

A Programmer's Reference Guide to **mysql** and Perl DBI

June 30, 1998

1.0 How to express Literals in **mysql**

- 1.1 Strings
- 1.2 Numbers
- 1.3 NULL
- 1.4 Database, table, index and column names

2.0 Column Types

- 2.1 More about data types
 - 2.1.1 Database size info.
 - 2.1.2 The numeric types
 - 2.1.3 TIMESTAMP type
 - 2.1.4 TEXT and BLOB types
 - 2.1.5 ENUM type
 - 2.1.6 SET type
- 2.2 Choosing the right type for a column.
- 2.3 Column indexes
- 2.4 Multiple column indexes
- 2.5 Type mapping to ease moving table definitions between different databases engines

3.0 Commands and Syntax

- 3.1 CREATE DATABASE syntax.
- 3.2 DROP DATABASE syntax.
- 3.3 CREATE TABLE syntax.
- 3.4 ALTER TABLE syntax
- 3.5 DROP TABLE syntax.
- 3.6 DELETE syntax.
- 3.7 SELECT syntax
- 3.8 JOIN syntax
- 3.9 INSERT syntax
- 3.10 REPLACE syntax
- 3.11 LOAD DATA INFILE syntax
- 3.12 UPDATE syntax
- 3.13 SHOW syntax. Get information about tables, columns...
- 3.14 EXPLAIN syntax. Get information about a SELECT.
- 3.15 DESCRIBE syntax. Get information about columns.
- 3.16 LOCK TABLES syntax
- 3.17 SET OPTION syntax.
- 3.18 CREATE INDEX syntax (Compatibility function).
- 3.19 DROP INDEX syntax (Compatibility function).
- 3.20 Comment syntax
- 3.21 CREATE FUNCTION syntax
- 3.22 Is MySQL picky about reserved words?

4.0 Functions for use in SELECT and WHERE clauses

- 4.1 Grouping functions.
- 4.2 Normal arithmetic operations.
- 4.3 Bit functions.
- 4.4 Logical operations.
- 4.5 Comparison operators.
- 4.6 String comparison functions.
- 4.7 Control flow functions.
- 4.8 Mathematical functions.
- 4.9 String functions.
- 4.10 Date and time functions.
- 4.11 Miscellaneous functions.
- 4.12 Functions for GROUP BY clause.

5.0 The DBI interface

- 5.1 The DBI interface
- 5.2 More DBI/DBD information

1.0

How to express literals in mysql

1.1 Strings

A string may have ' or " around it.

\ is a escape character. The following escape characters are recognized:

\0 An ascii 0 character.
\n A newline character.
\t A tab character.
\r A return character.
\b A backspace character.
\' A ' character.
\" A " character.
\ \ A \ character.
\% A % character. This is used in wildcard strings to search for %.
_ A _ character. This is used in wildcard strings to search for _.

A ' inside a string started with ' may be written as \".

A " inside a string started with " may be written as \"\".

Some example selects that shows how it works.

```
mysql> select 'hello', "'hello'", ""'hello'"" , "h'e'l'l'o", "hel''lo";
1 rows in set (0.00 sec)
```

```
+-----+-----+-----+-----+-----+
| hello | 'hello' | "'hello'" | 'h'e'l'l'o' | hel''lo |
+-----+-----+-----+-----+-----+
| hello | 'hello' | "'hello'" | 'h'e'l'l'o' | hel''lo |
+-----+-----+-----+-----+-----+
```

```
mysql> select 'hello', "hello", ""'hello'"" , 'ello', 'e'l'lo', '\ 'hello';
1 rows in set (0.00 sec)
```

```
+-----+-----+-----+-----+-----+
| hello | hello | "'hello'" | 'ello | e'l'lo | 'hello |
+-----+-----+-----+-----+-----+
| hello | hello | "'hello'" | 'ello | e'l'lo | 'hello |
+-----+-----+-----+-----+-----+
```

```
mysql> select "This\nIs\nFour\nlines";
1 rows in set (0.00 sec)
```

```
+-----+
| This
Is
Four
lines |
+-----+
| This
Is
Four
lines |
+-----+
```

If you want to insert binary data into a blob the following characters must be represented by escape sequences:

\0	Ascii 0. Should be replaced with "\0" (A backslash and a 0 digit).
\	Ascii 92, backslash
'	Ascii 39, Single quote
"	Ascii 33, Double quote

One doesn't have to escape " inside ' and ' inside '.

If you write C code you can use the C API function `mysql_escape_string(char *to,char *from,uint length)` to escape characters for the INSERT clause. (Note that 'to' must be at least 2 times bigger than from). In Perl you can use the quote function.

You should run the escape function on every possible string that may have a one of the above special characters!

1.2 Numbers

Integers are just a sequence of digits. Floats use . as a decimal separator.

Examples of valid numbers are: 1221, 294.42, -32032.6809e+10.

1.3 NULL

When using the text file export formats, NULL may be represented by \N.

1.4 Database, table, index and column names

Database, table, index and column names all follow the same rules in MySQL.

A name may use alphanumeric characters from the default character set. This is by default ISO-8859-1 Latin1 but may be changed when compiling MySQL.

Since MySQL needs to be able to decide if something is a name or a number the following special cases occurs:

- A name can not consist of only numbers.
- A name may start with a number. This is a difference from many other systems!
- It is not recommended to use names like 1e. This is because expressions like 1e+1 may be interpreted like the expression 1e + 1 or the number 1e+1.
- Punctuation characters like . and @ are not allowed in names since they will be used to extend MySQL.

In MySQL you can refer to a column with some of the following syntaxes:

```
column  
table.column  
database.table.column  
table@database.column
```

If you are using 'column' or 'table.column' you will get an error if the name is not unique among the used tables!

2.0 Column types.

The following column types are supported:

M means Max display size.

L means the actual length in a single row.

M means the maximum length.

D means the number of decimals.

Name	Description	Size
TINYINT[(M)] [UNSIGNED] [ZEROFILL]	A very small integer. Signed range -128 - 127. Unsigned range 0 - 255.	1
SMALLINT[(M)]. [UNSIGNED] [ZEROFILL]	A small integer. Signed range -32768 - 32767. Unsigned range 0 -	
65535.	2	
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]	A medium integer. Signed range -8388608-8388607. Unsigned range 0 - 16777215.	3
INT[(M)] [UNSIGNED] [ZEROFILL]	A normal integer. Signed range -2147483648 - 2147483647. Unsigned range 0 - 4294967295.	4
BIGINT[(M)] [UNSIGNED] [ZEROFILL]	A large integer. Signed range -9223372036854775808 - 9223372036854775807. Unsigned Range 0 - 18446744073709551615. Because all arithmetic is done with signed BIGINT or DOUBLE, one shouldn't use unsigned big integers bigger than 9223372036854775807 (63 bits) with anything else than bit functions!	8
FLOAT(Precision)	A small floating point number. Precision can be 4 or 8. FLOAT(4) is a single precision number and FLOAT(8) is a double precision number (se the DOUBLE entry). This syntax is for ODBC compatibility. Range - 3.402823466E+38F - -1.175494351E-38, 0, - 1.175494351E-38 - 3.402823466E+38F.	4
FLOAT[(M,D)]	A small floating point number. Cannot be unsigned. Range -3.402823466E+38F - -1.175494351E-38, 0, - 1.175494351E-38 - 3.402823466E+38F.	4

DOUBLE PRECISION[(M,D)]	A normal floating point number. Cannot be unsigned. Range -1.7976931348623157E+308 - - 2.2250738585072014E-308, 0, 2.2250738585072014E-308 - 1.7976931348623157E+308.	8
REAL[(M,D)]	Same as DOUBLE	8
DECIMAL [(M,D)]	An unpacked floating point number. Cannot be unsigned. Currently the same range maximum range as a double. Behaves as a CHAR column	M+D
NUMERIC [(M,D)]	Same as DECIMAL	M+D
TIMESTAMP [(M)]	An automatic timestamp. If you have many TIMESTAMP columns only the first one is automatic	4
DATE	A type to store date information. Uses the "YYYY-MM-DD" syntax, but may be updated with a number or a string. Understands at least the following syntaxes: 'YY-MM-DD', 'YYYY-MM-DD', 'YYMMDD' and full timestamps (YYYYMMDDHHMMDD). Range 0000-00-00 to 9999-12-31.	3
TIME	A type to store time information. Uses the "HH:MM:SS" syntax, but may be updated with a number or a string. Understands at least the following syntaxes: 'HH:MM:SS', 'HHMMSS', 'HHMM', 'HH'.	3
DATETIME	A type to store date and time information. Format "YYYY-MM-DD HH:MM:SS". Takes 8 bytes. Range '0000-01-01 00:00:00' - '9999-12-31 23:59:59'.	8
YEAR	A type to store years. Format "YYYY" or "YY". Takes 1 byte. Range 0, 1901-2155. 2 digits years in the range 00-69 is assumed to be 2000-2069 and will be sorted correctly. (now type for MySQL 3.22)	1
CHAR(M) [binary]	A fixed length string that is always filled up with spaces to the specified length. Range 1 - 255 characters. All end space are removed when retrieved. Is sorted and compared case insensitively unless the binary keyword is given.	M

VARCHAR(M) [binary]	A variable length string that is stored with its length. NOTE: All end space are removed when storing it (not ANSI SQL). Maximum range 1 - 255 characters. Is sorted and compared case insensitively unless the binary keyword is given. @tab	L+1
TINYTEXT and TINYBLOB	A TEXT/BLOB with max length of 255 characters.	L+1
TEXT and BLOB	A TEXT/BLOB with max length of 65535 characters.	L+2
MEDIUMTEXT and MEDIUMBLOB	A TEXT/BLOB with max length of 16777216 characters.	L+3
LONGTEXT and LONGBLOB	A TEXT/BLOB with max length of 4294967295 characters.	L+4
ENUM('value','value2',...)	A string object that can have only one set of allowed values (or NULL). See section 2.1 More about data types.	1 or 2
SET('value','value2',...)	A string object that can have one or many values of a set of allowed values. See section 2.1 More about data types.	1-8

2.1 More about data types

2.1.1 Database size info.

In the above table L means the actual length of a instance and M the maximum length. So L+1 for "abcd" means 5 bytes in the database.

If you use any data type with an L in the length field you will get a variable length record format.

2.1.2 The numeric types

All integer types can have an optional argument unsigned. This can be used when you only want to allow positive numbers in the column or you need a little bigger numerical range for the column.

Also for all integer columns, the optional argument ZEROFILL means that the column will be padded with zeroes up to the maximum length.

Max display size and decimals are for formatting and calculation of maximum column width.

When storing a value in an integer that is outside its range, MySQL stores the maximum (or minimum) possible value. When doing an ALTER TABLE or LOAD DATA INFILE one gets these conversions as 'warnings'. We have on the TODO to fix INSERT and UPDATE so they can return warnings, but this is scheduled for the next protocol change.

For example when storing -9999999999999999 into an int column the value ends up as -2147483648. And 9999999999999999 ends up as 2147483647.

And if the int is unsigned the stored values above becomes 0 and 4294967296.

The same rules go for all other integer types.

When returning data for an int(4) column that exceeds the allocated space, MySQL will return 9.99. If the operation is an UPDATE a warning will be issued.

Note that a type like decimal(4,2) means maximum 4 characters with two decimal points. That gives a range between -.99 -> 9.99.

To avoid some rounding problems, MySQL always rounds everything that it stores in any floating point column according to the number of decimals. This means that 2.333 stored into float(8,2) is stored as 2.33.

