t1 (a int, b char);

INSERT INTO t1 VALUES (1, 'a'), (2, NULL);

SELECT GROUP_CONCAT(a, b) FROM t1;
+--------------------+
| GROUP_CONCAT(a, b) |
+--------------------+
| 1a                 |
+--------------------+
```

1.2.4.9 JSON_ARRAYAGG

MariaDB starting with 10.5.0
JSON_ARRAYAGG was added in MariaDB 10.5.0.

Syntax

```
JSON_ARRAYAGG(column_or_expression)
```

Description

`JSON_ARRAYAGG` returns a JSON array containing an element for each value in a given set of JSON or SQL values. It acts on a column or an expression that evaluates to a single value.

The maximum returned length in bytes is determined by the group_concat_max_len server system variable.

Returns NULL in the case of an error, or if the result contains no rows.

`JSON_ARRAYAGG` cannot currently be used as a window function.

The full syntax is as follows:

```
JSON_ARRAYAGG([DISTINCT] expr
             [ORDER BY {unsigned_integer | col_name | expr}
                 [ASC | DESC] [,col_name ...]]
             [LIMIT {[offset,] row_count | row_count OFFSET offset}])
```

Examples

```
CREATE TABLE t1 (a INT, b INT);

INSERT INTO t1 VALUES (1, 1),(2, 1), (1, 1),(2, 1), (3, 2),(2, 2),(2, 2),(2, 2);

SELECT JSON_ARRAYAGG(a), JSON_ARRAYAGG(b) FROM t1;
+-------------------+-------------------+
| JSON_ARRAYAGG(a)  | JSON_ARRAYAGG(b)  |
+-------------------+-------------------+
| [1,2,1,2,3,2,2,2] | [1,1,1,1,2,2,2,2] |
+-------------------+-------------------+

SELECT JSON_ARRAYAGG(a), JSON_ARRAYAGG(b) FROM t1 GROUP BY b;
+-------------------+-------------------+
| JSON_ARRAYAGG(a) | JSON_ARRAYAGG(b) |
+-------------------+-------------------+
| [1,2,1,2]         | [1,1,1,1]         |
| [3,2,2,2]         | [2,2,2,2]         |
+-------------------+-------------------+
```

1.2.4.10 JSON_OBJECTAGG

MariaDB starting with 10.5.0
JSON_OBJECTAGG was added in MariaDB 10.5.0.

Syntax

```
JSON_OBJECTAGG(key, value)
```

Description

`JSON_OBJECTAGG` returns a JSON object containing key-value pairs. It takes two expressions that evaluate to a single value, or two column names, as arguments, the first used as a key, and the second as a value.

The maximum returned length in bytes is determined by the group_concat_max_len server system variable.

Returns NULL in the case of an error, or if the result contains no rows.

`JSON_OBJECTAGG` cannot currently be used as a window function.

Examples

```
select * from t1;
+------+-------+
| a    | b     |
+------+-------+
|    1 | Hello |
|    1 | World |
|    2 | This  |
+------+-------+

SELECT JSON_OBJECTAGG(a, b) FROM t1;
+---------------------------------------+
| JSON_OBJECTAGG(a, b)                  |
+---------------------------------------+
| {"1":"Hello", "1":"World", "2":"This"} |
+---------------------------------------+
```

1.2.4.11 MAX

Syntax

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

Contents

Description

Returns the largest, or maximum, value of *expr*. `MAX()` can also take a string argument in which case it returns the maximum string value. 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`.

Note that SET and ENUM fields are currently compared by their string value rather than their relative position in the set, so MAX() may produce a different highest result than ORDER BY DESC.

It is an aggregate function, and so can be used with the GROUP BY clause.

MAX() can be used as a window function.

`MAX()` returns `NULL` if there were no matching rows.

Examples

```
CREATE TABLE student (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student VALUES
  ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
  ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
  ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
  ('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);

SELECT name, MAX(score) FROM student GROUP BY name;
+---------+------------+
| name    | MAX(score) |
+---------+------------+
| Chun    |         75 |
| Esben   |         43 |
| Kaolin  |         88 |
| Tatiana |         87 |
+---------+------------+
```

MAX string:

```
SELECT MAX(name) FROM student;
+-----------+
| MAX(name) |
+-----------+
| Tatiana   |
+-----------+
```

Be careful to avoid this common mistake, not grouping correctly and returning mismatched data:

```
SELECT name,test,MAX(SCORE) FROM student;
+------+------+------------+
| name | test | MAX(SCORE) |
+------+------+------------+
| Chun | SQL  |         88 |
+------+------+------------+
```

Difference between ORDER BY DESC and MAX():

```
CREATE TABLE student2(name CHAR(10),grade ENUM('b','c','a'));

INSERT INTO student2 VALUES('Chun','b'),('Esben','c'),('Kaolin','a');

SELECT MAX(grade) FROM student2;
+------------+
| MAX(grade) |
+------------+
| c          |
+------------+

SELECT grade FROM student2 ORDER BY grade DESC LIMIT 1;
+-------+
| grade |
+-------+
| a     |
+-------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);
INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, MAX(score)
  OVER (PARTITION BY name) AS highest_score FROM student_test;
+---------+--------+-------+---------------+
| name    | test   | score | highest_score |
+---------+--------+-------+---------------+
| Chun    | SQL    |    75 |            75 |
| Chun    | Tuning |    73 |            75 |
| Esben   | SQL    |    43 |            43 |
| Esben   | Tuning |    31 |            43 |
| Kaolin  | SQL    |    56 |            88 |
| Kaolin  | Tuning |    88 |            88 |
| Tatiana | SQL    |    87 |            87 |
+---------+--------+-------+---------------+
```

1.2.4.12 MIN

Syntax

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

Contents

Description

Returns the minimum value of *expr*. `MIN()` may take a string argument, in which case it returns the minimum string value. 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`.

Note that SET and ENUM fields are currently compared by their string value rather than their relative position in the set, so MIN() may produce a different lowest result than ORDER BY ASC.

It is an aggregate function, and so can be used with the GROUP BY clause.

`MIN()` can be used as a window function.

`MIN()` returns `NULL` if there were no matching rows.

Examples

```
CREATE TABLE student (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student VALUES
  ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
  ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
  ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
  ('Tatiana', 'SQL', 87), ('Tatiana', 'Tuning', 83);

SELECT name, MIN(score) FROM student GROUP BY name;
+---------+------------+
| name    | MIN(score) |
+---------+------------+
| Chun    |         73 |
| Esben   |         31 |
| Kaolin  |         56 |
| Tatiana |         83 |
+---------+------------+
```

MIN() with a string:

```
SELECT MIN(name) FROM student;
+-----------+
| MIN(name) |
+-----------+
| Chun      |
+-----------+
```

Be careful to avoid this common mistake, not grouping correctly and returning mismatched data:

```
SELECT name,test,MIN(score) FROM student;
+------+------+------------+
| name | test | MIN(score) |
+------+------+------------+
| Chun | SQL  |         31 |
+------+------+------------+
```

Difference between ORDER BY ASC and MIN():

```
CREATE TABLE student2(name CHAR(10),grade ENUM('b','c','a'));

INSERT INTO student2 VALUES('Chun','b'),('Esben','c'),('Kaolin','a');

SELECT MIN(grade) FROM student2;
+------------+
| MIN(grade) |
+------------+
| a          |
+------------+

