MySQL Internals Manual

Copyright © 1998-2004 MySQL AB

Copyright © 1998-2004 MySQL AB

1 Coding Guidelines

The purpose of this chapter is to establish a set of guidelines that will help you, the developer, to follow the established MySQL coding styles and standards when writing code. Consistent style is important as it makes it easier to maintain our code and allows you to find specific routines much more quickly.

The standards contained within this chapter apply to the mysql server, and do not necessarily apply to other projects such as JDBC, ODBC, Administrator, etc.

Indentation and Spacing

The following rules apply to indentation and spacing of code within a source file:

- Do not use tab characters (\t), use spaces. All indentation should be two spaces. You should be able to configure your editor to use spaces instead of tabs. (See the editor configuration tips at the end of this chapter for instruction for vim and emacs).
- Line breaks should occur on column 80, with two line breaks between each function in a file.
- Do not use carriage return (\r\n) characters in a source document; this can cause problems for other users and for builds.
- Matching '`{}`' (left and right braces) should be in the same column. With the exception of a `switch` statement, all braces should appear on their own line. The only exception to having braces on separate lines is if there is nothing between the braces.

  Examples:

  ```
  if (is_valid(x))
  {
    x= 2;
  }

  switch (my_arg) {

  for (...)
  {}
  ```
- Put a space after evaluation keywords `if`, `for` and `while`.
- When a function has multiple arguments, place a space after each comma ('`,`') in the argument set.

  ```
  return_value= my_function(arg1, arg2, arg3);
  ```
- When performing assignments, place a space after the equals sign but make sure there is no space between the variable and the equals sign. This allows for easier searching through the code for variable assignments; It is easier to search for an assignment to '`x`' when you can search for `x=` .

  When you make multiple assignments indent the right-hand values to make them easier to read:

  ```
  int x=          27;
  int new_var=    18;
  ```
- Operators should have spaces on both sides, including the '`==`' operator.

  ```
  x + y;
  if (x == y);
  ```
- Do not do multiple tasks on the same line, whether you are declaring variables or performing commands.

```
int x= 11;
int y= 12;
int z= 0;

z= x;
y += x;
```

- Do not put a space between a pointer asterisk and its variable.

```
int *var;
```

- Use blank lines to separate functions and blocks of code. Successive functions should have two newlines between them.

Naming Conventions

- Use `my_var` as opposed to `myVar` or `MyVar` (underscore rather than capitalization is to be used for separating words in identifiers).
- Avoid capitalization except for class names; class names should begin with a capital letter.
- Don't have function names, structure elements, or variables that begin or end with '`_`', these make your code less readable.
- Use long function and variable names in English. This will make your code easier to read for all developers.
- Structure types are `typedef`'ed to an all-upper-case identifier.
- All `#define` declarations should be in upper case.

```
#define MY_CONSTANT 15
```

- Enumeration names should begin with `enum_`.

Commenting Code

- Comment your code when you do something that someone else may think is not "trivial".
- When writing multi-line comments, the beginning and ending markers should be on their own lines.

  In addition, do not use asterixes on left of the comment.

```
/*
  This is how
  a multi-line comment
  should look.
*/
```

- `//` comments are allowed at end of lines. In other cases, use `/* */` comments. In C files or headers used by C files, one should not use `//` comments.
- Function comments are important! When commenting a function, note the IN parameters (the word IN is implicit).

  Every function should have a description unless the function is very short and its purpose is obvious.
- All comments should be in English.
- Place a blank line between a function and its description.

```
/*
  This is a function.
  Use it wisely.
*/

int my_function()
```

General Development Guidelines

- We use BitKeeper for source management. Bitkeeper can be downloaded from `http://www.bitmover.com/cgi-bin/download.cgi`
- You should use the MySQL 5.0 source for all new developments. The public 5.0 development branch can be downloaded with `shell> bk clone bk://mysql.bkbits.net/mysql-5.0 mysql-5.0`
- If you have any questions about the MySQL source, you can post them to `internals@lists.mysql.com` and we will answer them.
- Before making big design decisions, please begin by posting a summary of what you want to do, why you want to do it, and how you plan to do it. This way we can easily provide you with feedback and also discuss it thoroughly. Perhaps another developer can assist you.
- Try to write code in a lot of black boxes that can be reused or at least use a clean, easy to change interface.
- Reuse code; There are already many algorithms in MySQL that can be reused for list handling, queues, dynamic and hashed arrays, sorting, etc.
- Use the `my_*` functions like `my_read()`/`my_write()`/ `my_malloc()` that you can find in the `mysys` library, instead of the direct system calls; This will make your code easier to debug and more portable.
- Use `libstring` functions (in the '`strings`' directory) instead of standard `libc` string functions whenever possible. For example, use `bfill()` and `bzero()` instead of `memset()`.
- Try to always write optimized code, so that you don't have to go back and rewrite it a couple of months later. It's better to spend 3 times as much time designing and writing an optimal function than having to do it all over again later on.
- Avoid CPU wasteful code, even when its use is trivial, to avoid developing sloppy coding habits.
- If you can do something in fewer lines, please do so (as long as the code will not be slower or much harder to read).
- Do not check the same pointer for `NULL` more than once.
- Never use a macro when an (inline) function would work as well.
- Do not make a function inline if you don't have a very good reason for it. In many cases, the extra code that is generated is more likely to slow down the resulting code than give a speed increase because the bigger code will cause more data fetches and instruction misses in the processor cache.

  The cases where you should use inline functions are when the function satisfies most of the following requirements:

  - Function is very short (just a few lines)
  - Function is used in a speed critical place and is executed over and over again.
  - Function is handling the normal case, not some extra functionallity that most users will not use.
  - The function is only called in very few cases. (This restriction must be followed unless the resulting assembler code of the inlined function < 16 assembler instructions).
  - If the compiler can do additional optimizations by inlining it and the resulting function will be only a fraction of the original one.
- Think assembly - make it easier for the compiler to optimize your code.
- Avoid using `malloc()`, which is very slow. For memory allocations that only need to live for the lifetime of one thread, use `sql_alloc()` instead.
- Functions return zero on success, and non-zero on error, so you can do:

```
if(a() || b() || c()) { error("something went wrong"); }
```

- Using `goto` is okay if not abused.
- Avoid default variable initializations. Use `LINT_INIT()` if the compiler complains after making sure that there is really no way the variable can be used uninitialized.
- Use `TRUE` and `FALSE` instead of `true` and `false` in C++ code. This makes the code more readable and makes it easier to use it later in a C library, if needed.
- `bool` exists only in C++. In C, you have to use `my_bool` (which is `char`); it has different cast rules than `bool`:

  ```
  int c= 256*2;
  bool a= c; /* a gets 'true' */
  my_bool b= c; /* b gets zero, i.e. 'false': BAD */
  my_bool b= test(c); /* b gets 'true': GOOD */
  ```

  In C++, use `bool`, unless the variable is used in C code (for example the variable is passed to a C function).
- Do not instantiate a class if you do not have to.
- Use pointers rather than array indexing when operating on strings.
- Never pass parameters with the `&variable_name` construct in C++. Alway use a pointer instead!

  The reason is that the above makes it much harder for the one reading the caller function code to know what is happening and what kind of code the compiler is generating for the call.
- Format function arguments in the following way:

  ```
  Return_value_type *Class_name::method_name(const char *arg1,
                                             size_t arg2, Type *arg3)

  return_value= function_name(argument1, argument2, long_argument3,
                              argument4,
                              function_name2(long_argument5,
                                             long_argument6));
  ```

  If function or argument names are too long:

  ```
  return_value=
    long_long_function_name(long_long_argument1, long_long_argument2,
                            long_long_long_argument3,
                            long_long_argument4,
                            long_function_name2(long_long_argument5,
                                                long_long_argument6));

  Long_long_return_value_type *
  Long_long_class_name::
  long_long_method_name(const char *long_long_arg1, size_t long_long_arg2,
                        Long_long_type *arg3)
  ```

  (you can but don't have to split `Class_name::method_name` into two lines)

  In such cases think also about renaming arguments.
- Format constructors in the following way:

  ```
  Item::Item(int a_arg, int b_arg, int c_arg)
    :a(a_arg), b(b_arg), c(c_arg)
  {}
  ```

  But keep lines short to make them more readable:

```
Item::Item(int longer_arg, int more_longer_arg)
  :longer(longer_arg),
  more_longer(more_longer_arg)
{}
```

If constructor can fit into one line:

```
Item::Item(int a_arg) :a(a_arg) {}
```

- You can align variable declarations and assignments, like this:

```
Type      value;
int       var2;
ulonglong var3;
```

Always align assignments of one structure to another, like this:

```
foo->member=      bar->member;
foo->name=        bar->name;
foo->name_length= bar->name_length;
```

When you write the assignments that way, you typically don't read the structures names more than once.

Suggested mode in emacs:

```
(require 'font-lock)
(require 'cc-mode)
(setq global-font-lock-mode t) ;;colors in all buffers that support it
(setq font-lock-maximum-decoration t) ;;maximum color
(c-add-style "MY"
 '("K&R"
   '("MY"
     (c-basic-offset . 2)
     (c-comment-only-line-offset . 0)
     (c-offsets-alist . ((statement-block-intro . +)
                         (knr-argdecl-intro . 0)
                         (substatement-open . 0)
                         (label . -)
                         (statement-cont . +)
                         (arglist-intro . c-lineup-arglist-intro-after-paren)
                         (arglist-close . c-lineup-arglist)
                         ))
     )))

(setq c-mode-common-hook '(lambda ()
                            (c-set-style "MY")
                            (setq tab-width 8)
                            (setq indent-tabs-mode t)
                            (setq comment-column 48)))

(c-set-style "MY")
(setq c-default-style "MY")
```

Basic vim setup:

```
set tabstop=8
set shiftwidth=2
set backspace=2
set softtabstop
```

```
set smartindent
set cindent
set cinoptions=g0:0t0c2C1(0f0l1
"set expandtab "uncomment if you don't want to use tabstops
```

Another `vim` setup:

```
set tabstop=8
set shiftwidth=2
set bs=2
set et
set sts=2
set tw=78
set formatoptions=cqroa1
set cinoptions=g0:0t0c2C1(0f0l1
set cindent

function InsertShiftTabWrapper()
  let num_spaces = 48 - virtcol('.')
  let line = ' '
  while (num_spaces > 0)
    let line = line . ' '
    let num_spaces = num_spaces - 1
  endwhile
  return line
endfunction
" jump to 48th column by Shift-Tab - to place a comment there
inoremap <S-tab> <c-r>=InsertShiftTabWrapper()<cr>
" highlight trailing spaces as errors
let c_space_errors=1
```

DBUG Tags

Here are some of the DBUG tags we now use:

`enter` Arguments to the function.

`exit` Results from the function.

`info` Something that may be interesting.

`warning` When something doesn't go the usual route or may be wrong.

`error` When something went wrong.

`loop` Write in a loop, that is probably only useful when debugging the loop. These should normally be deleted when you are satisfied with the code and it has been in real use for a while.

Some tags specific to `mysqld`, because we want to watch these carefully:

`trans` Starting/stopping transactions.

`quit` `info` when `mysqld` is preparing to die.

`query` Print query.

2 The Optimizer

Definitions

This description uses a narrow definition: The OPTIMISER is the set of routines which decide what execution path the DBMS should take for queries.

MySQL changes these routines frequently, so you should compare what is said here with what's in the current source code. To make that easy, this description includes notes referring to the relevant routine, for example "See: '/sql/select_cc', optimize_cond()".

When one query is changed into another query which delivers the same result, that is a TRANSFORMATION. For example, the DBMS could change

```
SELECT ... WHERE 5 = a
```

to

```
SELECT ...WHERE a = 5
```

Most transformations are less obvious. Some transformations result in faster execution.

The Code

Here is a diagram showing the code structure of `handle_select()` in '/sql/sql_select.cc', the server code that handles a query:

```
handle_select()
   mysql_select()
     JOIN::prepare()
       setup_fields()
     JOIN::optimize()            /* optimizer is from here ... */
       optimize_cond()
       opt_sum_query()
       make_join_statistics()
         get_quick_record_count()
         choose_plan()
           /* Find the best way to access tables */
           /* as specified by the user.          */
           optimize_straight_join()
             best_access_path()
           /* Find a (sub-)optimal plan among all or subset */
           /* of all possible query plans where the user    */
           /* controlls the exhaustiveness of the search.   */
           greedy_search()
             best_extension_by_limited_search()
               best_access_path()
           /* Perform an exhaustive search for an optimal plan */
           find_best()
       make_join_select()        /* ... to here */
     JOIN::exec()
```

The indentation in the diagram shows what calls what. Thus you can see that `handle_select()` calls `mysql_select()` which calls `JOIN::prepare()` which calls `setup_fields()`, and so on. The first part of `mysql_select()` is `JOIN::prepare()` which is for context analysis, metadata setup, and some subquery transformations. The optimizer is `JOIN::optimize()` and all its subordinate routines. When the optimizer finishes, `JOIN::exec()` takes over and does the job that `JOIN::optimize()` decides upon.

Although the word "JOIN" appears, these optimizer routines are for all query types.

The `optimize_cond()` and `opt_sum_query()` routines do transformations. The `make_join_statistics()` routine puts together all the information it can find about indexes that might be useful for accessing the query's tables.

Constant Propagation

A transformation takes place for expressions like this:

```
WHERE column1 = column2 AND column2 = 'x'
```

For such expressions, since it is known that "if A=B and B=C then A=C" (the Transitivity Law), the transformed condition becomes:

```
WHERE column1='x' AND column2='x'
```

This transformation occurs for `column1 <operator> column2` conditions if and only if `<operator>` is one of these operators:

```
=, <, >, <=, >=, <>, <=>, LIKE
```

That is, transitive transformations don't apply for `BETWEEN`. Probably they should not apply for `LIKE` either, but that's a story for another day.

Constant propagation happens in a loop, so the output from one "propagation step" can be input for the next step.

See: '`/sql/sql_select.cc`', `change_cond_ref_to_const()`. Or see: '`/sql/sql_select.cc`', `propagate_cond_constants()`.

Dead Code Elimination

A transformation takes place for always-true conditions:

```
WHERE 0=0 AND column1='y'
```

The first condition is always true, so it is removed, leaving:

```
WHERE column1='y'
```

See: '`/sql/sql_select.cc`', `remove_eq_conds()`.

A transformation takes place for always-false conditions:

```
WHERE (0 = 1 AND s1 = 5) OR s1 = 7
```

The parenthesized part is always false, so it is removed, reducing the expression above to:

```
WHERE s1 = 7
```

Sometimes the optimizer might eliminate the whole `WHERE` clause:

```
WHERE (0 = 1 AND s1 = 5)
```

The `EXPLAIN` statement will show the words `Impossible WHERE`. Informally, we at MySQL say: "The `WHERE` has been optimized away."

If a column cannot be `NULL`, the optimizer removes any non-relevant `IS NULL` conditions. Thus,

```
WHERE not_null_column IS NULL
```

is an always-false situation, and

```
WHERE not_null_column IS NOT NULL
```

is an always-true situation — so such columns are also eliminated from the conditional expression. This can be tricky. For example, in an `OUTER JOIN`, a column which is defined as `NOT NULL` might still contain a `NULL`. The optimizer leaves `IS NULL` conditions alone in such exceptional situations.

The optimizer will not detect all possible `impossible WHERE` situations — there are too many. For example:

```
CREATE TABLE Table1 (column1 CHAR(1));
...
SELECT * FROM Table1 WHERE column1 = 'Canada';
```

The optimizer will not eliminate the condition in the query, even though the `CREATE TABLE` definition makes it an impossible condition.

Constant Folding

A transformation takes place for this expression:

```
WHERE column1 = 1 + 2
```

which becomes:

```
WHERE column1 = 3
```

Before you say “but I never would write 1 + 2 in the first place” — remember what was said earlier about constant propagation. It is quite easy for the optimizer to put such expressions together. This process simplifies the result.

Constants and Constant Tables

A MySQL “constant” is something more than a mere literal in the query. It can also be the contents of a “constant table,” which is defined as follows:

1. A table with zero rows, or with only one row
2. A table expression that is restricted with a `WHERE` condition, containing expressions of the form `column = constant`, for all the columns of the table's `PRIMARY KEY`, or for all the columns of any of the table's `UNIQUE` keys (provided that the `UNIQUE` columns are also defined as `NOT NULL`).

For example, if the table definition for `Table0` contains

```
... PRIMARY KEY (column1,column2)
```

then this expression

```
FROM Table0 ... WHERE column1=5 AND column2=7 ...
```

returns a constant table. More simply, if the table definition for `Table1` contains

```
... unique_not_null_column INT NOT NULL UNIQUE
```

then this expression

```
FROM Table1 ... WHERE unique_not_null_column=5
```

returns a constant table.

These rules mean that a constant table has at most one row value. MySQL will evaluate a constant table in advance, to find out what that value is. Then MySQL will “plug” that value into the query. Here's an example:

```
SELECT Table1.unique_not_null_column, Table2.any_column
    FROM Table1, Table2
    WHERE Table1.unique_not_null_column = Table2.any_column
    AND Table1.unique_not_null_column = 5;
```

When evaluating this query, MySQL first finds that table `Table1` — after restriction with `Table1.unique_not_null_column` — is a constant table according to the second definition above. So it retrieves that value.

If the retrieval fails (there is no row in the table with `unique_not_null_column` = 5), then the constant table has zero rows and you will see this message if you run `EXPLAIN` for the statement:

```
Impossible WHERE noticed after reading const tables
```

Alternatively, if the retrieval succeeds (there is exactly one row in the table with `unique_not_null_column` = 5), then the constant table has one row and MySQL transforms the query to this:

```
SELECT 5, Table2.any_column
    FROM Table1, Table2
    WHERE 5 = Table2.any_column
    AND 5 = 5;
```

Actually this is a grand-combination example. The optimizer does some of the transformation because of constant propagation, which we described earlier. By the way, we described constant propagation first because it happens happens *before* MySQL figures out what the constant tables are. The sequence of optimizer steps sometimes makes a difference.

Although many queries have no constant-table references, it should be kept in mind that whenever the word "constant" is mentioned hereafter, it refers either to a literal or to the contents of a constant table.

See: '/sql/sql_select.cc', make_join_statistics().

Join Type

When evaluating a conditional expression, MySQL decides what "join type" the expression has. (Again: despite the word "join," this applies for all conditional expressions, not just join expressions. A term like "access type" would be clearer.) These are the documented join types, in order from best to worst:

```
system          ... a system table which is a constant table
const           ... a constant table
eq_ref          ... unique/primary index with '=' for joining
ref             ... index with '='
ref_or_null     ... index with '=', possibly NULL
range           ... index with BETWEEN, IN, >=, LIKE, etc.
index           ... sequential scan of index
ALL             ... sequential scan of table
```

See: '/sql/sql_select.h', enum join_type{}. Notice that there are a few other (undocumented) join types too, for subqueries.

The optimizer can use the join type to pick a "driver expression." For example, consider this query:

```
SELECT *
FROM Table1
WHERE indexed_column = 5 AND unindexed_column = 6
```

Since indexed_column has a better join type, it is more likely to be the driver. You'll see various exceptions as this description proceeds, but this is a simple first rule.

What is significant about a driver? Consider that there are two execution paths for the query:

(The Bad Execution Path) Read every row in the table. (This is called a "sequential scan of Table1" or just "table scan.") For each row, examine the values in indexed_column and in unindexed_column, to see if they meet the conditions.

(The Good Execution Plan) Via the index, look up the rows which have indexed_column = 5. (This is called an "indexed search.") For each row, examine the value in unindexed_column to see if it meets the condition.

An indexed search generally involves fewer accesses than a sequential scan, and far fewer accesses if the table is large but the index is UNIQUE. That is why it is better to access with "The Good Execution Plan," and that is why it is often good to choose indexed_column as the driver.

The 'range' Join Type

Some conditions can work with indexes, but over a (possibly wide) range of keys. These are known as "range" conditions, and are most often encountered with expressions involving these operators: >, >=, <, <=, IN, LIKE, BETWEEN

To the optimizer, this expression:

```
column1 IN (1,2,3)
```

is the same as this one:

```
column1 = 1 OR column1 = 2 OR column1 = 3
```

and MySQL treats them the same — there is no need to change IN to OR for a query, or vice versa.

The optimizer will use an index (range search) for

```
column1 LIKE 'x%'
```

but not for

```
column1 LIKE '%x'
```

That is, there is no range search if the first character in the pattern is a wildcard.

To the optimizer,

```
column1 BETWEEN 5 AND 7
```

is the same as this expression

```
column1 >= 5 AND column1 <= 7
```

and again, MySQL treats both expressions the same.

The optimizer may change a `Range` to an `ALL` join type if a condition would examine too many index keys. Such a change is particularly likely for `<` and `>` conditions and multiple-level secondary indexes. See: (for `MyISAM` indexes) '`/myisam/mi_range.c`', `mi_records_in_range()`.

The `index` Join Type

Consider this query:

```
SELECT column1 FROM Table1;
```

If `column1` is indexed, then the optimizer may choose to retrieve the values from the index rather than from the table. An index which is used this way is called a "covering index" in most texts. MySQL just uses the word "index" in `EXPLAIN` descriptions.

For this query:

```
SELECT column1, column2 FROM Table1;
```

the optimizer will use "join type = `index`" only if the index has this definition:

```
CREATE INDEX ... ON Table1 (column1, column2);
```

In other words, all columns in the select list must be in the index. (The order of the columns in the index does not matter.) Thus it might make sense to define a multiple-column index strictly for use as a covering index, regardless of search considerations.

Transposition

MySQL supports transpositions (reversing the order of operands around a relational operator) for simple expressions only. In other words:

```
WHERE - 5 = column1
```

becomes:

```
WHERE column1 = -5
```

However, MySQL does not support transpositions where arithmetic exists. Thus:

```
WHERE 5 = -column1
```

is not treated the same as:

```
WHERE column1 = -5
```

Transpositions to expressions of the form <column>=<constant> are ideal for index lookups. If an expression of this form refers to an indexed column, then MySQL always uses the index, regardless of the table size. (Exception: if the table has only zero rows or only one row, it is a

constant table and receives special treatment. See the earlier section "Constants and Constant Tables".)

AND

The ANDed search has the form `<condition> AND <condition>`, as in:

```
WHERE column1 = 'x' AND column2 = 'y'
```

Here the optimizer's decision is:

1. If (neither condition is indexed) use sequential scan.
2. Otherwise, if (one condition has better join type) then pick a driver based on join type (see the earlier section "Join Type").
3. Otherwise, since (both conditions are indexed and have equal join type) pick a driver based on the first index that was created.

Here's an example:

```
CREATE TABLE Table1 (s1 INT, s2 INT);
CREATE INDEX Index1 ON Table1 (s2);
CREATE INDEX Index2 ON Table1 (s1);
...
SELECT * FROM Table1 WHERE s1 = 5 AND s2 = 5;
```

When choosing a strategy to solve this query, the optimizer picks `s2 = 5` as the driver because the index for `s2` was created first. Regard this as an accidental effect rather than a rule — it could change at any moment.

OR

The ORed search has the form "<condition> OR <condition>" as in:

```
WHERE column1 = 'x' OR column2 = 'y'
```

Here the optimizer's decision is:

```
Use a sequential scan.
```

In theory there is another choice if both column1 and column2 are indexed:

```
index search for the first condition,
index search for the second condition,
merge the two result sets
```

MySQL never does that. But MySQL will do that in future, the code for doing so already exists.

The above warning does not apply if the same column is used in both conditions. For example:

```
WHERE column1 = 'x' OR column1 = 'y'
```

In such a case, the search is indexed because the expression is a range search. This subject will be revisited during the discussion of the `IN` predicate.

AND plus OR

Consider this search expression:

```
WHERE column1 = 5 AND (column2 = 5 OR column3 = 5)
```

If `column1` is indexed, then the optimizer will choose to use the index. In other words, the "sequential scan if `OR`" rule does not apply if the `OR` is subordinate to an `AND`.

UNION

All `SELECT` statements within a `UNION` are optimized separately. Therefore, for this query:

```
SELECT * FROM Table1 WHERE column1 = 'x'
UNION ALL
SELECT * FROM TABLE1 WHERE column2 = 'y'
```

if both `column1` and `column2` are indexed, then each `SELECT` is done using an indexed search, and the result sets are merged. Notice that this query might produce the same results as the query used in the `OR` example, which uses a sequential scan.

NOT, <>

It is a logical rule that

```
column1 <> 5
```

is the same as

```
column1 < 5 OR column1 > 5
```

However, MySQL does not transform in this circumstance. If you think that a range search would be better, then you should do your own transforming in such cases.

It is also a logical rule that

```
WHERE NOT (column1 != 5)
```

is the same as

```
WHERE column1 = 5
```

However, MySQL does not transform in this circumstance either.

We expect to add optimizations soon for both the above cases.

ORDER BY

In general, the optimizer will skip the sort procedure for the `ORDER BY` clause if it sees that the rows will be in order anyway. But let's examine some exceptional situations.

For the query:

```
SELECT column1 FROM Table1 ORDER BY 'x';
```

the optimizer will throw out the `ORDER BY` clause. This is another example of dead code elimination.

For the query:

```
SELECT column1 FROM Table1 ORDER BY column1;
```

the optimizer will use an index on `column1`, if it exists.

For the query:

```
SELECT column1 FROM Table1 ORDER BY column1+1;
```

the optimizer will use an index on `column1`, if it exists. But don't let that fool you! The index is only for finding the values. (It's cheaper to do a sequential scan of the index than a sequential scan of the table, that's why `index` is a better join type than `ALL` — see "The 'index' Join Type" section, earlier.) There will still be a full sort of the results.

For the query:

```
SELECT * FROM Table1
WHERE column1 > 'x' AND column2 > 'x'
ORDER BY column2;
```

if both `column1` and `column2` are indexed, the optimizer will choose an index on ... `column1`. The fact that ordering takes place by `column2` values does not affect the choice of driver in this case.

See: '`/sql/sql_select.cc`', `test_if_order_by_key()`, and '`/sql/sql_select.cc`', `test_if_skip_sort_order()`.

There is a description of the internal sort procedure in the MySQL Reference Manual, in section 5.2.8 "How MySQL optimizes `ORDER BY`." We will not repeat it here, but urge you to read it because it includes a description of how the buffering and the quicksort operate.

See: '`/sql/sql_select.cc`', `create_sort_index()`.

GROUP BY

These are the main optimizations that take place for GROUP BY and related items (HAVING, COUNT(), MAX(), MIN(), SUM(), AVG(), DISTINCT()).

- GROUP BY will use an index, if one exists.
- GROUP BY will use sorting, if there is no index. The optimizer may choose to use a hash table.
- For the case GROUP BY x ORDER BY x, the optimizer will realize that the ORDER BY is unnecessary, because the GROUP BY comes out in order by x.
- The optimizer contains code for shifting certain HAVING conditions to the WHERE clause; however, this code is not operative at time of writing. See: '/sql/sql_select.cc', JOIN::optimize(), after #ifdef HAVE_REF_TO_FIELDS.
- If the table handler has a quick row-count available, then the query

  ```
  SELECT COUNT(*) FROM Table1;
  ```

  gets the count without going through all the rows. This is true for MyISAM tables, but not for InnoDB tables. Note that the query

  ```
  SELECT COUNT(column1) FROM Table1;
  ```

  is not subject to the same optimization, unless column1 is defined as NOT NULL.
- New optimizations exist for MAX() and MIN(). For example, consider the query

  ```
  SELECT MAX(column1)
    FROM Table1
    WHERE column1 < 'a';
  ```

  If column1 is indexed, then it's easy to find the highest value by looking for 'a' in the index and going back to the key before that.
- The optimizer transforms queries of the form

  ```
  SELECT DISTINCT column1 FROM Table1;
  ```

  to

  ```
  SELECT column1 FROM Table1 GROUP BY column1;
  ```

  if and only if both of these conditions are true:
  - The GROUP BY can be done with an index. (This implies that there is only one table in the FROM clause, and no WHERE clause.)
  - There is no LIMIT clause.