2.1.3 TIMESTAMP type

Has a range of 1 Dec 1970 time 0.00 to sometime in the year 2106 and a resolution of one second. A TIMESTAMP column will automatically be updated on INSERT and UPDATE statements if set to NULL or if the column is not updated in the statement. Can be (part of) an index. Note that if you have many timestamp columns in a row, then only the first timestamp column will be automatically updated. Any timestamp column will be set to the current time if set to NULL. Depending on the display size one gets one of the following formats: "YYYY-MM-DD HH:MM:SS", "YY-MM-DD HH:MM:SS", "YYYY-MM-DD" or "YY-MM-DD".

2.1.4 TEXT and BLOB types

These are objects that can have a variable length without upper limit. All TEXT and BLOB objects are stored with their length (saved in 1 to 4 bytes depending on the type of object). The maximum TEXT and BLOB length you can use is dependent on available memory and client buffers. The only differences between TEXT and BLOB is that TEXT

is sorted and compared case insensitively while BLOB is compared case sensitively (by character values). TEXT and BLOB objects CANNOT be an index.

A BLOB is a binary large object which can hold any amount of data. There are 4 kinds of blobs See section 2.0 Column types.. Normally one can regard a BLOB as a VARCHAR without a specified limit.

TEXT is a BLOB that is sorted and compared case insensitively.

A BLOB/TEXT column may not be bigger than the message buffer. Note that you have to change the message buffer on both the server and the client.

MyODBC defines BLOBs as LONGVARBINARY and TEXTs as LONGVARCHAR.

Restrictions for BLOB and TEXT columns:

1. A BLOB or TEXT cannot be an index or a part of an index
2. When one sorts or groups a BLOB or TEXT, only the first max_sort_length (default 1024) of the blob is used.

This value can be changed by the -O option when starting the mysqld daemon. One can group on an expression involving a BLOB/ TEXT:

```
SELECT id,SUBSTR(blob,1,100) GROUP BY 2
```

3. There is no end space truncation for BLOB and TEXT as there is for CHAR and VARCHAR.

2.1.5 ENUM type

A string object that can have only one of a set of allowed values. The value to be stored may be given case independently. If one tries to store a non-existing value, "" is stored. If used in a number context this object returns/stores the value index. If there is less than 255 possible values this object occupies 1 byte, else two bytes (with a maximum of 65535 different values). Note that if an integer is put in the ENUM you get the corresponding string with the first counting as number 1. (0 is reserved for wrong enum values). Sorting on ENUM types are done according to the order of the strings in the enum. If declared NOT NULL the default value is the first value, else the default value is NULL.

For example the column test ENUM("one","two", "three") can have any of these values:

```
NULL  
"one"  
"two"  
"three"
```

2.1.6 SET type

A string object that can have one or many values from a set of allowed values. Each value is separated by a ','. If used in a number context this object returns/stores the bit positions of the used values. This object occupies $(\text{number_of_different_values}-1)/8+1$ bytes, rounded up to 1,2,3,4 or 8. One can't have more than 64 different values. Note that if an integer is put in the SET you get the corresponding string with the first bit corresponding to the first string. Sorting on SET types are done numerically.

For example the column test SET("one","two") NOT NULL can have any of these values:

```
""  
"one"  
"two"  
"one,two"
```

Normally on SELECT on a SET column with LIKE or FIND_IN_SET():

```
SELECT * from banner where banner_group LIKE '%value%';  
SELECT * from banner where FIND_IN_SET('value',banner_group)>0;
```

But the following will also work:

```
SELECT * from banner where banner_group = 'v1,v2'; ;Exact match  
SELECT * from banner where banner_group & 1; ;Is in first group
```

2.2 Choosing the right type for a column.

Try to use the most precise type in all cases. For example for an integer between 1-99999 an unsigned mediumint is the best type.

A common problem is representing monetary values accurately. In MySQL you should use the DECIMAL type. This is stored as a string so no loss of accuracy should occur. If accuracy is not too important the DOUBLE type may also be good enough.

For high precision you can always convert to a fixed point type stored in a BITINT. This allows you to do all calculations with integers and only convert the result back to floating point.

2.3 Column indexes

You can have indexes on all MySQL columns except BLOB and TEXT types. Using indexes on the relevant columns is the best way to improve the performance of selects.

For CHAR and VARCHAR columns you can have an index on a prefix. The example below shows how to create an index for the first 10 characters of a column. This is much faster and requires less disk space than having an index on the whole column.

```
CREATE TABLE test (name CHAR(200) NOT NULL, KEY index_name (name(10)));
```

2.4 Multiple column indexes

MySQL can have one index on parts of different columns.

A multiple-column index can be considered a sorted array where the columns are concatenated. This makes for fast queries where the first column in the index is a known quantity and the other columns are not.

Suppose that you have a table:

```
CREATE TABLE test (  
    id INT NOT NULL,  
    last_name CHAR(30) NOT NULL,  
    first_name CHAR(30) NOT NULL,  
    PRIMARY KEY (id),  
    INDEX name (last_name,first_name));
```

Then the index name is an index over last_name and first_name.

The name index will be used in the following queries:

```
SELECT * FROM test WHERE last_name="Widenius";
```

```
SELECT * FROM test WHERE last_name="Widenius" AND first_name="Michael";
```

```
SELECT * FROM test WHERE last_name="Widenius" AND  
    (first_name="Michael" OR first_name="Monty");
```

```
SELECT * FROM test WHERE last_name="Widenius" and  
    first_name >="M" and first_name < "N";
```

The name index will NOT be used in the following queries:

```
SELECT * FROM test WHERE first_name="Michael";
```

```
SELECT * FROM test WHERE last_name="Widenius" or first_name="Michael";
```

2.5 Type mapping to ease moving table definitions between different databases engines

To support easier use of code from different SQL vendors, MySQL does supports the following mappings:

binary(num)	char(num) binary
char varying	varchar
float4	float
float8	double
int1	tinyint
int2	smallint
int3	mediumint
int4	int
int8	bigint
long varbinary	blob
long varchar	text
middleint	mediumint
varbinary(num)	varchar(num) binary

3.0 Commands and Syntax

3.1 Create database syntax.

```
CREATE DATABASE database_name
```

Creates a database with the given name. The name can only contain letters, numbers or the '_' character and must start with a letter or a _. The maximum length of a database name is 64 characters. All databases in MySQL are directories, so a CREATE DATABASE only creates a directory in the MySQL database directory. You can also create databases with mysqladmin.

3.2 Drop database syntax.

```
DROP DATABASE [IF EXISTS] database_name
```

Drop all tables in the database and deleted the database. You have to be VERY careful with this command! DROP DATABASE returns how many files was removed from the directory. Normally this is number of tables*3. You can also drop databases with mysqladmin. In MySQL 3.22 one can use the new keywords IF EXISTS to not get an error for a database that doesn't exists.

3.3 CREATE TABLE syntax.

```
CREATE TABLE table_name ( create_definition,... )
```

create_definition:

```
column_name type [NOT NULL | NULL] [DEFAULT  
default_value][AUTO_INCREMENT]
```

```
[ PRIMARY KEY ] [reference_definition]
```

```
or PRIMARY KEY ( index_column_name,... )
```

```
or KEY [index_name] KEY( index_column_name,...)
```

```
or INDEX [index_name] ( index_column_name,...)
```

```
or UNIQUE [index_name] ( index_column_name,...)
```

```
or FOREIGN KEY index_name ( index_column_name,...) [reference_definition]
```

```
or CHECK (expr)
```

type:

- TINYINT[(length)] [UNSIGNED] [ZEROFILL]
- or SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
- or MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
- or INT[(length)] [UNSIGNED] [ZEROFILL]
- or INTEGER[(length)] [UNSIGNED] [ZEROFILL]
- or BIGINT[(length)] [UNSIGNED] [ZEROFILL]
- or REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
- or DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
- or FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
- or DECIMAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
- or NUMERIC[(length,decimals)] [UNSIGNED] [ZEROFILL]
- or CHAR(length) [BINARY],
- or VARCHAR(length) [BINARY],
- or DATE
- or TIME
- or TIMESTAMP
- or DATETIME
- or TINYBLOB
- or BLOB
- or MEDIUMBLOB
- or LONGBLOB
- or TINYTEXT
- or TEXT
- or MEDIUMTEXT
- or ENUM(value1,value2,value3...)
- or SET(value1,value2,value3...)

index_column_name:

column_name [(length)]

reference_definition:

REFERENCES table_name [(index_column_name,...)]
[MATCH FULL | MATCH PARTIAL]
[ON DELETE reference_option]
[ON UPDATE reference_option]

reference_option:

RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

See section 2.0 Column types.

The FOREIGN KEY, CHECK and REFERENCE syntax are only for compatibility. (To make it easier to port code from other SQL servers and run applications that create tables with references). They don't actually do anything.

If a column doesn't have a DEFAULT value and is not declared as NOT NULL, the default value is NULL.

If a column doesn't have a DEFAULT value and is declared as NOT NULL, MySQL will automatically assign a default value for the field.

ZEROFILL means that number is pre-zeroed to maximal length. With INT(5) ZEROFILL a value of 5 is retrieved as 00005.

BINARY means that the column will be compared case sensitive. The default is that all strings are compared case insensitive according to ISO-8859-1 Latin1. BINARY is 'sticky' which means that if a column marked BINARY is used in an expression, the whole expression is compared BINARY.

KEY is a synonym for INDEX.

UNIQUE is in MySQL a key that can only have distinct values. You will get an error if you try to add a new row with a key that matches an old row.

PRIMARY KEY is a unique KEY. One can only have one PRIMARY KEY in a table. If one doesn't assign a name to an index, the index will get the same name as the first key_column with an optional _# to make it unique.

Index columns and timestamp columns can't be NULL. For these columns the NULL attribute is silently removed.

With column_name(length) syntax one can specify an index which is only a part of a string column. This can make the index file much smaller.

A number column may have the additional attribute AUTO_INCREMENT to automatically get the largest value+1 for each insert where column value is NULL or 0.

One can insert NULL for timestamp and auto_increment columns. This results in the current time / the next number.

Blob columns can't be indexes. When one groups on a blob only the first 'max_sort_length' bytes are used.

Limitations of BLOB and TEXT types.

Deleted records are in a linked list and subsequent inserts will reuse old positions. To get smaller files one can use the isamchk utility to reorganize tables.

Each null column takes one bit extra, rounded up to the nearest byte.

The maximum record length can be calculated as follows: $1 + \text{sum_of_column_lengths} + \text{null_columns}/8 + \text{number of variable length columns}$.

In some cases an attribute may silently change after creation: VARCHAR columns with a length of one or two are changed to CHAR. When using one VARCHAR column all CHAR columns longer than 3 are changed to VARCHAR's. This doesn't affect the usage of the column in any way; In MySQL VARCHAR is just a different way to store characters. MySQL does the conversion because it will save space and make the table faster. See when to use VARCHAR/CHAR?

On INSERT/UPDATE all strings (CHAR and VARCHAR) are silently chopped/padded to the maximal length given by CREATE. All end spaces are also automatically removed. For example VARCHAR(10) means that the column can contain strings with a length up to 10 characters.

Something/0 gives a NULL value.

The regular expression function (REGEXP and RLIKE) uses ISO8859-1 (Latin1) when deciding the type of a character.