SELECT grade FROM student2 ORDER BY grade ASC LIMIT 1;
+-------+
| grade |
+-------+
| b     |
+-------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);
INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);


SELECT name, test, score, MIN(score)
  OVER (PARTITION BY name) AS lowest_score FROM student_test;
+---------+--------+-------+--------------+
| name    | test   | score | lowest_score |
+---------+--------+-------+--------------+
| Chun    | SQL    |    75 |           73 |
| Chun    | Tuning |    73 |           73 |
| Esben   | SQL    |    43 |           31 |
| Esben   | Tuning |    31 |           31 |
| Kaolin  | SQL    |    56 |           56 |
| Kaolin  | Tuning |    88 |           56 |
| Tatiana | SQL    |    87 |           87 |
+---------+--------+-------+--------------+
```

1.2.4.13 STD

Syntax

```
STD(expr)
```

Description

Returns the population standard deviation of *expr*. This is an extension to standard SQL. The standard SQL function `STDDEV_POP()` can be used instead.

It is an aggregate function, and so can be used with the GROUP BY clause.

STD() can be used as a window function.

This function returns `NULL` if there were no matching rows.

Examples

As an aggregate function:

```
CREATE OR REPLACE TABLE stats (category VARCHAR(2), x INT);

INSERT INTO stats VALUES
  ('a',1),('a',2),('a',3),
  ('b',11),('b',12),('b',20),('b',30),('b',60);

SELECT category, STDDEV_POP(x), STDDEV_SAMP(x), VAR_POP(x)
  FROM stats GROUP BY category;
+----------+--------------+---------------+------------+
| category | STDDEV_POP(x) | STDDEV_SAMP(x) | VAR_POP(x) |
+----------+--------------+---------------+------------+
| a        |       0.8165 |        1.0000 |     0.6667 |
| b        |      18.0400 |       20.1693 |   325.4400 |
+----------+--------------+---------------+------------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, STDDEV_POP(score)
  OVER (PARTITION BY test) AS stddev_results FROM student_test;
+---------+--------+-------+----------------+
| name    | test   | score | stddev_results |
+---------+--------+-------+----------------+
| Chun    | SQL    |    75 |        16.9466 |
| Chun    | Tuning |    73 |        24.1247 |
| Esben   | SQL    |    43 |        16.9466 |
| Esben   | Tuning |    31 |        24.1247 |
| Kaolin  | SQL    |    56 |        16.9466 |
| Kaolin  | Tuning |    88 |        24.1247 |
| Tatiana | SQL    |    87 |        16.9466 |
+---------+--------+-------+----------------+
```

1.2.4.14 STDDEV

Syntax

```
STDDEV(expr)
```

Contents

Description

Returns the population standard deviation of *expr*. This function is provided for compatibility with Oracle. The standard SQL function `STDDEV_POP()` can be used instead.

It is an aggregate function, and so can be used with the GROUP BY clause.

STDDEV() can be used as a window function.

This function returns `NULL` if there were no matching rows.

Examples

As an aggregate function:

```
CREATE OR REPLACE TABLE stats (category VARCHAR(2), x INT);

INSERT INTO stats VALUES
  ('a',1),('a',2),('a',3),
  ('b',11),('b',12),('b',20),('b',30),('b',60);

SELECT category, STDDEV_POP(x), STDDEV_SAMP(x), VAR_POP(x)
  FROM stats GROUP BY category;
+----------+--------------+---------------+------------+
| category | STDDEV_POP(x) | STDDEV_SAMP(x) | VAR_POP(x) |
+----------+--------------+---------------+------------+
| a        |       0.8165 |        1.0000 |     0.6667 |
| b        |      18.0400 |       20.1693 |   325.4400 |
+----------+--------------+---------------+------------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, STDDEV_POP(score)
  OVER (PARTITION BY test) AS stddev_results FROM student_test;
+---------+--------+-------+----------------+
| name    | test   | score | stddev_results |
+---------+--------+-------+----------------+
| Chun    | SQL    |    75 |        16.9466 |
| Chun    | Tuning |    73 |        24.1247 |
| Esben   | SQL    |    43 |        16.9466 |
| Esben   | Tuning |    31 |        24.1247 |
| Kaolin  | SQL    |    56 |        16.9466 |
| Kaolin  | Tuning |    88 |        24.1247 |
| Tatiana | SQL    |    87 |        16.9466 |
+---------+--------+-------+----------------+
```

1.2.4.15 STDDEV_POP

Syntax

```
STDDEV_POP(expr)
```

Contents

Description

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.

It is an aggregate function, and so can be used with the GROUP BY clause.

STDDEV_POP() can be used as a window function.

STDDEV_POP() returns `NULL` if there were no matching rows.

Examples

As an aggregate function:

```
CREATE OR REPLACE TABLE stats (category VARCHAR(2), x INT);

INSERT INTO stats VALUES
  ('a',1),('a',2),('a',3),
  ('b',11),('b',12),('b',20),('b',30),('b',60);

SELECT category, STDDEV_POP(x), STDDEV_SAMP(x), VAR_POP(x)
  FROM stats GROUP BY category;
+----------+--------------+---------------+------------+
| category | STDDEV_POP(x) | STDDEV_SAMP(x) | VAR_POP(x) |
+----------+--------------+---------------+------------+
| a        |       0.8165 |        1.0000 |     0.6667 |
| b        |      18.0400 |       20.1693 |   325.4400 |
+----------+--------------+---------------+------------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, STDDEV_POP(score)
  OVER (PARTITION BY test) AS stddev_results FROM student_test;
+---------+--------+-------+----------------+
| name    | test   | score | stddev_results |
+---------+--------+-------+----------------+
| Chun    | SQL    |    75 |        16.9466 |
| Chun    | Tuning |    73 |        24.1247 |
| Esben   | SQL    |    43 |        16.9466 |
| Esben   | Tuning |    31 |        24.1247 |
| Kaolin  | SQL    |    56 |        16.9466 |
| Kaolin  | Tuning |    88 |        24.1247 |
| Tatiana | SQL    |    87 |        16.9466 |
+---------+--------+-------+----------------+
```

1.2.4.16 STDDEV_SAMP

Syntax

```
STDDEV_SAMP(expr)
```

Description

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

It is an aggregate function, and so can be used with the GROUP BY clause.

STDDEV_SAMP() can be used as a window function.

STDDEV_SAMP() returns `NULL` if there were no matching rows.

1.2.4.17 SUM

Syntax

```
SUM([DISTINCT] expr)
```

Contents

Description

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`.

SUM() can be used as a window function, although not with the DISTINCT specifier.

Examples

```
CREATE TABLE sales (sales_value INT);
INSERT INTO sales VALUES(10),(20),(20),(40);

SELECT SUM(sales_value) FROM sales;
+------------------+
| SUM(sales_value) |
+------------------+
|               90 |
+------------------+

SELECT SUM(DISTINCT(sales_value)) FROM sales;
+---------------------------+
| SUM(DISTINCT(sales_value)) |
+---------------------------+
|                        70 |
+---------------------------+
```

Commonly, SUM is used with a GROUP BY clause:

```
CREATE TABLE sales (name CHAR(10), month CHAR(10), units INT);