  Because DISTINCT is not always transformed to GROUP BY, do not expect that queries with DISTINCT will always cause ordered result sets. (You can, however, rely on that rule with GROUP BY, unless the query includes ORDER BY NULL.)

See: '/sql/sql_select.cc', opt_sum_query(), and '/sql/sql_select.cc', remove_duplicates().

JOIN

Bad join choices can cause more damage than bad choices in single-table searches, so MySQL developers have spent proportionally more time making sure that the tables in a query are joined in an optimal order and that optimal access methods (often called "access paths") are chosen to retrieve table data. A combination of a fixed order in which tables are joined and the corresponding table access methods for each table is called "query execution plan" (QEP). The goal of the query optimizer is to find an optimal QEP among all possible such plans. There are several general ideas behind join optimization.

Each plan (or part of plan) is assigned a "cost." The cost of a plan reflects roughly the resources needed to compute a query according to the plan, where the main factor is the number of rows

that will be accessed while computing a query. Once we have a way to assign costs to different QEPs we have a way to compare them. Thus, the goal of the optimizer is to find a QEP with minimal cost among all possible plans.

In MySQL, the search for an optimal QEP is performed in a bottom-up manner. The optimizer first considers all plans for one table, then all plans for two tables, and so on, until it builds a complete optimal QEP. Query plans that consist of only some of the tables (and predicates) in a query are called “partial plans.” The optimizer relies on the fact that the more tables are added to a partial plan, the greater its cost. This allows the optimizer to expand with more tables only the partial plans with lower cost than the current best complete plan.

The key routine that performs the search for an optimal QEP is ‘`sql/sql_select.cc`’, `find_best()`. It performs an exhaustive search of all possible plans and thus guarantees it will find an optimal one.

Below we represent `find_best()` in an extremely free translation to pseudocode. It is recursive, so some input variables are labeled “so far” to indicate that they come from a previous iteration.

```
remaining_tables = {t1, ..., tn}; /* all tables referenced in a query */

procedure find_best(
   partial_plan in,      /* in, partial plan of tables-joined-so-far */
   partial_plan_cost,    /* in, cost of partial_plan */
   remaining_tables,     /* in, set of tables not referenced in partial_plan */
   best_plan_so_far,     /* in/out, best plan found so far */
   best_plan_so_far_cost)/* in/out, cost of best_plan_so_far */
{
   for each table T from remaining_tables
   {
     /* Calculate the cost of using table T. Factors that the
        optimizer takes into account may include:
          Many rows in table (bad)
          Many key parts in common with tables so far (very good)
          Restriction mentioned in the WHERE clause (good)
          Long key (good)
          Unique or primary key (good)
          Full-text key (bad)
        Other factors that may at some time be worth considering:
          Many columns in key
          Short average/maximum key length
          Small table file
          Few levels in index
          All ORDER BY / GROUP columns come from this table */
     cost = complex-series-of-calculations;
     /* Add the cost to the cost so far. */
     partial_plan_cost+= cost;

     if (partial_plan_cost >= best_plan_so_far_cost)
       /* partial_plan_cost already too great, stop search */
       continue;

     partial_plan= expand partial_plan by best_access_method;
     remaining_tables= remaining_tables - table T;
     if (remaining_tables is not an empty set)
     {
```

```
            find_best(partial_plan, partial_plan_cost,
                      remaining_tables,
                      best_plan_so_far, best_plan_so_far_cost);
          }
          else
          {
            best_plan_so_far_cost= partial_plan_cost;
            best_plan_so_far= partial_plan;
          }
        }
      }
```

Here the optimizer applies a "depth-first search algorithm." It tries estimates for every table in the `FROM` clause. It will stop a search early if the estimate becomes worse than the best estimate so far. The order of scanning will depend on the order that the tables appear in the `FROM` clause.

See: '`/sql/table.h`', `struct st_table`.

`ANALYZE TABLE` may affect some of the factors that the optimizer considers.

See also: '`/sql/sql_sqlect.cc`', `make_join_statistics()`.

The straightforward use of `find_best()` and `greedy_search()` will not apply for `LEFT JOIN` or `RIGHT JOIN`. For example, starting with MySQL 4.0.14, the optimizer may change a left join to a straight join and swap the table order in some cases. See also "5.2.7 How MySQL Optimises `LEFT JOIN` and `RIGHT JOIN`" in the MySQL Reference Manual.

2.1 The `Index Merge` Join Type

2.1.1 Overview

`Index Merge` is used when table condition can be converted to form:

```
cond_1 OR cond_2 ... OR cond_N
```

The conditions for conversion are that each `cond_i` can be used for a range scan, and no pair (`cond_i`, `cond_j`) uses the same index. (If `cond_i` and `cond_j` use the same index, then `cond_i OR cond_j` can be combined into a single range scan and no merging is necessary.)

For example, `Index Merge` can be used for the following queries:

```
SELECT * FROM t WHERE key1=c1 OR key2<c2 OR key3 IN (c3,c4);

SELECT * FROM t WHERE (key1=c1 OR key2<c2) AND nonkey=c3;
```

`Index Merge` is implemented as a "container" for range key scans constructed from `cond_i` conditions. When doing `Index Merge`, MySQL retrieves rows for each of the keyscans and then runs them through a duplicate elimination procedure. Currently the `Unique` class is used for duplicate elimination.

2.1.2 Index Merge Optimizer

2.1.2.1 Range Optimizer

For `Range`-type queries, the MySQL optimizer builds a `SEL_TREE` object which represents a condition in this form:

```
range_cond = (cond_key_1 AND cond_key_2 AND ... AND cond_key_N)
```

Each of cond_key_i is a condition that refers to components of one key. MySQL creates a cond_key_i condition for each of the usable keys. Then the cheapest condition cond_key_i is used for doing range scan.

A single cond_key_i condition is represented by a pointer-linked network of SEL_ARG objects. Each SEL_ARG object refers to particular part of the key and represents the following condition:

```
sel_arg_cond= (inf_val < key_part_n AND key_part_n < sup_val) (1)
              AND next_key_part_sel_arg_cond                  (2)
              OR left_sel_arg_cond                            (3)
              OR right_sel_arg_cond                           (4)
```

1. is for an interval, possibly without upper or lower bound, either including or not including boundary values.
2. is for a SEL_ARG object with condition on next key component.
3. is for a SEL_ARG object with interval on the same field as this SEL_ARG object. Intervals of current and "left" object are disjoint and left_sel_arg_cond.sup_val <= inf_val.
4. is for a SEL_ARG object with interval on the same field as this SEL_ARG object. Intervals of current and "right" object are disjoint and left_sel_arg_cond.min_val >= max_val.

MySQL is able to convert arbitrary-depth nested AND-OR conditions to the above conjunctive form.

2.1.2.2 Index Merge Optimizer

A single SEL_TREE object cannot be constructed for conditions that have different members of keys in the OR clause, like in condition:

```
key1 < c1 OR key2 < c2
```

Beginning with MySQL 5.0, these conditions are handled with the Index Merge method, and its range optimizer structure, class SEL_IMERGE. SEL_IMERGE represents a disjunction of several SEL_TREE objects, which can be expressed as:

```
sel_imerge_cond = (t_1 OR t_1 OR ... OR t_n)
```

where each of t_i stands for a SEL_TREE object, and no pair (t_i, t_j) of distinct SEL_TREE objects can be combined into single SEL_TREE object.

The current implementation builds SEL_IMERGE only if no single SEL_TREE object can be built for the part of the query condition it has analyzed, and discards SEL_TREE immediately if it discovers that a single SEL_TREE object can be constructed. This is actually a limitation, and can cause worse row retrieval strategy to be used. E.g. for query:

```
SELECT * FROM t WHERE (goodkey1=c1 OR goodkey1=c2) AND badkey=c3
```

scan on badkey will be chosen even if Index Merge on (goodkey1, goodkey) would be faster.

The Index Merge optimizer collects a list of possible ways to access rows with Index Merge. This list of SEL_IMERGE structures represents the following condition:

```
(t_11 OR t_12 OR ... OR t_1k) AND
(t_21 OR t_22 OR ... OR t_2l) AND
 ...                          AND
(t_M1 OR t_M2 OR ... OR t_mp)
```

where t_ij is one SEL_TREE and one line is for one SEL_IMERGE object.

The SEL_IMERGE object with *minimal* cost is used for row retrieval.

In 'sql/opt_range.cc', see imerge_list_and_list(), imerge_list_or_list(), and SEL_IMERGE class member functions for more details of Index Merge construction.

See the `get_index_merge_params` function in the same file for `Index Merge` cost calculation algorithm.

2.1.3 Row Retrieval Algorithm

`Index Merge` works in two steps:

Preparation step:

```
activate 'index only';
foreach key_i in (key_scans \ clustered_pk_scan)
{
  while (retrieve next (key, rowid) pair from key_i)
  {
    if (no clustered PK scan ||
        row doesn't match clustered PK scan condition)
      put rowid into Unique;
  }
}
deactivate 'index only';
```

Row retrieval step:

```
for each rowid in Unique
{
  retrieve row and pass it to output;
}
if (clustered_pk_scan)
{
  while (retrieve next row for clustered_pk_scan)
   pass row to output;
}
```

See: '`sql/opt_range.cc`', `QUICK_INDEX_MERGE_SELECT` class members for `Index Merge` row retrieval code.

3 Important Algorithms and Structures

MySQL uses many different algorithms and structures. This chapter tries to describe some of them.

3.1 How MySQL Does Sorting (`filesort`)

In those cases where MySQL must sort the result, it uses the following `filesort` algorithm before MySQL 4.1:

1. Read all rows according to key or by table scanning. Rows that don't match the `WHERE` clause are skipped.
2. For each row, store a pair of values in a buffer (the sort key and the row pointer). The size of the buffer is the value of the `sort_buffer_size` system variable.
3. When the buffer gets full, run a qsort (quicksort) on it and store the result in a temporary file. Save a pointer to the sorted block. (If all pairs fit into the sort buffer, no temporary file is created.)
4. Repeat the preceding steps until all rows have been read.
5. Do a multi-merge of up to `MERGEBUFF` (7) regions to one block in another temporary file. Repeat until all blocks from the first file are in the second file.
6. Repeat the following until there are fewer than `MERGEBUFF2` (15) blocks left.
7. On the last multi-merge, only the pointer to the row (the last part of the sort key) is written to a result file.
8. Read the rows in sorted order by using the row pointers in the result file. To optimize this, we read in a big block of row pointers, sort them, and use them to read the rows in sorted order into a row buffer. The size of the buffer is the value of the `read_rnd_buffer_size` system variable. The code for this step is in the '`sql/records.cc`' source file.

One problem with this approach is that it reads rows twice: One time when evaluating the `WHERE` clause, and again after sorting the pair values. And even if the rows were accessed successively the first time (for example, if a table scan is done), the second time they are accessed randomly. (The sort keys are ordered, but the row positions are not.)

In MySQL 4.1 and up, a `filesort` optimization is used that records not only the sort key value and row position, but also the columns required for the query. This avoids reading the rows twice. The modified `filesort` algorithm works like this:

1. Read the rows that match the `WHERE` clause, as before.
2. For each row, record a tuple of values consisting of the sort key value and row position, and also the columns required for the query.
3. Sort the tuples by sort key value
4. Retrieve the rows in sorted order, but read the required columns directly from the sorted tuples rather than by accessing the table a second time.

Using the modified `filesort` algorithm, the tuples are longer than the pairs used in the original method, and fewer of them fit in the sort buffer (the size of which is given by `sort_buffer_size`). As a result, it is possible for the extra I/O to make the modified approach slower, not faster. To avoid a slowdown, the optimization is used only if the total size of the extra columns in the sort tuple does not exceed the value of the `max_length_for_sort_data` system variable. (A symptom of setting the value of this variable too high is that you will see high disk activity and low CPU activity.)

3.2 Bulk Insert

The logic behind bulk insert optimization is simple.

Instead of writing each key value to B-tree (that is, to the key cache, although the bulk insert code doesn't know about the key cache), we store keys in a balanced binary (red-black) tree, in memory. When this tree reaches its memory limit, we write all keys to disk (to key cache, that is). But since the key stream coming from the binary tree is already sorted, inserting goes much faster, all the necessary pages are already in cache, disk access is minimized, and so forth.

3.3 How MySQL Does Caching

MySQL has the following caches. (Note that the some of the filenames contain an incorrect spelling of the word "cache.")

Key Cache
A shared cache for all B-tree index blocks in the different NISAM files. Uses hashing and reverse linked lists for quick caching of the most recently used blocks and quick flushing of changed entries for a specific table. ('`mysys/mf_keycash.c`')

Record Cache
This is used for quick scanning of all records in a table. ('`mysys/mf_iocash.c`' and '`isam/_cash.c`')

Table Cache
This holds the most recently used tables. ('`sql/sql_base.cc`')

Hostname Cache
For quick lookup (with reverse name resolving). This is a must when you have a slow DNS. ('`sql/hostname.cc`')

Privilege Cache
To allow quick change between databases, the last used privileges are cached for each user/database combination. ('`sql/sql_acl.cc`')

Heap Table Cache
Many uses of `GROUP BY` or `DISTINCT` cache all found rows in a `HEAP` table. (This is a very quick in-memory table with hash index.)

Join Buffer Cache
For every "full join" in a `SELECT` statement the rows found are cached in a join cache. (A "full join" here means there were no keys that could be used to find rows for the next table in the list.) In the worst case, one `SELECT` query can use many join caches.

3.4 How MySQL Uses the Join Buffer Cache

Basic information about the join buffer cache:

- The size of each join buffer is determined by the value of the `join_buffer_size` system variable.
- This buffer is used only when the join is of type `ALL` or `index` (in other words, when no possible keys can be used).
- A join buffer is never allocated for the first non-const table, even if it would be of type `ALL` or `index`.

- The buffer is allocated when we need to do a full join between two tables, and freed after the query is done.
- Accepted row combinations of tables before the `ALL/index` are stored in the cache and are used to compare against each read row in the `ALL` table.
- We only store the used columns in the join buffer, not the whole rows.

Assume you have the following join:

```
Table name      Type
t1              range
t2              ref
t3              ALL
```

The join is then done as follows:

```
- While rows in t1 matching range
 - Read through all rows in t2 according to reference key
  - Store used fields from t1, t2 in cache
  - If cache is full
    - Read through all rows in t3
      - Compare t3 row against all t1, t2 combinations in cache
        - If row satisfies join condition, send it to client
    - Empty cache

- Read through all rows in t3
 - Compare t3 row against all stored t1, t2 combinations in cache
   - If row satisfies join condition, send it to client
```

The preceding description means that the number of times table `t3` is scanned is determined as follows:

```
S = size-of-stored-row(t1,t2)
C = accepted-row-combinations(t1,t2)
scans = (S * C)/join_buffer_size + 1
```

Some conclusions:

- The larger the value of `join_buffer_size`, the fewer the scans of `t3`. If `join_buffer_size` is already large enough to hold all previous row combinations, there is no speed to be gained by making it larger.
- If there are several tables of join type `ALL` or `index`, then we allocate one buffer of size `join_buffer_size` for each of them and use the same algorithm described above to handle it. (In other words, we store the same row combination several times into different buffers.)

3.5 How MySQL Handles `FLUSH TABLES`

- `FLUSH TABLES` is handled in '`sql/sql_base.cc::close_cached_tables()`'.
- The idea of `FLUSH TABLES` is to force all tables to be closed. This is mainly to ensure that if someone adds a new table outside of MySQL (for example, by copying files into a database directory with `cp`), all threads will start using the new table. This will also ensure that all table changes are flushed to disk (but of course not as optimally as simply calling a sync for all tables)!
- When you do a `FLUSH TABLES`, the variable `refresh_version` is incremented. Every time a thread releases a table, it checks if the refresh version of the table (updated at open) is the same as the current `refresh_version`. If not, it will close it and broadcast a signal on `COND_refresh` (to await any thread that is waiting for all instances of a table to be closed).

- The current `refresh_version` is also compared to the open `refresh_version` after a thread gets a lock on a table. If the refresh version is different, the thread will free all locks, reopen the table and try to get the locks again. This is just to quickly get all tables to use the newest version. This is handled by '`sql/lock.cc::mysql_lock_tables()`' and '`sql/sql_base.cc::wait_for_tables()`'.
- When all tables have been closed, `FLUSH TABLES` returns an okay to the client.
- If the thread that is doing `FLUSH TABLES` has a lock on some tables, it will first close the locked tables, then wait until all other threads have also closed them, and then reopen them and get the locks. After this it will give other threads a chance to open the same tables.

3.6 Full-text Search in MySQL

Hopefully, sometime there will be complete description of full-text search algorithms. For now, it's just unsorted notes.

Weighting in boolean mode

The basic idea is as follows: In an expression of the form `A or B or (C and D and E)`, either `A` or `B` alone is enough to match the whole expression, whereas `C`, `D`, and `E` should **all** match. So it's reasonable to assign weight 1 to each of `A`, `B`, and `(C and D and E)`. Furthermore, `C`, `D`, and `E` each should get a weight of 1/3.

Things become more complicated when considering boolean operators, as used in MySQL full-text boolean searching. Obviously, `+A +B` should be treated as `A and B`, and `A B -` as `A or B`. The problem is that `+A B` can **not** be rewritten in and/or terms (that's the reason why this—extended—set of operators was chosen). Still, aproximations can be used. `+A B C` can be approximated as `A or (A and (B or C))` or as `A or (A and B) or (A and C) or (A and B and C)`. Applying the above logic (and omitting mathematical transformations and normalization) one gets that for `+A_1 +A_2 ... +A_N B_1 B_2 ... B_M` the weights should be: `A_i = 1/N`, `B_j=1` if `N==0`, and, otherwise, in the first rewriting approach `B_j = 1/3`, and in the second one - `B_j = (1+(M-1)*2^M)/(M*(2^(M+1)-1))`.

The second expression gives a somewhat steeper increase in total weight as number of matched `B_j` values increases, because it assigns higher weights to individual `B_j` values. Also, the first expression is much simpler, so it is the first one that is implemented in MySQL.

3.7 Functions in the `mysys` Library

Functions in `mysys`: (For flags see '`my_sys.h`')

`int my_copy _A((const char *from, const char *to, myf MyFlags));`
: Copy file from `from` to `to`.

`int my_rename _A((const char *from, const char *to, myf MyFlags));`
: Rename file from `from` to `to`.

`int my_delete _A((const char *name, myf MyFlags));`
: Delete file `name`.

`int my_redel _A((const char *from, const char *to, int MyFlags));`
: Delete `from` before rename of `to` to `from`. Copies state from old file to new file. If `MY_COPY_TIME` is set, sets old time.

`int my_getwd _A((string buf, uint size, myf MyFlags));`
`int my_setwd _A((const char *dir, myf MyFlags));`
: Get and set working directory.

`string my_tempnam _A((const char *dir, const char *pfx, myf MyFlags));`
: Make a unique temporary file name by using `dir` and adding something after `pfx` to make the name unique. The file name is made by adding a unique six character string and `TMP_EXT` after `pfx`. Returns pointer to `malloc()`'ed area for filename. Should be freed by `free()`.

`File my_open _A((const char *FileName,int Flags,myf MyFlags));`
`File my_create _A((const char *FileName, int CreateFlags, int AccsesFlags, myf MyFlags));`
`int my_close _A((File Filedes, myf MyFlags));`
`uint my_read _A((File Filedes, byte *Buffer, uint Count, myf MyFlags));`
`uint my_write _A((File Filedes, const byte *Buffer, uint Count, myf MyFlags));`
`ulong my_seek _A((File fd,ulong pos,int whence,myf MyFlags));`
`ulong my_tell _A((File fd,myf MyFlags));`
: Use instead of open, open-with-create-flag, close, read, and write to get automatic error messages (flag `MYF_WME`) and only have to test for != 0 if error (flag `MY_NABP`).

`FILE *my_fopen _A((const char *FileName,int Flags,myf MyFlags));`
`FILE *my_fdopen _A((File Filedes,int Flags,myf MyFlags));`
`int my_fclose _A((FILE *fd,myf MyFlags));`
`uint my_fread _A((FILE *stream,byte *Buffer,uint Count,myf MyFlags));`
`uint my_fwrite _A((FILE *stream,const byte *Buffer,uint Count, myf MyFlags));`
`ulong my_fseek _A((FILE *stream,ulong pos,int whence,myf MyFlags));`
`ulong my_ftell _A((FILE *stream,myf MyFlags));`
: Same read-interface for streams as for files.

`gptr _mymalloc _A((uint uSize,const char *sFile,uint uLine, myf MyFlag));`
`gptr _myrealloc _A((string pPtr,uint uSize,const char *sFile,uint uLine, myf MyFlag));`
`void _myfree _A((gptr pPtr,const char *sFile,uint uLine));`
`int _sanity _A((const char *sFile,unsigned int uLine));`
`gptr _myget_copy_of_memory _A((const byte *from,uint length,const char *sFile, uint uLine,myf MyFlag));`
: `malloc(size,myflag)` is mapped to these functions if not compiled with `-DSAFEMALLOC`.

`void TERMINATE _A((void));`
: Writes `malloc()` info on `stdout` if compiled with `-DSAFEMALLOC`.

`int my_chsize _A((File fd, ulong newlength, myf MyFlags));`
: Change size of file `fd` to `newlength`.

`void my_error _D((int nr, myf MyFlags, ...));`
: Writes message using error number (see 'mysys/errors.h') on `stdout`, or using curses, if `MYSYS_PROGRAM_USES_CURSES()` has been called.

`void my_message _A((const char *str, myf MyFlags));`
: Writes `str` on `stdout`, or using curses, if `MYSYS_PROGRAM_USES_CURSES()` has been called.

`void my_init _A((void ));`
: Start each program (in `main()`) with this.

`void my_end _A((int infoflag));`
: Gives info about program. If `infoflag & MY_CHECK_ERROR`, prints if some files are left open. If `infoflag & MY_GIVE_INFO`, prints timing info and `malloc()` info about program.

`int my_copystat _A((const char *from, const char *to, int MyFlags));`
: Copy state from old file to new file. If `MY_COPY_TIME` is set, sets old time.

`string my_filename _A((File fd));`
: Returns filename of open file.

`int dirname _A((string to, const char *name));`
: Copy name of directory from filename.

`int test_if_hard_path _A((const char *dir_name));`
: Test if `dir_name` is a hard path (starts from root).

`void convert_dirname _A((string name));`
: Convert dirname according to system. On Windows, changes all characters to capitals and changes '/' to '\'.

`string fn_ext _A((const char *name));`
: Returns pointer to extension in filename.

`string fn_format _A((string to,const char *name,const char *dsk,const char *form,int flag));`
: Format a filename with replacement of library and extension and convert between different systems. The `to` and `name` parameters may be identical. Function doesn't change name if `name` != `to`. `flag` may be:

1	Force replace filnames library with 'dsk'
2	Force replace extension with 'form' */
4	Force unpack filename (replace ~ with home directory)
8	Pack filename as short as possible for output to user

All open requests should always use at least `open(fn_format(temp_buffer, name, "", "", 4), ...)` to unpack home and convert filename to system-form.

`string fn_same _A((string toname, const char *name, int flag));`
: Copies directory and extension from `name` to `toname` if needed. Copying can be forced by same flags used in `fn_format()`.

`int wild_compare _A((const char *str, const char *wildstr));`
: Compare if `str` matches `wildstr`. `wildstr` can contain '*' and '?' as wildcard characters. Returns 0 if `str` and `wildstr` match.

`void get_date _A((string to, int timeflag));`
: Get current date in a form ready for printing.

`void soundex _A((string out_pntr, string in_pntr))`
: Makes `in_pntr` to a 5 char long string. All words that sound alike have the same string.

`int init_key_cache _A((ulong use_mem, ulong leave_this_much_mem));`
: Use caching of keys in MISAM, PISAM, and ISAM. `KEY_CACHE_SIZE` is a good size. Remember to lock databases for optimal caching.

`void end_key_cache _A((void));`
: End key caching.

4 Charsets and Related Issues

4.1 `CHARSET_INFO` Structure

5 How MySQL Performs Different Selects

5.1 Steps of Select Execution

Every select is performed in these base steps:

- `JOIN::prepare`
  - Initialization and linking `JOIN` structure to `st_select_lex`.
  - `fix_fields()` for all items (after `fix_fields()`, we know everything about item).
  - Moving `HAVING` to `WHERE` if possible.
  - Initialization procedure if there is one.
- `JOIN::optimize`
  - Single select optimization.
  - Creation of first temporary table if needed.
- `JOIN::exec`
  - Performing select (a second temporary table may be created).
- `JOIN::cleanup`
  - Removing all temporary tables, other cleanup.
- `JOIN::reinit`
  - Prepare all structures for execution of `SELECT` (with `JOIN::exec`).

5.2 `select_result` Class

This class has a very important role in `SELECT` performance with `select_result` class and classes inherited from it (usually called with a `select_` prefix). This class provides the interface for transmitting results.

The key methods in this class are the following:

- `send_fields` sends given item list headers (type, name, etc.).
- `send_data` sends given item list values as row of table of result.
- `send_error` is used mainly for error interception, making some operation and then `::send_error` will be called.

For example, there are the following `select_result` classes:

- `select_send` used for sending results though network layer.
- `select_export` used for exporting data to file.
- `multi_delete` used for multi-delete.
- `select_insert` used for `INSERT` ... `SELECT` ...
- `multi_update` used for multi-update.
- `select_singlerow_subselect` used for row and scalar subqueries..
- `select_exists_subselect` used for `EXISTS`/`IN`/`ALL`/`ANY`/`SOME` subqueries.
- `select_max_min_finder_subselect` used for min/max subqueries (`ALL`/`ANY` subquery optimization).

5.3 SIMPLE or PRIMARY SELECT

For performing single primary select, `SELECT` uses the `mysql_select` function, which does:

- allocate `JOIN`
- `JOIN::prepare`
- `JOIN::optimize`
- `JOIN::exec`
- `JOIN::cleanup`

In previous versions of MySQL, all `SELECT` operations were performed with the help of this function and `mysql_select()` was not divided into parts.

5.4 Structure Of Complex Select

There are two structures that describe selects:

- `st_select_lex` (`SELECT_LEX`) for representing `SELECT` itself
- `st_select_lex_unit` (`SELECT_LEX_UNIT`) for grouping several selects in a bunch

The latter item represents `UNION` operation (the absence of `UNION` is a union with only one `SELECT` and this structure is present in any case). In the future, this structure will be used for `EXCEPT` and `INTERSECT` as well.