3.4 ALTER TABLE syntax

```
ALTER [IGNORE] TABLE table_name alter_spec [, alter_spec ...]
```

alter_specification:

```
    ADD [COLUMN] create_definition [AFTER column_name | FIRST]
or   CHANGE [COLUMN] old_column_name create_definition
or   ALTER [COLUMN] column_name { SET DEFAULT literal | DROP DEFAULT }
or   ADD INDEX [index_name] ( index_column_name,...)
or   ADD UNIQUE [index_name] ( index_column_name,...)
or   DROP [COLUMN] column_name
or   DROP PRIMARY KEY
or   DROP INDEX key_name
or   RENAME [AS] new_table_name
```

ALTER TABLE works by creating a temporary table and copying all information to it and then the old table is deleted and the new one is renamed. This is done in such a

way that all updates are automatically redirect to the new table without any failed updates. While the ALTER TABLE is working, the old table is readable for other clients. Table updates/writes to the table are stalled and only executed after the new table is ready.

If IGNORE isn't specified then the copy will be aborted and rolled back if there exists any duplicated unique keys in the new table. In case of duplicates the first found row will be used. This is a MySQL extension.

The CHANGE column_name, DROP column_name and DROP INDEX are MySQL extensions to ANSI SQL92.

The optional word COLUMN is a pure noise word and can be omitted.

The ADD and CHANGE takes the same create_definition as CREATE TABLE. See section 3.3 CREATE TABLE syntax. In MySQL 3.22 you can use ADD ... AFTER column_name or FIRST to add a column at some specific location in your table. The default is the add the column last.

ALTER COLUMN sets a new default value or removes the old default value for a column.

DROP INDEX removes an index. This is a MySQL extension.

The FOREIGN KEY syntax in MySQL exists only for compatibility. If one drops a column_name which is part of some index, this index part is removed. If all index parts are removed then the index is removed.

DROP PRIMARY KEY drops index named PRIMARY or if no such index exists, it drops the first UNIQUE index in the table.

CHANGE tries to convert data to the new format as good as possible. With mysql_info(MYSQL*) one can retrieve how many records were copied and how many records were deleted because of multiple indexes.

To use ALTER TABLE one needs select, insert, delete, update, create and drop privileges on the table. If one uses ALTER TABLE table_name RENAME AS new_name without any other options, MySQL will only do a fast rename of table table.

Some examples of using ALTER TABLE:

```
CREATE TABLE t1 (a INTEGER,b CHAR(10));
INSERT INTO t1 VALUES(1,"testing");
ALTER TABLE t1 RENAME t2;
ALTER TABLE t2 CHANGE a a TINYINT NOT NULL, CHANGE b c CHAR(20);
ALTER TABLE t2 ADD d TIMESTAMP;
ALTER TABLE t2 ADD INDEX (d), ADD PRIMARY KEY (a);
ALTER TABLE t2 DROP COLUMN c;
ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT, ADD
INDEX (c);
DROP TABLE t2;
```

3.5 DROP TABLE syntax.

```
DROP TABLE [IF EXISTS] table_name [, table_name...]
```

Removes one or more tables. All the data and the definition are removed so take it easy with this command!

In MySQL 3.22 one can use the new keywords IF EXISTS to not get an error for tables that doesn't exists.

3.6 DELETE syntax.

```
DELETE FROM table_name WHERE where_definition
```

Returns records affected.

If one does a delete without a WHERE clause then the table is recreated, which is much faster than doing a delete for each row. In these cases, the command returns zero as affected records. MySQL can't return the number of deleted row because the recreate is done without opening the data files to make sure that one can recreate the table as long as the table definition file table_name.frm is valid.

3.7 SELECT syntax

```
SELECT [STRAIGHT_JOIN] [DISTINCT | ALL] select_expression,... [INTO OUTFILE
'file_name' ...] [ FROM table_references [WHERE
where_definition ] [GROUP BY column,...] [HAVING where_definition] [
ORDER BY column [ASC | DESC] ,...] [LIMIT [offset,] rows]
[PROCEDURE procedure_name]]
```

All used keywords must come in exactly the above order. For example a HAVING clause must come after any GROUP BY and before any ORDER BY clause.

Strings are automatically converted to numbers and numbers to strings when needed (a-la Perl). If in a compare operation ((=, <>, <=, <, >=, >)) either of the arguments are numerical, the arguments are compared as numbers, else the arguments are compared as strings. All string comparisons are by default done case-independent by ISO8859-1 (The Scandinavian letter set which also works excellently with English).

```
select 1 > '6x';    -> 0
```

```
select 7 > '6x';    -> 1
```

```
select 0 > 'x6';    -> 0
```

```
select 0 = 'x6';    -> 1
```

A column name does not need a table prefix if the given column name is unique.

A select expression may be given an alias which will be its column name and can be used when sorting and grouping or in the HAVING clause.

```
select concat(last_name, ' ', first_name) as name from table order by name
```

Table_references is a list of tables to join. This may also contain LEFT OUTER JOIN references. See section 3.8 Join syntax.

In LIKE expressions % and _ may be preceded with '\' to skip the wildcard meaning and get a literal % or _.

A DATE is a string with one of the following syntaxes:

YYMMDD (Year is assumed to be 2000 if YY < 70.)

YYYYMMDD

YY.MM.DD Where '.' may be any non-numerical separator.

YYYY.MM.DD Where '.' may be any non-numerical separator.

IFNULL() and IF() return number or string value according to use.

ORDER and GROUP columns may be given as column names, column aliases or column numbers in SELECT clauses.

The HAVING clause can take any column or alias in the select_expressions. It is applied last, just before items are sent to the client, without any optimisation. Don't use it for items that should be in the WHERE clause. You can't write (yet):

```
SELECT user,MAX(salary) FROM users GROUP BY user HAVING max(salary)>10
```

Change it to:

```
SELECT user,MAX(salary) AS sum FROM users GROUP BY user HAVING sum > 10
```

STRAIGHT_JOIN forces the optimizer to join the tables in the same order that the tables are given in the FROM clause. One can use this to get a query to be done more quickly if the optimizer joins the tables in non-optimal order. See section 3.14 EXPLAIN syntax. Get information about a SELECT.

LIMIT takes one or two numerical arguments. If one argument, the argument indicates the maximum number of rows in a result. If two arguments, the first argument says the offset to the first row to return, the second is the maximum number of rows.

INTO OUTFILE 'filename' writes the given set to a file. The file can not already exist from before. See section 3.11 LOAD DATA INFILE syntax.

3.8 Join syntax

MySQL supports the following JOIN syntaxes:

```
table_reference, table_reference  
table_reference [CROSS] JOIN table_reference  
table_reference STRAIGHT_JOIN table_reference  
table_reference LEFT [OUTER] JOIN table_reference ON conditional-expr  
table_reference LEFT [OUTER] JOIN table_reference USING (column-commalist)  
table_reference NATURAL LEFT [OUTER] JOIN table_reference  
{ oj table_reference LEFT OUTER JOIN table_reference ON conditional-expr }
```

The last version is ODBC syntax.

A table reference may be aliased with table_reference AS alias or table_reference alias.

, and JOIN are semantically identical. This does a full join between the used tables. One normally specifies in the WHERE condition how the tables should be linked.

The ON conditional is any WHERE conditional. If there is no matching record for the right table in a LEFT JOIN a row with all columns set to NULL will be used for the right table.

The USING column-list is a list of fields that must exist in both tables. A LEFT JOIN B USING (C1,C2,C3...) is defined to be semantically identical to using an ON expression A.C1=B.C1 AND A.C2=B.C2 AND A.C3=B.C3... .

The NATURAL LEFT JOIN of two tables is defined to be semantically identical to a USING with all column names that exist in both tables.

The last LEFT JOIN syntax exists only for compatibility with ODBC.

STRAIGHT_JOIN is identical as JOIN, except that the left table will always be read before the right table. This can be used in the few cases when the join optimizer puts the tables in the wrong order.

Some examples:

```
SELECT * from table1,table2 where table1.id=table2.id;
SELECT * from table1 LEFT JOIN table2 ON table1.id=table2.id;
SELECT * from table1 LEFT JOIN table2 USING (id);
SELECT table1.* from table1 LEFT JOIN table2 ON table1.id=table2.id WHERE
table2.id IS NULL;
SELECT * from table1 LEFT JOIN table2 ON table1.id=table2.id LEFT JOIN table3 ON
table3.id=table2.id;
```

The fourth example is noteworthy as it finds all rows in table1 that doesn't have an row in table2.

3.9 INSERT syntax

```
INSERT INTO table [ (column_name,...) ] VALUES (expression,...)
or INSERT INTO table [ (column_name,...) ] SELECT ...
```

An expression may use any previous column in column_name list (or table if no column name list is given).

The following holds for a multi-row INSERT statement:

The query cannot contain an ORDER BY clause.

The target table of the INSERT statement cannot appear in the FROM clause of the query.

If one uses INSERT INTO ... SELECT ... then one can get the following info string with the C API function mysql_info(). Records: 100 Duplicates: 0 Warnings: 0 Duplicates are rows which couldn't be written because some index would be duplicated. Warnings are columns which were set to NULL, but have been declared NOT NULL. These will be set to their

default value. In this case it's also forbidden in ANSI SQL to SELECT from the same table that you are inserting into. The problem that there may be problems if the SELECT finds records that is inserted at the same run. When using sub selects the situation could easily be very confusing! If one sets a time stamp value to anything other than NULL, the time stamp value will be copied to the result table. Auto increment columns works as usual.

3.10 REPLACE syntax

```
REPLACE INTO table [ (column_name,...) ] VALUES (expression,...)
or REPLACE INTO table [ (column_name,...) ] SELECT ...
```

This works exactly like INSERT, except that if there was some old record in the table with the same unique index the old record or records will be deleted before this record is inserted. See section 3.9 INSERT syntax.

3.11 LOAD DATA INFILE syntax

```
LOAD DATA INFILE 'text_file_name.text' [REPLACE | IGNORE] INTO TABLE
table_name [FIELDS [TERMINATED BY ','
[OPTIONALLY] ENCLOSED BY '"' ESCAPED BY '\\\ ']]
[LINES TERMINATED BY '\n'] [(Field1, Field2...)]
```

This is used to read rows from a text file, which must be located on the server, at a very high speed. The server-client protocol doesn't yet support files over a connection. If you only have the file on the client, use rcp or ftp to copy it, possibly compressed, to the server before using LOAD DATA INFILE. All paths to the text file are relative to the database directory.

To write data to a text file, use the SELECT ... INTO OUTFILE 'interval.txt' fields terminated by ',' enclosed by '"' escaped by '\\\ ' lines terminated by '\n' FROM ... syntax.

Normally you don't have to specify any of the text file type options. The default is a compact text file with columns separated with tab characters and all rows end with a newline. Tabs, newlines and \ inside fields are prefixed with a \. NULLs are read and written as \N.

FIELDS TERMINATED BY has the default value of \t.

FIELDS [OPTIONALLY] ENCLOSED BY has the default value of ".

FIELDS ESCAPED BY has the default value of '\\\'.

LINES TERMINATED BY has the default value of '\n'.

FIELDS TERMINATED BY and LINES TERMINATED BY may be more than one character.

If LINES TERMINATED BY is an empty string and FIELDS TERMINATED BY is non-empty then lines are also terminated with FIELDS TERMINATED BY.

If FIELDS TERMINATED BY and FIELDS ENCLOSED BY both are empty strings (") then this gives a fixed row format ("not delimited" import format). With a fixed row size NULL values are output as a blank string. If you specify OPTIONALLY in ENCLOSED BY, then only strings are enclosed in ENCLOSED BY by the SELECT ... INTO statement.