INSERT INTO sales VALUES
  ('Chun', 'Jan', 75), ('Chun', 'Feb', 73),
  ('Esben', 'Jan', 43), ('Esben', 'Feb', 31),
  ('Kaolin', 'Jan', 56), ('Kaolin', 'Feb', 88),
  ('Tatiana', 'Jan', 87), ('Tatiana', 'Feb', 83);

SELECT name, SUM(units) FROM sales GROUP BY name;
+---------+------------+
| name    | SUM(units) |
+---------+------------+
| Chun    |        148 |
| Esben   |         74 |
| Kaolin  |        144 |
| Tatiana |        170 |
+---------+------------+
```

The GROUP BY clause is required when using an aggregate function along with regular column data, otherwise the result will be a mismatch, as in the following common type of mistake:

```
SELECT name,SUM(units) FROM sales
;+------+------------+
| name | SUM(units) |
+------+------------+
| Chun |        536 |
+------+------------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);
INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, SUM(score) OVER (PARTITION BY name) AS total_score FROM student_test;
+---------+--------+-------+-------------+
| name    | test   | score | total_score |
+---------+--------+-------+-------------+
| Chun    | SQL    |    75 |         148 |
| Chun    | Tuning |    73 |         148 |
| Esben   | SQL    |    43 |          74 |
| Esben   | Tuning |    31 |          74 |
| Kaolin  | SQL    |    56 |         144 |
| Kaolin  | Tuning |    88 |         144 |
| Tatiana | SQL    |    87 |          87 |
+---------+--------+-------+-------------+
```

1.2.4.18 VARIANCE

Syntax

```
VARIANCE(expr)
```

Contents

Description

Returns the population standard variance of `expr`. This is an extension to standard SQL. The standard SQL function VAR_POP() can be used instead.

Variance is calculated by

- working out the mean for the set
- for each number, subtracting the mean and squaring the result
- calculate the average of the resulting differences

It is an aggregate function, and so can be used with the GROUP BY clause.

VARIANCE() can be used as a window function.

VARIANCE() returns `NULL` if there were no matching rows.

Examples

```
CREATE TABLE v(i tinyint);

INSERT INTO v VALUES(101),(99);

SELECT VARIANCE(i) FROM v;
+-------------+
| VARIANCE(i) |
+-------------+
|      1.0000 |
+-------------+

INSERT INTO v VALUES(120),(80);

SELECT VARIANCE(i) FROM v;
+-------------+
| VARIANCE(i) |
+-------------+
|    200.5000 |
+-------------+
```

As an aggregate function:

```
CREATE OR REPLACE TABLE stats (category VARCHAR(2), x INT);

INSERT INTO stats VALUES
  ('a',1),('a',2),('a',3),
  ('b',11),('b',12),('b',20),('b',30),('b',60);

SELECT category, STDDEV_POP(x), STDDEV_SAMP(x), VAR_POP(x)
  FROM stats GROUP BY category;
+----------+--------------+---------------+------------+
| category | STDDEV_POP(x) | STDDEV_SAMP(x) | VAR_POP(x) |
+----------+--------------+---------------+------------+
| a        |       0.8165 |        1.0000 |     0.6667 |
| b        |      18.0400 |       20.1693 |   325.4400 |
+----------+--------------+---------------+------------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, VAR_POP(score)
  OVER (PARTITION BY test) AS variance_results FROM student_test;
+---------+--------+-------+------------------+
| name    | test   | score | variance_results |
+---------+--------+-------+------------------+
| Chun    | SQL    |    75 |         287.1875 |
| Chun    | Tuning |    73 |         582.0000 |
| Esben   | SQL    |    43 |         287.1875 |
| Esben   | Tuning |    31 |         582.0000 |
| Kaolin  | SQL    |    56 |         287.1875 |
| Kaolin  | Tuning |    88 |         582.0000 |
| Tatiana | SQL    |    87 |         287.1875 |
+---------+--------+-------+------------------+
```

1.2.4.19 VAR_POP

Syntax

```
VAR_POP(expr)
```

Contents

Description

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.

Variance is calculated by

- working out the mean for the set
- for each number, subtracting the mean and squaring the result
- calculate the average of the resulting differences

It is an aggregate function, and so can be used with the GROUP BY clause.

VAR_POP() can be used as a window function.

VAR_POP() returns `NULL` if there were no matching rows.

Examples

```
CREATE TABLE v(i tinyint);

INSERT INTO v VALUES(101),(99);

SELECT VAR_POP(i) FROM v;
+------------+
| VAR_POP(i) |
+------------+
|     1.0000 |
+------------+

INSERT INTO v VALUES(120),(80);

SELECT VAR_POP(i) FROM v;
+------------+
| VAR_POP(i) |
+------------+
|   200.5000 |
+------------+
```

As an aggregate function:

```
CREATE OR REPLACE TABLE stats (category VARCHAR(2), x INT);

INSERT INTO stats VALUES
  ('a',1),('a',2),('a',3),
  ('b',11),('b',12),('b',20),('b',30),('b',60);

SELECT category, STDDEV_POP(x), STDDEV_SAMP(x), VAR_POP(x)
  FROM stats GROUP BY category;
+----------+---------------+---------------+------------+
| category | STDDEV_POP(x) | STDDEV_SAMP(x) | VAR_POP(x) |
+----------+---------------+---------------+------------+
| a        |        0.8165 |        1.0000 |     0.6667 |
| b        |       18.0400 |       20.1693 |   325.4400 |
+----------+---------------+---------------+------------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, VAR_POP(score)
  OVER (PARTITION BY test) AS variance_results FROM student_test;
+---------+--------+-------+------------------+
| name    | test   | score | variance_results |
+---------+--------+-------+------------------+
| Chun    | SQL    |    75 |         287.1875 |
| Esben   | SQL    |    43 |         287.1875 |
| Kaolin  | SQL    |    56 |         287.1875 |
| Tatiana | SQL    |    87 |         287.1875 |
| Chun    | Tuning |    73 |         582.0000 |
| Esben   | Tuning |    31 |         582.0000 |
| Kaolin  | Tuning |    88 |         582.0000 |
+---------+--------+-------+------------------+
```

1.2.4.20 VAR_SAMP

Syntax

```
VAR_SAMP(expr)
```

Contents

Description

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

It is an aggregate function, and so can be used with the GROUP BY clause.

VAR_SAMP() can be used as a window function.

VAR_SAMP() returns `NULL` if there were no matching rows.

Examples

As an aggregate function:

```
CREATE OR REPLACE TABLE stats (category VARCHAR(2), x INT);

INSERT INTO stats VALUES
  ('a',1),('a',2),('a',3),
  ('b',11),('b',12),('b',20),('b',30),('b',60);

SELECT category, STDDEV_POP(x), STDDEV_SAMP(x), VAR_POP(x)
  FROM stats GROUP BY category;
+----------+---------------+----------------+------------+
| category | STDDEV_POP(x) | STDDEV_SAMP(x) | VAR_POP(x) |
+----------+---------------+----------------+------------+
| a        |        0.8165 |         1.0000 |     0.6667 |
| b        |       18.0400 |        20.1693 |   325.4400 |
+----------+---------------+----------------+------------+
```

As a window function:

```
CREATE OR REPLACE TABLE student_test (name CHAR(10), test CHAR(10), score TINYINT);

INSERT INTO student_test VALUES
    ('Chun', 'SQL', 75), ('Chun', 'Tuning', 73),
    ('Esben', 'SQL', 43), ('Esben', 'Tuning', 31),
    ('Kaolin', 'SQL', 56), ('Kaolin', 'Tuning', 88),
    ('Tatiana', 'SQL', 87);