For example:

```
(SELECT ...) UNION (SELECT ... (SELECT...)...(SELECT...UNION...SELECT))
   1            2     3           4             5        6       7
```

will be represented as:

```
------------------------------------------------------------------------
                                                                 level 1
SELECT_LEX_UNIT(2)
|
+---------------+
|               |
SELECT_LEX(1)   SELECT_LEX(3)
                |
--------------- | ------------------------------------------------------
                |                                                level 2
                +-------------------+
                |                   |
                SELECT_LEX_UNIT(4)  SELECT_LEX_UNIT(6)
                |                   |
                |                   +--------------+
                |                   |              |
                SELECT_LEX(4)       SELECT_LEX(5)  SELECT_LEX(7)

------------------------------------------------------------------------
```

Note: Single subquery 4 has its own `SELECT_LEX_UNIT`.

The uppermost `SELECT_LEX_UNIT` (#2 in example) is stored in LEX. The first and uppermost `SELECT_LEX` (#1 in example) is stored in LEX, too. These two structures always exist.

At the time of creating or performing any `JOIN::*` operation, `LEX::current_select` points to an appropriate `SELECT_LEX`.

Only during parsing of global ORDER BY and LIMIT clauses (for the whole UNION), LEX::current_select points to SELECT_LEX_UNIT of this unit, in order to store this parameter in this SELECT_LEX_UNIT. SELECT_LEX and SELECT_LEX_UNIT are inherited from st_select_lex_node.

5.5 Non-Subquery UNION Execution

Non-subquery unions are performed with the help of mysql_union(). For now, it is divided into the following steps:

- st_select_lex_unit::prepare (the same procedure can be called for single SELECT for derived table => we have support for it in this procedure, but we will not describe it here):
  - Create select_union (inherited from select_result) which will write select results in this temporary table, with empty temporary table entry. We will need this object to store in every JOIN structure link on it, but we have not (yet) temporary table structure.
  - Allocate JOIN structures and execute JOIN::prepare() for every SELECT to get full information about types of elements of SELECT list (results). Merging types of result fields and storing them in special Items (Item_type_holder) will be done in this loop, too. Result of this operation (list of types of result fields) will be stored in st_select_lex_unit::types).
  - Create a temporary table for storing union results (if UNION without ALL option, 'distinct' parameter will be passed to the table creation procedure).
  - Assign a temporary table to the select_union object created in the first step.
- st_select_lex_unit::exec
  - Delete rows from the temporary table if this is not the first call.
  - if this is the first call, call JOIN::optimize else JOIN::reinit and then JOIN::exec for all SELECTs (select_union will write a result for the temporary table). If union is cacheable and this is not the first call, the method will do nothing.
  - Call mysql_select on temporary table with global ORDER BY and LIMIT parameters after collecting results from all SELECTs. A special fake_select_lex (SELECT_LEX) which is created for every UNION will be passed for this procedure (this SELECT_LEX also can be used to store global ORDER BY and LIMIT parameters if brackets used in a query).

5.6 Derived Table Execution

"Derived tables" is the internal name for subqueries in the FROM clause.

The processing of derived tables is now included in the table opening process (open_and_lock_tables() call). Routine of execution derived tables and substituting temporary table instead of it (mysql_handle_derived()) will be called just after opening and locking all real tables used in query (including tables used in derived table query).

If lex->derived_tables flag is present, all SELECT_LEX structures will be scanned (there is a list of all SELECT_LEX structures in reverse order named lex->all_selects_list, the first SELECT in the query will be last in this list).

There is a pointer for the derived table, SELECT_LEX_UNIT stored in the TABLE_LIST structure (TABLE_LIST::derived). For any table that has this pointer, mysql_derived() will be called.

mysql_derived():

- Creates union_result for writing results in this table (with empty table entry, same as for UNIONs).

- call `unit->prepare()` to get list of types of result fields (it work correctly for single `SELECT`, and do not create temporary table for `UNION` processing in this case).
- Creates a temporary table for storing results.
- Assign this temporary table to `union_result` object.
- Calls `mysql_select` or `mysql_union` to execute the query.
- If it is not explain, then cleanup `JOIN` structures after execution (`EXPLAIN` needs data of optimization phase and cleanup them after whole query processing).
- Stores pointer to this temporary table in `TABLE_LIST` structure, then this table will be used by outer query.
- Links this temporary table in `thd->derived_tables` for removing after query execution. This table will be closed in `close_thread_tables` if its second parameter (`bool skip_derived`) is true.

5.7 Subqueries

In expressions, subqueries (that is, subselects) are represented by `Item` inherited from `Item_subselect`.

To hide difference in performing single `SELECT`s and `UNION`s, `Item_subselect` uses two different engines, which provide uniform interface for access to underlying `SELECT` or `UNION` (`subselect_single_select_engine` and `subselect_union_engine`, both are inherited from `subselect_engine`).

The engine will be created at the time `Item_select` is constructed (`Item_subselect::init` method).

On `Item_subselect::fix_fields()`, `engine->prepare()` will be called.

Before calling any value-getting method (`val`, `val_int`, `val_str`, `bring_value` (in case of row result)) `engine->exec()` will be called, which executes the query or just does nothing if subquery is cacheable and has already been executed.

Inherited items have their own select_result classes. There are two types of them:

- `select_singlerow_subselect`, to store values of given rows in `Item_singlerow_subselect` cache on `send_data()` call, and report error if `Item_subselect` has 'assigned' attribute.
- `select_exists_subselect` just store 1 as value of `Item_exists_subselect` on `send_data()` call. Since `Item_in_subselect` and `Item_allany_subselect` are inherited from `Item_exists_subselect,` they use the same `select_result` class.

`Item_select` will never call the `cleanup()` procedure for `JOIN`. Every `JOIN::cleanup` will call `cleanup()` for inner `JOIN`s. The uppermost `JOIN::cleanup` will be called by `mysql_select()` or `mysql_union()`.

5.8 Single Select Engine

subselect_single_select_engine:

- `constructor` allocate `JOIN` and store pointers on `SELECT_LEX` and `JOIN`.
- `prepare()` call `JOIN::prepare`.
- `fix_length_and_dec()` prepare cache and receive type and parameters of returning items (called only by `Item_singlerow_subselect`).
- `exec()` drop 'assigned' flag of `Item_subselect`. If this is the first time, call `JOIN::optimize` and `JOINexec()`, else do nothing or `JOIN::reinit() JOIN::exec()` depending on type of subquery.

5.9 Union Engine

`subselect_union_engine`:

- `constructor` just store pointer to `st_select_lex_union` (`SELECT_LEX_UNION`).
- `prepare()` call `st_select_lex_unit::prepare`.
- `fix_length_and_dec()` prepare cache and receive type and parameters (maximum of length) of returning items (called only by `Item_singlerow_subselect`).
- `exec()` call `st_select_lex_unit::exec()`. `st_select_lex_unit::exec()` can drop 'assigned' flag of `Item_subselect` if `st_select_lex_unit::item` is not 0.

5.10 Special Engines

There are special engines used for optimization purposes. These engines do not have a full range of features. They can only fetch data. The normal engine can be replaced with such special engines only during the optimization process.

Now we have two such engines:

- `subselect_uniquesubquery_engine` used for:

  ```
  left_expression IN (SELECT primary_key FROM table WHERE conditions)
  ```

  This looks for the given value once in a primary index, checks the `WHERE` condition, and returns "was it found or not?"
- `subselect_indexsubquery_engine` used for:

  ```
  left_expression IN (SELECT any_key FROM table WHERE conditions)
  ```

  This first looks up the value of the left expression in an index (checking the `WHERE` condition), then if value was not found, it checks for `NULL` values so that it can return `NULL` correctly (only if a `NULL` result makes sense, for example if an `IN` subquery is the top item of the `WHERE` clause then `NULL` will not be sought)

The decision about replacement of the engine happens in `JOIN::optimize`, after calling `make_join_readinfo`, when we know what the best index choice is.

5.11 Explain Execution

For an `EXPLAIN` statement, for every `SELECT`, `mysql_select` will be called with option `SELECT_DESCRIBE`.

For main `UNION`, `mysql_explain_union` will be called.

For every `SELECT` in a given union, `mysql_explain_union` will call `mysql_explain_select`.

`mysql_explain_select` will call `mysql_select` with option `SELECT_DESCRIBE`.

`mysql_select` creates a `JOIN` for select if it does not already exist (it might already exist because if it called for subselect `JOIN` can be created in `JOIN::optimize` of outer query when it decided to calculate the value of the subquery). Then it calls `JOIN::prepare`, `JOIN::optimize`, `JOIN::exec` and `JOIN::cleanup` as usual.

`JOIN::exec` is called for `SELECT` with `SELECT_DESCRIBE` option call `select_describe`.

`select_describe` returns the user description of `SELECT` and calls `mysql_explain_union` for every inner `UNION`.

PROBLEM: how it will work with global query optimization?

6 How MySQL Transforms Subqueries

`Item_subselect` virtual method `select_transformer` is used to rewrite subqueries. It is called from `Item_subselect::init` (which is called just after call to `fix_fields()` method for all items in `JOIN::prepare`).

6.1 Item_in_subselect::select_transformer

`Item_in_subselect::select_transformer` is divided into two parts, for the scalar left part and the row left part.

6.1.1 Scalar IN Subquery

To rewrite a scalar `IN` subquery, the method used is `Item_in_subselect::single_value_transformer`. Scalar `IN` subquery will be replaced with `Item_in_optimizer`.

`Item_in_optimizer` item is a special boolean function. On a value request (one of `val`, `val_int`, or `val_str` methods) it evaluates left expression of `IN` by storing its value in cache item (one of `Item_cache*` items), then it tests the cache to see whether it is `NULL`. If left expression (cache) is `NULL`, then `Item_in_optimizer` returns `NULL`, else it evaluates `Item_in_subselect`.

Example queries.

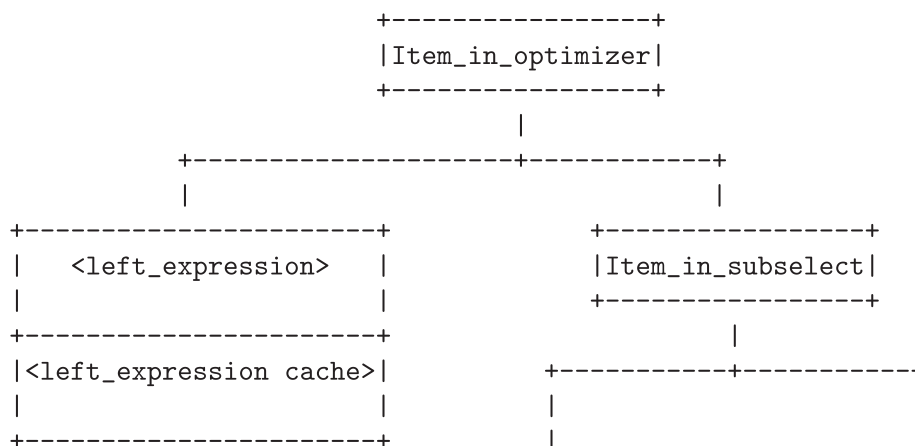
```
a) SELECT * from t1 where t1.a in (SELECT t2.a FROM t2);
b) SELECT * from t1 where t1.a in (SELECT t2.a FROM t2 GROUP BY t2.a);
```

- `Item_in_subselect` inherits the mechanism for getting a value from `Item_exists_subselect`.
- `Select_transformer` stores a reference to the left expression in its conditions:

  ```
  (in WHERE and HAVING in case 'a' and in HAVING in case 'b')
  ```

- Item from item list of this select (`t2.a`) can be referenced with a special reference (`Item_ref_null_helper` or `Item_null_helper`). This reference informs `Item_in_optimizer` whether item (`t2.a`) is `NULL` by setting the 'was_null' flag.
- The return value from `Item_in_subselect` will be evaluated as follows:
  - If TRUE, return true
  - If NULL, return null (that is, unknown)
  - If FALSE, and 'was_null' is set, return null
  - Return FALSE

<left_expression> IN (SELECT <item> ...) will be represented as follows:



```
          ^                 +-----------+       +---------------------+
          +<<<<<<<<<<<<<<<<<| Item_ref  |    +<<<|Item_ref_null_helper|
                            +-----------+    V   +---------------------+
                                             V   +---------------------+
                                             +>>>|       <item>        |
                                                 +---------------------+
```

where <<<<<<<<< is reference in meaning of Item_ref.

Item_ref is used to point to <left_expression cache>, because at the time of transformation we know only the address of variable where the cache pointer will be stored.

If the select statement has an ORDER BY clause, it will be wiped out, because there is no sense in ORDER BY without LIMIT here.

If IN subquery union, the condition of every select in the UNION will be changed individually.

If a condition needs to be added to the WHERE clause, it will be presented as (item OR item IS NULL) and Item_is_not_null_test(item) will be added to the HAVING clause. Item_is_not_null_test registers NULL value the way Item_ref_null_helper does it, and returns FALSE if argument is NULL. With the above trick, we will register NULL value of Item even for the case of index optimization of a WHERE clause (case 'a' in the following example).

The following are examples of IN transformations:

- Example 1:

  ```
  <left_expression> IN (SELECT <item> FROM t WHERE <where_exp>)
  ```

  If returning NULL correctly would make sense, the above will become:

  ```
  (SELECT 1 FROM t
    WHERE
      <where_exp> and
      (Item_ref(<cached_left_expression>)=<item> or
       <Item> is null)
    HAVING Item_is_not_null_test(<item>))
  ```

  When subquery is marked as the top item of the WHERE clause, it will become:

  ```
  (SELECT 1 FROM t
    WHERE
       <where_exp> and
       Item_ref(<cached_left_expression>)=<item>)
  ```

- Example 2:

  ```
  <left_expression> IN (SELECT <item> FROM t
                               HAVING <having_expr>
                               ORDER BY 1)
  ```

  will be represented as

  ```
  (SELECT <item> as ref_null_helper FROM t
     HAVING <having_exp> AND
       Item_ref(<cached_left_expression>) = Item_ref_null_helper(item))
  ```

- Example 3:

  ```
  <left_expression> IN (SELECT <item> UNION ...)
  ```

  will become

  ```
  (SELECT 1
     HAVING Item_ref(<cached_left_expression>)=
            <Item_null_helper(<Item>)>
   UNION ...)
  ```

(`HAVING` without `FROM` is a syntax error, but a `HAVING` condition is checked even for subquery without `FROM`)

- Example 4:

```
<left_expression> IN (select <item>)
```

will be completely replaced with `<left_expression> = <item>`

Now conditions `(WHERE (a) or HAVING (b))` will be changed, depending on the select, in the following way:

If subquery contains a `HAVING` clause, `SUM()` function or `GROUP BY` (example 1), then the item list will be unchanged and `Item_ref_null_helper` reference will be created on item list element. A condition will be added to the `HAVING`.

If the subquery does not contain `HAVING`, `SUM()` function or `GROUP BY` (example 2), then:

- `item list` will be replaced with 1.
- `left_expression cache> = <item> or is null <item>` will be added to the `WHERE` clause and a special `is_not_null(item)` will be added to the `HAVING`, so null values will be registered. If returning `NULL` wouldn't make correct sense, then only `left_expression cache> = <item>` will be added to the `WHERE` clause. If this subquery does not contain a `FROM` clause or if the subquery contains `UNION` (example 3), then `left_expression cache> = Item_null_helper(<item>)` will be added to the `HAVING` clause.

A single select without a `FROM` clause will be reduced to just `<left_expression> = <item>` without use of `Item_in_optimizer`.

6.1.2 Row `IN` Subquery

To rewrite a row `IN` subquery, the method used is `Item_in_subselect::row_value_transformer`. It works in almost the same way as the scalar analog, but works with `Item_cache_row` for caching left expression and uses references for elements of `Item_cache_row`. To refer to item list it uses `Item_ref_null_helper(ref_array+i)`.

Subquery with `HAVING`, `SUM()` function, or `GROUP BY` will transformed in the following way:

```
ROW(l1, l2, ... lN) IN (SELECT i1, i2, ... iN FROM t HAVING <having_expr>)
```

will become:

```
(SELECT i1, i2, ... iN FROM t
   HAVING <having_expr> and
     <cache_l0> = <Item_ref_null_helper(ref_array[0]> AND
     <cache_l1> = <Item_ref_null_helper(ref_array[1])> AND
     ...
     <cache_lN-1> = <Item_ref_null_helper(ref_array[N-1]>)
```

`SELECT` without `FROM` will be transformed in this way, too.

It will be the same for other subqueries, except for the `WHERE` clause.

6.2 `Item_allany_subselect`

`Item_allany_subselect` is inherited from `Item_in_subselect`. `ALL`/`ANY`/`SOME` use the same algorithm (and the same method of `Item_in_subselect`) as scalar `IN`, but use a different function instead of `=`.

`ANY`/`SOME` use the same function that was listed after the left expression.

`ALL` uses an inverted function, and all subqueries passed as arguments to `Item_func_not_all` (`Item_func_not_all` is a special `NOT` function used in optimization, see following).

But before above transformation ability of independent ALL/ANY/SOME optimization will be checked (query is independent, operation is one of <, =<, >, >=, returning correct NULL have no sense (top level of WHERE clause) and it is not row subquery).

For such queries, the following transformation can be done:

```
val >  ALL (SELECT...) -> val >  MAX (SELECT...)
val <  ALL (SELECT...) -> val <  MIN (SELECT...)
val >  ANY (SELECT...) -> val >  MIN (SELECT...)
val <  ANY (SELECT...) -> val <  MAX (SELECT...)
val >= ALL (SELECT...) -> val >= MAX (SELECT...)
val <= ALL (SELECT...) -> val <= MIN (SELECT...)
val >= ANY (SELECT...) -> val >= MIN (SELECT...)
val <= ANY (SELECT...) -> val <= MAX (SELECT...)
```

ALL subqueries already have NOT before them. This problem can be solved with help of special NOT, which can bring 'top' tag to its argument and correctly process NULL if it is 'top' item (return TRUE if argument is NULL if it is 'top' item). Let's call this operation _NOT_. Then we will have following table of transformation:

```
val >  ALL (SELECT...) -> _NOT_ val >= MAX (SELECT...)
val <  ALL (SELECT...) -> _NOT_ val <= MIN (SELECT...)
val >  ANY (SELECT...) ->       val <  MIN (SELECT...)
val <  ANY (SELECT...) ->       val >  MAX (SELECT...)
val >= ALL (SELECT...) -> _NOT_ val >  MAX (SELECT...)
val <= ALL (SELECT...) -> _NOT_ val <  MIN (SELECT...)
val >= ANY (SELECT...) ->       val <= MIN (SELECT...)
val <= ANY (SELECT...) ->       val >= MAX (SELECT...)
```

If subquery does not contain grouping and aggregate function, above subquery can be rewritten with MAX()/MIN() aggregate function, for example:

```
val > ANY (SELECT item ...) -> val < (SELECT MIN(item)...)
```

For queries with aggregate function and/or grouping, special Item_maxmin_subselect will be used. This subquery will return maximum (minimum) value of result set.

6.3 Item_singlerow_subselect

Item_singlerow_subselect will be rewritten only if it contains no FROM clause, and it is not part of UNION, and it is a scalar subquery. For now, there will be no conversion of subqueries with field or reference on top of item list (on the one hand we can't change the name of such items, but on the other hand we should assign to it the name of the whole subquery which will be reduced);

The following will not be reduced:

```
SELECT a;
SELECT 1 UNION SELECT 2;
SELECT 1 FROM t1;
```

The following select will be reduced:

```
SELECT 1;
SELECT a+2;
```

Such a subquery will be completely replaced by its expression from item list and its SELECT_LEX and SELECT_LEX_UNIT will be removed from SELECT_LEX's tree.

But every Item_field and Item_ref of that expression will be marked for processing by a special fix_fields() procedure. The fix_fields() procedures for such Items will be performed in

the same way as for items of an inner subquery. Also, if this expression is `Item_fields` or `Item_ref`, then the name of this new item will be the same as the name of this item (but not `(SELECT ...)`). This is done to prevent broken references on such items from more inner subqueries.

7 MySQL Client/Server Protocol

7.1 Raw Packet Without Compression

```
+-----------------------------------------------+
| Packet Length | Packet no     | Data          |
| 3 Bytes       | 1 Byte        | n Bytes       |
+-----------------------------------------------+
```

3 Byte packet length
: The length is calculated with int3store See include/global.h for details. The max packetsize can be 16MB.

1 Byte packet no
: If no compression is used the first 4 bytes of each packet is the header of the packet. The packet number is incremented for each sent packet. The first packet starts with 0.

n Byte data

The packet length can be recalculated with:

```
length = byte1 + (256 * byte2) + (256 * 256 * byte3)
```

7.2 Raw Packet With Compression

```
+---------------------------------------------------+
| Packet Length | Packet no | Uncomp. Packet Length |
| 3 Bytes       | 1 Byte    | 3 Bytes               |
+---------------------------------------------------+
```

3 Byte packet length
: The length is calculated with int3store See include/global.h for details. The max packetsize can be 16MB.

1 Byte packet no

3 Byte uncompressed packet length

If compression is used the first 7 bytes of each packet is the header of the packet.

7.3 Basic Packets

7.3.1 OK Packet

For details, see '`sql/net_pkg.cc::send_ok()`'.

```
+-----------------------------------------------+
| Header        | No of Rows    | Affected Rows |
|               | 1 Byte        | 1-8 Byte      |
|-----------------------------------------------|
| ID (last_insert_id)   | Status | Length       |
| 1-8 Byte              | 2 Byte | 1-8 Byte     |
|-----------------------------------------------|
| Messagetext                                   |
| n Byte                                        |
+-----------------------------------------------+
```

Header

1 byte number of rows ? (always 0 ?)
1-8 bytes affected rows
1-8 byte id (last_insert_id)
2 byte Status (usually 0)
If the OK-packege includes a message:
1-8 bytes length of message
n bytes messagetext

7.3.2 Error Packet

```
+-------------------------------------------------+
| Header         | Status code    | Error no      |
|                | 1 Byte         | 2 Byte        |
|-------------------------------------------------|
| Messagetext                          | 0x00   |
| n Byte                               | 1 Byte |
+-------------------------------------------------+
```

Header

1 byte status code (0xFF = ERROR)
2 byte error number (is only sent to new 3.23 clients.
n byte errortext
1 byte 0x00

7.4 Communication

```
> Packet from server to client
< Paket from client tor server

Login
------
> 1. packet
Header
1 byte protocolversion
n byte serverversion
1 byte 0x00
4 byte threadnumber
8 byte crypt seed
1 byte 0x00
2 byte CLIENT_xxx options (see include/mysql_com.h
that is supported by the server
1 byte number of current server charset
2 byte server status variables (SERVER_STATUS_xxx flags)
13 byte 0x00 (not used yet).

< 2. packet
Header
2 byte CLIENT_xxx options
3 byte max_allowed_packet for the client
n byte username
1 byte 0x00
```

```
8 byte crypted password
1 byte 0x00
n byte databasename
1 byte 0x00

> 3. packet
OK-packet


Command
--------
< 1. packet
Header
1 byte command type (e.g.0x03 = query)
n byte query

Result set (after command)
--------------------------
> 2. packet
Header
1-8 byte field_count (packed with net_store_length())

If field_count == 0 (command):
1-8 byte affected rows
1-8 byte insert id
2 bytes server_status (SERVER_STATUS_xx)

If field_count == NULL_LENGTH (251)
LOAD DATA LOCAL INFILE

If field_count > 0 Result Set:

> n packets
Header Info
Column description: 5 data object /column
(See code in unpack_fields())

Columninfo for each column:
1 data block table_name
    1 byte length of block
    n byte data
1 data block field_name
    1 byte length of block...
    n byte data
1 data block display length of field
    1 byte length of block
    3 bytes display length of filed
1 data block type field of type (enum_field_types)
    1 byte length of block
    1 bytexs field of type
1 data block flags
    1 byte length of block
```

```
    2 byte flags for the columns (NOT_NULL_FLAG, ZEROFILL_FLAG....)
    1 byte decimals

if table definition:
1 data block default value

Actual result (one packet per row):
4 byte header
1-8 byte length of data
n data
```

7.5 Fieldtype Codes

```
                display_length  |enum_field_type        |flags
                ----------------------------------------------------
Blob            03 FF FF 00     |01 FC                  |03 90 00 00
Mediumblob      03 FF FF FF     |01 FC                  |03 90 00 00
Tinyblob        03 FF 00 00     |01 FC                  |03 90 00 00
Text            03 FF FF 00     |01 FC                  |03 10 00 00
Mediumtext      03 FF FF FF     |01 FC                  |03 10 00 00
Tinytext        03 FF 00 00     |01 FC                  |03 10 00 00
Integer         03 0B 00 00     |01 03                  |03 03 42 00
Mediumint       03 09 00 00     |01 09                  |03 00 00 00
Smallint        03 06 00 00     |01 02                  |03 00 00 00
Tinyint         03 04 00 00     |01 01                  |03 00 00 00
Varchar         03 XX 00 00     |01 FD                  |03 00 00 00
Enum            03 05 00 00     |01 FE                  |03 00 01 00
Datetime        03 13 00 00     |01 0C                  |03 00 00 00
Timestamp       03 0E 00 00     |01 07                  |03 61 04 00
Time            03 08 00 00     |01 0B                  |03 00 00 00
Date            03 0A 00 00     |01 0A                  |03 00 00 00
```

7.6 Functions Used to Implement the Protocol

```
Raw packets
-----------

- The my_net_xxxx() functions handles the packaging of a stream of data
  into a raw packet that contains a packet number, length and data.

- This is implemented for the server in sql/net_serv.cc.
  The client file, libmysql/net.c, is symlinked to this file

The important functions are:

my_net_write() Store a packet (= # number of bytes) to be sent
net_flush() Send the packets stored in the buffer
net_write_command() Send a command (1 byte) + packet to the server.
my_net_read() Read a packet
```

```
Include files
-------------

- include/mysql.h is included by all MySQL clients.  It includes the
  MYSQL and MYSQL_RES structures.
- include/mysql_com.h is include by mysql.h and mysql_priv.h (the
  server) and includes a lot of common functions and structures to
  handle the client/server protocol.


Packets from server to client:
-----------------------------

sql/net_pkg.cc:

 - Sending of error packets
 - Sending of OK packets (= end of data)
 - Storing of values in a packet


sql/sql_base.cc:

 - Function send_fields() sends the field description to the client.

sql/sql_show.cc:

 - Sends results for a lot of SHOW commands, including:
   SHOW DATABASES [like 'wildcard']
   SHOW TABLES    [like 'wildcard']


Packets from client to server:
------------------------------

This is done in libmysql/libmysql.c

The important ones are:

- mysql_real_connect()          Connects to a mysqld server
- mysql_real_query()            Sends a query to the server and
                                reads the OK packet or columns header.
- mysql_store_result()          Read a result set from the server to memory
- mysql_use_result()            Read a result set row by row from the server.

- net_safe_read()               Read a packet from the server with
                                error handling.
- net_field_length()            Reads the length of a packet string.
- simple_command()              Sends a command/query to the server.
```

```
Connecting to mysqld (the MySQL server)
---------------------------------------

- On the client side: libmysql/libmysql.c::mysql_real_connect().
- On the server side: sql/sql_parse.cc::check_connections()

The packets sent during a connection are as follows

Server:  Send greeting package (includes server capabilites, server
 version and a random string of bytes to be used to scramble
 the password.
Client:  Sends package with client capabilites, user name, scrambled
 password, database name

Server:  Sends ok package or error package.

Client:  If init command specified, send it t the server and read
 ok/error package.


Password functions
------------------

The passwords are scrambled to a random number and are stored in hex
format on the server.

The password handling is done in sql/password.c. The important
function is 'scramble()', which takes the a password in clear text
and uses this to 'encrypt' the random string sent by the server
to a new message.

The encrypted message is sent to the server which uses the stored
random number password to encrypt the random string sent to the
client. If this is equal to the new message the client sends to the
server then the password is accepted.
```