Duplicated ENCLOSED BY chars are removed from strings that start with ENCLOSED BY. For example: With ENCLOSED BY ""':

```
"The ""BIG"" boss" -> The "BIG" boss  
The "BIG" boss    -> The "BIG" boss
```

If ESCAPED BY is not empty then the following characters will be prefixed with the escape character: ESCAPED BY, ASCII 0, and the first character in any of FIELDS TERMINATED BY, FIELDS ENCLOSED BY and LINES TERMINATED BY.

If FIELDS ENCLOSED BY is not empty then NULL is read as a NULL value. If FIELDS ESCAPED BY is not empty then \N is also read as a NULL value.

If REPLACE is used, then the new row will replace all rows which have the same unique index. If IGNORE is used, the row will then be skipped if a record already exists with an identical unique key. If none of the above options are used an error will be issued. The rest of the text file will be ignored if one gets a duplicate index error.

Some possible cases that are not supported by LOAD DATA:

Fixed size rows (FIELDS TERMINATED BY and FIELDS ENCLOSED BY both are empty) and BLOB columns.

If some of the separators are a prefix of another.

FIELDS ESCAPED BY is empty and the data contains LINES TERMINATED BY or FIELDS ENCLOSED BY followed by FIELDS TERMINATED BY.

All rows are read into the table. If a row has too few columns, the rest of the columns are set to default values. TIMESTAMP columns are only set to the current time if there is a NULL value for the column or if the TIMESTAMP column is left out from the field list when the field list is used (the last case only holds for the first TIMESTAMP column).

For security reasons the text file must either reside in the database directory or be readable by all. Each user that wants to use LOAD DATA INFILE must also have 'Y' in the 'File_priv' column in the user privilege table!

Because LOAD DATA INFILE regards all input as strings you can't use number values for enum or set columns as you can with INSERT statements. All enum and set must be given as strings!

For more information about the escaped syntax, See section 1.0 Literals. How do you write strings and numbers?.

When the LOAD DATA query is done, one can get the following info string with the C API function `mysql_info()`.

```
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

Warnings are incremented for each column which can't be stored without loss of precision, for each column which didn't get a value from the read text line (happens if the line is too short) and for each line which has more data than can fit into the given columns. A warning is also given for any time, date, timestamp or datetime column that is set to 0.

An example that loads all columns:

```
LOAD DATA INFILE 'persondata.text' INTO TABLE persondata;
```

3.12 UPDATE syntax

UPDATE table SET column=expression,... WHERE where_definition

All updates are done from left to right. If one accesses a column in the expression, update will then use the current value (a given value or the default value) of the column.

```
UPDATE persondata SET count=count+1
```

A UPDATE statements returns how many rows was actually changed. In MySQL 3.22 mysql_info() returns the number of rows that was matched and updated and how warnings one got during the update.

3.13 SHOW syntax. Get information about tables, columns...

```
SHOW DATABASES [LIKE wild]
or SHOW TABLES [FROM database] [LIKE wild]
or SHOW COLUMNS FROM table [FROM database] [LIKE wild]
or SHOW INDEX FROM table [FROM database]
or SHOW STATUS
or SHOW VARIABLES [LIKE wild]
```

Gives information about databases, tables or columns. If the LIKE wild part is used the wild string is a normal SQL wildcard (with % and _). FIELDS may be used as an alias for COLUMNS and KEYS may be used as an alias for INDEXES.

STATUS gives status information from the server like mysqladmin status). The output may differ from the following:

Uptime	Running_threads	Questions	Reloads	Open_tables
119	1	4	1	3

VARIABLES shows the values of the some of MySQL system variables. Most of these variables can be changed by different options to mysqld!

3.14 EXPLAIN syntax. Get information about a SELECT.

```
EXPLAIN SELECT select_options
```

Gives information about how and in which order tables are joined. With the help of EXPLAIN one can see when one has to add more indexes to tables to get a faster select that uses indexes to find the records. You can also see if the optimizer joins the tables in an optimal order. One can force the optimizer to use a specific join order with the STRAIGHT_JOIN option to select.

The different join types are:

- system The table has only one record (= system table)
- const The table has at most one matching record which will be read at the start of the query. All columns in this table will be regarded as constants by the rest of the optimizer.
- eq_ref One record will be read from this table for each combination of the previous tables.
- ref All rows with matching indexes will be read from this table for each combination of the previous tables.
- range Only rows that is in a given index range will be retrieved through an index. The extra column will tell which index is used.
- all A full table scan will be done for each combination of the previous tables.

Here is an example of a join which is optimised with the help of EXPLAIN.

```
EXPLAIN SELECT tt.TicketNumber, tt.TimeIn,
  tt.ProjectReference, tt.EstimatedShipDate,
  tt.ActualShipDate, tt.ClientID,
  tt.ServiceCodes, tt.RepetitiveID,
  tt.CurrentProcess, tt.CurrentDPPerson,
  tt.RecordVolume, tt.DPPrinted, et.COUNTRY,
  et_1.COUNTRY, do.CUSTNAME
FROM tt, et, et AS et_1,
do
WHERE tt.SubmitTime Is Null and tt.ActualPC =
et.EMPLOYID and tt.AssignedPC =
et_1.EMPLOYID and tt.ClientID =
do.CUSTNMBR;
```

The EXPLAIN returns the following:

```
table type possible_keys key key_len ref rows Extra
et ALL PRIMARY NULL NULL NULL 74
do ALL PRIMARY NULL NULL NULL 2135
et_1 ALL PRIMARY NULL NULL NULL 74
tt ALL AssignedPC,ClientID,ActualPC NULL NULL NULL 3872
    range checked for each record (key map: 35)
```

In this case MySQL is doing a full join for all tables! This will take quite a long time as the product of the number of rows in each table must be examined! So if all tables had 1000 records MySQL has to look at $1000^4 = 1000000000000$ rows. If the tables are bigger you can only imagine how long it would take...

In this case the first error is that MySQL can't yet use efficiently indexes on columns that are declared differently: (varchar()) and char() are not different in this context)

In this case tt.ActualPC is char(10) and et.EMPLOYID is char(15).

Fix:

```
mysql> alter table tt change ActualPC ActualPC varchar(15);
```

And the above explanation shows:

```
table type possible_keys key key_len ref rows Extra
tt ALL AssignedPC,ClientID,ActualPC NULL NULL NULL 3872
    where used
do ALL PRIMARY NULL NULL NULL 2135
    range checked for each record (key map: 1)
et_1 ALL PRIMARY NULL NULL NULL 74
    range checked for each record (key map: 1)
et eq_ref PRIMARY PRIMARY 15 tt.ActualPC 1
```

Which is not perfect but much better. This version is executed in a couple of seconds.

After

```
mysql> alter table tt change AssignedPC AssignedPC varchar(15),
    change ClientID Clientid varchar(15);
```

You get the following from EXPLAIN:

```
table type possible_keys key key_len ref rows Extra
et ALL PRIMARY NULL NULL NULL 74
tt ref AssignedPC,ClientID,ActualPC ActualPC 15
   et.EMPLOYID 52 where used
et_1 eq_ref PRIMARY PRIMARY 15 tt.AssignedPC 1
do eq_ref PRIMARY PRIMARY 15 tt.Clientid 1
```

Which is 'almost' as good as it can get. The problem is that MySQL assumes that tt.AcutralPC is evenly distributed which isn't the case in the tt.

Fortunately it is easy to tell MySQL about this:

```
shell> isamchk --analyze PATH_TO_MYSQL_DATABASE/tt
shell> mysqladmin refresh
```

And now the join is 'perfect':

```
table type possible_keys key key_len ref rows Extra
tt ALL AssignedPC,ClientID,ActualPC NULL NULL NULL
   3872 where used
et eq_ref PRIMARY PRIMARY 15 tt.ActualPC 1
et_1 eq_ref PRIMARY PRIMARY 15 tt.AssignedPC 1
do eq_ref PRIMARY PRIMARY 15 tt.Clientid 1
```

3.15 DESCRIBE syntax. Get information about columns.

(DESCRIBE | DESC) table [column]

Gives information about columns. This command is for Oracle compatibility. See section 3.13 SHOW syntax.

Get information about tables, columns.... Column may be a column name or a string. Strings may contain wild cards.

3.16 LOCK TABLES syntax

```
LOCK TABLES table_name [AS alias] READ | WRITE [, table_name READ | WRITE]
...
UNLOCK TABLES
```

Locks tables for this thread. If a thread has a READ lock on a table, the thread (and all other threads) can only read from the table. If a thread has a WRITE lock on a table, then only this thread can READ and WRITE on the table. All threads wait until they get all locks (no timeouts).

When one uses LOCK TABLES one must lock all tables one is going to use! This policy ensures that table locking is deadlock free.

```
LOCK TABLES trans READ, customer AS c WRITE
SELECT SUM(value) FROM trans WHERE customer_id= #some_id#;
UPDATE customer SET total_value=#value_from_last_statement# WHERE
customer_id=#some_id#;
UNLOCK TABLES
```

All tables are automatically unlocked when one issues another LOCK TABLES or if the connection to the server is closed.

Normally you don't have to lock tables. There is a couple of cases when you would like to lock tables anyway:

If you are going to run many operations on a bunch of tables, it's much faster to lock the tables you are going to use. The downside is of course that no other thread can update a READ locked table and no other thread can read a WRITE locked table.

As MySQL doesn't support a transaction environment, you must use lock tables if you want to ensure that no other thread comes between a read and a update. For example the previous example requires

LOCK TABLES to be safe! If one didn't use LOCK TABLES there is a chance that someone inserts a new 'trans' row between the SELECT and UPDATE statements.

By using incremental updates (UPDATE customer set value=value+new_value) or the LAST_INSERT_ID() function you can avoid using LOCK TABLES in many cases.

You can also solve some cases by using user level locks: GET_LOCK() and RELEASE_LOCK(). These locks are saved in a hash table in the server and implemented with pthread_mutex for high speed. See section 7.3.11 Miscellaneous functions.

3.17 SET OPTION syntax.

SET [OPTION] SQL_VALUE_OPTION=value, ...

The used options remain in effect for the whole current session.

The different options are:

SQL_SELECT_LIMIT=value

The maximum number of records to return in any select. If a select has a limit clause it overrides this statement. The default value for a new connection is 'unlimited'.

SQL_BIG_TABLES= 0 | 1

If set to 1 then all temporary tables are stored on disk instead of in memory. This will be a little slower, but one will not get the error The table ### is full for big selects that require a big temporary table. The default value for a new connection is 0 (use in memory temporary tables).

SQL_BIG_SELECTS= 0 | 1

If set to 1 then MySQL will abort if a select is attempted that will probably take a very long time. This is useful when an erroneous WHERE statement has been issued. A big query is defined as a SELECT that will probably have to examine more than max_join_size rows. The default value for a new connection is 0 (which will allow all SELECT's).

CHARACTER SET character_set_name | DEFAULT

This maps all strings from and to the client with the given mapping. Currently the only option for character_set_name is cp1251_koi8, but one can easily add new mappings by editing the file mysql_source_directory/sql/convert.cc. One can restore the default mapping by using DEFAULT as the character_set_name.

SQL_LOG_OFF= 0 | 1

If set to 1 then no logging will be done to the standard log for this client if the client has process list privileges. This doesn't affect the update log!

TIMESTAMP= timestamp_value | DEFAULT

Set the time for this client. This is used to get the original timestamp if one uses the update log to restore rows.