SELECT name, test, score, VAR_SAMP(score)
  OVER (PARTITION BY test) AS variance_results FROM student_test;
+---------+--------+-------+------------------+
| name    | test   | score | variance_results |
+---------+--------+-------+------------------+
| Chun    | SQL    |    75 |         382.9167 |
| Chun    | Tuning |    73 |         873.0000 |
| Esben   | SQL    |    43 |         382.9167 |
| Esben   | Tuning |    31 |         873.0000 |
| Kaolin  | SQL    |    56 |         382.9167 |
| Kaolin  | Tuning |    88 |         873.0000 |
| Tatiana | SQL    |    87 |         382.9167 |
+---------+--------+-------+------------------+
```

1.2.5 Numeric Functions

Functions dealing with numerals, including ABS, CEIL, DIV, EXP, PI, SIN, etc.

Addition Operator (+)

Addition.

Subtraction Operator (-)

Subtraction and unary minus.

Division Operator (/)

Division.

Multiplication Operator (*)

Multiplication.

Modulo Operator (%)

Modulo operator. Returns the remainder of N divided by M.

DIV

Integer division.

ABS

Returns an absolute value.

ACOS

Returns an arc cosine.

ASIN

Returns the arc sine.

ATAN

Returns the arc tangent.

ATAN2

Returns the arc tangent of two variables.

CEIL

Synonym for CEILING().

CEILING

Returns the smallest integer not less than X.

CONV

Converts numbers between different number bases.

COS

Returns the cosine.

COT

Returns the cotangent.

CRC32

Computes a cyclic redundancy check (CRC) value.

CRC32C

Computes a cyclic redundancy check (CRC) value using the Castagnoli polynomial.

DEGREES

Converts from radians to degrees.

EXP

e raised to the power of the argument.

FLOOR

Largest integer value not greater than the argument.

GREATEST

Returns the largest argument.

LEAST

Returns the smallest argument.

LN
Returns natural logarithm.

LOG
Returns the natural logarithm.

LOG10
Returns the base-10 logarithm.

LOG2
Returns the base-2 logarithm.

MOD
Modulo operation. Remainder of N divided by M.

OCT
Returns octal value.

PI
Returns the value of π (pi).

POW
Returns X raised to the power of Y.

POWER
Synonym for POW().

RADIANS
Converts from degrees to radians.

RAND
Random floating-point value.

ROUND
Rounds a number.

SIGN
Returns 1, 0 or -1.

SIN
Returns the sine.

SQRT
Square root.

TAN
Returns the tangent.

TRUNCATE
The TRUNCATE function truncates a number to a specified number of decimal places.

There are 1 related questions.

1.1.5.1.1 Addition Operator (+)

1.1.5.1.7 Subtraction Operator (-)

1.1.5.1.3 Division Operator (/)

1.1.5.1.6 Multiplication Operator (*)

1.1.5.1.5 Modulo Operator (%)

1.2.5.6 DIV

Syntax

```
DIV
```

Description

Integer division. Similar to FLOOR(), but is safe with BIGINT values. Incorrect results may occur for non-integer operands that exceed BIGINT range.

If the `ERROR_ON_DIVISION_BY_ZERO` SQL_MODE is used, a division by zero produces an error. Otherwise, it returns NULL.

The remainder of a division can be obtained using the MOD operator.

Examples

```
SELECT 300 DIV 7;
+-----------+
| 300 DIV 7 |
+-----------+
|        42 |
+-----------+

SELECT 300 DIV 0;
+-----------+
| 300 DIV 0 |
+-----------+
|      NULL |
+-----------+
```

1.2.5.7 ABS

Syntax

```
ABS(X)
```

Contents

Description

Returns the absolute (non-negative) value of `X`. If `X` is not a number, it is converted to a numeric type.

Examples

```
SELECT ABS(42);
+---------+
| ABS(42) |
+---------+
|      42 |
+---------+

SELECT ABS(-42);
+----------+
| ABS(-42) |
+----------+
|       42 |
+----------+

SELECT ABS(DATE '1994-01-01');
+------------------------+
| ABS(DATE '1994-01-01') |
+------------------------+
|               19940101 |
+------------------------+
```

1.2.5.8 ACOS

Syntax

```
ACOS(X)
```

Contents

Description

Returns the arc cosine of `X`, that is, the value whose cosine is `X`. Returns `NULL` if `X` is not in the range `-1` to `1`.

Examples

```
SELECT ACOS(1);
+---------+
| ACOS(1) |
+---------+
|       0 |
+---------+

SELECT ACOS(1.0001);
+--------------+
| ACOS(1.0001) |
+--------------+
|         NULL |
+--------------+

SELECT ACOS(0);
+-----------------+
| ACOS(0)         |
+-----------------+
| 1.5707963267949 |
+-----------------+

SELECT ACOS(0.234);
+-------------------+
| ACOS(0.234)       |
+-------------------+
| 1.33460644244679 |
+-------------------+
```

1.2.5.9 ASIN

Syntax

```
ASIN(X)
```

Contents

Description

Returns the arc sine of X, that is, the value whose sine is X. Returns NULL if X is not in the range -1 to 1.

Examples

```
SELECT ASIN(0.2);
+--------------------+
| ASIN(0.2)          |
+--------------------+
| 0.2013579207903308 |
+--------------------+

SELECT ASIN('foo');
+-------------+
| ASIN('foo') |
+-------------+
|           0 |
+-------------+

SHOW WARNINGS;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'foo' |
+---------+------+----------------------------------------+
```

1.2.5.10 ATAN

Syntax

```
ATAN(X)
```

Contents

Description

Returns the arc tangent of X, that is, the value whose tangent is X.

Examples

```
SELECT ATAN(2);
+--------------------+
| ATAN(2)            |
+--------------------+
| 1.1071487177940904 |
+--------------------+

SELECT ATAN(-2);
+---------------------+
| ATAN(-2)            |
+---------------------+
| -1.1071487177940904 |
+---------------------+
```

1.2.5.11 ATAN2

Syntax

```
ATAN(Y,X), ATAN2(Y,X)
```

Description

Returns the arc tangent of the two variables X and Y. It is similar to calculating the arc tangent of Y / X, except that the signs of both arguments are used to determine the quadrant of the result.

Examples

```
SELECT ATAN(-2,2);
+---------------------+
| ATAN(-2,2)          |
+---------------------+
| -0.7853981633974483 |
+---------------------+

SELECT ATAN2(PI(),0);
+--------------------+
| ATAN2(PI(),0)      |
+--------------------+
| 1.5707963267948966 |
+--------------------+
```

1.2.5.12 CEIL

Syntax

```
CEIL(X)
```

Description

CEIL() is a synonym for CEILING().

1.2.5.13 CEILING

Syntax

```
CEILING(X)
```

Description

Returns the smallest integer value not less than X.

Examples

```
SELECT CEILING(1.23);
+---------------+
| CEILING(1.23) |
+---------------+
|             2 |
+---------------+

SELECT CEILING(-1.23);
+----------------+
| CEILING(-1.23) |
+----------------+
|             -1 |
+----------------+
```

1.2.5.14 CONV

Syntax