7.7 Another Description of the Protocol

```
*****************************
*
* PROTOCOL OVERVIEW
*
*****************************

The MySQL protocol is relatively simple, and is designed for high performance
through minimization of overhead, and extensibility through versioning and
options flags. It is a request-response protocol, and does not allow
multitasking or multiplexing over a single connection. There are two packet
formats, 'raw' and 'compressed' (which is used when both client and
server support zlib compression, and the client requests that data be
compressed):
```

```
* RAW PACKET, shorter than 16 M *

+-----------------------------------------------+
| Packet Length | Packet no     | Data          |
| 3 Bytes       | 1 Byte        | n Bytes       |
+-----------------------------------------------+
^                               ^
|          'HEADER'             |
+-------------------------------+


 * Packet Length: Calculated with int3store. See include/global.h for
                  details. The basic computation is length = byte1 +
                  (256 * byte2) + (256 * 256 * byte3). The max packetsize
                  can be 16MB.

 * Packet no:     The packet number is incremented for each sent packet.
                  The first packet for each query from the client
  starts with 0.

 * Data:          Specific to the operation being performed. Most often
                  used to send string data, such as a SQL query.

* COMPRESSED PACKET *

+-------------------------------------------------------+-----------------+
| Packet Length | Packet no | Uncomp. Packet Length | Compressed Data |
| 3 Bytes       | 1 Byte    | 3 Bytes               | n bytes         |
+-------------------------------------------------------+-----------------+
^                                                       ^
|                    'HEADER'                           |
+-------------------------------------------------------+

 * Packet Length: Calculated with int3store. See include/my_global.h for
                  details. The basic computation is length = byte1 +
                  (256 * byte2) + (256 * 256 * byte3). The max packetsize
                  can be 16MB.

 * Packet no:     The packet number is incremented for each sent packet.
                  The first packet starts with 0.

 * Uncomp. Packet Length: The length of the original, uncompressed packet
   If this is zero then the data is not compressed.

 * Compressed Data: The original packet, compressed with zlib compression


When using the compressed protocol, the client/server will only compress
send packets where the new packet is smaller than the not compressed one.
In other words, some packets may be compressed while others will not.
```

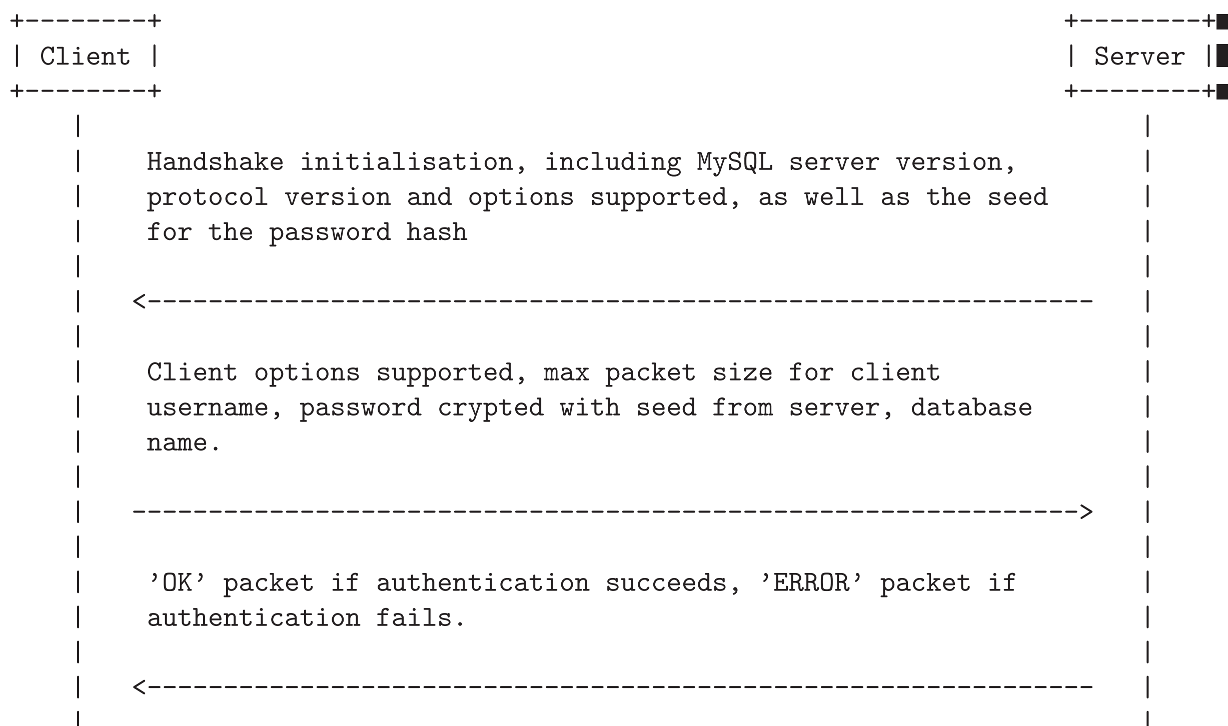
The 'compressed data' is one or more packets in *RAW PACKET* format.

```
****************************
*
* FLOW OF EVENTS
*
****************************
```

To understand how a client communicates with a MySQL server, it is easiest to start with a high-level flow of events. Each event section will then be followed by details of the exact contents of each type of packet involved in the event flow.

```
*                          *
* CONNECTION ESTABLISHMENT *
*                          *
```

Clients connect to the server via a TCP/IP socket (port 3306 by default), a Unix Domain Socket, or named pipes (on Windows). Once connected, the following connection establishment sequence is followed:



```
+--------+                                                              +--------+
| Client |                                                              | Server |
+--------+                                                              +--------+
    |                                                                         |
    |    Handshake initialisation, including MySQL server version,            |
    |    protocol version and options supported, as well as the seed          |
    |    for the password hash                                                |
    |                                                                         |
    |   <--------------------------------------------------------------       |
    |                                                                         |
    |    Client options supported, max packet size for client                 |
    |    username, password crypted with seed from server, database           |
    |    name.                                                                |
    |                                                                         |
    |   -------------------------------------------------------------->       |
    |                                                                         |
    |    'OK' packet if authentication succeeds, 'ERROR' packet if            |
    |    authentication fails.                                                |
    |                                                                         |
    |   <--------------------------------------------------------------       |
    |                                                                         |
```

```
* HANDSHAKE INITIALISATION PACKET *
```

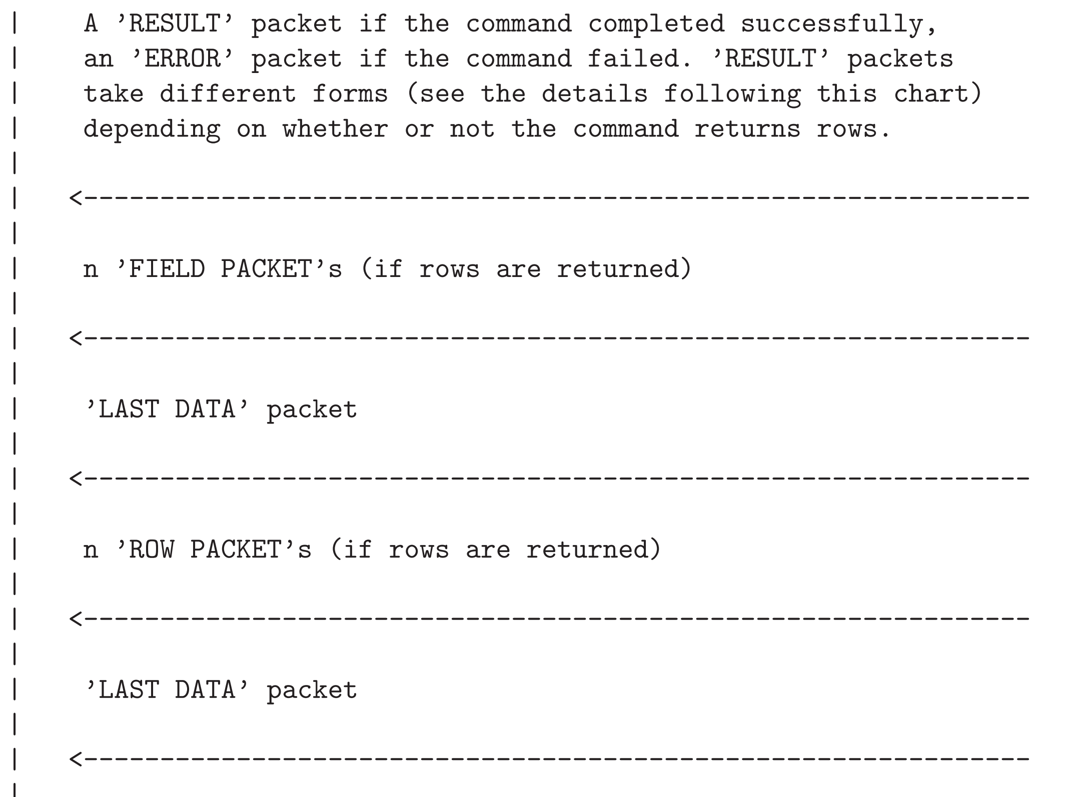
```
+---------------------------------------------------------------------+
| Header         | Prot. Version | Server Version String | 0x00       |
|                | 1 Byte        | n bytes | 1 byte     |
|---------------------------------------------------------------------|
```

```
| Thread Number | Crypt Seed | 0x00    | CLIENT_xxx options          |
|               |            |         | supported by server         |
| 4 Bytes       | 8 Bytes    | 1 Byte  | 2 Bytes                     |
|-------------------------------------------------------------------|
| Server charset no.   | Server status variables | 0x00 padding       |
| 1 Byte               | 2 Bytes                 | 13 bytes           |
+-------------------------------------------------------------------+

 * Protocol version (currently '10')
 * Server Version String (e.g. '4.0.5-beta-log').  Can be any length as
   it's followed by a 0 byte.
 * Thread Number - ID of server thread handling this connection
 * Crypt seed - seed used to crypt password in auth packet from client
 * CLIENT_xxx options - see include/mysql_com.h
 * Server charset no. - Index of charset in use by server
 * Server status variables - see include/mysql_com.h
 * The padding bytes are reserverd for future extensions to the protocol

* CLIENT AUTH PACKET *


+-------------------------------------------------------------------+
| Header        | CLIENT_xxx options supported | max_allowed_packet  |
|               | by client                    | for client          |
|               | 4 Bytes                      | 4 bytes             |
|-------------------------------------------------------------------|
| Character set | Reserved for the future      |
| 1 Bytes       | 23 bytes                                          |
|-------------------------------------------------------------------|
| User Name | 0x00   | Crypted Password | 0x00   | Database Name     |
| n Bytes   | 1 Byte | 8 Bytes          | 1 Byte | n Bytes           |
|-------------------------------------------------------------------|
| 0x00                                                              |
| 1 Byte                                                            |
+-------------------------------------------------------------------+

 * CLIENT_xxx options that this client supports:

#define CLIENT_LONG_PASSWORD    1       /* new more secure passwords */
#define CLIENT_FOUND_ROWS       2       /* Found instead of affected rows */
#define CLIENT_LONG_FLAG        4       /* Get all column flags */
#define CLIENT_CONNECT_WITH_DB  8       /* One can specify db on connect */
#define CLIENT_NO_SCHEMA        16      /* Don't allow database.table.column */
#define CLIENT_COMPRESS         32      /* Can use compression protocol */
#define CLIENT_ODBC             64      /* Odbc client */
#define CLIENT_LOCAL_FILES      128     /* Can use LOAD DATA LOCAL */
#define CLIENT_IGNORE_SPACE     256     /* Ignore spaces before '(' */
#define CLIENT_INTERACTIVE      1024    /* This is an interactive client */
#define CLIENT_SSL              2048     /* Switch to SSL after handshake */
#define CLIENT_IGNORE_SIGPIPE   4096     /* IGNORE sigpipes */
#define CLIENT_TRANSACTIONS     8192    /* Client knows about transactions */
```

```
 * max_allowed_packet for the client (in 'int3store' form)
 * User Name - user to authenticate as.  Is followed by a null byte.
 * Crypted Password - password crypted with seed given in packet from
                      server, see scramble() in sql/password.c
 * Database name (optional) - initial database to use once connected
   Is followed by a null byte

At the end of every client/server exchange there is either an 'OK' packet
or an 'ERROR' packet sent from the server. To determine whether a packet is
an 'OK' packet, or an 'ERROR' packet, check if the first byte (after the
header) is 0xFF. If it has the value of 0xFF, the packet is an 'ERROR'
packet.


* OK PACKET *

For details, see sql/net_pkg.cc::send_ok()

+-----------------------------------------------+
| Header        | No of Rows    | Affected Rows |
|               | 1 Byte        | 1-9 Byte      |
|-----------------------------------------------|
| ID (last_insert_id)   | Status | Length       |
| 1-9 Byte              | 2 Byte | 1-9 Byte     |
|-----------------------------------------------|
| Messagetext                                   |
| n Byte                                        |
+-----------------------------------------------+

 * Number of rows, always 0
 * Affected rows
 * ID (last_insert_id) - value for auto_increment column (if any)
 * Status (usually 0)

In general, in the MySQL protocol, fields in a packet that that
represent numeric data, such as lengths, that are labeled as '1-9'
bytes can be decoded by the following logic:

If the first byte is '251', the
        corresponding column value is NULL (only appropriate in
        'ROW DATA' packets).

If the first byte is '252', the value stored can be read
        from the following 2 bytes as a 16-bit integer.


        If the first byte is '253' the value stored can be read
        from the following 4 bytes as a 32-bit long integer


        If the first byte is '254', the value stored can be read
        from the following 8 bytes as a 64-byte long
```

```
Otherwise (values 0-250), the value stored is the value of the
first byte itself.


If the OK-packet includes a message:

 * Length of message
 * Message Text


* ERROR PACKET *

+-------------------------------------------------+
| Header         | Status code   | Error no        |
|                | 1 Byte        | 2 Byte          |
|-------------------------------------------------|
| Messagetext                           |         |
| n Byte                                |         |
+-------------------------------------------------+

 * Status code (0xFF = ERROR)
 * Error number (is only sent to 3.23 and newer clients)
 * Error message text (ends at end of packet)

Note that the error message is not null terminated.
The client code can however assume that the packet ends with a null
as my_net_read() will always add an end-null to all read packets to
make things easier for the client.

Example:

Packet dump of client connecting to server:

+------------------------- Protocol Version (10)
|
|  +---------------------- Server Version String (0x00 terminated)
|  |
|  |
0a 34 2e 30 2e 35 2d 62     . 4 . 0 . 5 - b
65 74 61 2d 6c 6f 67 00     e t a - l o g .
15 00 00 00 2b 5a 65 6c     . . . . + Z e l
 |          |
 |          +------------ First 4 bytes of crypt seed
 |
 +------------------------ Thread Number

+------------------------- Last 4 bytes of crypt seed
|
|                +-------- CLIENT_XXX Options supported by server
|                |
|              +-+--+ +--- Server charset index
```

```
|             |    | |
6f 69 41 46 00 2c 28 08     o i A F . , ( .
02 00 00 00 00 00 00 00     . . . . . . . .
|  |
|  +---------------------- 0x00 padding begins
|
+------------------------- Server status (0x02 =
                           SERVER_STATUS_AUTOCOMMIT)

00 00 00 00 00 00 00 00     . . . . . . . .

* Client Authentication Response (Username 'test', no database
  selected) *

    +--------------------- Packet Length (0x13 = 19 bytes)
    |
    |     +--------------- Packet Sequence #
    |     |
    |     |   +----------- CLIENT_XXX Options supported by client
          |   |
+---+---+ | +-+-+
|       | | |   |
13 00 00 01 03 00 1e 00     . . . . . . . .
00 74 65 73 74 00 48 5e     . t e s t . H ^
   |          |   |
   +----+-----+   +------- Scrambled password, 0x00 terminated
        |
        +----------------- Username, 0x00 terminated

57 4a 4e 41 4a 4e 00 00     W J N A J N . .
00                          .


>From this point on, the server waits for 'commands' from the client
which include queries, database shutdown, quit, change user, etc (see
the COM_xxxx values in include/mysql_com.h for the latest
command codes).

*                    *
* COMMAND PROCESSING *
*                    *

+--------+                                                             +--------+
| Client |                                                             | Server |
+--------+                                                             +--------+
    |                                                                       |
    |    A command packet, with a command code, and string data            |
    |    when appropriate (e.g. a query), (see the COM_xxxx values         |
    |    in include/mysql_com.h for the command codes)                     |
    |                                                                       |
    |   ----------------------------------------------------------->   |
    |                                                                       |
```



```
|    A 'RESULT' packet if the command completed successfully,         |
|    an 'ERROR' packet if the command failed. 'RESULT' packets        |
|    take different forms (see the details following this chart)      |
|    depending on whether or not the command returns rows.            |
|                                                                      |
|   <--------------------------------------------------------------  |
|                                                                      |
|    n 'FIELD PACKET's (if rows are returned)                          |
|                                                                      |
|   <--------------------------------------------------------------  |
|                                                                      |
|    'LAST DATA' packet                                                |
|                                                                      |
|   <--------------------------------------------------------------  |
|                                                                      |
|    n 'ROW PACKET's (if rows are returned)                            |
|                                                                      |
|   <--------------------------------------------------------------  |
|                                                                      |
|    'LAST DATA' packet                                                |
|                                                                      |
|   <--------------------------------------------------------------  |
|                                                                      |
```

```
* Command Packet *

+------------------------------------------------------+
| Header         | Command type | Query (if applicable) |
|                | 1 Byte       | n Bytes               |
+------------------------------------------------------+

 * Command type:  (e.g.0x03 = query, see the COM_xxxx values in
                   include/mysql_com.h)
 * Query (if applicable)

Note that my_net_read() null-terminates all packets on the
receiving side of the channel to make it easier for the code
examining the packets.

The current command codes are:

   0x00   COM_SLEEP
   0x01   COM_QUIT
   0x02   COM_INIT_DB
   0x03   COM_QUERY
   0x04   COM_FIELD_LIST
   0x05   COM_CREATE_DB
   0x06   COM_DROP_DB
   0x07   COM_REFRESH
   0x08   COM_SHUTDOWN
   0x09   COM_STATISTICS
```

```
   0x0a   COM_PROCESS_INFO
   0x0b   COM_CONNECT
   0x0c   COM_PROCESS_KILL
   0x0d   COM_DEBUG
   0x0e   COM_PING
   0x0f   COM_TIME
   0x10   COM_DELAYED_INSERT
   0x11   COM_CHANGE_USER
   0x12   COM_BINLOG_DUMP
   0x13   COM_TABLE_DUMP
   0x14   COM_CONNECT_OUT
   0x15   COM_REGISTER_SLAVE

* Result Packet *

Result packet for a command returning _no_ rows:

+-----------------------------------------------+
| Header         | Field Count   | Affected Rows |
|                | 1-9 Bytes     | 1-9 Bytes     |
|-----------------------------------------------|
| ID (last_insert_id)   | Server Status          |
| 1-9 Bytes             | 2 Bytes                |
+-----------------------------------------------+

 * Field Count: Has value of '0' for commands returning _no_ rows
 * Affected rows: Count of rows affected by INSERT/UPDATE/DELETE, etc.
 * ID: value of auto_increment column in row (if any).  0 if
 * Server Status: Usually 0

Result packet for a command returning rows:

+-------------------------------+
| Header         | Field Count   |
|                | 1-9 Bytes     |
+-------------------------------+

 * Field Count: number of columns/fields in result set,
                 (packed with net_store_length() in sql/net_pkg.cc)

This is followed by as many packets as the number of fields ('Field Count')
that contain the metadata for each column/field (see unpack_fields() in
libmysql/libmysql.c):


* FIELD PACKET *

+-----------------------------------------------+
| Header         | Table Name          |
|                | length-coded-string           |
|-----------------------------------------------|
| Field Name |
```

```
| length-code-string |
|----------------------------------------------|
| Display length of field
| length-coded-binary (4 bytes) |
|----------------------------------------------|
| Field Type (enum_field_types in mysql_com.h)  |
| length-coded-binary (2 bytes) |
|----------------------------------------------|
| Field Flags                 | Decimal Places|
| length-coded-binary (3 bytes) | 1 Byte       |
+--------------+------------+------------------+

 * A length coded string is a string where we first have a packet
   length (1-9 bytes, packed_with net_store_length()) followed
   by a string.
 * A length coded binary is a length (1 byte) followed by an integer
   value in low-byte-first order.  For the moment this type is always
   fixed length in this packet.

 * Table Name - the name of the table the column comes from
 * Field Name - the name of the column/field
 * Display length of field - length of field
 * Field Type - Type of field, see enum_field_types in
                include/mysql_com.h

   Current field types are:

      0x00   FIELD_TYPE_DECIMAL
      0x01   FIELD_TYPE_TINY
      0x02   FIELD_TYPE_SHORT
      0x03   FIELD_TYPE_LONG
      0x04   FIELD_TYPE_FLOAT
      0x05   FIELD_TYPE_DOUBLE
      0x06   FIELD_TYPE_NULL
      0x07   FIELD_TYPE_TIMESTAMP
      0x08   FIELD_TYPE_LONGLONG
      0x09   FIELD_TYPE_INT24
      0x0a   FIELD_TYPE_DATE
      0x0b   FIELD_TYPE_TIME
      0x0c   FIELD_TYPE_DATETIME
      0x0d   FIELD_TYPE_YEAR
      0x0e   FIELD_TYPE_NEWDATE
      0xf7   FIELD_TYPE_ENUM
      0xf8   FIELD_TYPE_SET
      0xf9   FIELD_TYPE_TINY_BLOB
      0xfa   FIELD_TYPE_MEDIUM_BLOB
      0xfb   FIELD_TYPE_LONG_BLOB
      0xfc   FIELD_TYPE_BLOB
      0xfd   FIELD_TYPE_VAR_STRING
      0xfe   FIELD_TYPE_STRING
      0xff   FIELD_TYPE_GEOMETRY
```

```
 * Field Flags - NOT_NULL_FLAG, PRI_KEY_FLAG, xxx_FLAG in
                 include/mysql_com.h


Note that the packet format in 4.1 has slightly changed to allow more values.


* ROW PACKET *

+-----------------------------------------------+
| Header        | Data Length   | Column Data   | ....for each column
|               | 1-9 Bytes     | n Bytes       |
+-----------------------------------------------+

 * Data Length: (packed with net_store_length() in sql/net_pkg.cc)

If 'Data Length' == 0, this is an 'ERROR PACKET'.

 * Column Data: String representation of data. MySQL always sends result set
                data as strings.

* LAST DATA PACKET *

Packet length is < 9 bytes, and first byte is 0xFE

+--------+
| 0xFE   |
| 1 Byte |
+--------+

Examples:

***********
*
* INITDB Command
*
***********

A client issuing an 'INITDB' (select the database to use) command,
followed by an 'OK' packet with no rows and no affected rows from
the server:

* INITDB (select database to use) 'COMMAND' Packet *

    +---------------------- Packet Length (5 bytes)
    |
    |     +---------------- Packet Sequence #
    |     |
    |     |  +------------- Command # (INITDB = 0x02)
          |  |
+---+---+ |  | +----------- Beginning of query data
|       | |  | |
```

```
05 00 00 00 02 74 65 73     . . . . . t e s
74                          t

* 'OK' Packet with no rows, and no rows affected *

    +--------------------- Packet Length (3 bytes)
    |
    |     +--------------- Packet Sequence #
    |     |
+---+---+ |
|       | |
03 00 00 01 00 00 00        . . . . . . .


***********
*
* SELECT query example
*
***********

Client issuing a 'SELECT *' query on the following table:

    CREATE TABLE number_test (minBigInt bigint,
                              maxBigInt bigint,
                              testBigInt bigint)

* 'COMMAND' Packet with QUERY (select ...) *

    +--------------------- Packet Length (26)
    |
    |     +--------------- Packet Sequence #
    |     |
    |     |  +------------ Command # (QUERY = 0x03)
          |  |
+---+---+ |  | +---------- Beginning of query data
|       | |  | |
1a 00 00 00 03 53 45 4c     . . . . . S E L
45 43 54 20 2a 20 66 72     E C T . * . f r
6f 6d 20 6e 75 6d 62 65     o m . n u m b e
72 5f 74 65 73 74           r _ t e s t


and receiving an 'OK' packet with a 'FIELD COUNT' of 3


* 'OK' Packet with 3 fields *

    +--------------------- Packet Length (3 bytes)
    |
    |     +--------------- Packet Sequence #
    |     |
+---+---+ |
```

```
|       | |
01 00 00 01 03              . . . . .

Followed immediately by 3 'FIELD' Packets. Note, the individual packets
are delimitted by =======, so that all fields can be annotated in the first
'FIELD' packet example:

==============================================================

    +--------------------- Packet Length (0x1f = 31 bytes)
    |
    |     +--------------- Packet Sequence #
    |     |
    |     |  +------------ Block Length (0x0b = 11 bytes)
    |     |  |
+---+---+ |  |  +--------- Table Name (11 bytes long)
|       | |  |  |
1f 00 00 02 0b 6e 75 6d     . . . . . n u m
62 65 72 5f 74 65 73 74     b e r _ t e s t

 +------------------------ Block Length (9 bytes)
 |
 |  +--------------------- Column Name (9 bytes long)
 |  |
09 6d 69 6e 42 69 67 49     . m i n B i g I
6e 74 03 14 00 00 01 08     n t . . . . . .
       | |       | |  |
       | +---+---+ |  +--- Field Type (0x08 = FIELD_TYPE_LONGLONG)
       |     |     |
       |     |     +------ Block Length (1)
       |     |
       |     +--------------- Display Length (0x14 = 20 chars)
       |
       +------------------ Block Length (3)

 +------------------------ Block Length (2)
 |
 |   +-------------------- Field Flags (0 - no flags set)
 |   |
 | +---+  +--------------- Decimal Places (0)
 | |   |  |
02 00 00 00                 . . . .

==============================================================

'FIELD' packet for the 'number_Test.maxBigInt' column

1f 00 00 03 0b 6e 75 6d     . . . . . n u m
62 65 72 5f 74 65 73 74     b e r _ t e s t
09 6d 61 78 42 69 67 49     . m a x B i g I
6e 74 03 14 00 00 01 08     n t . . . . . .
02 00 00 00                 . . . .
```

```
============================================================

'FIELD' packet for the 'number_test.testBigInt' column

20 00 00 04 0b 6e 75 6d     . . . . . n u m
62 65 72 5f 74 65 73 74     b e r _ t e s t
0a 74 65 73 74 42 69 67     . t e st B i g
49 6e 74 03 14 00 00 01     I n t . . . . .
08 02 00 00 00              . . . . .
============================================================

Followed immediately by one 'LAST DATA' packet:

fe 00                       . .

Followed immediately by 'n' row packets (in this case, only
one packet is sent from the server, for simplicity's sake):


    +--------------------- Packet Length (0x52 = 82 bytes)
    |
    |     +--------------- Packet Sequence #
    |     |
    |     |  +------------ Data Length (0x14 = 20 bytes)
    |     |  |
+---+---+ |  |  +--------- String Data '-9223372036854775808'
|       | |  |  |          (repeat Data Length/Data sequence)

52 00 00 06 14 2d 39 32     . . . . . - 9 2
32 33 33 37 32 30 33 36     2 3 3 7 2 0 3 6
38 35 34 37 37 35 38 30     8 5 4 7 7 5 8 0
38 13 39 32 32 33 33 37     8 . 9 2 2 3 3 7
32 30 33 36 38 35 34 37     2 0 3 6 8 5 4 7
37 35 38 30 37 0a 36 31     7 5 8 0 7 . 6 1
34 37 34 38 33 36 34 37     4 7 4 8 3 6 4 7

Followed immediately by one 'LAST DATA' packet:

fe 00                       . .
```

7.8 Changes to 4.0 Protocol in 4.1

All basic packet handling is identical to 4.0. When communication with an old 4.0 or 3.x client we will use the old protocol.