```
LAST_INSERT_ID= #
```

Set the value to be returned from LAST_INSERT_ID(). This is stored in the update log when one uses LAST_INSERT_ID() in a command that updates a table.

3.18 CREATE INDEX syntax (Compatibility function).

```
CREATE [UNIQUE] INDEX index_name ON table_name ( column_name[(length)],... )
```

This function doesn't do anything in MySQL version before version 3.22. This is mapped to a ALTER TABLE call to create indexes. See section 3.4 ALTER TABLE syntax

Normally one creates all INDEX at the same time with CREATE TABLE See section 3.3 CREATE TABLE syntax.

(col1, col2) creates a multiple index over the two columns. The index can be seen as a concatenation of the given columns. If you in CREATE TABLE use INDEX(col1), INDEX(col2) instead of INDEX(col1,col2) you get two separate indexes instead of one multiple index.

```
SELECT * FROM table WHERE col1=# AND col2=#
```

In a case of an index on (col1,col2) the right row(s) can be fetched directly. In a case of (col1), (col2) the optimizer decides which index will find fewer rows and this index will be used to fetch the rows.

If the table has an index (col1,col2,col3...) the prefix of this can be used by the optimizer to find the rows. This means that the above gives you search capabilities on: (col1) and (col1,col2) and (col1,col2,col3)...

MySQL can't use a portion of an index to locate rows through an index.

With the definition (col1,col2,col3):

```
SELECT * FROM table WHERE col1=#  
SELECT * FROM table WHERE col2=#  
SELECT * FROM table WHERE col2=# and col3=#
```

only the first query will use indexes.

MySQL will also use indexes if the LIKE argument is a constant string that doesn't start with a wild character:

The following will use indexes:

```
SELECT * from table WHERE key_column like "Patrick%";  
SELECT * from table WHERE key_column like "Pat%_ck%";
```

In the above cases only rows with Patrick <= key_column < Patricl and Pat <= key_column < Pau will be considered.

The following selects will not use indexes:

```
SELECT * from table WHERE key_column like "%Patrick%";  
SELECT * from table WHERE key_column like other_column;
```

With column_name(length) syntax one can specify an index which is only a part of a string column. This can make the index file much smaller.

```
CREATE INDEX part_of_name ON customer (name(10))
```

As it's quite normal that most names differs in the first 10 characters, the above definition should not slow down searches on names, but it could save a lot of disk and even speed up inserts!

3.19 DROP INDEX syntax (Compatibility function).

```
DROP INDEX index_name
```

This function doesn't do anything in MySQL before version 3.22. This is mapped to a ALTER TABLE call to drop the INDEX or UNIQUE definition. See section 3.4 ALTER TABLE syntax.

3.20 Comment syntax

MySQL supports the # to end of line and /* multiple line */ comment styles.

```
select 1+1; # This comment is to the end of line  
select 1 /* in-line-comment */ + 1;  
select 1+/* This will be ignored  
*/1;
```

MySQL doesn't support the -- ANSI SQL style comments.

3.21 CREATE FUNCTION syntax

```
CREATE FUNCTION <function_name> RETURNS [string | real | integer]
    SONAME <name_of_shared_library>
```

```
DROP FUNCTION <function_name>
```

User definable functions (UDF) is way to extend MySQL with new functions that works as native MySQL functions like ABS() and concat(). UDF's are written in C or C++ and require that dynamic loading works on the operating system. The source distribution includes the file `udf_example.cc' that defines 5 new functions.

The functions name, type and shared library is saved in the new system table 'func' in the 'mysql' database.

To be able to create new functions one must have write privilege for the database 'mysql'. If one starts MySQL with --skip-grant-tables, then UDF initialization will also be skipped.

Each defined function may have a xxxx_init function and a xxxx_deinit function. The init function should alloc memory for the function and tell the main function about the max length of the result (for string functions), number of decimals (for double functions) and if the result may be a null value.

If a function sets the 'error' argument to 1 the function will not be called anymore and mysql will return NULL for all calls to this instance of the function.

All strings arguments to functions are given as string pointer + length to allow handling of binary data. Remember that all functions must be thread safe. This means that one is not allowed to alloc any global or static variables that changes! If one needs memory one should alloc this in the init function and free this on the __deinit function.

A dynamically loadable file should be compiled sharable (something like: gcc -shared -o udf_example.so myfunc.cc). You can easily get all switches right by doing: cd sql ; make udf_example.o Take the compile line that make writes, remove the '-c' near the end of the line and add -o udf_example.so to the end of the compile line. The resulting library (udf_example.so) should be copied to some dir searched by ld, for example /usr/lib.

Some notes about the example functions:

Function `metaphon` returns a metaphon string of the string argument. This is something like a soundex string, but it's more tuned for English.

Function `myfunc_double` returns summary of codes of all letters of arguments divided by summary length of all its arguments.

Function `myfunc_int` returns summary length of all its arguments.

Function `lookup` returns the IP number for an hostname.

Function `reverse_lookup` returns the hostname for a IP number. The function may be called with a string `"xxx.xxx.xxx.xxx"` or four numbers.

After the library is installed one must notify `mysqld` about the new functions with the commands:

```
CREATE FUNCTION metaphon RETURNS STRING SONAME "udf_example.so";
CREATE FUNCTION myfunc_double RETURNS REAL SONAME "udf_example.so";
CREATE FUNCTION myfunc_int RETURNS INTEGER SONAME "udf_example.so";
CREATE FUNCTION lookup RETURNS STRING SONAME "udf_example.so";
CREATE FUNCTION reverse_lookup RETURNS STRING SONAME "udf_example.so";
```

Functions should be created only once. The functions can be deleted by:

```
DROP FUNCTION metaphon;
DROP FUNCTION myfunc_double;
DROP FUNCTION myfunc_int;
DROP FUNCTION lookup;
DROP FUNCTION reverse_lookup;
```

The `CREATE FUNCTION` and `DROP FUNCTION` update the func table. All active function will be reloaded on everyrestart of server (if `--skip-grant-tables` is not given).

3.22 Is MySQL picky about reserved words?

A common problem stems from trying to create a table with column names `timestamp` or `group`, the names of datatypes and functions built into MySQL. You're allowed to do it (for example, `ABS` is an allowed column name), but whitespace is not allowed between a function name and the `'` when using the functions whose names are also column names.

The following are explicitly reserved words in MySQL. Most of them (for example) `group`, are forbidden by ANSI SQL92 as column and/or table names. A few are because MySQL needs them and is (currently) using a yacc parser:

action	add	all	alter
and	as	asc	auto_increment
between	bigint	bit	binary
blob	both	by	cascade
char	character	change	check
column	columns	create	data
database	databases	date	datetime
day	day_hour	day_minute	day_second
dayofweek	dec	decimal	default
delete	desc	describe	distinct
double	drop	escaped	enclosed
enum	explain	fields	float
float4	float8	foreign	from
for	full	grant	group
having	hour	hour_minute	hour_second
ignore	in	index	infile
insert	int	integer	interval
int1	int2	int3	int4
int8	into	is	join
key	keys	leading	left
like	lines	limit	lock
load	long	longblob	longtext
match	mediumblob	mediumtext	mediumint
middleint	minute	minute_second	month
natural	numeric	no	not
null	on	option	optionally
or	order	outer	outfile
partial	precision	primary	procedure
privileges	read	real	references
rename	regexp `repeat	replace	
restrict	rlike	select	set
show	smallint	sql_big_tables	sql_big_selects
sql_select_limit	sql_log_off	straight_join	starting
table	tables terminated	text	
time	timestamp	tinyblob	tinytext
tinyint	trailing	to	use
using	unique	unlock	unsigned
update	usage	values	varchar
varying	varbinary	with	write
where	year	year_month	zerofill

The following symbols (from the table above) are disallowed by ANSI SQL but allowed by MySQL as column/table names. This is because some of these names are very natural names and a lot of people have already used them.

ACTION
BIT
DATE
ENUM
NO
TEXT
TIME
TIMESTAMP

4.0 Functions for use in SELECT and WHERE clauses

A select_expression or where_definition can consist of any expression using the following functions:

In the examples below the output of the mysql program has been shortened. So this:

```
mysql> select mod(29,9);
1 rows in set (0.00 sec)
```

```
+-----+
| mod(29,9) |
+-----+
|      2 |
+-----+
```

Has been converted to:

```
mysql> select mod(29,9);      -> 2
```

4.1 Grouping functions.

() Parenthesis. Force order of evaluation in a expression.

```
mysql> select 1+2*3;          -> 7
mysql> select (1+2)*3;        -> 9
```

4.2 Normal arithmetic operations.

+ Addition

- Subtraction.

* Multiplication

/ Division. A division by zero results in a NULL.

```
mysql> select 102/(1-1);      -> NULL
```

4.3 Bit functions.

These have a range of maximum 64 bits because MySQL uses bigint (64 bit) arithmetic.

| Bitwise OR.

```
mysql> select 29 | 15;          ->  31
```

& Bitwise and.

```
mysql> select 29 & 15;         ->  13
```

<< Shift a longlong number to the right

```
mysql > select 1 << 2         ->  4
```

<< Shift a longlong number to the left

```
mysql > select 4 >> 2        ->  1
```

BIT_COUNT()

Number of set bits in an argument.

```
mysql> select bit_count(29);   ->  4
```

4.4 Logical operations.

All logical function return 1 (TRUE) or 0 (FALSE).

NOT

! Logical NOT. Return 1 if argument is 0 else return 0.

```
mysql> select NOT 1;          ->  0
mysql> select NOT NULL;      ->  NULL
mysql> select !(1+1);        ->  0
mysql> select ! 1+1;         ->  1
```

OR

|| Logical OR. Return 1 if any of the arguments are non 0 and not NULL.

```
mysql> select 1 || 0;        ->  1
mysql> select 0 || 0;        ->  0
mysql> select 1 || NULL;     ->  1
```

AND

&& Logical AND. Return 1 if all of the arguments are non 0 or NULL

```
mysql> select 1 && NULL;      ->  0
mysql> select 1 && 0;         ->  0
```

4.5 Comparison operators.

Returns 1 (TRUE), 0 (FALSE) or NULL. These functions work for both numbers and strings. MySQL uses the following rules to decide how the compare is done:

If both arguments to a compare operation are strings, compare as strings.

If both arguments are integers, compare as integers.

If one of the arguments is a `TIMESTAMP` or `DATETIME` column and the other argument is a constant. In this case the constant is converted to a timestamp before the comparasion. This is to be more ODBC friendly.

In all other cases compare as floating point numbers (real).

If one or both of the arguments are `NULL` the result of the comparison is `NULL`.

= Equal.

```
mysql> select 1 = 0;          ->  0
mysql> select '0' = 0;       ->  1
mysql> select '0.0' = 0;     ->  1
mysql> select '0.01' = 0;    ->  0
mysql> select '.01' = 0.01;  ->  1
```

<>

!= Not equal.

```
mysql> select '.01' <> '0.01'; ->  1
mysql> select .01 <> '0.01';   ->  0
mysql> select 'zapp' <> 'zappp'; ->  1
```

<= Smaller than or equal.

```
mysql> select 0.1 <= 2;       ->  1
```

< Smaller than.

```
mysql> select 2 <= 2;         ->  1
```

>= Bigger than or equal.

```
mysql> select 2 >= 2;          -> 1
```

> Bigger than.