```
CONV(N,from_base,to_base)
```

Description

Converts numbers between different number bases. Returns a string representation of the number `N`, converted from base `from_base` to base `to_base`.

Returns `NULL` if any argument is `NULL`, or if the second or third argument are not in the allowed range.

The argument `N` is interpreted as an integer, but may be specified as an integer or a string. The minimum base is 2 and the maximum base is 36 (prior to MariaDB 11.4.0) or 62 (from MariaDB 11.4.0). If `to_base` is a negative number, `N` is regarded as a signed number. Otherwise, `N` is treated as unsigned. `CONV()` works with 64-bit precision.

Some shortcuts for this function are also available: BIN(), OCT(), HEX(), UNHEX(). Also, MariaDB allows binary literal values and hexadecimal literal values.

Examples

```
SELECT CONV('a',16,2);
+----------------+
| CONV('a',16,2) |
+----------------+
| 1010           |
+----------------+

SELECT CONV('6E',18,8);
+-----------------+
| CONV('6E',18,8) |
+-----------------+
| 172             |
+-----------------+

SELECT CONV(-17,10,-18);
+------------------+
| CONV(-17,10,-18) |
+------------------+
| -H               |
+------------------+

SELECT CONV(12+'10'+'10'+0xa,10,10);
+-------------------------------+
| CONV(12+'10'+'10'+0xa,10,10) |
+-------------------------------+
| 42                            |
+-------------------------------+
```

1.2.5.15 COS

Syntax

```
COS(X)
```

Description

Returns the cosine of X, where X is given in radians.

Examples

```
SELECT COS(PI());
+-----------+
| COS(PI()) |
+-----------+
|        -1 |
+-----------+
```

1.2.5.16 COT

Syntax

```
COT(X)
```

Description

Returns the cotangent of X.

Examples

```
SELECT COT(42);
+--------------------+
| COT(42)            |
+--------------------+
| 0.4364167060752729 |
+--------------------+

SELECT COT(12);
+---------------------+
| COT(12)             |
+---------------------+
| -1.5726734063976893 |
+---------------------+

SELECT COT(0);
ERROR 1690 (22003): DOUBLE value is out of range in 'cot(0)'
```

1.2.5.17 CRC32

Syntax

<= MariaDB 10.7

```
CRC32(expr)
```

From MariaDB 10.8

```
CRC32([par,]expr)
```

Description

Computes a cyclic redundancy check (CRC) value and returns a 32-bit unsigned value. The result is NULL if the argument is NULL. The argument is expected to be a string and (if possible) is treated as one if it is not.

Uses the ISO 3309 polynomial that used by zlib and many others. MariaDB 10.8 introduced the CRC32C() function, which uses the alternate Castagnoli polynomia.

MariaDB starting with 10.8

Often, CRC is computed in pieces. To facilitate this, MariaDB 10.8.0 introduced an optional parameter: CRC32('MariaDB')=CRC32(CRC32('Maria'),'DB').

Examples

```
SELECT CRC32('MariaDB');
+------------------+
| CRC32('MariaDB') |
+------------------+
|       4227209140 |
+------------------+

SELECT CRC32('mariadb');
+------------------+
| CRC32('mariadb') |
+------------------+
|       2594253378 |
+------------------+
```

From MariaDB 10.8.0

```
SELECT CRC32(CRC32('Maria'),'DB');
+----------------------------+
| CRC32(CRC32('Maria'),'DB') |
+----------------------------+
|                 4227209140 |
+----------------------------+
```

1.2.5.18 CRC32C

MariaDB starting with 10.8

Introduced in MariaDB 10.8.0 to compute a cyclic redundancy check (CRC) value using the Castagnoli polynomial.

Syntax

```
CRC32C([par,]expr)
```

Description

MariaDB has always included a native unary function CRC32() that computes the CRC-32 of a string using the ISO 3309 polynomial that used by zlib and many others.

InnoDB and MyRocks use a different polynomial, which was implemented in SSE4.2 instructions that were introduced in the Intel Nehalem microarchitecture. This is commonly called CRC-32C (Castagnoli).

The CRC32C function uses the Castagnoli polynomial.

This allows `SELECT…INTO DUMPFILE` to be used for the creation of files with valid checksums, such as a logically empty InnoDB redo log file `ib_logfile0` corresponding to a particular log sequence number.

The optional parameter allows the checksum to be computed in pieces:
CRC32C('MariaDB')=CRC32C(CRC32C('Maria'),'DB').

Examples

```
SELECT CRC32C('MariaDB');
+--------------------+
| CRC32C('MariaDB') |
+--------------------+
|          809606978 |
+--------------------+

SELECT CRC32C(CRC32C('Maria'),'DB');
+-----------------------------+
| CRC32C(CRC32C('Maria'),'DB') |
+-----------------------------+
|                   809606978 |
+-----------------------------+
```

1.2.5.19 DEGREES

Syntax

```
DEGREES(X)
```

Description

Returns the argument *X*, converted from radians to degrees.

This is the converse of the RADIANS() function.

Examples

```
SELECT DEGREES(PI());
+---------------+
| DEGREES(PI()) |
+---------------+
|           180 |
+---------------+

SELECT DEGREES(PI() / 2);
+-------------------+
| DEGREES(PI() / 2) |
+-------------------+
|                90 |
+-------------------+

SELECT DEGREES(45);
+-----------------+
| DEGREES(45)     |
+-----------------+
| 2578.3100780887 |
+-----------------+
```

1.2.5.20 EXP

Syntax

```
EXP(X)
```

Description

Returns the value of e (the base of natural logarithms) raised to the power of X. The inverse of this function is LOG() (using a single argument only) or LN().

If `X` is `NULL`, this function returns `NULL`.

Examples

```
SELECT EXP(2);
+------------------+
| EXP(2)           |
+------------------+
| 7.38905609893065 |
+------------------+

SELECT EXP(-2);
+--------------------+
| EXP(-2)            |
+--------------------+
| 0.1353352832366127 |
+--------------------+

SELECT EXP(0);
+--------+
| EXP(0) |
+--------+
|      1 |
+--------+

SELECT EXP(NULL);
+-----------+
| EXP(NULL) |
+-----------+
|      NULL |
+-----------+
```

1.2.5.21 FLOOR

Syntax

```
FLOOR(X)
```

Description

Returns the largest integer value not greater than X.

Examples

```
SELECT FLOOR(1.23);
+-------------+
| FLOOR(1.23) |
+-------------+
|           1 |
+-------------+

SELECT FLOOR(-1.23);
+--------------+
| FLOOR(-1.23) |
+--------------+
|           -2 |
+--------------+
```

1.1.5.4.10 GREATEST

1.1.5.4.18 LEAST

1.2.5.24 LN

Syntax

```
LN(X)
```

Description

Returns the natural logarithm of X; that is, the base-e logarithm of X. If X is less than or equal to 0, or `NULL`, then NULL is returned.

The inverse of this function is `EXP()`.

Examples

```
SELECT LN(2);
+-------------------+
| LN(2)             |
+-------------------+
| 0.693147180559945 |
+-------------------+

SELECT LN(-2);
+--------+
| LN(-2) |
+--------+
|   NULL |
+--------+
```

1.2.5.25 LOG

Syntax