The new things that we support with 4.1 are:

- Warnings
- Prepared statements
- Binary protocol (is be faster than the current protocol that converts everything to strings)

What has changed in 4.1 are:

- A lot of new field information (catalog, database, real table name etc)
- The 'ok' packet has more status fields
- The 'end' packet (send last for each result set) now contains some extra information
- New protocol for prepared statements. In this case all parameters and results will sent as binary (low-byte-first).

7.9 4.1 Field Description Packet

The field description packet is sent as a response to a query that contains a result set. It can be distinguished from an OK packet by the fact that the first byte can't be 0 for a field packet. See Section 7.11 [4.1 OK packet], page 60.

The header packet has the following structure:

Size	Comment
1-9	Number of columns in result set (never 0)
1-9	Extra information sent be some command (SHOW COLUMNS uses this to send the number of rows in the table)

This packet is always followed by a field description set. See Section 7.10 [4.1 field desc], page 60.

7.10 4.1 Field Description Result Set

The field description result set contains the meta info for a result set.

Type	Comment
string	Catalog name (for 5.x)
string	Database name
string	Table name alias (or table name if no alias)
string	Real table name
string	Alias for column name (or column name if not used)
12 byte	Fixed length fields in one field part: • 2 byte int - Character set number • 4 byte int - Length of column definition • 1 byte int - Enum value for field type • 3 byte int - 2 byte column flags (NOT_NULL_FLAG etc..) + 1 byte number of decimals. • 2 byte int - zero (reserved for future use)
string int	Default value, only set when using mysql_list_fields().

7.11 4.1 OK Packet

The OK packet is the first that is sent as an response for a query that didn't return a result set.

The OK packet has the following structure:

Size	Comment
1	0 ; Marker for OK packet
1-9	Affected rows
1-9	Last insert id (0 if one wasn't used)
2	Server status; Can be used by client to check if we are inside an transaction
2	Warning count

1-9	Message length (optional)
xxx	Message (optional)

Size 1-9 means that the parameter is packed in to 1-9 bytes depending on the value. (See function sql/net_pkg.cc::net_store_length).

The message is optional. For example, for multiple-line `INSERT`, it contains a string for how many rows were inserted / deleted.

7.12 4.1 End Packet

The end packet is sent as the last packet for

- End of field information
- End of parameter type information
- End of result set

The end packet has the following structure:

Size	Comment
1	254 ; Marker for EOF packet
2	Warning count
2	Status flags (For flags like SERVER_STATUS_MORE_RESULTS)

Note that a normal packet may start with byte 254, which means 'length stored in 9 bytes'. One can different between these cases by checking the packet length `<` 9 bytes (in which case it's and end packet).

7.13 4.1 Error Packet

The error packet is sent when something goes wrong. The error packet has the following structure:

Size	Comment
1	255 Error packet marker
2	Error code
1	'#' marker that SQLSTATE follows
6	SQLSTATE code (000000 for many messages)
1-512	Null terminated error message

The client/server protocol is designed in such a way that a packet can only start with 255 if it's an error packet.

7.14 4.1 Prepared Statement Init Packet

The client issues COM_PREPARE packet with statement string to prepare. The packet has the following format:

Size	Comment
1 byte	COM_PREPARE packet code
#	Statement string

This is the server return packet to COM_PREPARE command: (packet length is always '9').

Size	Comment
1	0 ; Marker for OK packet
4	Statement handler id
2	Number of columns in result set

2	Number of parameters in query

After this, there is a packet that contains the following for each parameter in the query:

Size	Comment
2	Enum value for field type. (MYSQL_TYPE_UNKNOWN if not known)
2	2 byte column flags (NOT_NULL_FLAG etc)
1	Number of decimals
4	Max column length.

Note that the above is not yet in 4.1 but will be added this month.

As MySQL can have a parameter 'anywhere' it will in many cases not be able to provide the optimal information for all parameters.

If number of columns, in the header packet, is not 0 then the prepared statement will contain a result set. In this case the packet is followed by a field description result set. See Section 7.10 [4.1 field desc], page 60.

7.15 4.1 Long Data Handling

This is used by mysql_send_long_data() to set any parameter to a string value. One can call mysql_send_long_data() multiple times for the same parameter; The server will concatenate the results to a one big string.

The server will not require an end packet for the string. mysql_send_long_data() is responsible updating a flag that all data has been sent. (Ie; That the last call to mysql_send_long_data() has the 'last_data' flag set).

This packet is sent from client `->` server:

Size	Comment
4	Statement handler
2	Parameter number
2	Type of parameter (not used at this point)
#	data (Rest of packet)

The server will NOT send an `ok` or `error` packet in responce for this. If there is any errors (like to big string), one will get the error when calling execute.

7.16 4.1 Execute

On execute we send all parameters to the server in a COM_EXECUTE packet.

The packet contains the following information:

Size	Comment
1 byte	COM_EXECUTE packet code
4 bytes	statement id
1 byte	Flags, reserved for future use. In 5.0 is used to send 'open cursor' request to the server. In 4.1 is always 0.
4 bytes	iteration count. Reserved for future use. In 4.1 is always 1.
(param_count+7)/8	Null bit map; if parameter N is null, bit `1 << (N & 7)` of `N/8`th byte is set.
1	new_parameter_bound flag. Is set to 1 for first execute or if one has rebound the parameters.
2*param_count	Type of parameters (only given if new_parameter_bound flag is 1). Each type is encoded in first 15 bits (low-byte-first), the last bit is set if type is unsigned.

#	Parameter data, repeated for each parameter that are NOT NULL and not used with mysql_send_long_data().

The null-bit-map is for all parameters (including parameters sent with 'mysql_send_long_data). If parameter 0 is NULL, then bit 0 in the null-bit-map should be 1 (ie: first byte should be 1)

The parameters are stored the following ways:

Type	Size	Comment
tinyint	1	One byte integer
short	2	
int	4	
longlong	8	
float	4	
double	8	
string	1-9 + #	Packed string length + string

The result for this will be either an OK packet or a binary result set.

7.17 4.1 Binary Result Set

A binary result are sent the following way.

For each result row:

- null bit map with first two bits set to 01 (bit 0,1 value 1)
- parameter data, repeated for each not null result column.

The idea with the reserving two bits in the null map is that we can use standard error (first byte 255) and OK packets (first byte 0) to end a result sets.

Except that the null-bit-map is shifted two steps, the server is sending the data to the client the same way that the server is sending bound parameters to the client. The server is always sending the data as type given for 'column type' for respective column. It's up to the client to convert the parameter to the requested type.

DATETIME, DATE and TIME are sent to the server in a binary format as follows:

Type	Size	Comment
date	1 + 0-11	Length + 2 byte year, 1 byte MMDDHHMMSS, 4 byte billionth of a second
datetime	1 + 0-11	Length + 2 byte year, 1 byte MMDDHHMMSS, 4 byte billionth of a second
time	1 + 0-14	Length + sign (0 = pos, 1= neg), 4 byte days, 1 byte HHMMDD, 4 byte billionth of a second

The first byte is a length byte and then comes all parameters that are not 0. (Always counted from the beginning).

8 Replication

This chapter describes MySQL replication principles and code, as it is in version 4.1.1.

MySQL replication works like this: Every time the master executes a query that updates data (`UPDATE`, `INSERT`, `DELETE`, etc.), it packs this query into an **event**, which consists of the query plus a few bytes of information (timestamp, thread id of the thread which issued the query etc., defined later in this chapter). Then the master writes this event to a file (the "binary log"). When the slave is connected, the master re-reads its binary log and sends the events to the slaves. The slave unpacks the event and executes the query.

8.1 Main Code Files

These file are all in the '`sql`' directory:

- '`log.cc`': creating/writing/deleting a binlog.
- '`log_event.*`': all event types and their methods.
- '`slave.*`': all the slave threads' code.
- '`sql_repl.*`': all SQL commands related to replication (`START SLAVE`, `CHANGE MASTER TO`). Also all the master's high-level code about replication (binlog sending, a.k.a. `COM_BINLOG_DUMP`). For example, binlog sending code is in '`sql_repl.cc`', but uses low-level commands (single event reading) which are in '`log_event.cc`'.
- '`repl_failsafe.*`': unfinished code about failsafe (master election if the primary master fails). This file will probably be heavily reworked. Presently it's almost unused.

8.2 The Binary Log

When started with `--log-bin`, `mysqld` creates a binary log ("binlog") of all updates. Only updates that really change the data are written (a `DELETE` issued on an empty table won't be written to the binary log). Every query is written in a packed form: an event. The binary log is a sequence of **events**. The '`mysqlbinlog`' utility can be used to print human-readable data from the binary log.

```
[guilhem@gbichot2 1]$ mysqlbinlog gbichot2-bin.005
# at 4
#030710 21:55:35 server id 1  log_pos 4        Start: binlog v 3, server v 4.0.14-debu
# at 79
#030710 21:55:59 server id 1  log_pos 79       Query   thread_id=2     exec_time=16
SET TIMESTAMP=1057866959;
drop database test;
# at 128
#030710 21:56:20 server id 1  log_pos 128      Query   thread_id=2     exec_time=0
SET TIMESTAMP=1057866980;
create database test;
# at 179
#030710 21:57:00 server id 1  log_pos 179      Query   thread_id=2     exec_time=1
use test;
SET TIMESTAMP=1057867020;
create table u(a int primary key, b int, key(b), foreign key (b) references u(a));
# at 295
#030710 21:57:19 server id 1  log_pos 295      Query   thread_id=2     exec_time=0
SET TIMESTAMP=1057867039;
```

```
drop table u;
# at 342
#030710 21:57:24 server id 1  log_pos 342       Query   thread_id=2     exec_time=0
SET TIMESTAMP=1057867044;
create table u(a int primary key, b int, key(b), foreign key (b) references u(a)) type=
# at 470
#030710 21:57:52 server id 1  log_pos 470       Query   thread_id=2     exec_time=0
SET TIMESTAMP=1057867072;
insert into u values(4,NULL);
# at 533
#030710 21:57:59 server id 1  log_pos 533       Query   thread_id=2     exec_time=0
SET TIMESTAMP=1057867079;
insert into u values(3,4);
# at 593
#030710 21:58:34 server id 1  log_pos 593       Query   thread_id=4     exec_time=0
SET TIMESTAMP=1057867114;
delete from u;
# at 641
#030710 21:58:57 server id 1  log_pos 641       Query   thread_id=4     exec_time=0
SET TIMESTAMP=1057867137;
drop table u;
# at 688
#030710 21:59:18 server id 1  log_pos 688       Query   thread_id=4     exec_time=0
SET TIMESTAMP=1057867158;
create table v(c int primary key) type=innodb;
# at 768
#030710 21:59:24 server id 1  log_pos 768       Query   thread_id=4     exec_time=0
SET TIMESTAMP=1057867164;
create table u(a int primary key, b int, key(b), foreign key (b) references v(c)) type=
# at 896
#030710 21:59:47 server id 1  log_pos 896       Query   thread_id=8     exec_time=0
SET TIMESTAMP=1057867187;
DROP TABLE IF EXISTS u;
# at 953
#030710 21:59:47 server id 1  log_pos 953       Query   thread_id=8     exec_time=0
SET TIMESTAMP=1057867187;
CREATE TABLE u (
  a int(11) NOT NULL default '0',
  b int(11) default NULL,
  PRIMARY KEY  (a),
  KEY b (b),
  CONSTRAINT `0_41` FOREIGN KEY (`b`) REFERENCES `v` (`c`)
) TYPE=InnoDB;
# at 1170
#030710 21:59:47 server id 1  log_pos 1170      Query   thread_id=8     exec_time=0
SET TIMESTAMP=1057867187;
DROP TABLE IF EXISTS v;
# at 1227
#030710 21:59:47 server id 1  log_pos 1227      Query   thread_id=8     exec_time=0
SET TIMESTAMP=1057867187;
CREATE TABLE v (
  c int(11) NOT NULL default '0',
```

```
  PRIMARY KEY  (c)
) TYPE=InnoDB;
# at 1345
#030710 22:00:06 server id 1  log_pos 1345      Query   thread_id=9     exec_time=0
SET TIMESTAMP=1057867206;
drop table u,v;
# at 1394
#030710 22:00:29 server id 1  log_pos 1394      Query   thread_id=13    exec_time=0
SET TIMESTAMP=1057867229;
create table v(c int primary key) type=innodb;
# at 1474
#030710 22:00:32 server id 1  log_pos 1474      Query   thread_id=13    exec_time=0
SET TIMESTAMP=1057867232;
create table u(a int primary key, b int, key(b), foreign key (b) references v(c)) type=
# at 1602
#030710 22:00:44 server id 1  log_pos 1602      Query   thread_id=16    exec_time=0
SET TIMESTAMP=1057867244;
drop table v,u;
# at 1651
#030710 22:00:51 server id 1  log_pos 1651      Query   thread_id=16    exec_time=0
SET TIMESTAMP=1057867251;
CREATE TABLE v (
  c int(11) NOT NULL default '0',
  PRIMARY KEY  (c)
) TYPE=InnoDB;
# at 1769
#030710 22:12:50 server id 1  log_pos 1769      Stop
```

Here are the possible types of events:

```
enum Log_event_type
{
  START_EVENT = 1, QUERY_EVENT =2, STOP_EVENT=3, ROTATE_EVENT = 4,
  INTVAR_EVENT=5, LOAD_EVENT=6, SLAVE_EVENT=7, CREATE_FILE_EVENT=8,
  APPEND_BLOCK_EVENT=9, EXEC_LOAD_EVENT=10, DELETE_FILE_EVENT=11,
  NEW_LOAD_EVENT=12, RAND_EVENT=13, USER_VAR_EVENT=14
};

enum Int_event_type
{
  INVALID_INT_EVENT = 0, LAST_INSERT_ID_EVENT = 1, INSERT_ID_EVENT = 2
};
```

`START_EVENT`
: Written when `mysqld` starts.

`STOP_EVENT`
: Written when `mysqld` stops.

`QUERY_EVENT`
: Written when an updating query is done.

`ROTATE_EVENT`
: Written when `mysqld` switches to a new binary log (because someone issued `FLUSH LOGS` or the current binary log's size becomes too large. The maximum size is determined as described in Section 8.3.1 [Slave I/O thread], page 68.

`CREATE_FILE_EVENT`
: Written when a `LOAD DATA INFILE` statement starts.

`APPEND_BLOCK_EVENT`
: Written for each loaded block.

`DELETE_FILE_EVENT`
: Written if the load finally failed

`EXECUTE_LOAD_EVENT`
: Written if the load finally succeeded.

`SLAVE_EVENT`
: Not used yet.

`INTVAR_EVENT, RAND_EVENT, USER_VAR_EVENT`
: Written every time a query or `LOAD DATA` used them. They are written together with the `QUERY_EVENT` or the events written by the `LOAD DATA INFILE`. `INTVAR_EVENT` is in fact two types: `INSERT_ID_EVENT` and `LAST_INSERT_ID_EVENT`.

`INSERT_ID_EVENT`
: Used to tell the slave the value that should be used for an `auto_increment` column for the next query.

`LAST_INSERT_ID_EVENT`
: Used to tell the slave the value that should be used for the `LAST_INSERT_ID()` function if the next query uses it.

`RAND_EVENT`
: Used to tell the slave the value it should use for the `RAND()` function if the next query uses it.

`USER_VAR_EVENT`
: Used to tell the slave the value it should use for a user variable if the next query uses it.

The event's format is described in detail in Section 8.7 [Replication event format], page 74.

There is a C++ class for every type of event (`class Query_log_event` etc). Prototypes are in '`sql/log_event.h`'. Code for methods of these classes is in '`log_event.cc`'. Code to create, write, rotate, or delete a binary log is in '`log.cc`'.

8.3 Replication Threads

Every time replication is started on the slave `mysqld`, i.e. when `mysqld` is started with some replication options (`--master-host=this_hostname` etc.) or some existing `master.info` and `relay-log.info` files, or when the user does `START SLAVE` on the slave, two threads are created on the slave, in '`slave.cc`':

```
extern "C" pthread_handler_decl(handle_slave_io,arg)
{ ... }

extern "C" pthread_handler_decl(handle_slave_sql,arg)
{ ... }
```

8.3.1 The Slave I/O Thread

The I/O thread connects to the master using a user/password. When it has managed to connect, it asks the master for its binary logs:

```
thd->proc_info = "Requesting binlog dump";
if (request_dump(mysql, mi, &suppress_warnings))
```

Then it enters this loop:

```
while (!io_slave_killed(thd,mi))
{
  <cut>
  thd->proc_info = "Reading master update";
  ulong event_len = read_event(mysql, mi, &suppress_warnings);
  <cut>
  thd->proc_info = "Queueing event from master";
  if (queue_event(mi,(const char*)mysql->net.read_pos + 1,
                  event_len))
  {
    sql_print_error("Slave I/O thread could not queue event
                     from master");
    goto err;
  }
  flush_master_info(mi);
  if (mi->rli.log_space_limit && mi->rli.log_space_limit <
      mi->rli.log_space_total &&
      !mi->rli.ignore_log_space_limit)
    if (wait_for_relay_log_space(&mi->rli))
    {
      sql_print_error("Slave I/O thread aborted while
                       waiting for relay log space");
      goto err;
    }
  <cut>
}
```

`read_event()` calls `net_safe_read()` to read what the master has sent over the network. `queue_event()` writes the read event to the relay log, and also updates the counter which keeps track of the space used by all existing relay logs. `flush_master_info()` writes to the '`master.info`' file the new position up to which the thread has read in the master's binlog. Finally, if relay logs take too much space, the I/O thread blocks until the SQL thread signals it's okay to read and queue more events. The `<cut>` code handles network failures and reconnections.

When the relay log gets too large, it is "rotated": The I/O thread stops writing to it, closes it, opens a new relay log and writes to the new one. The old one is kept, until the SQL thread (see below) has finished executing it, then it is deleted. The meaning of "too large" is determined as follows:

- `max_relay_log_size`, if `max_relay_log_size` > 0
- `max_binlog_size`, if `max_relay_log_size` = 0 or MySQL is older than 4.0.14

8.3.2 The Slave SQL Thread

```
while (!sql_slave_killed(thd,rli))
{
  thd->proc_info = "Processing master log event";
  DBUG_ASSERT(rli->sql_thd == thd);
  THD_CHECK_SENTRY(thd);
```

```
        if (exec_relay_log_event(thd,rli))
        {
          // do not scare the user if SQL thread was simply killed or stopped
          if (!sql_slave_killed(thd,rli))
            sql_print_error("Error running query, slave SQL thread
                             aborted. Fix the problem, and restart
                             the slave SQL thread with "SLAVE START".
                             We stopped at log '%s' position %s",
                             RPL_LOG_NAME, llstr(rli->master_log_pos, llbuff));
          goto err;
        }
      }
```

`exec_relay_log_event()` reads an event from the relay log (by calling `next_event()`). `next_event()` will start reading the next relay log file if the current one is finished; it will also wait if there is no more relay log to read (because the I/O thread is stopped or the master has nothing more to send to the slave). Finally `exec_relay_log_event()` executes the read event (all `::exec_event()` methods in '`log_event.cc`') (mostly this execution goes through `sql_parse.cc`), thus updating the slave database and writing to '`relay-log.info`' the new position up to which it has executed in the relay log. The `::exec_event()` methods in '`log_event.cc`' will take care of filter options like `replicate-do-table` and such.

When the SQL thread hits the end of the relay log, it checks whether a new one exists (that is, whether a rotation has occurred). If so, it deletes the already-read relay log and starts reading the new one. Otherwise, it just waits until there's more data in the relay log.

8.3.3 Why 2 Threads?

In MySQL 3.23, we had only one thread on the slave, which did the whole job: read one event from the connection to the master, executed it, read another event, executed it, etc.

In MySQL 4.0.2 we split the job into two threads, using a relay log file to exchange between them.

This makes code more complicated. We have to deal with the relay log being written at the end, read at another position, at the same time. Plus handling the detection of EOF on the relay log, switching to the new relay log. Also the SQL thread must do different reads, depending on whether the relay log it is reading

- is being written to by the I/O thread; then the relay log is partly in memory, not all on disk, and mutexes are needed to avoid confusion between threads.
- has already been rotated (the I/O thread is not writing to it anymore), in which case it is a normal file that no other threads touches.

The advantages of having 2 threads instead of one:

- **Helps having a more up-to-date slave**. Reading a query is fast, executing it is slow. If the master dies (burns), there are good chances that the I/O thread has caught almost all updates issued on the master, and saved them in the relay log, for use by the SQL thread.
- **Reduces the required master-slave connection time**. If the slave has not been connected for a long time, it is very late compared to the master. It means the SQL thread will have a lot of executing to do. So with the single-thread read-execute-read-execute technique, the slave will have to be connected for a long time to be able to fetch all updates from the master. Which is stupid, as for a significant part of the time, the connection will be idle, because the single thread is busy executing the query. Whereas with 2 threads, the I/O thread will fetch the binlogs from the master in a shorter time. Then the connection is not needed anymore, and the SQL thread can continue executing the relay log.

8.3.4 The Binlog Dump Thread

This thread is created by the master when it receives a COM_BINLOG_DUMP request.

```
void mysql_binlog_send(THD* thd, char* log_ident, my_off_t pos,
                       ushort flags)
{
  <cut>
  if ((file=open_binlog(&log, log_file_name, &errmsg)) < 0)
  {
    my_errno= ER_MASTER_FATAL_ERROR_READING_BINLOG;
    goto err;
  }
  if (pos < BIN_LOG_HEADER_SIZE || pos > my_b_filelength(&log))
  {
    errmsg= "Client requested master to start replication from
             impossible position";
    my_errno= ER_MASTER_FATAL_ERROR_READING_BINLOG;
    goto err;
  }

  my_b_seek(&log, pos);                 // Seek will done on next read
  <cut>
  // if we are at the start of the log
  if (pos == BIN_LOG_HEADER_SIZE)
  {
    // tell the client log name with a fake rotate_event
    if (fake_rotate_event(net, packet, log_file_name, &errmsg))
    {
      my_errno= ER_MASTER_FATAL_ERROR_READING_BINLOG;
      goto err;
    }
    <cut>
  }

  while (!net->error && net->vio != 0 && !thd->killed)
  {
    pthread_mutex_t *log_lock = mysql_bin_log.get_log_lock();

    while (!(error = Log_event::read_log_event(&log, packet, log_lock)))
    {
      <cut>
      if (my_net_write(net, (char*)packet->ptr(), packet->length()) )
      {
        errmsg = "Failed on my_net_write()";
        my_errno= ER_UNKNOWN_ERROR;
        goto err;
      }
      <cut>
```

If this thread starts reading from the beginning of a binlog, it is possible that the slave does not know the binlog's name (for example it could have just asked "give me the FIRST binlog"). Using `fake_rotate_event()`, the master tells the slave the binlog's name (required for 'master.info'

and SHOW SLAVE STATUS) by building a Rotate_log_event and sending this event to the slave. In this event the slave will find the binlog's name. This event has zeros in the timestamp (shows up as written in year "1970" when reading the relay log with mysqlbinlog).

8.4 How Replication Deals With...

This section describes how replication handles various problematic issues.

8.4.1 auto_increment Columns, LAST_INSERT_ID()

When a query inserts into such a column, or uses LAST_INSERT_ID(), one or two Intvar_log_event are written to the binlog just before the Query_log_event.

8.4.2 User Variables (Since 4.1)

When a query uses a user variable, a User_var_log_event is written to the binlog just before the Query_log_event.

8.4.3 System Variables

Example: SQL_MODE, FOREIGN_KEY_CHECKS. Not dealt with. Guilhem is working on it for version 5.0.

8.4.4 Some Functions

USER(), LOAD_FILE(). Not dealt with. Will be solved with row-level binlogging (presently we have query-level binlogging, but in the future we plan to support row-level binlogging too).

8.4.5 Non-repeatable UDF Functions

"Non repeatable" means that they have a sort of randomness, for example they depend on the machine (to generate a unique ID for example). Not dealt with. Will be solved with row-level binlogging.

8.4.6 Prepared Statements

For the moment, a substituted normal query is written to the master's binlog. Using prepared statements on the slave as well is on the TODO.

8.4.7 Temporary Tables

Temporary tables depend on the thread which created them, so any query event which uses such tables is marked with the LOG_EVENT_THREAD_SPECIFIC_F flag. All events have in their header the id of the thread which created them, so the slave knows which temporary table the query refers to.

When the slave is stopped (STOP SLAVE or even mysqladmin shutdown), the in-use replicated temporary tables are not dropped (like clients' temporary tables are). This way, when the slave restarts they are still available.

When a connection using temporary tables terminates on the master, the master automatically writes some DROP TEMPORARY TABLE statements for them so that they are dropped on the slave as well.

When the master brutally dies, then restarts, it drops all temporary tables which remained in tmpdir, but without writing it to the binlog, so these temporary tables are still on the slave, and they will not be dropped before the next slave's mysqld restart. To avoid this, the slave drops all replicated temporary tables when it executes a Start_log_event read from the master. Indeed such an event means the master's mysqld has restarted so all preceding temporary tables have been dropped.

Presently we have a bug: if the slave mysqld is stopped while it was replicating a temporary table, then at restart it deletes this table (like a normal temporary table), which may cause a problem if subsequent queries on the master refer to this table.

8.4.8 LOAD DATA [LOCAL] INFILE (Since 4.0)

The master writes the loaded file to the binlog, but in small blocks rather than all at once. The slave creates a temporary file, the concatenation of each block. When the slave reads the final Execute_load_log_event, it loads all temporary files into the table and deletes the temporary files. If the final event was instead a Delete_file_log_event then these temporary files are deleted without loading.

8.5 How a Slave Asks Its Master to Send Its Binary Log

The slave server must open a normal connection to its master. The MySQL account used to connect to the master must have the REPLICATION SLAVE privilege on the master. Then the slave must send the COM_BINLOG_DUMP command, as in this example taken from function request_dump():

```
static int request_dump(MYSQL* mysql, MASTER_INFO* mi,
                        bool *suppress_warnings)
{
  char buf[FN_REFLEN + 10];
  int len;
  int binlog_flags = 0; // for now
  char* logname = mi->master_log_name;
  DBUG_ENTER("request_dump");

  // TODO if big log files: Change next to int8store()
  int4store(buf, (longlong) mi->master_log_pos);
  int2store(buf + 4, binlog_flags);
  int4store(buf + 6, server_id);
  len = (uint) strlen(logname);
  memcpy(buf + 10, logname,len);
  if (simple_command(mysql, COM_BINLOG_DUMP, buf, len + 10, 1))
  {
     // act on errors
  }
}
```

Here variable buf contains the arguments for COM_BINLOG_DUMP. It's the concatenation of:

- 4 bytes: the position in the master's binlog from which we want to start (i.e. "please master send me the binlog, starting from this position").