```
mysql> select 2 > 2;          -> 0
```

ISNULL(A)

Returns 1 if A is NULL else 0.

```
mysql> select isnull(1+1);    -> 0
```

```
mysql> select isnull(1/0);    -> 1
```

A BETWEEN B AND C

A is bigger or equal as B and A is smaller or equal to C. Does the same thing as (A >= B AND A <= C) if all arguments are of the same type. It's the first argument (A) that decides how the comparison should be done! If A is a string expression, compare as case insensitive strings. If A is a binary string, compare as binary strings. If A is an integer expression compare as integers, else compare as reals.

```
mysql> select 1 between 2 and 3; -> 0
```

```
mysql> select 'b' between 'a' and 'c'; -> 1
```

```
mysql> select 2 between 2 and '3'; -> 1
```

```
mysql> select 2 between 2 and 'x-3'; -> 0
```

4.6 String comparison functions.

Normally if one expression that is compared is not case sensitive than the compare is done case insensitively.

expr IN (value,...)

Returns 1 if expr is any of the values in the IN list, else it returns 0. If all values are constants, then all values are evaluated according to the type of expr and sorted. The search for the item is then done by using a binary search. This means IN is very quick when used with constants in the IN part. If expr is a string expression then string compare is done case sensitively if expr is case sensitive.

```
mysql> select 2 in (0,3,5,'wefwf'); -> 0
```

```
mysql> select 'wefwf' in (0,3,5,'wefwf'); -> 1
```

expr NOT IN (value,...)

Same as NOT (expr IN (value,...))

expr LIKE expr

SQL simple regular expression comparison. Returns 1 (TRUE) or 0 (FALSE). With LIKE you have two wild characters.

% Matches any number of characters, even zero characters.

_ Matches exactly one character.

\% Matches one %.

_ Matches one _.

```
mysql> select 'David!' like 'David_';      -> 1
mysql> select 'David!' like 'David\_';     -> 0
mysql> select 'David_' like 'David\_';     -> 1
mysql> select 'David!' like '%D%v%';      -> 1
mysql> select 10 like '1%';                -> 1
```

LIKE is allowed on numerical expressions! (Extension)

expr NOT LIKE expr Same as NOT (expr LIKE expr).

expr REGEXP expr

expr RLIKE expr Checks string against extended regular expr. RLIKE is for mSQL compatibility. NOTE: Because MySQL uses the C escape syntax in strings (\n) You must double any '\' that you uses in your REGEXP strings.

```
mysql> select 'Monty!' regexp 'm%y%%';    -> 0
mysql> select 'Monty!' regexp '.*';        -> 1
mysql> select 'new*\n*line' regexp 'new\\*.*line'
```

expr NOT REGEXP expr Same as NOT (expr REGEXP expr).

STRCMP() Returns 0 if the strings are the same. Otherwise return -1 if the first argument is smaller according to the current sort-order, otherwise return 1.

```
mysql> select strcmp('text', 'text2');     -> -1
mysql> select strcmp('text2', 'text');     -> 1
mysql> select strcmp('text', 'text');     -> 0
```

4.7 Control flow functions.

IFNULL(A,B)

If A is not NULL it returns A, else B.

```
mysql> select ifnull(1,0);    -> 1
mysql> select ifnull(0,10);   -> 0
mysql> select ifnull(1/0,10); -> 10
```

IF(A,B,C)

If A is true (A <> 0 and A <> NULL) then return B, else return C. A is evaluated as an INTEGER, which means that if you are using floats you should also use a comparison operation.

```
mysql> select if(1>2,2,3);    -> 3
```

4.8 Mathematical functions.

All mathematical functions returns NULL in the case of a error.

- Sign. Changes sign of argument.

```
mysql> select - 2;           -> -2
```

ABS() Absolute value.

```
mysql> select abs(2);        -> 2
mysql> select abs(-32);      -> 32
```

SIGN() Sign of argument. Returns -1, 0 or 1.

```
mysql> select sign(-32);     -> -1
mysql> select sign(0);       -> 0
mysql> select sign(234);     -> 1
```

MOD() % Modulo (like % in C).

```
mysql> select mod(234, 10);  -> 4
mysql> select 253 % 7;       -> 1
mysql> select mod(29,9);     -> 2
```

FLOOR() Largest integer value not greater than x.

```
mysql> select floor(1.23);      -> 1
mysql> select floor(-1.23);     -> -2
```

CEILING() Smallest integer value not less than x.

```
mysql> select ceiling(1.23);    -> 2
mysql> select ceiling(-1.23);   -> -1
```

ROUND(N) Round argument N to an integer.

```
mysql> select round(-1.23);     -> -1
mysql> select round(-1.58);     -> -2
mysql> select round(1.58);      -> 2
```

ROUND(Number,Decimals) Round argument Number to a number with Decimals decimals.

```
mysql> select ROUND(1.298, 1);  -> 1.3
```

EXP(N) Returns the value of e (the base of natural logarithms) raised to the power of N.

```
mysql> select exp(2);           -> 7.389056
mysql> select exp(-2);          -> 0.135335
```

LOG(X) Return the natural logarithm of X.

```
mysql> select log(2);           -> 0.693147
mysql> select log(-2);          -> NULL
```

LOG10(X) return the base-10 logarithm of X.

```
mysql> select log10(2);         -> 0.301030
mysql> select log10(100);       -> 2.000000
mysql> select log10(-100);      -> NULL
```

POW(X,Y)

POWER(X,Y) Return the value of X raised to the power of Y.

```
mysql> select pow(2,2);         -> 4.000000
mysql> select pow(2,-2);        -> 0.250000
```

sqrt(X) Returns the non-negative square root of X.

```
mysql> select sqrt(4);      -> 2.000000
mysql> select sqrt(20);    -> 4.472136
```

PI() Return the value of PI.

```
mysql> select PI();       -> 3.141593
```

COS(X) Return the cosine of X, where X is given in radians.

```
mysql> select cos(PI());  -> -1.000000
```

SIN(X) Return the sine of X, where X is given in radians.

```
mysql> select sin(PI());  -> 0.000000
```

TAN(X) Returns the tangent of X, where X is given in radians.

```
mysql> select tan(PI()+1); -> 1.557408
```

ACOS(X) Return the arc cosine of X; that is the value whose cosine is X. If X is not in the range -1 to 1 NULL is returned.

```
mysql> select ACOS(1);    -> 0.000000
mysql> select ACOS(1.0001); -> NULL
mysql> select ACOS(0);    -> 1.570796
```

ASIN(X) Return the arc sine of X; that is the value whose sine is X. If X is not in the range -1 to 1 NULL is returned.

```
mysql> select ASIN(0.2);  -> 0.201358
mysql> select ASIN('foo'); -> 0.000000
```

ATAN(X) Return the arc tangent of X; that is the value whose tangent is X.

```
mysql> select ATAN(2);    -> 1.107149
mysql> select ATAN(-2);   -> -1.107149
```

ATAN2(X,Y) Return 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.

```
mysql> select ATAN(-2,2);      -> -0.785398
mysql> select ATAN(PI(),0);    -> 1.570796
```

COT(N) Return the cotangens of N.

```
mysql> select COT(12);        -> -1.57267341
mysql> select COT(0);         -> NULL
```

RAND([X]) Returns a random float, $0 \leq x \leq 1.0$, using the integer expression X as the optional seed value.

```
mysql> SELECT RAND();        -> 0.5925
mysql> SELECT RAND(20);      -> 0.1811
mysql> SELECT RAND(20);      -> 0.1811
mysql> SELECT RAND();        -> 0.2079
mysql> SELECT RAND();        -> 0.7888
```

One can't do an ORDER BY on a column with RAND() values because ORDER BY would evaluate the column multiple times.

MIN(X,Y...)

Min value of arguments. Must have 2 or more arguments, else these are GROUP BY functions. The arguments are compared as numbers. If no records are found NULL is returned.

```
mysql> SELECT MIN(2,0);      -> 0
mysql> SELECT MIN(34,3,5,767); -> 3
mysql> SELECT MIN(a) from table where 1=0; -> NULL
```

MAX(X,Y...) Max value of arguments. Must have 2 or more arguments, else these are GROUP BY functions. The arguments are compared as numbers. If no records are found NULL is returned.

```
mysql> SELECT MAX(34,3,5,767); -> 767
mysql> SELECT MAX(2,0,4,5,34); -> 34
mysql> SELECT MAX(a) from table where 1=0; -> NULL
```

DEGREES(N) Return N converted from radians to degrees.

```
mysql> select DEGREES(PI()); -> 180.000000
```

RADIANS(N) Return N converted from degrees to radians.

```
mysql> select RADIANS(90);          -> 1.570796
```

TRUNCATE(Number, Decimals) Truncate number Number to Decimals decimals.

```
mysql> select TRUNCATE(1.223,1);    -> 1.2
mysql> select TRUNCATE(1.999,1);    -> 1.9
mysql> select TRUNCATE(1.999,0);    -> 1
```

4.9 String functions.

ASCII(S) Returns the ASCII code value of the leftmost character of S. If S is NULL return NULL.

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

CHAR(X,...) Returns a string that consists of the characters given by the ASCII code values of the arguments. NULLs are skipped.

```
mysql> SELECT char(77,121,83,81,'76'); -> 'MySQL'
```

CONCAT(X,Y...) Concatenates strings. May have more than 2 arguments.

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

LENGTH(S) Length of string.

OCTET_LENGTH(S)

CHAR_LENGTH(S)

CHARACTER_LENGTH(S)

```
mysql> SELECT length('text');      -> 4
mysql> SELECT octet_length('text'); -> 4
```

LOCATE(A,B) Returns position of A substring in B. The first position is 1.
POSITION(B IN A) Returns 0 if A is not in B.

```
mysql> select locate('bar', 'foobarbar'); -> 4
mysql> select locate('xbar', 'foobar'); -> 0
```

INSTR(A,B) Returns position of first substring B in string A. This is the same as LOCATE with swapped parameters.

```
mysql> select instr('foobarbar', 'bar');    -> 4
mysql> select instr('xbar', 'foobar');     -> 0
```

LPAD(A,B,C) Left pad (at the start) the string A with C until A's length is B.

```
mysql> select lpad('hi',4,'??')           -> '??hi'
```

RPAD(A,B,C) Right pad (at the end) the string A with C until A's length is B.

```
mysql> select rpad('hi',5,'?')           -> 'hi???'
```

LOCATE(A,B,C) Returns position of first substring A in string B starting at C.

```
mysql> select locate('bar', 'foobarbar',5); -> 7
```

LEFT(str,length) Gets length in characters from beginning of string.

```
mysql> select left('foobarbar', 5);       -> 'fooba'
```

RIGHT(A,B)

SUBSTRING(A FROM B) Gets B characters from end of string A.

```
mysql> select right('foobarbar', 5);      -> 'arbar'
mysql> select substring('foobarbar' from 5); -> 'arbar'
```

LTRIM(str) Removes space characters from the beginning of string.

```
mysql> select ltrim(' barbar');           -> 'barbar'
```

RTRIM(str) Removes space characters from the end of string.

```
mysql> select rtrim('barbar ');          -> 'barbar'
```

TRIM([[BOTH | LEADING | TRAILING] [A] FROM] B)

Returns a character string with all A prefixes and/or suffixes removed from B. If BOTH, LEADING and TRAILING isn't used BOTH are assumed. If A is not given, then spaces are removed.