```
LOG(X), LOG(B,X)
```

Description

If called with one parameter, this function returns the natural logarithm of X. If X is less than or equal to 0, then NULL is returned.

If called with two parameters, it returns the logarithm of X to the base B. If B is <= 1 or X <= 0, the function returns NULL.

If any argument is `NULL`, the function returns `NULL`.

The inverse of this function (when called with a single argument) is the EXP() function.

Examples

LOG(X):

```
SELECT LOG(2);
+-------------------+
| LOG(2)            |
+-------------------+
| 0.693147180559945 |
+-------------------+

SELECT LOG(-2);
+---------+
| LOG(-2) |
+---------+
|    NULL |
+---------+
```

LOG(B,X)

```
SELECT LOG(2,16);
+-----------+
| LOG(2,16) |
+-----------+
|         4 |
+-----------+

SELECT LOG(3,27);
+-----------+
| LOG(3,27) |
+-----------+
|         3 |
+-----------+

SELECT LOG(3,1);
+----------+
| LOG(3,1) |
+----------+
|        0 |
+----------+

SELECT LOG(3,0);
+----------+
| LOG(3,0) |
+----------+
|     NULL |
+----------+
```

1.2.5.26 LOG10

Syntax

```
LOG10(X)
```

Description

Returns the base-10 logarithm of X.

Examples

```
SELECT LOG10(2);
+-------------------+
| LOG10(2)          |
+-------------------+
| 0.3010299995663981 |
+-------------------+

SELECT LOG10(100);
+------------+
| LOG10(100) |
+------------+
|          2 |
+------------+

SELECT LOG10(-100);
+-------------+
| LOG10(-100) |
+-------------+
|        NULL |
+-------------+
```

1.2.5.27 LOG2

Syntax

```
LOG2(X)
```

Description

Returns the base-2 logarithm of X.

Examples

```
SELECT LOG2(4398046511104);
+---------------------+
| LOG2(4398046511104) |
+---------------------+
|                  42 |
+---------------------+

SELECT LOG2(65536);
+-------------+
| LOG2(65536) |
+-------------+
|          16 |
+-------------+

SELECT LOG2(-100);
+------------+
| LOG2(-100) |
+------------+
|       NULL |
+------------+
```

1.2.5.28 MOD

Syntax

```
MOD(N,M), N % M, N MOD M
```

Description

Modulo operation. Returns the remainder of N divided by M. See also Modulo Operator.

If the `ERROR_ON_DIVISION_BY_ZERO` SQL_MODE is used, any number modulus zero produces an error. Otherwise, it returns NULL.

The integer part of a division can be obtained using DIV.

Examples

```
SELECT 1042 % 50;
+-----------+
| 1042 % 50 |
+-----------+
|        42 |
+-----------+

SELECT MOD(234, 10);
+--------------+
| MOD(234, 10) |
+--------------+
|            4 |
+--------------+

SELECT 253 % 7;
+---------+
| 253 % 7 |
+---------+
|       1 |
+---------+

SELECT MOD(29,9);
+-----------+
| MOD(29,9) |
+-----------+
|         2 |
+-----------+

SELECT 29 MOD 9;
+----------+
| 29 MOD 9 |
+----------+
|        2 |
+----------+
```

1.2.5.29 OCT

Syntax

```
OCT(N)
```

Contents

Description

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.

Examples

```
SELECT OCT(34);
+---------+
| OCT(34) |
+---------+
| 42      |
+---------+

SELECT OCT(12);
+---------+
| OCT(12) |
+---------+
| 14      |
+---------+
```

1.2.5.30 PI

Syntax

```
PI()
```

Description

Returns the value of π (pi). The default number of decimal places displayed is six, but MariaDB uses the full double-precision value internally.

Examples

```
SELECT PI();
+----------+
| PI()     |
+----------+
| 3.141593 |
+----------+

SELECT PI()+0.00000000000000000000;
+---------------------------------+
| PI()+0.00000000000000000000 |
+---------------------------------+
|      3.141592653589793115980 |
+---------------------------------+
```

1.2.5.31 POW

Syntax

```
POW(X,Y)
```

Description

Returns the value of X raised to the power of Y.

POWER() is a synonym.

Examples

```
SELECT POW(2,3);
+----------+
| POW(2,3) |
+----------+
|        8 |
+----------+

SELECT POW(2,-2);
+-----------+
| POW(2,-2) |
+-----------+
|      0.25 |
+-----------+
```

1.2.5.32 POWER

Syntax

```
POWER(X,Y)
```

Description

This is a synonym for POW(), which returns the value of X raised to the power of Y.

1.2.5.33 RADIANS

Syntax

```
RADIANS(X)
```

Description

Returns the argument X, converted from degrees to radians. Note that π radians equals 180 degrees.

This is the converse of the DEGREES() function.

Examples

```
SELECT RADIANS(45);
+-------------------+
| RADIANS(45)       |
+-------------------+
| 0.785398163397448 |
+-------------------+

SELECT RADIANS(90);
+------------------+
| RADIANS(90)      |
+------------------+
| 1.5707963267949 |
+------------------+

SELECT RADIANS(PI());
+--------------------+
| RADIANS(PI())      |
+--------------------+
| 0.0548311355616075 |
+--------------------+

SELECT RADIANS(180);
+------------------+
| RADIANS(180)     |
+------------------+
| 3.14159265358979 |
+------------------+
```

1.2.5.34 RAND

Contents

Syntax

```
RAND(), RAND(N)
```

Description

Returns a random `DOUBLE` precision floating point value v in the range 0 <= v < 1.0. If a constant integer argument N is specified, it is used as the seed value, which produces a repeatable sequence of column values. In the example below, note that the sequences of values produced by RAND(3) is the same both places where it occurs.

In a WHERE clause, RAND() is evaluated each time the WHERE is executed.

Statements using the RAND() function are not safe for statement-based replication.

Practical uses

The expression to get a random integer from a given range is the following:

```
FLOOR(min_value + RAND() * (max_value - min_value +1))
```

RAND() is often used to read random rows from a table, as follows:

```
SELECT * FROM my_table ORDER BY RAND() LIMIT 10;
```

Note, however, that this technique should never be used on a large table as it will be extremely slow. MariaDB will read all rows in the table, generate a random value for each of them, order them, and finally will apply the LIMIT clause.

Examples

```
CREATE TABLE t (i INT);

INSERT INTO t VALUES(1),(2),(3);

SELECT i, RAND() FROM t;
+------+------------------+
| i    | RAND()           |
+------+------------------+
|    1 | 0.255651095188829 |
|    2 | 0.833920199269355 |
|    3 |  0.40264774151393 |
+------+------------------+

SELECT i, RAND(3) FROM t;
+------+------------------+
| i    | RAND(3)          |
+------+------------------+
|    1 |  0.90576975597606 |
|    2 | 0.373079058130345 |
|    3 | 0.148086053457191 |
+------+------------------+

SELECT i, RAND() FROM t;
+------+------------------+
| i    | RAND()           |
+------+------------------+
|    1 | 0.511478140495232 |
|    2 | 0.349447508668012 |
|    3 | 0.212803152588013 |
+------+------------------+
```

Using the same seed, the same sequence will be returned:

```
SELECT i, RAND(3) FROM t;
+------+------------------+
| i    | RAND(3)          |
+------+------------------+
|    1 |  0.90576975597606 |
|    2 | 0.373079058130345 |
|    3 | 0.148086053457191 |
+------+------------------+
```

Generating a random number from 5 to 15:

```
SELECT FLOOR(5 + (RAND() * 11));
```

1.2.5.35 ROUND

Syntax

```
ROUND(X), ROUND(X,D)
```

Description

Rounds the argument `X` to `D` decimal places. `D` defaults to `0` if not specified. `D` can be negative to cause `D` digits left of the decimal point of the value `X` to become zero.

The rounding algorithm depends on the data type of `X`:

- for floating point types (FLOAT, DOUBLE) the C libraries rounding function is used, so the behavior *may* differ between operating systems
- for fixed point types (DECIMAL, DEC/NUMBER/FIXED) the "round half up" rule is used, meaning that e.g. a value ending in exactly .5 is always rounded up.

Examples

```
SELECT ROUND(-1.23);
+--------------+
| ROUND(-1.23) |
+--------------+
|           -1 |
+--------------+

SELECT ROUND(-1.58);
+--------------+
| ROUND(-1.58) |
+--------------+
|           -2 |
+--------------+

SELECT ROUND(1.58);
+-------------+
| ROUND(1.58) |
+-------------+
|           2 |
+-------------+

SELECT ROUND(1.298, 1);
+-----------------+
| ROUND(1.298, 1) |
+-----------------+
|             1.3 |
+-----------------+

SELECT ROUND(1.298, 0);
+-----------------+
| ROUND(1.298, 0) |
+-----------------+
|               1 |
+-----------------+

SELECT ROUND(23.298, -1);
+-------------------+
| ROUND(23.298, -1) |
+-------------------+
|                20 |
+-------------------+
```

1.2.5.36 SIGN

Syntax

```
SIGN(X)
```

Description

Returns the sign of the argument as -1, 0, or 1, depending on whether X is negative, zero, or positive.

Examples

```
SELECT SIGN(-32);
+-----------+
| SIGN(-32) |
+-----------+
|        -1 |
+-----------+

SELECT SIGN(0);
+---------+
| SIGN(0) |
+---------+
|       0 |
+---------+

SELECT SIGN(234);
+-----------+
| SIGN(234) |
+-----------+
|         1 |
+-----------+
```

1.2.5.37 SIN

Syntax

```
SIN(X)
```

Description

Returns the sine of X, where X is given in radians.

Examples

```
SELECT SIN(1.5707963267948966);
+--------------------------+
| SIN(1.5707963267948966) |
+--------------------------+
|                        1 |
+--------------------------+

SELECT SIN(PI());
+----------------------+
| SIN(PI())            |
+----------------------+
| 1.22460635382238e-16 |
+----------------------+

SELECT ROUND(SIN(PI()));
+------------------+
| ROUND(SIN(PI())) |
+------------------+
|                0 |
+------------------+
```

1.2.5.38 SQRT

Syntax

```
SQRT(X)
```

Description

Returns the square root of X. If X is negative, NULL is returned.

Examples

```
SELECT SQRT(4);
+---------+
| SQRT(4) |
+---------+
|       2 |
+---------+

SELECT SQRT(20);
+------------------+
| SQRT(20)         |
+------------------+
| 4.47213595499958 |
+------------------+

SELECT SQRT(-16);
+-----------+
| SQRT(-16) |
+-----------+
|      NULL |
+-----------+

SELECT SQRT(1764);
+------------+
| SQRT(1764) |
+------------+
|         42 |
+------------+
```

1.2.5.39 TAN

Syntax

```
TAN(X)
```

Description

Returns the tangent of X, where X is given in radians.

Examples

```
SELECT TAN(0.7853981633974483);
+--------------------------+
| TAN(0.7853981633974483) |
+--------------------------+
|       0.9999999999999999 |
+--------------------------+

SELECT TAN(PI());
+-----------------------+
| TAN(PI())             |
+-----------------------+
| -1.22460635382238e-16 |
+-----------------------+

SELECT TAN(PI()+1);
+-----------------+
| TAN(PI()+1)     |
+-----------------+
| 1.5574077246549 |
+-----------------+

SELECT TAN(RADIANS(PI()));
+--------------------+
| TAN(RADIANS(PI())) |
+--------------------+
| 0.0548861508080033 |
+--------------------+
```

1.2.5.40 TRUNCATE

This page documents the TRUNCATE function. See TRUNCATE TABLE for the DDL statement.

Syntax

```
TRUNCATE(X,D)
```

Contents

Description

Returns the number X, truncated to D decimal places. If D is 0, the result has no decimal point or fractional part. D can be negative to cause D digits left of the decimal point of the value X to become zero.

Examples

```
SELECT TRUNCATE(1.223,1);
+-------------------+
| TRUNCATE(1.223,1) |
+-------------------+
|               1.2 |
+-------------------+

SELECT TRUNCATE(1.999,1);
+-------------------+
| TRUNCATE(1.999,1) |
+-------------------+
|               1.9 |
+-------------------+

SELECT TRUNCATE(1.999,0);
+-------------------+
| TRUNCATE(1.999,0) |
+-------------------+
|                 1 |
+-------------------+

SELECT TRUNCATE(-1.999,1);
+--------------------+
| TRUNCATE(-1.999,1) |
+--------------------+
|               -1.9 |
+--------------------+

SELECT TRUNCATE(122,-2);
+------------------+
| TRUNCATE(122,-2) |
+------------------+
|              100 |
+------------------+

SELECT TRUNCATE(10.28*100,0);
+------------------------+
| TRUNCATE(10.28*100,0) |
+------------------------+
|                   1028 |
+------------------------+
```

1.2.6 Control Flow Functions

Built-In functions for assessing data to determine what results to return.

CASE OPERATOR

Returns the result where value=compare_value or for the first condition that is true.

DECODE

Decrypts a string encoded with ENCODE(), or, in Oracle mode, matches expressions.

DECODE_ORACLE

Synonym for the Oracle mode version of DECODE().

IF Function

If expr1 is TRUE, returns expr2; otherwise it returns expr3.

IFNULL

Check whether an expression is NULL.

NULLIF

Returns NULL if expr1 = expr2.

NVL

Synonym for IFNULL.

NVL2

Returns a value based on whether a specified expression is NULL or not.

There are 1 related questions.

1.2.6.1 CASE OPERATOR

Syntax

```
CASE value WHEN [compare_value] THEN result [WHEN [compare_value] THEN
result ...] [ELSE result] END

CASE WHEN [condition] THEN result [WHEN [condition] THEN result ...]
[ELSE result] END
```

Contents

Description

The first version returns the result where value=compare_value. The second version returns the result for the first condition that is true. If there was no matching result value, the result after ELSE is returned, or NULL if there is no ELSE part.

There is also a CASE statement, which differs from the CASE operator described here.

Examples

```
SELECT CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END;
+-------------------------------------------------------------+
| CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END |
+-------------------------------------------------------------+
| one                                                         |
+-------------------------------------------------------------+

SELECT CASE WHEN 1>0 THEN 'true' ELSE 'false' END;
+----------------------------------------+
| CASE WHEN 1>0 THEN 'true' ELSE 'false' END |
+----------------------------------------+
| true                                   |
+----------------------------------------+


SELECT CASE BINARY 'B' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END;
+-----------------------------------------------------+
| CASE BINARY 'B' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END |
+-----------------------------------------------------+
|                                                NULL |
+-----------------------------------------------------+
```

1.2.6.2 DECODE

Syntax

```
DECODE(crypt_str,pass_str)
```

In Oracle mode from MariaDB 10.3.2:

```
DECODE(expr, search_expr, result_expr [, search_expr2, result_expr2 ...] [default_expr])
```

In all modes from MariaDB 10.3.2:

```
DECODE_ORACLE(expr, search_expr, result_expr [, search_expr2, result_expr2 ...] [default_expr])
```

Description

In the default mode, `DECODE` decrypts the encrypted string *crypt_str* using *pass_str* as the password. *crypt_str* should be a string returned from ENCODE(). The resulting string will be the original string only if *pass_str* is the same.

In Oracle mode from MariaDB 10.3.2, `DECODE` compares *expr* to the search expressions, in order. If it finds a match, the corresponding result expression is returned. If no matches are found, the default expression is returned, or NULL if no default is provided.

NULLs are treated as equivalent.

`DECODE_ORACLE` is a synonym for the Oracle-mode version of the function, and is available in all modes.

Examples

From MariaDB 10.3.2:

```
SELECT DECODE_ORACLE(2+1,3*1,'found1',3*2,'found2','default');
+-----------------------------------------------------+
| DECODE_ORACLE(2+1,3*1,'found1',3*2,'found2','default') |
+-----------------------------------------------------+
| found1                                              |
+-----------------------------------------------------+

SELECT DECODE_ORACLE(2+4,3*1,'found1',3*2,'found2','default');
+-----------------------------------------------------+
| DECODE_ORACLE(2+4,3*1,'found1',3*2,'found2','default') |
+-----------------------------------------------------+
| found2                                              |
+-----------------------------------------------------+

SELECT DECODE_ORACLE(2+2,3*1,'found1',3*2,'found2','default');
+-----------------------------------------------------+
| DECODE_ORACLE(2+2,3*1,'found1',3*2,'found2','default') |
+-----------------------------------------------------+
| default                                             |
+-----------------------------------------------------+
```

Nulls are treated as equivalent:

```
SELECT DECODE_ORACLE(NULL,NULL,'Nulls are equivalent','Nulls are not equivalent');
+-------------------------------------------------------------------------+
| DECODE_ORACLE(NULL,NULL,'Nulls are equivalent','Nulls are not equivalent') |
+-------------------------------------------------------------------------+
| Nulls are equivalent                                                    |
+-------------------------------------------------------------------------+
```

1.2.6.3 DECODE_ORACLE

MariaDB starting with 10.3.2

`DECODE_ORACLE` is a synonym for the Oracle mode version of the DECODE function, and is available in all modes.

1.2.6.4 IF Function

Syntax

```
IF(expr1,expr2,expr3)
```

Contents

Description

If `expr1` is `TRUE` (`expr1 <> 0` and `expr1 <> NULL`) then `IF()` returns `expr2` ; otherwise it returns `expr3` . `IF()` returns a numeric or string value, depending on the context in which it is used.

Note: There is also an IF statement which differs from the `IF()` function described here.

Examples

```
SELECT IF(1>2,2,3);
+-------------+
| IF(1>2,2,3) |
+-------------+
|           3 |
+-------------+
```

```
SELECT IF(1<2,'yes','no');
+--------------------+
| IF(1<2,'yes','no') |
+--------------------+
| yes                |
+--------------------+
```

```
SELECT IF(STRCMP('test','test1'),'no','yes');
+---------------------------------------+
| IF(STRCMP('test','test1'),'no','yes') |
+---------------------------------------+
| no                                    |
+---------------------------------------+
```

1.2.6.5 IFNULL

Syntax

```
IFNULL(expr1,expr2)
NVL(expr1,expr2)
```

Contents

Description

If *`expr1`* is not NULL, IFNULL() returns *`expr1`* ; otherwise it returns *`expr2`* . IFNULL() returns a numeric or string value, depending on the context in which it is used.

From MariaDB 10.3, NVL() is an alias for IFNULL().

Examples

```
SELECT IFNULL(1,0);
+-------------+
| IFNULL(1,0) |
+-------------+
|           1 |
+-------------+

SELECT IFNULL(NULL,10);
+-----------------+
| IFNULL(NULL,10) |
+-----------------+
|              10 |
+-----------------+

SELECT IFNULL(1/0,10);
+----------------+
| IFNULL(1/0,10) |
+----------------+
|        10.0000 |
+----------------+

SELECT IFNULL(1/0,'yes');
+-------------------+
| IFNULL(1/0,'yes') |
+-------------------+
| yes               |
+-------------------+
```

1.2.6.6 NULLIF

Syntax

```
NULLIF(expr1,expr2)
```

Contents

Description

Returns NULL if expr1 = expr2 is true, otherwise returns expr1. This is the same as CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END.

Examples

```
SELECT NULLIF(1,1);
+-------------+
| NULLIF(1,1) |
+-------------+
|        NULL |
+-------------+

SELECT NULLIF(1,2);
+-------------+
| NULLIF(1,2) |
+-------------+
|           1 |
+-------------+
```

1.2.6.7 NVL

MariaDB starting with 10.3

From MariaDB 10.3, NVL is a synonym for IFNULL.

1.2.6.8 NVL2

MariaDB starting with 10.3

The NLV2 function was introduced in MariaDB 10.3.0.

Syntax

```
NVL2(expr1,expr2,expr3)
```

Contents

Description

The `NVL2` function returns a value based on whether a specified expression is NULL or not. If *expr1* is not NULL, then NVL2 returns *expr2*. If *expr1* is NULL, then NVL2 returns *expr3*.

Examples

```
SELECT NVL2(NULL,1,2);
+----------------+
| NVL2(NULL,1,2) |
+----------------+
|              2 |
+----------------+

SELECT NVL2('x',1,2);
+---------------+
| NVL2('x',1,2) |
+---------------+
|             1 |
+---------------+
```

1.2.7 Pseudo Columns

MariaDB has pseudo columns that can be used for different purposes.

_rowid

_rowid is an alias for the primary key column

1.2.7.1 _rowid

Syntax

_rowid

Description

The `_rowid` pseudo column is mapped to the primary key in the related table. This can be used as a replacement of the `rowid` pseudo column in other databases. Another usage is to simplify sql queries as one doesn't have to know the name of the primary key.

Examples

```
create table t1 (a int primary key, b varchar(80));
insert into t1 values (1,"one"),(2,"two");
select * from t1 where _rowid=1;
```

```
+---+------+
| a | b    |
+---+------+
| 1 | one  |
+---+------+
```

```
update t1 set b="three" where _rowid=2;
select * from t1 where _rowid>=1 and _rowid<=10;
```

```
+---+-------+
| a | b     |
+---+-------+
| 1 | one   |
| 2 | three |
+---+-------+
```

1.2.8 Secondary Functions

These are commonly used functions, but they are not primary functions.

Bit Functions and Operators

Operators for comparison and setting of values, and related functions.

Encryption, Hashing and Compression Functions

Functions used for encryption, hashing and compression.

Information Functions

Functions which return information on the server, the user, or a given query.

Miscellaneous Functions

Functions for very singular and specific needs.

1.2.8.1 Bit Functions and Operators

Operators for comparison and setting of values, and related functions. They all return a result of the `BIGINT UNSIGNED` type

Operator Precedence

Precedence of SQL operators

&

Bitwise AND

<<

Left shift

>>

Shift right

BIT_COUNT

Returns the number of set bits

^

Bitwise XOR