- 2 bytes: 0 for the moment.
- 4 bytes: this slave's server id. This is used by the master to delete old `Binlog Dump` threads which were related to this slave (see function `kill_zombie_dump_threads()` for details).
- variable-sized part: the name of the binlog we want. The dump will start from this binlog, at the position indicated in the first four bytes.

Then send the command, and start reading the incoming packets from the master, like `read_event()` does (using `net_safe_read()` like explained below). One should also, to be safe, handle all possible cases of network problems, disconnections/reconnections, malformed events.

8.6 Network Packets in Detail

The communication protocol between the master and slave is the one that all other normal connections use, as described earlier in this document. See Chapter 7 [Client/Server Protocol], page 41. So after the `COM_BINLOG_DUMP` command has been sent, the communication between the master and slave is a sequence of packets, each of which contains an event. In `slave.cc`, function `read_event()`, one has an example: `net_safe_read()` is called; it is able to detect wrong packets. After `net_safe_read()`, the event is ready to be interpreted; it starts at pointer `(char*) mysql->net.read_pos + 1`. That is, `(char*) mysql->net.read_pos + 1` is the first byte of the event's timestamp, etc.

8.7 Replication Event Format in Detail

8.7.1 The Common Header

Each event starts with a header of size `LOG_EVENT_HEADER_LEN=19` (defined in '`log_event.h`'), which contains:

timestamp
: 4 bytes, seconds since 1970.

event type
: 1 byte. 1 means `START_EVENT`, 2 means `QUERY_EVENT`, etc (these numbers are defined in an `enum Log_event_type` in '`log_event.h`').

server ID
: 4 bytes. The server ID of the `mysqld` which created this event. When using circular replication (with option `--log-slave-updates` on), we use this server ID to avoid endless loops. Suppose tthat M1, M2, and M3 have server ID values of 1, 2, and 3, and that they are replicating in circular fashion: M1 is the master for M2, M2 is the master for M3, and M3 is that master for M1. The master/server relationships look like this:

```
M1---->M2
^      |
|      |
+--M3<-+
```

A client sends an `INSERT` query to M1. Then this is executed on M1, then written in the binary log of M1, and the event's server ID is 1. The event is sent to M2, which executes it and writes it to the binary log of M2; the event written still has server ID 1 (because that is the ID of the server that originally created the event). The event is sent to M3, which executes it and writes it to the binary log of M3, with server ID 1. This last event is sent to M1, which sees "server ID = 1" and understands this event comes from itself, so has to be ignored.

event total size

4 bytes. Size of this event in bytes. Most events are 10-1000 bytes, except when using `LOAD DATA INFILE` (where events contain the loaded file, so they can be big).

position of the event in the binary log

4 bytes. Offset in bytes of the event in the binary log, as returned by `tell()`. It is the offset in the binary log where this event was created `in the first place`. That is, it is copied as-is to the relay log. It is used on the slave, for `SHOW SLAVE STATUS` to be able to show coordinates of the last executed event **in the master's coordinate system**. If this value were not stored in the event, we could not know these coordinates because the slave cannot invoke `tell()` for the master's binary log.

flags 2 bytes of flags. Almost unused for now. The only one which is used in 4.1 is `LOG_EVENT_THREAD_SPECIFIC_F`, which is used only by '`mysqlbinlog`' (not by the replication code at all) to be able to deal properly with temporary tables. '`mysqlbinlog`' prints queries from the binary log, so that one can feed these queries into '`mysql`' (the command-line interpreter), to achieve incremental backup recovery. But if the binary log looks like this:

```
<thread id 1>
create temporary table t(a int);
<thread id 2>
create temporary table t(a int)
```

(two simultaneous threads used temporary tables with the same name, which is allowed as temporary tables are visible only in the thread which created them), then simply feeding this into '`mysql`' will lead to the "table t already exists" error. This is why events which use temporary tables are marked with the flag, so that '`mysqlbinlog`' knows it has to set the pseudo_thread_id before, like this:

```
SET PSEUDO_THREAD_ID=1;
create temporary table t(a int);
SET PSEUDO_THREAD_ID=2;
create temporary table t(a int);
```

This way there is no confusion for the server which receives these queries. Always printing `SET PSEUDO_THREAD_ID`, even when temporary tables are not used, would cause no bug, it would just slow down.

8.7.2 The "Post-headers" (Event-specific Headers)

After the common header, each event has an event-specific header of fixed size (0 or more bytes) and a variable-sized part (0 or more bytes). It's easy for the slave to know the size of the variable-sized part: it is the event's size (contained in the common header) minus the size of the common header, minus the size of the event-specific header.

`START_EVENT`

In MySQL 4.0 and 4.1, such events are written only for the first binary log since `mysqld` startup. Binlogs created afterwards (by `FLUSH LOGS`) do not contain this event. In MySQL 5.0 we will change this; all binary logs will start with a `START_EVENT`, but there will be a way to distinguish between a `START_EVENT` created at `mysqld` startup and other `START_EVENT`s; such distinction is needed because the first category of `START_EVENT`, which means the master has started, should trigger some cleaning tasks on the slave (suppose the master died brutally and restarted: the slave must delete old replicated temporary tables).

- 2 bytes: The binary log format version. This is 3 in MySQL 4.0 and 4.1; it will be 4 in MySQL 5.0.
- 50 bytes: The MySQL server's version (example: 4.0.14-debug-log).
- 4 bytes: Timestamp in seconds when this event was created (this is the moment when the binary log was created). In fact this is useless information as we already have the timestamp in the common header, so this useless timestamp should NOT be used, because we plan to change its meaning soon.
- No variable-sized part.

`QUERY_EVENT`

- 4 bytes: The thread ID of the thread that issued this query. Needed for temporary tables. This is also useful for a DBA for knowing who did what on the master.
- 4 bytes: The time in seconds which the query took for execution. Only useful for inspection by the DBA.
- 1 byte: The length of the name of the database which was the default database when the query was executed (later in the event we store this name; this is necessary for queries like `INSERT INTO t VALUES(1)` which don't specify the database, relying on the default database previously selected by `USE`).
- 2 bytes: The error code which the query got on the master. Error codes are defined in '`include/mysqld_error.h`'. 0 means no error. How come queries with a non-zero error code can exist in the binary log? This is mainly due to the non-transactional nature of `MyISAM` tables. If an `INSERT SELECT` fails after inserting 1000 rows (for example, with a duplicate-key violation), then we have to write this query to the binary log, because it truly modified the `MyISAM` table. For transactional tables, there should be no event with a non-zero error code (though it can happen, for example if the connection was interrupted (Control-C)). The slave checks the error code: After executing the query itself, it compares the error code it got with the error code in the event, and if they are different it stops replicating (unless `--slave-skip-errors` was used).
- Variable-sized part: The concatenation of the name of the default database (null-terminated) and the query. The slave knows the size of the name of the default database (it's in the event-specific header) so by subtraction it can know the size of the query.

`STOP_EVENT`

No event-specific header, no variable-sized part. It just means "Stop" and the event's type says it all. This event is purely for informational purposes, it is not even queued into the relay log.

`ROTATE_EVENT`

This event is information for the slave to know the name of the next binary log it is going to receive.

- 8 bytes: Useless, alway contains the number 4 (meaning the next event starts at position 4 in the next binary log).
- variable-sized part: The name of the next binary log.

`INTVAR_EVENT`

- 8 bytes: the value to be used for the `auto_increment` counter or `LAST_INSERT_ID()`. 8 bytes corresponds to the size of MySQL's `BIGINT` type.
- No variable-sized part.

`LOAD_EVENT`
This is an event for internal use. One should only need to be able to read `CREATE_FILE_EVENT` (see below).

`SLAVE_EVENT`
This event is never written so it cannot exist in a binlog. It was meant for failsafe replication which will be reworked.

`CREATE_FILE_EVENT`
`LOAD DATA INFILE` is not written to the binlog like other queries; it is written in the form of a `CREATE_FILE_EVENT`; the command does not appear in clear-text in the binlog, it's in a packed format. This event tells the slave to create a temporary file and fill it with a first data block. Later, zero or more `APPEND_BLOCK_EVENT` events append blocks to this temporary file. `EXEC_LOAD_EVENT` tells the slave to load the temporary file into the table, or `DELETE_FILE_EVENT` tells the slave not to do the load and to delete the temporary file. `DELETE_FILE_EVENT` occurs is when the `LOAD DATA` failed on the master: on the master we start to write loaded blocks to the binlog before the end of the command. If for some reason there is an error, we have to tell the slave to abort the load. The format for this event is more complicated than for others, because the command has many options. Unlike other events, fixed headers and variable-sized parts are intermixed; this is due to the history of the `LOAD DATA INFILE` command.

- 4 bytes: The thread ID of the thread that issued this `LOAD DATA INFILE`. Needed for temporary tables. This is also useful for a DBA for knowing who did what on the master.
- 4 bytes: The time in seconds which the `LOAD DATA INFILE` took for execution. Only useful for inspection by the DBA.
- 4 bytes: The number of lines to skip at the beginning of the file (option `IGNORE number LINES` of `LOAD DATA INFILE`).
- 1 byte: The size of the name of the table which is to be loaded.
- 1 byte: The size of the name of the database where this table is.
- 4 bytes: The number of columns to be loaded (option `(col_name,...)`). Will be non-zero only if the columns to load were explicitly mentioned in the command.
- 4 bytes: An ID for this file (1, 2, 3, etc). This is necessary in case several `LOAD DATA INFILE` commands have been run in parallel on the master: in that case the binlog contains events for the first command and for the second command intermixed; the ID is used to resolve to which file the blocks in `APPEND_BLOCK_EVENT` must be appended, and which file must be loaded by the `EXEC_LOAD_EVENT` event, and which file must be deleted by the `DELETE_FILE_EVENT`.
- 1 byte: The size of the field-terminating string (`FIELDS TERMINATED BY` option).
- variable-sized part: The field-terminating string (null-terminated).
- 1 byte: The size of the field-enclosing string (`FIELDS ENCLOSED BY` option).
- variable-sized part: The field-enclosing string (null-terminated).
- 1 byte: The size of the line-terminating string (`LINES TERMINATED BY` option).
- variable-sized part: The line-terminating string (null-terminated).
- 1 byte: The size of the line-starting string (`LINES STARTING BY` option).
- variable-sized part: The line-starting string (null-terminated).
- 1 byte: The size of the escaping string (`FIELDS ESCAPED BY` option).
- variable-sized part: The escaping string (null-terminated).

- 1 byte: Flags: `OPT_ENCLOSED_FLAG` (`FIELD OPTIONALLY ENCLOSED BY` option), `REPLACE_FLAG` (`LOAD DATA INFILE REPLACE`), `IGNORE_FLAG` (`LOAD DATA INFILE IGNORE`), `DUMPFILE_FLAG` (unused). All these are defined in '`log_event.h`'.
- 1 byte: The size of the name of the first column to load.
- etc
- 1 byte: The size of the name of the last column to load.
- Variable-sized part: The name of the first column to load (null-terminated).
- etc
- Variable-sized part: The name of the last column to load (null-terminated).
- Variable-sized part: The name of the table which is to be loaded (null-terminated).
- Variable-sized part: The name of the database containing the table (null-terminated).
- Variable-sized part: The name of the file which was loaded (that's the original name, from the master) (null-terminated).
- Variable-sized part: The block of raw data to load.

Here is a concrete example:

```
On the master we have file '/m/tmp/u.txt' which contains:
>1,2,3
>4,5,6
>7,8,9
>10,11,12

And we issue this command on the master:
load data infile '/m/tmp/u.txt' replace into table x fields
terminated by ',' optionally enclosed by '"' escaped by '\\'
lines starting by '>' terminated by '\n' ignore 2 lines (a,b,c);

Then in the master's binlog we have this event (hexadecimal dump):
00000180:                     db4f 153f 0801 0000  .........O.?....
00000190: 006f 0000 0088 0100 0000 0004 0000 0000  .o..............
000001a0: 0000 0002 0000 0001 0403 0000 0003 0000  ................
000001b0: 0001 2c01 2201 0a01 3e01 5c06 0101 0161  ..,."...>.\....a
000001c0: 0062 0063 0078 0074 6573 7400 2f6d 2f74  .b.c.x.test./m/t
000001d0: 6d70 2f75 2e74 7874 003e 312c 322c 330a  mp/u.txt.>1,2,3.
000001e0: 3e34 2c35 2c36 0a3e 372c 382c 390a 3e31  >4,5,6.>7,8,9.>1
000001f0: 302c 3131 2c31 32db 4f15 3f0a 0100 0000  0,11,12.O.?.....
00000200: 1700 0000 f701 0000 0000 0300 0000       .............
```

- Line 180: timestamp db4f153f, event's type (08), server id (01 0000 00).
- Line 190: event's size (6f 0000 00), position in the binlog (88 0100 00) (that's 392 in decimal base), flags (00 00), thread id (04 0000 00), time it took (00 0000 00).
- Line 1a0: number of lines to skip at the beginning of the file (02 0000 00), size of the table's name (01), size of the database's name (04), number of columns to load (03 0000 00), the file's id (03 0000 00).
- Line 1b0: size of the field terminating string (01), field terminating string (2c i.e. ,), size of the field enclosing string (01), field enclosing string (22 i.e. `"`), size of the line terminating string (01), line terminating string (0a i.e. newline),

size of the line starting string (01), line starting string (3e i.e. `>`), size of the escaping string (01), escaping string (5c i.e. backslash), flags (06) (that's `OPT_ENCLOSED_FLAG | REPLACE_FLAG`), size of the name of the first column to load (01), size of the name of the second column to load (01), size of the name of the third column to load (01), name of the first column to load (61 00 i.e. `"a"`).

- Line 1c0: name of the second column to load (62 00), name of the third column to load (63 00), name of the table to load (78 00), name of the database to load (74 6573 7400), name of the file loaded on the master (2f6d 2f74 6d70 2f75 2e74 7874 00).
- Line 1d0 and following: raw data to load (3e 312c 322c 330a 3e34 2c35 2c36 0a3e 372c 382c 390a 3e31 302c 3131 2c31 32). The next byte is the beginning of the `EXEC_LOAD_EVENT` event.

`APPEND_BLOCK_EVENT`
- 4 bytes: The ID of the file this block should be appended to.
- Variable-sized part: the raw data to load.

`EXEC_LOAD_EVENT`
- 4 bytes: the ID of the file to be loaded.
- No variable-sized part.

`DELETE_FILE_EVENT`
- 4 bytes: The ID of the file to be deleted.
- No variable-sized part.

`NEW_LOAD_EVENT`

For internal use.

`RAND_EVENT`

`RAND()` in MySQL uses 2 seeds to compute the random number.
- 8 bytes: Value for the first seed.
- 8 bytes:
  Value for the second seed.
- No variable-sized part.

`USER_VAR_EVENT`
- 4 bytes: the size of the name of the user variable.
- variable-sized part: A concatenation. First is the name of the user variable. Second is one byte, non-zero if the content of the variable is the SQL value `NULL`, ASCII 0 otherwise. If this bytes was ASCII 0, then the following parts exist in the event. Third is one byte, the type of the user variable, which corresponds to elements of `enum Item_result` defined in '`include/mysql_com.h`'. Fourth is 4 bytes, the number of the character set of the user variable (needed for a string variable). Fifth is 4 bytes, the size of the user variable's value (corresponds to member `val_len` of class `Item_string`). Sixth is variable-sized: for a string variable it is the string, for a float or integer variable it is its value in 8 bytes.

8.8 Plans for MySQL 5.0

We plan (Guilhem is presently working on it) to add more information in the events. For example, the number of rows the query modified. We also plan to have a more dynamic binlog format, i.e. if we need to add 1 more byte in the header, replication between an old and a new server would still be possible.

9 MyISAM Record Structure

9.1 Introduction

When you say:

```
CREATE TABLE Table1 ...
```

MySQL creates files named 'Table1.MYD' ("MySQL Data"), 'Table1.MYI' ("MySQL Index"), and 'Table1.frm' ("Format"). These files will be in the directory:

```
/<datadir>/<database>/
```

For example, if you use Linux, you might find the files in the '/usr/local/var/test' directory (assuming your database name is test). if you use Windows, you might find the files in the '\mysql\data\test\' directory.

Let's look at the '.MYD' Data file (MyISAM SQL Data file) more closely. There are three possible formats — fixed, dynamic, and packed. First, let's discuss the fixed format.

Page Size Unlike most DBMSs, MySQL doesn't store on disk using pages. Therefore you will not see filler space between rows. (Reminder: This does not refer to BDB and InnoDB tables, which do use pages).

Record Header

The minimal record header is a set of flags:

- "X bit" = 0 if row is deleted, = 1 if row is not deleted
- "Null Bits" = 0 if column is not NULL, = 1 if column is NULL
- "Filler Bits" = 1

The length of the record header is thus:

```
(1 + number of NULL columns + 7) / 8 bytes
```

After the header, all columns are stored in the order that they were created, which is the same order that you would get from SHOW COLUMNS.

Here's an example. Suppose you say:

```
CREATE TABLE Table1 (column1 CHAR(1), column2 CHAR(1), column3 CHAR(1));
INSERT INTO Table1 VALUES ('a', 'b', 'c');
INSERT INTO Table1 VALUES ('d', NULL, 'e');
```

A CHAR(1) column takes precisely one byte (plus one bit of overhead that is assigned to every column — I'll describe the details of column storage later). So the file 'Table1.MYD' looks like this:

Hexadecimal Display of 'Table1.MYD' file

```
F1 61 62 63 00 F5 64 00 66 00                 ... .abc..d e.
```

Here's how to read this hexadecimal-dump display:

- The hexadecimal numbers F1 61 62 63 00 F5 64 20 66 00 are byte values and the column on the right is an attempt to show the same bytes in ASCII.
- The F1 byte means that there are no null fields in the first row.
- The F5 byte means that the second column of the second row is NULL.

(It's probably easier to understand the flag setting if you restate F5 as 11110101 binary, and (a) notice that the third flag bit from the right is on, and (b) remember that the first flag bit is the X bit.)

There are complications — the record header is more complex if there are variable-length fields — but the simple display shown in the example is exactly what you'd see if you looked at the MySQL Data file with a debugger or a hexadecimal file dumper.

So much for the fixed format. Now, let's discuss the dynamic format.

The dynamic file format is necessary if rows can vary in size. That will be the case if there are `BLOB` columns, or "true" `VARCHAR` columns. (Remember that MySQL may treat `VARCHAR` columns as if they're `CHAR` columns, in which case the fixed format is used.) A dynamic row has more fields in the header. The important ones are "the actual length", "the unused length", and "the overflow pointer". The actual length is the total number of bytes in all the columns. The unused length is the total number of bytes between one physical record and the next one. The overflow pointer is the location of the rest of the record if there are multiple parts.

For example, here is a dynamic row:

```
03, 00              start of header
04                  actual length
0c                  unused length
01, fc              flags + overflow pointer
****                data in the row
************        unused bytes
                    <-- next row starts here)
```

In the example, the actual length and the unused length are short (one byte each) because the table definition says that the columns are short — if the columns were potentially large, then the actual length and the unused length could be two bytes each, three bytes each, and so on. In this case, actual length plus unused length is 10 hexadecimal (sixteen decimal), which is a minimum.

As for the third format — packed — we will only say briefly that:

- Numeric values are stored in a form that depends on the range (start/end values) for the data type.
- All columns are packed using either Huffman or enum coding.

For details, see the source files '`/myisam/mi_statrec.c`' (for fixed format), '`/myisam/mi_dynrec.c`' (for dynamic format), and '`/myisam/mi_packrec.c`' (for packed format).

Note: Internally, MySQL uses a format much like the fixed format which it uses for disk storage. The main differences are:

1. `BLOB` values have a length and a memory pointer rather than being stored inline.
2. "True `VARCHAR`" (a column storage which will be fully implemented in version 5.0) will have a 16-bit length plus the data.
3. All integer or floating-point numbers are stored with the low byte first. Point (3) does not apply for `ISAM` storage or internals.

9.2 Physical Attributes of Columns

Next I'll describe the physical attributes of each column in a row. The format depends entirely on the data type and the size of the column, so, for every data type, I'll give a description and an example.

The character data types

`CHAR`

- Storage: fixed-length string with space padding on the right.

- Example: a `CHAR(5)` column containing the value `'A'` looks like:
  `hexadecimal` 41 20 20 20 20 – (length = 5, value = `'A    '`)

`VARCHAR`

- Storage: variable-length string with a preceding length.
- Example: a `VARCHAR(7)` column containing `'A'` looks like:
  `hexadecimal` 01 41 – (length = 1, value = `'A'`)

The numeric data types

Important: MySQL almost always stores multi-byte binary numbers with the low byte first. This is called "little-endian" numeric storage; it's normal on Intel x86 machines; MySQL uses it even for non-Intel machines so that databases will be portable.

`TINYINT`

- Storage: fixed-length binary, always one byte.
- Example: a `TINYINT` column containing `65` looks like:
  `hexadecimal` 41 – (length = 1, value = 65)

`SMALLINT`

- Storage: fixed-length binary, always two bytes.
- Example: a `SMALLINT` column containing `65` looks like:
  `hexadecimal` 41 00 – (length = 2, value = 65)

`MEDIUMINT`

- Storage: fixed-length binary, always three bytes.
- Example: a `MEDIUMINT` column containing `65` looks like:
  `hexadecimal` 41 00 00 – (length = 3, value = 65)

`INT`

- Storage: fixed-length binary, always four bytes.
- Example: an `INT` column containing `65` looks like:
  `hexadecimal` 41 00 00 00 – (length = 4, value = 65)

`BIGINT`

- Storage: fixed-length binary, always eight bytes.
- Example: a `BIGINT` column containing `65` looks like:
  `hexadecimal` 41 00 00 00 00 00 00 00 – (length = 8, value = 65)

`FLOAT`

- Storage: fixed-length binary, always four bytes.
- Example: a `FLOAT` column containing approximately `65` looks like:
  `hexadecimal` 00 00 82 42 – (length = 4, value = 65)

`DOUBLE PRECISION`

- Storage: fixed-length binary, always eight bytes.
- Example: a `DOUBLE PRECISION` column containing approximately `65` looks like:
  `hexadecimal` 00 00 00 00 00 40 50 40 – (length = 8, value = 65)

`REAL`

- Storage: same as `FLOAT`, or same as `DOUBLE PRECISION`, depending on the setting of the `--ansi` option.

`DECIMAL`

- Storage: fixed-length string, with a leading byte for the sign, if any.
- Example: a `DECIMAL(2)` column containing `65` looks like:
  `hexadecimal 20 36 35` – (length = 3, value = `' 65'`)
- Example: a `DECIMAL(2) UNSIGNED` column containing `65` looks like:
  `hexadecimal 36 35` – (length = 2, value = `'65'`)
- Example: a `DECIMAL(4,2) UNSIGNED` column containing `65` looks like:
  `hexadecimal 36 35 2E 30 30` – (length = 5, value = `'65.00'`)

`NUMERIC`

- Storage: same as `DECIMAL`.

`BOOL`

- Storage: same as `TINYINT`.

The temporal data types

`DATE`

- Storage: 3 byte integer, low byte first. Packed as: 'day + month*32 + year*16*32'
- Example: a `DATE` column containing `'1962-01-02'` looks like:
  `hexadecimal 22 54 0F`

`DATETIME`

- Storage: eight bytes.
- Part 1 is a 32-bit integer containing year*10000 + month*100 + day.
- Part 2 is a 32-bit integer containing hour*10000 + minute*100 + second.
- Example: a `DATETIME` column for `'0001-01-01 01:01:01'` looks like:
  `hexadecimal B5 2E 11 5A 02 00 00 00`

`TIME`

- Storage: 3 bytes, low byte first. This is stored as seconds: days*24*3600+hours*3600+minutes*60+seconds
- Example: a `TIME` column containing `'1 02:03:04'` (1 day 2 hour 3 minutes and 4 seconds) looks like:
  `hexadecimal 58 6E 01`

`TIMESTAMP`

- Storage: 4 bytes, low byte first. Stored as unix `time()`, which is seconds since the Epoch (00:00:00 UTC, January 1, 1970).
- Example: a `TIMESTAMP` column containing `'2003-01-01 01:01:01'` looks like:
  `hexadecimal 4D AE 12 23`

`YEAR`

- Storage: same as unsigned `TINYINT` with a base value of 0 = 1901.

Others

`SET`

- Storage: one byte for each eight members in the set.
- Maximum length: eight bytes (for maximum 64 members).
- This is a bit list. The least significant bit corresponds to the first listed member of the set.
- Example: a `SET('A','B','C')` column containing `'A'` looks like:
  `01` – (length = 1, value = 'A')

ENUM

- Storage: one byte if less than 256 alternatives, else two bytes.
- This is an index. The value 1 corresponds to the first listed alternative. (Note: ENUM always reserves the value 0 for an erroneous value. This explains why 'A' is 1 instead of 0.)
- Example: an ENUM('A','B','C') column containing 'A' looks like:
  01 – (length = 1, value = 'A')

The Large-Object data types

Warning: Because TINYBLOB's preceding length is one byte long (the size of a TINYINT) and MEDIUMBLOB's preceding length is three bytes long (the size of a MEDIUMINT), it's easy to think there's some sort of correspondence between the the BLOB and INT types. There isn't — a BLOB's preceding length is not four bytes long (the size of an INT).

TINYBLOB

- Storage: variable-length string with a preceding one-byte length.
- Example: a TINYBLOB column containing 'A' looks like:
  hexadecimal 01 41 – (length = 2, value = 'A')

TINYTEXT

- Storage: same as TINYBLOB.

BLOB

- Storage: variable-length string with a preceding two-byte length.
- Example: a BLOB column containing 'A' looks like:
  hexadecimal 01 00 41 – (length = 2, value = 'A')

TEXT

- Storage: same as BLOB.

MEDIUMBLOB

- Storage: variable-length string with a preceding length.
- Example: a MEDIUMBLOB column containing 'A' looks like:
  hexadecimal 01 00 00 41 – (length = 4, value = 'A')

MEDIUMTEXT

- Storage: same as MEDIUMBLOB.

LONGBLOB

- Storage: variable-length string with a preceding four-byte length.
- Example: a LONGBLOB column containing 'A' looks like:
  hexadecimal 01 00 00 00 41 – (length = 5, value = 'A')

LONGTEXT

- Storage: same as LONGBLOB.

9.3 Where to Look For More Information

References:

Most of the formatting work for MyISAM columns is visible in the program '/sql/field.cc' in the source code directory. And in the MyISAM directory, the files that do formatting work for different record formats are: '/myisam/mi_statrec.c', '/myisam/mi_dynrec.c', and '/myisam/mi_packrec.c'.

10 The '.MYI' file

A '.MYI' file for a MyISAM table contains the table's indexes.