```
mysql> select trim(' barbar ');           -> 'bar'
mysql> select trim(leading 'x' from 'xxxbarxxx'); -> 'barxxx'
mysql> select trim(both 'x' from 'xxxbarxxx');   -> 'bar'
mysql> select trim(trailing 'xyz' from 'barxyz'); -> 'barx'
```

SOUNDEX(S)

Gets a soundex string from S. Two strings that sound 'about the same' should have identical soundex strings. A 'standard' soundex string is 4 characters long, but this function returns an arbitrary long string. One can use SUBSTRING on the result to get a 'standard' soundex string. All non alpha characters are ignored in the given string. All characters outside the A-Z range are treated as vocals.

```
mysql> select soundex('Hello');          -> 'H400'  
mysql> select soundex('Bättre');        -> 'B360'  
mysql> select soundex('Quadratically'); -> 'Q36324'
```

SUBSTRING(A, B, C)

SUBSTRING(A FROM B FOR C)

MID(A, B, C)

Returns substring from A starting at B with C chars. The variant with FROM is ANSI SQL 92 syntax.

```
mysql> select substring('Quadratically',5,6);    -> ratica
```

SUBSTRING_INDEX(String, Delimiter, Count)

Returns the substring from String after Count Delimiters. If Count is positive the strings are searched from left else if count is negative the substrings are searched and returned from right.

```
mysql> select substring_index('www.tcx.se', '.', 2); -> 'www.tcx'  
mysql> select substring_index('www.tcx.se', '.', -2); -> 'tcx.se'
```

SPACE(N) Return a string of N spaces.

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

REPLACE(A, B, C) Replaces all occurrences of string B in string A with string C.

```
mysql> select replace('www.tcx.se', 'w', 'Ww'); -> 'WwWwWw.tcx.se'
```

REPEAT(String, Count)

Repeats String Count times. If Count <= 0 returns a empty string. If String or Count is NULL or LENGTH(string)*count > max_allowed_size returns NULL.

```
mysql> select repeat('MySQL', 3);      -> 'MySQLMySQLMySQL'
```

REVERSE(String) Reverses all characters in string.

```
mysql> select reverse('abc'); -> 'cba'
```

INSERT(Org, Start, Length, New)

Replaces substring in Org starting at Start and Length long with New. First position in Org is numbered 1.

```
mysql> select insert('Quadratic', 3, 4, 'What'); -> 'QuWhattic'
```

INTERVAL(N, N1, N2, N3...)

It is required that $N_n > N_3 > N_2 > N_1$ is this function shall work. This is because a binary search is used (Very fast). Returns 0 if $N < N_1$, 1 if $N < N_2$ and so on. All arguments are treated as numbers.

```
mysql> select INTERVAL(23, 1, 15, 17, 30, 44, 200); -> 3
```

```
mysql> select INTERVAL(10, 1, 10, 100, 1000); -> 2
```

```
mysql> select INTERVAL(22, 23, 30, 44, 200); -> 0
```

ELT(N, A1, A2, A3...)

Returns A1 if N = 1, A2 if N = 2 and so on. If N is less than 1 or bigger than the number of arguments NULL is returned.

```
mysql> select elt(1, 'ej', 'Heja', 'hej', 'foo'); -> 'ej'
```

```
mysql> select elt(4, 'ej', 'Heja', 'hej', 'foo'); -> 'foo'
```

FIELD(S, S1, S2, S3...)

Returns index of S in S1, S2, S3... list. The complement of ELT(). Return 0 when S is not found.

```
mysql> select FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo'); -> 2
```

```
mysql> select FIELD('fo', 'Hej', 'ej', 'Heja', 'hej', 'foo'); -> 0
```

FIND_IN_SET(string,string of strings)

Returns a value 1 to N if the 'string' is in 'string of strings'. A 'string of strings' is a string where each different value is separated with a ','. If the first argument is a constant string and the second is a column of type SET, the FIND_IN_SET is optimized to use bit arithmetic!

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

This function will not work properly if the first argument contains a ','.

LCASE(A)

LOWER(A)

Changes A to lower case according to current character set ,dmappings (Default Latin1).

```
mysql> select lcase('QUADRATICALLY');      -> 'quadratically'
```

UCASE(A) Changes A to upper case.

UPPER(A)

```
mysql> select ucase('Hej');      -> 'HEJ'
```

4.10 Date and time functions.

Some examples using more than one date function:

Select all record with a date_field from the last 30 days.

```
SELECT something FROM table
WHERE TO_DAYS(NOW()) - TO_DAYS(date_field) <= 30;
```

A Date expression may be a date string, a datetime string, a timestamp([6 | 8 | 14]) or a number of format YYMMDD or YYYYMMDD.

In a date expression a year may be 2 or 4 digits. 2 digits is assumed to be in the range 1970-2069. Dates 100-199 00' can be stored and retrieved as 0000-00-00.

If you use a date function with a number, then if the length of the number is 4, 8 or >= 14 then the year is assumed to have 4 digits. In all other cases the year is assumed to be the 2 first digits in the given number. To be on the safe side when using dates as numbers (not strings) one should always use 4 digit dates! If not you will get in trouble

with year 2000 when a number 002001 is sent to the date functions as '2001' instead of the date '20002001'. '002001' will of course work correctly!

A Time expression may be a date string, a datetime string, a timestamp([6 | 8 | 14]) or a number of format HHMMSS or YYYYMMDDHHMMSS.

DAYOFWEEK(date expr)

Gets weekday for Date (1 = Sunday, 2 = Monday, 2 = Tuesday ..) This is according to the ODBC standard.

```
mysql> select dayofweek('1998-02-03');      -> 3
```

WEEKDAY(date expr) Gets weekday for Date (0 = Monday, 1 = Tuesday ..)

```
mysql> select WEEKDAY('1997-10-04 22:23:00'); -> 5
mysql> select WEEKDAY('1997-11-05');        -> 2
```

DAYOFMONTH(date expr) Returns day of month (1-31)

```
mysql> select DAYOFMONTH('1998-02-03');     -> 3
```

DAYOFYEAR(date expr) Returns day of year (1-366)

```
mysql> select DAYOFYEAR('1998-02-03');     -> 34
```

MONTH(date expr) Returns month (1-12)

```
mysql> select MONTH('1998-02-03');        -> 02
```

DAYNAME(date expr) Returns the name of the day.

```
mysql> select dayname("1998-02-05");      -> Thursday
```

MONTHNAME(date expr) Returns the name of the month.

```
mysql> select monthname("1998-02-05");    -> February
```

QUARTER(date expr) Returns quarter (1-4).

```
mysql> select QUARTER('98-04-01');        -> 2
```

WEEK(date expr)

Returns week (1-53) on locations where Sunday is the first day of the year

```
mysql> select WEEK('98-02-20');      -> 7
```

YEAR(date expr) Returns year (1000-9999).

```
mysql> select YEAR('98-02-03');     -> 1998
```

HOUR(time expr) Returns hour (0-23)

```
mysql> select HOUR('10:05:03');     -> 10
```

MINUTE(time expr) Returns minute (0-59).

```
mysql> select MINUTE('98-02-03 10:05:03'); -> 5
```

SECOND(time expr) Returns seconds (1000-9999).

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

PERIOD_ADD(P, N)

Adds N months to period P (of type YYMM or YYYYMM). Returns YYYYMM.

```
mysql> select PERIOD_ADD(9801,2);    -> 199803
```

PERIOD_DIFF(A, B)

Returns months between periods A and B. A and B should be of format YYMM or YYYYMM.

```
mysql> select PERIOD_DIFF(9802,199703); -> 11
```

TO_DAYS(Date)

Changes a Date to a daynumber (Number of days since year 0). Date may be a DATE string, a DATETIME string, a TIMESTAMP([6 | 8 | 14]) or a number of format YYMMDD or YYYYMMDD.

```
mysql> select TO_DAYS(9505);         -> 733364
mysql> select TO_DAYS('1997-10-07'); -> 729669
```

FROM_DAYS() Changes a daynumber to a DATE.

```
mysql> select from_days(729669);            -> 1997-10-07
```

DATE_FORMAT(Date, Format)

Formats the Date (a date or a timestamp) according to the Format string. The following format commands are known:

- M Month name
- W Weekday name
- D Day of the month with english suffix
- Y Year with 4 digits
- y Year with 2 digits
- a Abbreviated weekday name (Sun..Sat)
- d Day of the month, numeric
- m Month, numeric
- b Abbreviated month name (Jan.Dec)
- j Day of year (001..366)
- H Hour (00..23)
- k Hour (0..23)
- h Hour (01..12)
- I Hour (01..12)
- l Hour (1..12)
- i Minutes, numeric
- r Time, 12-hour (hh:mm:ss [AP]M)
- T Time, 24-hour (hh:mm:ss)
- S Seconds (00..59)
- s Seconds (00..59)
- p AM or PM
- w Day of the week (0=Sunday..)
- % single % are ignored. Use %% for a % (for future extensions).

All other characters are copied to the result.

```
mysql> select date_format('1997-10-04 22:23:00', '%W %M %Y %h:%i:%s');  
-> 'Saturday October 1997 22:23:00'
```

```
mysql> select date_format('1997-10-04 22:23:00', '%D %y %a %d %m %b  
%j %H %k %I %r %T %S %w');  
-> '4th 97 Sat 04 10 Oct 277 22 22 10 10:23:00 PM 22:23:00 00 6'
```

For the moment % is optional. In future versions of MySQL % will be required.

TIME_FORMAT(time expr, format) This can be used like the DATE_FORMAT above, but only with the format options which handle hours, minutes and seconds. Other options give NULL value or 0.

CURDATE()
CURRENT_DATE

Returns today's date. In form YYYYMMDD or 'YYYY-MM-DD' depending on whether CURDATE() is used in a number or string context.

```
mysql> select CURDATE();          -> '1997-12-15'  
mysql> select CURDATE()+0;       -> 19971215
```

CURTIME()
CURRENT_TIME

Returns the current time in the form HHMMSS or 'HH:MM:SS', depending on whether CURTIME() is used in a number or string context.

```
mysql> select CURTIME();         -> '23:50:20'  
mysql> select CURTIME()+0;      -> 235026
```

NOW()
SYSDATE()
CURRENT_TIMESTAMP

Returns the current time. In format YYYYMMDDHHMMSS or 'YYYY-MM-DD HH:MM:SS' depending on whether NOW() is used in a number or string context.

```
mysql> select NOW();            -> '1997-12-15 23:51:26'  
mysql> select NOW()+0;         -> 19971215235131
```

UNIX_TIMESTAMP([date expression])

If called without any arguments, a unix timestamp (seconds in GMT since 1970.01.01 00:00:00). Normally it is called with a TIMESTAMP column as an argument in which case it returns the columns value in seconds. Date may also be a date string, a datetime string, or a number of format YYMMDD or YYYYMMDD in local time.

```
mysql> select UNIX_TIMESTAMP();  -> 882226357  
mysql> select UNIX_TIMESTAMP('1997-10-04 22:23:00'); -> 875996580
```

FROM_UNIXTIME(Unix_timestamp)

Returns a string of the timestamp in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format depending on context (numeric/string).

```
mysql> select FROM_UNIXTIME(875996580); -> '1997-10-04 22:23:00'
```

FROM_UNIXTIME(Unix_timestamp, Format_string) Returns a string of the timestamp formatted according to the Format_string. The format string may contain:

M	Month, textual
W	Day (of the week), textual
D	Day (of the month), numeric plus english suffix
Y	Year, numeric, 4 digits
y	Year, numeric, 2 digits
m	Month, numeric
d	Day (of the month), numeric
h	Hour, numeric
i	Minutes, numeric
s	Seconds, numeric
w	Day (of the week), numeric
All other	All other characters are just copied.

```
mysql> select FROM_UNIXTIME(UNIX_TIMESTAMP(), 'Y D M h:m:s x');  
-> '1997 23rd December 03:12:30 x'
```

SEC_TO_TIME(Seconds)

Returns the hours, minutes and seconds of the argument in H:MM:SS or HMMSS format depending on context.

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

TIME_TO_SEC(Time)

Converts Time to seconds.

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

4.11 Miscellaneous functions.

DATABASE() Returns current database name.

```
mysql> select DATABASE();      -> 'test'
```

USER()

SYSTEM_USER()

SESSION_USER() Returns current user name.

```
mysql> select USER();          -> 'davida'
```

PASSWORD(String)

Calculates a password string from plaintext password String. This must be used to store a password in the 'user' grant table.

```
mysql> select PASSWORD('badpwd');  -> '7f84554057dd964b'
```

ENCRYPT(String[, Salt])

Crypt String with the unix crypt() command. The Salt should be a string with 2 characters. If crypt() was not found NULL will always be returned.

LAST_INSERT_ID()

Returns the last automatically generated value that was set in an auto_increment column.

```
mysql> select LAST_INSERT_ID();    -> 1
```

FORMAT(Nr, Num)

Formats number Nr to a Format like '#,###,###.##' with Num decimals.

```
mysql> select FORMAT(12332.33, 2);  -> '12,332.33'
```

VERSION Return the version of the MySQL server.

```
mysql> select version();           -> '3.21.16-beta-log'
```

GET_LOCK(String,timeout)

Tries to get a lock on named 'String' with a timeout of 'timeout' seconds. Returns 1 if one got the lock, 0 on timeout and NULL on error (like out of memory or if thread was killed with mysqladmin kill. A lock is released if one executes RELEASE_LOCK, executes a new GET_LOCK or if the thread ends. This function can be used to implement application locks or simulate record locks.

```
mysql> select get_lock("automaticly released",10);    -> 1
mysql> select get_lock("test",10);                   -> 1
mysql> select release_lock("test");                   -> 1
mysql> select release_lock("automaticly released")    -> NULL
```

RELEASE_LOCK(String)

Releases a lock this thread has got with GET_LOCK. Returns 1 if the lock was released, 0 if lock wasn't locked by this thread and NULL if the lock 'String' didn't exist.

4.12 Functions for GROUP BY clause.

COUNT(Expr)

Count number of non NULL rows. count(*) is optimised to return very quickly if no other column is used in the SELECT.

```
select count(*) from student;
select count(if(length(name)>3,1,NULL)) from student;
```

AVG(expr) Average value of expr.

MIN(expr)

MAX(expr) Minimum/Maximum value of expr. min() and max() may take a string argument and will then return the minimum/maximum string value.

SUM(expr) Sum of expr.

STD(expr)

STDDEV(expr) (Oracle format) Standard derivative of expression. This is a extension to ANSI SQL.

BIT_OR(expr) The bitwise OR of all bits in expr. Calculation done with 64 bit precision.

BIT_AND(expr) The bitwise AND of all bits in expr. Calculation done with 64 bit precision.

MySQL has extended the use of GROUP BY. You can use columns or calculations in the SELECT expressions which don't appear in the GROUP BY part. This stands for 'any possible value for this group'. By using this, one can get a higher performance by avoiding sorting and grouping on unnecessary items. For example, in the following query one doesn't need to group on b.name:

```
SELECT a.id,b.name,COUNT(*) from a,b WHERE a.id=b.id GROUP BY a.id
```

In ANSI SQL you would have to add the customer.name in the GROUP BY for the following query. In MySQL the name redundant.

```
SELECT order.custid,customer.name,max(payments) from order,customer  
WHERE order.custid = customer.custid GROUP BY order.custid;
```

Note that you can't use expressions in the GROUP BY or ORDER BY clause. You can on the other hand use an alias on a expression and use this to solve the problem:

```
SELECT id,FLOOR(value/100) AS val FROM table_name GROUP BY id,val ORDER BY  
val
```

5.0 The DBI interface

5.1 The DBI Interface for Perl

Portable DBI methods.

connect	Establish a connection to a database server
prepare	Get a SQL statement ready for execution
do	Prepares and executes a SQL statement
disconnect	Disconnect from the database server
quote	Quote strings/blobs to be inserted
execute	Executes prepared statements
fetchrow_array	fetch the next row as an array of fields.
fetchrow_arrayref	fetch next row as a reference array of fields
fetchrow_hashref	fetch next row as a reference to a hashtable
fetchall_arrayref	Get all data as a array of arrays
finish	finish a statement and let the system free resources
rows	Returns the number of rows affected
data_sources	Return an array of databases available on localhost
ChopBlanks	Shall fetchrow trim spaces
NUM_OF_PARAMS	Number of placeholders in the prepared statement
NULLABLE	Which columns can be NULL

MySQL specific methods.

insertid	The latest auto_increment value
is_blob	Which column are BLOBs
is_key	Which columns are keys
is_num	Which columns are numeric
is_pri_key	Which columns are primary keys
is_not_null	Which columns can NOT be NULL. See NULLABLE
length	Maximum theoretically possible column sizes
max_length	Maximum physical present column sizes
NAME	Column names
NUM_OF_FIELDS	Number of fields returned.
table	Table names in returned set
type	All column types
_CreateDB	Create a database
_DropDB	Drop a database. THIS IS DANGEROUS

connect You use the connect method to make a database connection to the data source. The `$data_source` value should begin with `DBI:driver_name:`. Example connect methods with the `DBD::mysql` driver:

```
$dbh = DBI->connect("DBI:mysql:$database", $user, $password);  
$dbh = DBI->connect("DBI:mysql:$database:$hostname",  
$user, $password);  
$dbh = DBI->connect("DBI:mysql:$database:$hostname:$port",  
$user, $password);
```

If the username and/or password are undefined, then the DBI will use the values of the `DBI_USER`, `DBI_PASS` environment variables respectively. If you don't specify a hostname, then it will default to `"localhost"`. If you don't specify a port, then it defaults to the default mysql port (3306).

prepare Prepare gets a SQL statement ready for execution by the database engine and returns a statement handle (`$sth`) which invokes the `execute` method. Example:

```
$sth = $dbh->prepare($statement) or die "Can't prepare  
$statement:  
$dbh->errstr\n";
```

do The "do" method prepares and executes a SQL statement and returns the number of rows effected.

This method is generally used for "non-select" statements which can not be prepared in advance (driver limitation) or which do not need to be executed more than once (inserts, deletes, etc.).

Examples:

```
$rc = $dbh->do($statement) or die "Can't execute $statement:  
$dbh->errstr\n";
```

disconnect Disconnect will disconnect the database handle from the database. This is typically called right before you exit from the program.

Example:

```
$rc = $dbh->disconnect;
```

quote	<p>The quote method is used to "escape" any special characters contained in the string and to add the required outer quotation marks.</p> <pre>\$sql = \$dbh->quote(\$string)</pre>
execute	<p>This method executes the prepared statement. For non-select statements, it returns the number of rows affected. For select statements, execute only starts the SQL query in the database. You need to use one of the fetch_* methods below to retrieve the data. Example:</p> <pre>\$rv = \$sth->execute or die "can't execute the query: \$sth->errstr;</pre>
fetchrow_array	<p>This method "fetches" the next row of data and returns it as an array of field values. Example:</p> <pre>while(@row = \$sth->fetchrow_array) { print qw(\$row[0]\t\$row[1]\t\$row[2]\n); }</pre>
fetchrow_arrayref	<p>This method "fetches" the next row of data and returns it as a reference to an array of field values.</p> <p>Example:</p> <pre>while(\$row_ref = \$sth->fetchrow_arrayref) { print qw(\$row_ref->[0]\t\$row_ref->[1]\t\$row_ref->[2]\n); }</pre>
fetchrow_hashref	<p>This method fetches a row of data and returns a reference to a hash table containing field name/value pairs. This method is not nearly as efficient as using array references as demonstrated above.</p> <p>Example:</p> <pre>while(\$hash_ref = \$sth->fetchrow_hashref) { print qw(\$hash_ref->{firstname}\t\$hash_ref->{lastname}\t\ \$hash_ref->{title}\n); }</pre>

`fetchall_arrayref` This method is used to get all the data (rows) to be returned from the SQL statement. It returns a reference to an array of arrays of references to each row. You access/print the data by using a nested loop. Example:

```
my $table = $sth->fetchall_arrayref or die "$sth->errstr\n";
my($i, $j);
for $i ( 0 .. ${$table} ) {
    for $j ( 0 .. ${$table->[$i]} ) {
        print "$table->[$i][$j]\t";
    }
}
print "\n";
}
```

`finish` Indicates that no more data will be fetched from this statement handle. You call this method to free up the statement handle and any system resources it may be holding. Example:

```
$rc = $sth->finish;
```

`rows` Returns the number of rows affected (updated, deleted, etc.) from the last command. This is usually used after a `do()` or `non-select execute()` statement.

```
$rv = $sth->rows;
```

`NULLABLE` A reference to an array of boolean values; TRUE indicates that this column may contain NULLs.

```
$null_possible = $sth->{NULLABLE};
```

`NUM_OF_FIELDS` Number of fields returned by a `SELECT` or `LISTFIELDS` statement. You may use this for checking whether a statement returned a result: A zero value indicates a non-`SELECT` statement like `INSERT`, `DELETE` or `UPDATE`.

```
$nr_of_fields = $sth->{NUM_OF_FIELDS};
```

`data_sources` This method returns an array of databases available to the `mysql` daemon on `localhost`.

```
@dbs = DBI->data_sources("mysql");
```

ChopBlanks This determines whether a fetchrow will chop preceding and trailing blanks off the returned values.

```
$sth->{'ChopBlanks'} =1;
```

MySQL specific methods.

insertid If you use the auto-increment feature of mysql, the new auto-incremented values will be stored here.

```
$new_id = $sth->{insertid};
```

is_blob Reference to an array of boolean values; TRUE indicates that the respective column is a blob.

```
$keys = $sth->{is_blob};
```

is_key Reference to an array of boolean values; TRUE indicates, that the respective column is a key.

```
$keys = $sth->{is_key};
```

is_num Reference to an array of boolean values; TRUE indicates, that the respective column contains numeric values.

```
$nums = $sth->{is_num};
```

is_pri_key Reference to an array of boolean values; TRUE indicates, that the respective column is a primary key.

```
$pri_keys = $sth->{is_pri_key};
```

is_not_null A reference to an array of boolean values; FALSE indicates that this column may contain NULLs.

You should better use the NULLABLE attribute above which is a DBI standard.

```
$not_nulls = $sth->{is_not_null};
```

max_length length	A reference to an array of maximum column sizes. The max_length is the maximum physically present in the result table, length gives the theoretically possible maximum. <code>\$max_lengths = \$sth->{max_length};</code> <code>\$lengths = \$sth->{length};</code>
NAME	A reference to an array of column names. <code>\$names = \$sth->{NAME};</code>
table	Returns a reference to an array of table names. <code>\$tables = \$sth->{table};</code>

5.2 More DBI/DBD information

You can use the perldoc command to get more information about DBI.

```
perldoc DBI
perldoc DBI::FAQ
perldoc mysql
```

You can also use the pod2man, pod2html, etc.. tools to translate to other formats.