The '.MYI' file has two parts: the header information and the key values. So the next sub-sections will be "The '.MYI' Header" and "The '.MYI' Key Values".

The '.MYI' Header

A '.MYI' file begins with a header, with information about options, about file sizes, and about the "keys". In MySQL terminology, a "key" is something that you create with CREATE [UNIQUE] INDEX.

Program files which read and write '.MYI' headers are in the './myisam' directory: 'mi_open.c' has the routines that write each section of the header, 'mi_create.c' has a routine that calls the 'mi_open.c' routines in order, and 'myisamdef.h' has structure definitions corresponding to what we're about to describe.

These are the main header sections:

```
Section                        Occurrences
-------                        -----------
state                          Occurs 1 time
base                           Occurs 1 time
keydef (including keysegs)     Occurs once for each key
recinfo                        Occurs once for each field
```

Now we will look at each of these sections, showing each field.

We are going to use an example table throughout the description. To make the example table, we executed these statements:

```
CREATE TABLE T (S1 CHAR(1), S2 CHAR(2), S3 CHAR(3));
CREATE UNIQUE INDEX I1 ON T (S1);
CREATE INDEX I2 ON T (S2,S3);
INSERT INTO T VALUES ('1', 'aa', 'b');
INSERT INTO T VALUES ('2', 'aa', 'bb');
INSERT INTO T VALUES ('3', 'aa', 'bbb');
DELETE FROM T WHERE S1 = '2';
```

We took a hexadecimal dump of the resulting file, 'T.MYI'.

In all the individual descriptions below, the column labeled "Dump From Example File" has the exact bytes that are in 'T.MYI'. You can verify that by executing the same statements and looking at a hexadecimal dump yourself. With Linux this is possible using od -h T.MYI; with Windows you can use the command-line debugger.

Along with the typical value, we may include a comment. The comment usually explains why the value is what it is. Sometimes the comment is derived from the comments in the source code.

state

This section is written by 'mi_open.c', mi_state_info_write().

```
Name                         Size Dump From Example File  Comment
----                         ---- ----------------------  -------

file_version                  4   FE FE 07 01             from myisam_file_magic
options                       2   00 02                   HA_OPTION_COMPRESS_RECORD
                                                          etc.
header_length                 2   01 A2                   this header example has
                                                          0x01A2 bytes
```

```
state_info_length            2   00 B0                    = MI_STATE_INFO_SIZE
                                                          defined in myisamdef.h
base_info_length             2   00 64                    = MI_BASE_INFO_SIZE
                                                          defined in myisamdef.h
base_pos                     2   00 D4                    = where the base
                                                          section starts
key_parts                    2   00 03                    a key part is a column
                                                          within a key
unique_key_parts             2   00 00                    key-parts+unique-parts
keys                         1   02                       here are 2 keys --
                                                          I1 and I2
uniques                      1   00                       number of hash unique
                                                          keys used internally
                                                          in temporary tables
                                                          (nothing to do with
                                                          'UNIQUE' definitions)
language                     1   08                       "language for indexes"
max_block_size               1   01
fulltext_keys                1   00                       # of fulltext keys.
                                                          = 0 if version <= 4.0
not_used                     1   00                       to align to 8-byte
                                                          boundary

state->open_count            2   00 01
state->changed               1   39                       set if table updated;
                                                          reset if shutdown (so
                                                          one can examine this
                                                          to see if there was an
                                                          update without proper
                                                          shutdown)
state->sortkey               1   FF                       "sorted by this key"
                                                          (not used)
state->state.records         8   00 00 00 00 00 00 00 02 number of actual,
                                                          un-deleted, records
state->state.del             8   00 00 00 00 00 00 00 01 # of deleted records
state->split                 8   00 00 00 00 00 00 00 03 # of "chunks" (e.g.
                                                          records or spaces left
                                                          after record deletion)
state->dellink               8   00 00 00 00 00 00 00 07 "Link to next removed
                                                          "block". Initially =
                                                          HA_OFFSET_ERROR
state->state.key_file_length 8   00 00 00 00 00 00 0c 00 2048
state->state.data_file_length 8  00 00 00 00 00 00 00 15 = size of .MYD file
state->state.empty           8   00 00 00 00 00 00 00 00
state->state.key_empty       8   00 00 00 00 00 00 00 00
state->auto_increment        8   00 00 00 00 00 00 00 00
state->checksum              8   00 00 00 00 00 00 00 00
state->process               4   00 00 09 E6              from getpid(). process
                                                          of last update
state->unique                4   00 00 00 0B              initially = 0
state->status                4   00 00 00 00
state->update_count          4   00 00 00 04              updated for each write
```

```
                                                        lock (there were 3
                                                        inserts + 1 delete,
                                                        total 4 operations)
state->key_root              8   00 00 00 00 00 00 04 00 offset in file where
                                                        I1 keys start, can be
                                                        = HA_OFFSET_ERROR
                                 00 00 00 00 00 00 08 00 state->key_root occurs
                                                        twice because there
                                                        are two keys
state->key_del               8   FF FF FF FF FF FF FF FF delete links for keys
                                                        (occurs many times if
                                                        many delete links)
state->sec_index_changed     4   00 00 00 00             sec_index = secondary
                                                        index (presumably)
                                                        not currently used
state->sec_index_used        4   00 00 00 00             "which extra indexes
                                                        are in use"
                                                        not currently used
state->version               4   3F 3F EB F7             "timestamp of create"
state->key_map               8   00 00 00 03             "what keys are in use"
state->create_time           8   00 00 00 00 3F 3F EB F7 "time when database
                                                        created" (actually:
                                                        time when file made)
state->recover_time          8   00 00 00 00 00 00 00 00 "time of last recover"
state->check_time            8   00 00 00 00 3F 3F EB F7 "time of last check"
state->rec_per_key_rows      8   00 00 00 00 00 00 00 00
state->rec_per_key_parts     4   00 00 00 00             (key_parts = 3, so
                                 00 00 00 00              rec_per_key_parts
                                 00 00 00 00              occurs 3 times)
```

base

This section is written by 'mi_open.c', `mi_base_info_write()`. The corresponding structure in 'myisamdef.h' is `MI_BASE_INFO`.

In our example 'T.MYI' file, the first byte of the `base` section is at offset 0x00d4. That's where it's supposed to be, according to the header field `base_pos` (above).

```
Name                          Size Dump From Example File  Comment
----                          ---- ----------------------  -------

base->keystart                8    00 00 00 00 00 00 04 00 keys start at offset
                                                           1024 (0x0400)
base->max_data_file_length    8    00 00 00 00 00 00 00 00
base->max_key_file_length     8    00 00 00 00 00 00 00 00
base->records                 8    00 00 00 00 00 00 00 00
base->reloc                   8    00 00 00 00 00 00 00 00
base->mean_row_length         4    00 00 00 00
base->reclength               4    00 00 00 07             length(s1)+length(s2)
                                                           +length(s3)=7
base->pack_reclength          4    00 00 00 07
base->min_pack_length         4    00 00 00 07
base->max_pack_length         4    00 00 00 07
base->min_block_length        4    00 00 00 14
base->fields                  4    00 00 00 04             4 fields: 3 defined,
```

```
                                                                     plus 1 extra
base->pack_fields             4    00 00 00 00
base->rec_reflength           1    04
base->key_reflength           1    04
base->keys                    1    02                                was 0 at start
base->auto_key                1    00
base->pack_bits               2    00 00
base->blobs                   2    00 00
base->max_key_block_length    2    04 00                             length of block = 1024
                                                                     bytes (0x0400)
base->max_key_length          2    00 10                             including length of
                                                                     pointer
base->extra_alloc_bytes       2    00 00
base->extra_alloc_procent     1    00
base->raid_type               1    00
base->raid_chunks             2    00 00
base->raid_chunksize          4    00 00 00 00
[extra] i.e. filler           6    00 00 00 00 00 00
```

keydef

This section is written by 'mi_open.c', mi_keydef_write(). The corresponding structure in 'myisamdef.h' is MI_KEYDEF.

This is a multiple-occurrence structure. Since there are two indexes in our example (I1 and I2), we will see that keydef occurs two times below. There is a subordinate structure, keyseg, which also occurs multiple times (once within the keydef for I1 and two times within the keydef for I2).

```
Name                          Size Dump From Example File  Comment
----                          ---- ----------------------  -------

/* key definition for I1 */

keydef->keysegs               1    01                      there is 1 keyseg (for
                                                           column S1).
keydef->key_alg               1    01                      algorithm = Rtree or
                                                           Btree
keydef->flag                  2    00 49                   HA_NOSAME +
                                                           HA_SPACE_PACK_USED +
                                                           HA_NULL_PART_KEY
keydef->block_length          2    04 00                   i.e. 1024
key def->keylength            2    00 06                   field-count+sizeof(S1)
                                                           sizeof(ROWID)
keydef->minlength             2    00 06
keydef->maxlength             2    00 06
  /* keyseg for S1 in I1 */
  keyseg->type                1    01                      /* I1(S1) size(S1)=1,
                                                              column = 1 */
                                                           = HA_KEYTYPE_TEXT

  keyseg->language            1    08
  keyseg->null_bit            1    02
  keyseg->bit_start           1    00
  keyseg->bit_end             1    00
  [0] i.e. filler             1    00
```

```
    keyseg->flag              2    00 14              HA_NULL_PART +
                                                      HA_PART_KEY
    keyseg->length            2    00 01              length(S1) = 1
    keyseg->start             4    00 00 00 01        offset in the row
    keyseg->null_pos          4    00 00 00 00

  /* key definition for I2 */

  keydef->keysegs             1    02                 keysegs=2, for columns
                                                      S2 and S3
  keydef->key_alg             1    01                 algorithm = Rtree or
                                                      Btree
  keydef->flag                2    00 48              HA_SPACE_PACK_USED +
                                                      HA_NULL_PART_KEY
  keydef->block_length        2    04 00              i.e. 1024
  key def->keylength          2    00 0B              field-count+ sizeof(all fiel
                                                        sizeof(RID)
  keydef->minlength           2    00 0B
  keydef->maxlength           2    00 0B
    /* keyseg for S2 in I2 */
    keyseg->type              1    01                 /* I2(S2) size(S2)=2,
                                                         column = 2 */
    keyseg->language          1    08
    keyseg->null_bit          1    04
    keyseg->bit_start         1    00
    keyseg->bit_end           1    00
    [0] i.e. filler           1    00
    keyseg->flag              2    00 14              HA_NULL_PART +
                                                      HA_PART_KEY
    keyseg->length            2    00 02              length(S2) = 2
    keyseg->start             4    00 00 00 02
    keyseg->null_pos          4    00 00 00 00
    /* keyseg for S3 in I2 */
    keyseg->type              1    01                 /* I2(S3) size(S3)=3,
                                                         column = 3 */
    keyseg->language          1    08
    keyseg->null_bit          1    08
    keyseg->bit_start         1    00
    keyseg->bit_end           1    00
    [0] i.e. filler           1    00
    keyseg->flag              2    00 14              HA_NULL_PART +
                                                      HA_PART_KEY
    keyseg->length            2    00 03              length(S3) = 3
    keyseg->start             4    00 00 00 04
    keyseg->null_pos          4    00 00 00 00
```

recinfo

The `recinfo` section is written by '`mi_open.c`', `mi_recinfo_write()`. The corresponding structure in '`myisamdef.h`' is `MI_COLUMNDEF`.

This is another multiple-occurrence structure. It appears once for each field that appears in a key, including an extra field that appears at the start and has flags (for deletion and for null fields).

```
Name                       Size Dump From Example File  Comment
----                       ---- ----------------------  -------

recinfo->type              2    00 00                   extra
recinfo->length            2    00 01
recinfo->null_bit          1    00
recinfo->null_pos          2    00 00

recinfo->type              2    00 00                   I1 (S1)
recinfo->length            2    00 01
recinfo->null_bit          1    02
recinfo->null_pos          2    00 00

recinfo->type              2    00 00                   I2 (S2)
recinfo->length            2    00 02
recinfo->null_bit          1    04
recinfo->null_pos          2    00 00

recinfo->type              2    00 00                   I2 (S3)
recinfo->length            2    00 03
recinfo->null_bit          1    08
recinfo->null_pos          2    00 00
```

We are now at offset 0xA2 within the file '`T.MYI`'. Notice that the value of the third field in the header, `header_length`, is 0xA2. Anything following this point, up till the first key value, is filler.

The '`.MYI`' Key Values

And now we look at the part which is not the information header: we look at the key values. The key values are in blocks (MySQL's term for pages). A block contains values from only one index. To continue our example: there is a block for the I1 key values, and a block for the I2 key values.

According to the header information (`state->key_root` above), the I1 block starts at offset 0x0400 in the file, and the I2 block starts at offset 0x0800 in the file.

At offset 0x0400 in the file, we have this:

```
Name                       Size Dump From Example File  Comment
----                       ---- ----------------------  -------

(block header)             2    00 0E                   = size (inclusive)
                                                        (first bit of word =
                                                        0 meaning this is a
                                                        B-Tree leaf, see the
                                                        mi_test_if_nod macro)
(first key value)          2    01 31                   Value is "1" (0x31).
(first key pointer)        4    00 00 00 00             Pointer is to Record
                                                        #0000.
(second key value)         2    01 33                   Value is "3" (0x33).
(second key pointer)       4    00 00 00 02             Pointer is to Record
                                                        #0002.
(junk)                     1010 .. .. .. .. .. .. ..    rest of the 1024-byte
                                                        block is unused
```

At offset 0800x in the file, we have this:

```
Name                          Size Dump From Example File  Comment
----                          ---- ----------------------  -------

(block header)                2    00 18                   = size (inclusive)
(first key value)             7    01 61 61 01 62 20 20    Value is "aa/b  "
(first key pointer)           4    00 00 00 00             Pointer is to Record
                                                           #0000.
(second key value)            7    01 61 61 01 62 62 62    Value is "aa/bbb"
(second key pointer)          4    00 00 00 02             Pointer is to Record
                                                           #0002.
(junk)                        1000 .. .. .. .. .. .. ..    rest of the 1024-byte
                                                           block is unused
```

From the above illustrations, these facts should be clear:

- Each key contains the entire contents of all the columns, including trailing spaces in `CHAR` columns. There is no front truncation. There is no back truncation. (There can be space truncation if `keyseg->flag` `HA_SPACE_PACK` flag is on.)
- For fixed-row tables: The pointer is a fixed-size (4-byte) number which contains an ordinal row number. The first row is Record #0000. This item is analogous to the ROWID, or RID (row identifier), which other DBMSs use. For dynamic-row tables: The pointer is an offset in the '`.MYD`' file.
- The normal block length is 0x0400 (1024) bytes.

These facts are not illustrated, but are also clear:

- If a key value is `NULL`, then the first byte is 0x00 (instead of 001 as in the above examples) and that's all. Even for a fixed `CHAR(3)` column, the size of the key value is only 1 byte.
- Initially the junk at the end of a block is filler bytes, value = 0xA5. If MySQL shifts key values up after a `DELETE`, the end of the block is not overwritten.
- A normal block is at least 65% full, and typically 80% full. (This is somewhat denser than the typical B-tree algorithm would cause, it is thus because `myisamchk -rq` will make blocks nearly 100% full.)
- There is a pool of free blocks, which increases in size when deletions occur. If all blocks have the same normal block length (1024), then MySQL will always use the same pool.
- The maximum number of keys is 32 (`MI_MAX_KEY`). The maximum number of segments in a key is 16 (`MI_MAX_KEY_SEG`). The maximum key length is 500 (`MI_MAX_KEY_LENGTH`). The maimum block length is 16384 (`MI_MAX_KEY_BLOCK_LENGTH`). All these MI_... constants are expressed by #defines in the '`myisamdef.h`' file.

10.1 `MyISAM` Files

Some notes about `MyISAM` file handling:

- If a table is never updated, MySQL will never touch the table files, so it would never be marked as closed or corrupted.
- If a table is marked readonly by the OS, it will only be opened in readonly mode. Any updates to it will fail.
- When a normal table is opened for reading by a `SELECT`, MySQL will open it in read/write mode, but will not write anything to it.
- A table can be closed during one of the following events:
  - Out of space in table cache
  - Someone executed flush tables

- MySQL was shut down
- flush_time expired (which causes an automatic flush-tables to be executed)

- When MySQL opens a table, it checks if the table is clean. If it isn't and the server was started with the `--myisam-recover` option, check the table and try to recover it if it's crashed. (The safest automatic recover option is probably `--myisam-recover=BACKUP`.)

11 InnoDB Record Structure

This page contains:

- A high-altitude "summary" picture of the parts of a MySQL/InnoDB record structure.
- A description of each part.
- An example.

After reading this page, you will know how MySQL/InnoDB stores a physical record.

11.1 High-Altitude Picture

The chart below shows the three parts of a physical record.

Name	**Size**
Field Start Offsets	(F*1) or (F*2) bytes
Extra Bytes	6 bytes
Field Contents	depends on content

Legend: The letter 'F' stands for 'Number Of Fields'.

The meaning of the parts is as follows:

- The FIELD START OFFSETS is a list of numbers containing the information "where a field starts".
- The EXTRA BYTES is a fixed-size header.
- The FIELD CONTENTS contains the actual data.

An Important Note About The Word "Origin"

The "Origin" or "Zero Point" of a record is the first byte of the Field Contents — not the first byte of the Field Start Offsets. If there is a pointer to a record, that pointer is pointing to the Origin. Therefore the first two parts of the record are addressed by subtracting from the pointer, and only the third part is addressed by adding to the pointer.

11.1.1 FIELD START OFFSETS

The Field Start Offsets is a list in which each entry is the position, relative to the Origin, of the start of the next field. The entries are in reverse order, that is, the first field's offset is at the end of the list.

An example: suppose there are three columns. The first column's length is 1, the second column's length is 2, and the third column's length is 4. In this case, the offset values are, respectively, 1, 3 (1+2), and 7 (1+2+4). Because values are reversed, a core dump of the Field Start Offsets would look like this: `07,03,01`.

There are two complications for special cases:

- Complication #1: The size of each offset can be either one byte or two bytes. One-byte offsets are only usable if the total record size is less than 127. There is a flag in the "Extra Bytes" part which will tell you whether the size is one byte or two bytes.
- Complication #2: The most significant bits of an offset may contain flag values. The next two paragraphs explain what the contents are.

When The Size Of Each Offset Is One Byte

- 1 bit = 0 if field is non-NULL, = 1 if field is NULL

- 7 bits = the actual offset, a number between 0 and 127

When The Size Of Each Offset Is Two Bytes

- 1 bit = 0 if field is non-NULL, = 1 if field is NULL
- 1 bit = 0 if field is on same page as offset, = 1 if field and offset are on different pages
- 14 bits = the actual offset, a number between 0 and 16383

It is unlikely that the "field and offset are on different pages" unless the record contains a large BLOB.

11.1.2 EXTRA BYTES

The Extra Bytes are a fixed six-byte header.

Name	**Size**	**Description**
info_bits:		
()	1 bit	unused or unknown
()	1 bit	unused or unknown
deleted_flag	1 bit	1 if record is deleted
min_rec_flag	1 bit	1 if record is predefined minimum record
n_owned	4 bits	number of records owned by this record
heap_no	13 bits	record's order number in heap of index page
n_fields	10 bits	number of fields in this record, 1 to 1023
1byte_offs_flag	1 bit	1 if each Field Start Offsets is 1 byte long (this item is also called the "short" flag)
next 16 bits	16 bits	pointer to next record in page
TOTAL	48 bits	

Total size is 48 bits, which is six bytes.

If you're just trying to read the record, the key bit in the Extra Bytes is 1byte_offs_flag — you need to know if 1byte_offs_flag is 1 (i.e.: "short 1-byteoffsets") or 0 (i.e.: "2-byte offsets").

Given a pointer to the Origin, InnoDB finds the start of the record as follows:

- Let X = n_fields (the number of fields is by definition equal to the number of entries in the Field Start Offsets Table).
- If 1byte_offs_flag equals 0, then let X = X * 2 because there are two bytes for each entry instead of just one.
- Let X = X + 6, because the fixed size of Extra Bytes is 6.
- The start of the record is at (pointer value minus X).

11.1.3 FIELD CONTENTS

The Field Contents part of the record has all the data. Fields are stored in the order they were defined in.

There are no markers between fields, and there is no marker or filler at the end of a record.

Here's an example.

- I made a table with this definition:

```
CREATE TABLE T
    (FIELD1 VARCHAR(3), FIELD2 VARCHAR(3), FIELD3 VARCHAR(3))
    Type=InnoDB;
```

To understand what follows, you must know that table T has six columns — not three — because InnoDB automatically added three "system columns" at the start for its own housekeeping. It happens that these system columns are the row ID, the transaction ID, and the rollback pointer, but their values don't matter now. Regard them as three black boxes.

- I put some rows in the table. My last three INSERT statements were:

```
INSERT INTO T VALUES ('PP', 'PP', 'PP');
INSERT INTO T VALUES ('Q', 'Q', 'Q');
INSERT INTO T VALUES ('R', NULL, NULL);
```

- I ran Borland's TDUMP to get a hexadecimal dump of the contents of '\mysql\data\ibdata1', which (in my case) is the MySQL/InnoDB data file (on Windows).

Here is an extract of the dump:

```
Address Values in Hexadecimal                          Values in ASCII
0D4280: 00 00 2D 00 84 4F 4F 4F 4F 4F 4F 4F 4F 4F 19 17  ..-..OOOOOOOOO..
0D4290: 15 13 0C 06 00 00 78 0D 02 BF 00 00 00 00 04 21  ......x........!
0D42A0: 00 00 00 00 09 2A 80 00 00 00 2D 00 84 50 50 50  .....*....-..PPP
0D42B0: 50 50 50 16 15 14 13 0C 06 00 00 80 0D 02 E1 00  PPP.............
0D42C0: 00 00 00 04 22 00 00 00 00 09 2B 80 00 00 00 2D  ....".....+....-
0D42D0: 00 84 51 51 51 94 94 14 13 0C 06 00 00 88 0D 00  ..QQQ...........
0D42E0: 74 00 00 00 00 04 23 00 00 00 00 09 2C 80 00 00  t.....#.....,...
0D42F0: 00 2D 00 84 52 00 00 00 00 00 00 00 00 00 00 00  .-..R...........
```

A reformatted version of the dump, showing only the relevant bytes, looks like this (I've put a line break after each field and added labels):

Reformatted Hexadecimal Dump

```
19 17 15 13 0C 06 Field Start Offsets /* First Row */
00 00 78 0D 02 BF Extra Bytes
00 00 00 00 04 21 System Column #1
00 00 00 00 09 2A System Column #2
80 00 00 00 2D 00 84 System Column #3
50 50 Field1 'PP'
50 50 Field2 'PP'
50 50 Field3 'PP'

16 15 14 13 0C 06 Field Start Offsets /* Second Row */
00 00 80 0D 02 E1 Extra Bytes
00 00 00 00 04 22 System Column #1
00 00 00 00 09 2B 80 System Column #2
00 00 00 2D 00 84 System Column #3
51 Field1 'Q'
51 Field2 'Q'
51 Field3 'Q'

94 94 14 13 0C 06 Field Start Offsets /* Third Row */
00 00 88 0D 00 74 Extra Bytes
00 00 00 00 04 23 System Column #1
00 00 00 00 09 2C System Column #2
80 00 00 00 2D 00 84 System Column #3
```

```
52 Field1 'R'
```

You won't need explanation if you followed everything I've said, but I'll add helpful notes for the three trickiest details.

- Helpful Notes About "Field Start Offsets":

  Notice that the sizes of the record's fields, in forward order, are: 6, 6, 7, 2, 2, 2. Since each offset is for the start of the "next" field, the hexadecimal offsets are 06, 0c (6+6), 13 (6+6+7), 15 (6+6+7+2), 17 (6+6+7+2+2), 19 (6+6+7+2+2+2). Reversing the order, the Field Start Offsets of the first record are: `19,17,15,13,0c,06`.

- Helpful Notes About "Extra Bytes":

  Look at the Extra Bytes of the first record: `00 00 78 0D 02 BF`. The fourth byte is `0D hexadecimal`, which is `1101 binary` ... the 110 is the last bits of n_fields (`110 binary` is 6 which is indeed the number of fields in the record) and the final 1 bit is 1byte_offs_flag. The fifth and sixth bytes, which contain `02 BF`, constitute the "next" field. Looking at the original hexadecimal dump, at address `0D42BF` (which is position `02BF` within the page), you'll see the beginning bytes of System Column #1 of the second row. In other words, the "next" field points to the "Origin" of the following row.

- Helpful Notes About NULLs:

  For the third row, I inserted `NULL`s in FIELD2 and FIELD3. Therefore in the Field Start Offsets the top bit is `on` for these fields (the values are `94 hexadecimal`, `94 hexadecimal`, instead of `14 hexadecimal`, `14 hexadecimal`). And the row is shorter because the `NULL`s take no space.

11.2 Where to Look For More Information

References:

The most relevant `InnoDB` source-code files are '`rem0rec.c`', '`rem0rec.ic`', and '`rem0rec.h`' in the '`rem`' ("Record Manager") directory.

12 InnoDB Page Structure

InnoDB stores all records inside a fixed-size unit which is commonly called a "page" (though InnoDB sometimes calls it a "block" instead). Currently all pages are the same size, 16KB.

A page contains records, but it also contains headers and trailers. I'll start this description with a high-altitude view of a page's parts, then I'll describe each part of a page. Finally, I'll show an example. This discussion deals only with the most common format, for the leaf page of a data file.

12.1 High-Altitude View

An InnoDB page has seven parts:

- Fil Header
- Page Header
- Infimum + Supremum Records
- User Records
- Free Space
- Page Directory
- Fil Trailer

As you can see, a page has two header/trailer pairs. The inner pair, "Page Header" and "Page Directory", are mostly the concern of the \page program group, while the outer pair, "Fil Header" and "Fil Trailer", are mostly the concern of the \fil program group. The "Fil" header also goes goes by the name of "File Page Header".

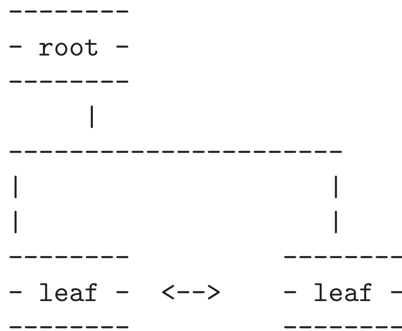
Sandwiched between the headers and trailers, are the records and the free (unused) space. A page always begins with two unchanging records called the Infimum and the Supremum. Then come the user records. Between the user records (which grow downwards) and the page directory (which grows upwards) there is space for new records.

12.1.1 Fil Header

The Fil Header has eight parts, as follows:

Name	Size	Remarks
FIL_PAGE_SPACE	4	4 ID of the space the page is in
FIL_PAGE_OFFSET	4	ordinal page number from start of space
FIL_PAGE_PREV	4	offset of previous page in key order
FIL_PAGE_NEXT	4	offset of next page in key order
FIL_PAGE_LSN	8	log serial number of page's latest log record
FIL_PAGE_TYPE	2	current defined types are: FIL_PAGE_INDEX, FIL_PAGE_UNDO_LOG, FIL_PAGE_INODE, FIL_PAGE_IBUF_FREE_LIST
FIL_PAGE_FILE_FLUSH_LSN	8	"the file has been flushed to disk at least up to this lsn" (log serial number), valid only on the first page of the file
FIL_PAGE_ARCH_LOG_NO	4	the latest archived log file number at the time that FIL_PAGE_FILE_FLUSH_LSN was written (in the log)

- FIL_PAGE_SPACE is a necessary identifier because different pages might belong to different (table) spaces within the same file. The word "space" is generic jargon for either "log" or "tablespace".
- FIL_PAGE_PREV and FIL_PAGE_NEXT are the page's "backward" and "forward" pointers. To show what they're about, I'll draw a two-level B-tree.



Everyone has seen a B-tree and knows that the entries in the root page point to the leaf pages. (I indicate those pointers with vertical '`|`' bars in the drawing.) But sometimes people miss the detail that leaf pages can also point to each other (I indicate those pointers with a horizontal two-way pointer '<–>' in the drawing). This feature allows `InnoDB` to navigate from leaf to leaf without having to back up to the root level. This is a sophistication which you won't find in the classic B-tree, which is why `InnoDB` should perhaps be called a B+-tree instead.

- The fields `FIL_PAGE_FILE_FLUSH_LSN`, `FIL_PAGE_PREV`, and `FIL_PAGE_NEXT` all have to do with logs, so I'll refer you to my article "How Logs Work With MySQL And InnoDB" on `devarticles.com`.
- `FIL_PAGE_FILE_FLUSH_LSN` and `FIL_PAGE_ARCH_LOG_NO` are valid only for the first page of a data file.

12.1.2 Page Header

The Page Header has 14 parts, as follows:

Name	**Size**	**Remarks**
`PAGE_N_DIR_SLOTS`	2	number of directory slots in the Page Directory part; initial value = 2
`PAGE_HEAP_TOP`	2	record pointer to first record in heap
`PAGE_N_HEAP`	2	number of heap records; initial value = 2
`PAGE_FREE`	2	record pointer to first free record
`PAGE_GARBAGE`	2	"number of bytes in deleted records"
`PAGE_LAST_INSERT`	2	record pointer to the last inserted record
`PAGE_DIRECTION`	2	either `PAGE_LEFT`, `PAGE_RIGHT`, or `PAGE_NO_DIRECTION`
`PAGE_N_DIRECTION`	2	number of consecutive inserts in the same direction, e.g. "last 5 were all to the left"
`PAGE_N_RECS`	2	number of user records
`PAGE_MAX_TRX_ID`	8	the highest ID of a transaction which might have changed a record on the page (only set for secondary indexes)
`PAGE_LEVEL`	2	level within the index (0 for a leaf page)
`PAGE_INDEX_ID`	8	identifier of the index the page belongs to
`PAGE_BTR_SEG_LEAF`	10	"file segment header for the leaf pages in a B-tree" (this is irrelevant here)
`PAGE_BTR_SEG_TOP`	10	"file segment header for the non-leaf pages in a B-tree" (this is irrelevant here)

(Note: I'll clarify what a "heap" is when I discuss the User Records part of the page.)

Some of the Page Header parts require further explanation:

- PAGE_FREE:

  Records which have been freed (due to deletion or migration) are in a one-way linked list. The PAGE_FREE pointer in the page header points to the first record in the list. The "next" pointer in the record header (specifically, in the record's Extra Bytes) points to the next record in the list.
- PAGE_DIRECTION and PAGE_N_DIRECTION:

  It's useful to know whether inserts are coming in a constantly ascending sequence. That can affect InnoDB's efficiency.
- PAGE_HEAP_TOP and PAGE_FREE and PAGE_LAST_INSERT:

  Warning: Like all record pointers, these point not to the beginning of the record but to its Origin (see the earlier discussion of Record Structure).
- PAGE_BTR_SEG_LEAF and PAGE_BTR_SEG_TOP:

  These variables contain information (space ID, page number, and byte offset) about index node file segments. InnoDB uses the information for allocating new pages. There are two different variables because InnoDB allocates separately for leaf pages and upper-level pages.

12.1.3 The Infimum and Supremum Records

"Infimum" and "supremum" are real English words but they are found only in arcane mathematical treatises, and in InnoDB comments. To InnoDB, an infimum is lower than the the lowest possible real value (negative infinity) and a supremum is greater than the greatest possible real value (positive infinity). InnoDB sets up an infimum record and a supremum record automatically at page-create time, and never deletes them. They make a useful barrier to navigation so that "get-prev" won't pass the beginning and "get-next" won't pass the end. Also, the infimum record can be a dummy target for temporary record locks.

The InnoDB code comments distinguish between "the infimum and supremum records" and the "user records" (all other kinds).

It's sometimes unclear whether InnoDB considers the infimum and supremum to be part of the header or not. Their size is fixed and their position is fixed, so I guess so.

12.1.4 User Records

In the User Records part of a page, you'll find all the records that the user inserted.

There are two ways to navigate through the user records, depending whether you want to think of their organization as an unordered or an ordered list.

An unordered list is often called a "heap". If you make a pile of stones by saying "whichever one I happen to pick up next will go on top" — rather than organizing them according to size and colour – then you end up with a heap. Similarly, InnoDB does not want to insert new rows according to the B-tree's key order (that would involve expensive shifting of large amounts of data), so it inserts new rows right after the end of the existing rows (at the top of the Free Space part) or wherever there's space left by a deleted row.

But by definition the records of a B-tree must be accessible in order by key value, so there is a record pointer in each record (the "next" field in the Extra Bytes) which points to the next record in key order. In other words, the records are a one-way linked list. So InnoDB can access rows in key order when searching.

12.1.5 Free Space

I think it's clear what the Free Space part of a page is, from the discussion of other parts.

12.1.6 Page Directory

The Page Directory part of a page has a variable number of record pointers. Sometimes the record pointers are called "slots" or "directory slots". Unlike other DBMSs, `InnoDB` does not have a slot for every record in the page. Instead it keeps a sparse directory. In a fullish page, there will be one slot for every six records.

The slots track the records' logical order (the order by key rather than the order by placement on the heap). Therefore, if the records are `'A' 'B' 'F' 'D'` the slots will be `(pointer to 'A') (pointer to 'B') (pointer to 'D') (pointer to 'F')`. Because the slots are in key order, and each slot has a fixed size, it's easy to do a binary search of the records on the page via the slots.

(Since the Page Directory does not have a slot for every record, binary search can only give a rough position and then `InnoDB` must follow the "next" record pointers. `InnoDB`'s "sparse slots" policy also accounts for the n_owned field in the Extra Bytes part of a record: n_owned indicates how many more records must be gone through because they don't have their own slots.)

12.1.7 Fil Trailer

The Fil Trailer has one part, as follows:

Name	**Size**	**Remarks**
`FIL_PAGE_END_LSN`	8	low 4 bytes = checksum of page, last 4 bytes = same as `FIL_PAGE_LSN`

The final part of a page, the fil trailer (or File Page Trailer), exists because `InnoDB`'s architect worried about integrity. It's impossible for a page to be only half-written, or corrupted by crashes, because the log-recovery mechanism restores to a consistent state. But if something goes really wrong, then it's nice to have a checksum, and to have a value at the very end of the page which must be the same as a value at the very beginning of the page.

12.2 Example

For this example, I used Borland's TDUMP again, as I did for the earlier chapter on Record Format. This is what a page looked like:

Address Values in Hexadecimal **Values in ASCII**

```
0D4000: 00 00 00 00 00 00 00 35 FF FF FF FF FF FF FF FF     .......5........
0D4010: 00 00 00 00 00 00 E2 64 45 BF 00 00 00 00 00 00     .......dE.......
0D4020: 00 00 00 00 00 00 00 05 02 F5 00 12 00 00 00 00     ................
0D4030: 02 E1 00 02 00 0F 00 10 00 00 00 00 00 00 00 00     ................
0D4040: 00 00 00 00 00 00 00 00 00 14 00 00 00 00 00 00     ................
0D4050: 00 02 16 B2 00 00 00 00 00 00 00 02 15 F2 08 01     ................
0D4060: 00 00 03 00 89 69 6E 66 69 6D 75 6D 00 09 05 00     .....infimum....
0D4070: 08 03 00 00 73 75 70 72 65 6D 75 6D 00 22 1D 18     ....supremum."..
0D4080: 13 0C 06 00 00 10 0D 00 B7 00 00 00 00 04 14 00     ................
0D4090: 00 00 00 09 1D 80 00 00 00 2D 00 84 41 41 41 41     .........-..AAAA
0D40A0: 41 41 41 41 41 41 41 41 41 41 41 1F 1B 17 13 0C    AAAAAAAAAAA.....
 ...
 ...
0D7FE0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 74     ...............t
0D7FF0: 02 47 01 AA 01 0A 00 65 3A E0 AA 71 00 00 E2 64     .G.....e:..q...d
```

Let's skip past the first 38 bytes, which are Fil Header. The bytes of the Page Header start at location 0d4026 hexadecimal:

Location	Name	Description
00 05	PAGE_N_DIR_SLOTS	There are 5 directory slots.
02 F5	PAGE_HEAP_TOP	At location 0402F5, not shown, is the beginning of free space. Maybe a better name would have been PAGE_HEAP_END.
00 12	PAGE_N_HEAP	There are 18 (hexadecimal 12) records in the page.
00 00	PAGE_FREE	There are zero free (deleted) records.
00 00	PAGE_GARBAGE	There are zero bytes in deleted records.
02 E1	PAGE_LAST_INSERT	The last record was inserted at location 02E1, not shown, within the page.
00 02	PAGE_DIRECTION	A glance at page0page.h will tell you that 2 is the #defined value for PAGE_RIGHT.
00 0F	PAGE_N_DIRECTION	The last 15 (hexadecimal 0F) inserts were all done "to the right" because I was inserting in ascending order.
00 10	PAGE_N_RECS	There are 16 (hexadecimal 10) user records. Notice that PAGE_N_RECS is smaller than the earlier field, PAGE_N_HEAP.
00 00 00 00 00 00 00	PAGE_MAX_TRX_ID	
00 00	PAGE_LEVEL	Zero because this is a leaf page.
00 00 00 00 00 00 00 14	PAGE_INDEX_ID	This is index number 20.
00 00 00 00 00 00 00 02 16 B2	PAGE_BTR_SEG_LEAF	
00 00 00 00 00 00 00 02 15 F2	PAGE_BTR_SEG_TOP	

Immediately after the page header are the infimum and supremum records. Looking at the "Values In ASCII" column in the hexadecimal dump, you will see that the contents are in fact the words "infimum" and "supremum" respectively.

Skipping past the User Records and the Free Space, many bytes later, is the end of the 16KB page. The values shown there are the two trailers.

- The first trailer (00 74, 02 47, 01 AA, 01 0A, 00 65) is the page directory. It has 5 entries, because the header field PAGE_N_DIR_SLOTS says there are 5.
- The next trailer (3A E0 AA 71, 00 00 E2 64) is the fil trailer. Notice that the last four bytes, 00 00 E2 64, appeared before in the fil header.

12.3 Where to Look For More Information

References:

The most relevant InnoDB source-code files are 'page0page.c', 'page0page.ic', and 'page0page.h' in the 'page' directory.

13 Adding New Error Messages to MySQL

To add new error messages, use the following procedure:

- Open the file 'sql/share/english.txt' in an editor.
- Add new error messages at the end of this file. Each message should be on a separate line, it should be quoted within double quote ('"') characters and should end with a ',' following the second double quote.
- For each new error message, add a define to the file 'include/mysqld_error.h' before the last line (#define ER_ERROR_MESSAGES).
- Adjust the value of ER_ERROR_MESSAGES to the new number of error messages.
- Add last to 'include/sql_state.h' the SQL states for the new error messages. Note that this file must be kept sorted according to the value of the error number. That is, although the 'sql_state.h' file might not contain an entry for every symbol in 'mysqld_error.h', those entries that are present in 'sql_state.h' must appear in the same order as those for the corresponding entries in 'mysqld_error.h'.
- Go to the 'sql' directory in a terminal window and type ./add_errmsg #. This will copy the last # error messages from 'share/english.txt' to all the other language files in 'share/'.
- Translate the error message in the languages you know by editing the files share/XXXXXX/errmsg.txt.

The maximum error message is MYSQL_ERRMSG_SIZE = 512. Ensure by using constructs like "%s-.64s" that there are no buffer overflows!

14 Annotated List of Files in the MySQL Source Code Distribution

This is a description of the files that you get when you download the source code of MySQL. This description begins with a list of the main directories and a short comment about each one. Then, for each directory, in alphabetical order, a longer description is supplied. When a directory contains significant program files, a list of each C program is given along with an explanation of its intended function.

14.1 Directory Listing

Directory – Short Comment

- bdb – The Berkeley Database table handler
- BitKeeper – BitKeeper administration (not part of the source distribution)
- BUILD – Frequently used build scripts
- Build-tools – Build tools (not part of the source distribution)
- client – Client library
- cmd-line-utils – Command-line utilities (libedit and readline)
- dbug – Fred Fish's dbug library
- Docs – Preliminary documents about internals and new modules; will eventually be moved to the mysqldoc repository
- extra – Some minor standalone utility programs
- heap – The HEAP table handler
- include – Header (*.h) files for most libraries; includes all header files distributed with the MySQL binary distribution
- innobase – The Innobase (InnoDB) table handler
- libmysql – For producing MySQL as a library (e.g. a Windows .DLL)
- libmysql_r – For building a thread-safe libmysql library
- libmysqld – The MySQL Server as an embeddable library
- man – Some user-contributed manual pages
- myisam – The `MyISAM` table handler
- myisammrg – The `MyISAM` Merge table handler
- mysql-test – A test suite for mysqld
- mysys – MySQL system library (Low level routines for file access etc.)
- netware – Files related to the Novell NetWare version of MySQL
- NEW-RPMS – Directory to place RPMs while making a distribution
- os2 – Routines for working with the OS/2 operating system
- pstack – Process stack display (not currently used)
- regex – Henry Spencer's Regular Expression library for support of REGEXP function
- SCCS – Source Code Control System (not part of source distribution)
- scripts – SQL batches, e.g. mysqlbug and mysql_install_db
- sql – Programs for handling SQL commands; the `"core"` of MySQL
- sql-bench – The MySQL benchmarks
- SSL – Secure Sockets Layer; includes an example certification one can use to test an SSL (secure) database connection

- strings – Library for C string routines, e.g. atof, strchr
- support-files – Files used to build MySQL on different systems
- tests – Tests in Perl and in C
- tools – mysqlmanager.c (tool under development, not yet useful)
- VC++Files – Includes this entire directory, repeated for VC++ (Windows) use
- vio – Virtual I/O Library
- zlib – Data compression library, used on Windows

14.1.1 bdb

The Berkeley Database table handler.

The Berkeley Database (BDB) is maintained by Sleepycat Software. MySQL AB maintains only a few small patches to make BDB work better with MySQL.

The documentation for BDB is available at http://www.sleepycat.com/docs/. Since it's reasonably thorough documentation, a description of the BDB program files is not included in this document.

14.1.2 BitKeeper

BitKeeper administration.

Bitkeeper administration is not part of the source distribution. This directory may be present if you downloaded the MySQL source using BitKeeper rather than via the mysql.com site. The files in the BitKeeper directory are for maintenance purposes only – they are not part of the MySQL package.

The MySQL Reference Manual explains how to use Bitkeeper to get the MySQL source. Please see `http://www.mysql.com/doc/en/Installing_source_tree.html` for more information.

14.1.3 BUILD

Frequently used build scripts.

This directory contains the build switches for compilation on various platforms. There is a subdirectory for each set of options. The main ones are:

- alpha
- ia64
- pentium (with and without debug or bdb, etc.)
- solaris

14.1.4 Build-tools

Build tools.

Build-tools is not part of the source distribution. This directory contains batch files for extracting, making directories, and making programs from source files. There are several subdirectories with different scripts – for building Linux executables, for compiling, for performing all build steps, and so on.

14.1.5 client

Client library.

The client library includes 'mysql.cc' (the source of the mysql executable) and other utilities. Most of the utilities are mentioned in the MySQL Reference Manual. Generally these are standalone C programs which one runs in "client mode", that is, they call the server.

The C program files in the directory are:

- get_password.c – ask for a password from the console
- mysql.cc – "The MySQL command tool"
- mysqladmin.c – maintenance of MySQL databases
- mysqlcheck.c – check all databases, check connect, etc.
- mysqldump.c – dump table's contents as SQL statements, suitable to backup a MySQL database
- mysqlimport.c – import text files in different formats into tables
- mysqlmanager-pwgen.c – pwgen stands for "password generation" (not currently maintained)
- mysqlmanagerc.c – entry point for mysql manager (not currently maintained)
- mysqlshow.c – show databases, tables or columns
- mysqltest.c – test program used by the mysql-test suite, mysql-test-run
- password.c – password checking routines (version 4.1 and up)

14.1.6 cmd-line-utils

Command-line utilities (libedit and readline).

There are two subdirectories: '\readline' and '\libedit'. All the files here are "non-MySQL" files, in the sense that MySQL AB didn't produce them, it just uses them. It should be unnecessary to study the programs in these files unless you are writing or debugging a tty-like client for MySQL, such as mysql.exe.

The '\readline' subdirectory contains the files of the GNU Readline Library, "a library for reading lines of text with interactive input and history editing". The programs are copyrighted by the Free Software Foundation.

The '\libedit' (library of edit functions) subdirectory has files written by Christos Zoulas. They are distributed and modifed under the BSD License. These files are for editing the line contents.

These are the program files in the \libedit subdirectory:

- chared.c – character editor
- common.c – common editor functions
- el.c – editline interface functions
- emacs.c – emacs functions
- fgetln.c – get line
- hist.c – history access functions
- history.c – more history access functions
- key.c – procedures for maintaining the extended-key map
- map.c – editor function definitions
- parse.c – parse an editline extended command

- prompt.c – prompt printing functions
- read.c – terminal read functions
- readline.c – read line
- refresh.c – "lower level screen refreshing functions"
- search.c – "history and character search functions"
- sig.c – for signal handling
- strlcpy.c – string copy
- term.c – "editor/termcap-curses interface"
- tokenizer.c – Bourne shell line tokenizer
- tty.c – for a tty interface
- vi.c – commands used when in the vi (editor) mode

14.1.7 dbug

Fred Fish's dbug library.

This is not really part of the MySQL package. Rather, it's a set of public-domain routines which are useful for debugging MySQL programs. The MySQL Server and all .c and .cc programs support the use of this package.

How it works: One inserts a function call that begins with DBUG_* in one of the regular MYSQL programs. For example, in get_password.c, you will find this line:

```
DBUG_ENTER("get_tty_password");
```

at the start of a routine, and this line:

```
DBUG_RETURN(my_strdup(to,MYF(MY_FAE)));
```

at the end of the routine. These lines don't affect production code. Features of the dbug library include extensive reporting and profiling (the latter has not been used by the MySQL team).

The C programs in this directory are:

- dbug.c – The main module
- dbug_analyze.c – Reads a file produced by trace functions
- example1.c – A tiny example
- example2.c – A tiny example
- example3.c – A tiny example
- factorial.c – A tiny example
- main.c – A tiny example
- sanity.c – Declaration of a variable

14.1.8 Docs

Preliminary documents about internals and new modules, which will eventually be moved to the mysqldoc repository.

This directory doesn't have much at present that's very useful to the student, but the plan is that some documentation related to the source files and the internal workings of MySQL, including perhaps some documentation from developers themselves, will be placed here. Files in this directory will eventually be moved to the MySQL documentation repository.

These sub-directories are part of this directory:

- books – .gif images and empty .txt files; no real information

- flags – images of flags of countries
- images – flag backgrounds and the MySQL dolphin logo
- mysql-logos – more MySQL-related logos, some of them moving
- raw-flags – more country flags, all .gif files
- support – various files for generating texinfo/docbook documentation
- to-be-included... – contains a MySQL-for-dummies file
- translations – some Portuguese myodbc documentation

In the main directory, you'll find some .txt files related to the methods that MySQL uses to produce its printed and html documents, odd bits in various languages, and the single file in the directory which has any importance – '`internals.texi`' – The "MySQL Internals" document.

Despite the name, '`internals.texi`' is not yet much of a description of MySQL internals although work is in progress to make it so. However, there is some useful description of the functions in the mysys directory (see below), and of the structure of client/server messages (doubtless very useful for eople who want to make their own JDBC drivers, or just sniff).

14.1.9 extra

Some minor standalone utility programs.

These programs are all standalone utilities, that is, they have a main() function and their main role is to show information that the MySQL server needs or produces. Most are unimportant. They are as follows:

- my_print_defaults.c – print parameters from my.ini files. Can also be used in scripts to enable processing of my.ini files.
- mysql_waitpid.c – wait for a program to terminate. Useful for shell scripts when one needs to wait until a process terminates.
- perror.c – "print error" – given error number, display message
- replace.c – replace strings in text files or pipe
- resolve_stack_dump.c – show symbolic information from a MySQL stack dump, normally found in the mysql.err file
- resolveip.c – convert an IP address to a hostname, or vice versa

14.1.10 heap

The HEAP table handler.

All the MySQL table handlers (i.e. the handlers that MySQL itself produces) have files with similar names and functions. Thus, this (heap) directory contains a lot of duplication of the myisam directory (for the `MyISAM` table handler). Such duplicates have been marked with an "*" in the following list. For example, you will find that '`\heap\hp_extra.c`' has a close equivalent in the myisam directory ('`\myisam\mi_extra.c`') with the same descriptive comment. (Some of the differences arise because `HEAP` has different structures. `HEAP` does not need to use the sort of B-tree indexing that `ISAM` and `MyISAM` use; instead there is a hash index. Most importantly, `HEAP` is entirely in memory. File-I/O routines lose some of their vitality in such a context.)

- hp_block.c – Read/write a block (i.e. a page)
- hp_clear.c – Remove all records in the table
- hp_close.c – * close database
- hp_create.c – * create a table
- hp_delete.c – * delete a row

- hp_extra.c – * for setting options and buffer sizes when optimizing
- hp_hash.c – Hash functions used for saving keys
- hp_info.c – * Information about database status
- hp_open.c – * open database
- hp_panic.c – * the hp_panic routine, for shutdowns and flushes
- hp_rename.c – * rename a table
- hp_rfirst.c – * read first row through a specific key (very short)
- hp_rkey.c – * read record using a key
- hp_rlast.c – * read last row with same key as previously-read row
- hp_rnext.c – * read next row with same key as previously-read row
- hp_rprev.c – * read previous row with same key as previously-read row
- hp_rrnd.c – * read a row based on position
- hp_rsame.c – * find current row using positional read or key-based read
- hp_scan.c – * read all rows sequentially
- hp_static.c – * static variables (very short)
- hp_test1.c – * testing basic functions
- hp_test2.c – * testing database and storing results
- hp_update.c – * update an existing row
- hp_write.c – * insert a new row

There are fewer files in the heap directory than in the myisam directory, because fewer are necessary. For example, there is no need for a \myisam\mi_cache.c equivalent (to cache reads) or a \myisam\log.c equivalent (to log statements).

14.1.11 include

Header (*.h) files for most libraries; includes all header files distributed with the MySQL binary distribution.

These files may be included in C program files. Note that each individual directory will also have its own *.h files, for including in its own *.c programs. The *.h files in the include directory are ones that might be included from more than one place.

For example, the mysys directory contains a C file named rijndael.c, but does not include rijndael.h. The include directory contains rijndael.h. Looking further, you'll find that rijndael.h is also included in other places: by my_aes.c and my_aes.h.

The include directory contains 51 *.h (header) files.

14.1.12 innobase

The Innobase (InnoDB) table handler.

A full description of these files can be found elsewhere in this document.

14.1.13 libmysql

The MySQL Library, Part 1.

The files here are for producing MySQL as a library (e.g. a Windows DLL). The idea is that, instead of producing separate `mysql` (client) and `mysqld` (server) programs, one produces a library. Instead of sending messages, the client part merely calls the server part.

The `libmysql` files are split into three directories: '`libmysql`' (this one), '`libmysql_r`' (the next one), and '`libmysqld`' (the next one after that).

The "library of mysql" has some client-connection modules. For example, as described in an earlier section of this manual, there is a discussion of '`libmysql/libmysql.c`' which sends packets from the client to the server. Many of the entries in the '`libmysql`' directory (and in the following '`libmysqld`' directory) are 'symlinks' on Linux, that is, they are in fact pointers to files in other directories.

The program files on this directory are:

- conf_to_src.c – has to do with charsets
- dll.c – initialization of the dll library
- errmsg.c – English error messages, compare \mysys\errors.c
- get_password.c – get password
- libmysql.c – the code that implements the MySQL API, i.e. the functions a client that wants to connect to MySQL will call
- manager.c – initialize/connect/fetch with MySQL manager

14.1.14 libmysql_r

The MySQL Library, Part 2.

There is only one file here, used to build a thread-safe libmysql library:

- makefile.am

14.1.15 libmysqld

The MySQL library, Part 3.

The Embedded MySQL Server Library. The product of `libmysqld` is not a client/server affair, but a library. There is a wrapper to emulate the client calls. The program files on this directory are:

- libmysqld.c – The called side, compare the mysqld.exe source
- lib_vio.c – Emulate the vio directory's communication buffer

14.1.16 man

Some user-contributed manual pages

These are user-contributed "man" (manual) pages in a special markup format. The format is described in a document with a heading like "man page for man" or "macros to format man pages" which you can find in a Linux directory or on the Internet.

14.1.17 myisam

The `MyISAM` table handler.

The C files in this subdirectory come in six main groups:

- ft*.c files – ft stands for "Full Text", code contributed by Sergei Golubchik
- mi*.c files – mi stands for "My Isam", these are the main programs for Myisam
- myisam*.c files – for example, "myisamchk" utility routine functions source
- rt*.c files – rt stands for "rtree", some code was written by Alexander Barkov

- sp*.c files – sp stands for "spatial", some code was written by Ramil Kalimullin
- sort.c – this is a single file that sorts keys for index-create purposes

The "full text" and "rtree" and "spatial" program sets are for special purposes, so this document focuses only on the mi*.c "myisam" C programs. They are:

- mi_cache.c – for reading records from a cache
- mi_changed.c – a single routine for setting a "changed" flag (very short)
- mi_check.c – for checking and repairing tables. Used by the myisamchk program and by the MySQL server.
- mi_checksum.c – calculates a checksum for a row
- mi_close.c – close database
- mi_create.c – create a table
- mi_dbug.c – support routines for use with "dbug" (see \dbug description)
- mi_delete.c – delete a row
- mi_delete_all.c – delete all rows
- mi_delete_table.c – delete a table (very short)
- mi_dynrec.c – functions to handle space-packed records and blobs
- mi_extra.c – setting options and buffer sizes when optimizing
- mi_info.c – return useful base information for an open table
- mi_key.c – for handling keys
- mi_locking.c – lock database
- mi_log.c – save commands in a log file which myisamlog program can read. Can be used to exactly replay a set of changes to a table.
- mi_open.c – open database
- mi_packrec.c – read from a data file compresed with myisampack
- mi_page.c – read and write pages containing keys
- mi_panic.c – the mi_panic routine, probably for sudden shutdowns
- mi_range.c – approximate count of how many records lie between two keys
- mi_rename.c – rename a table
- mi_rfirst.c – read first row through a specific key (very short)
- mi_rkey.c – read a record using a key
- mi_rlast.c – read last row with same key as previously-read row
- mi_rnext.c – read next row with same key as previously-read row
- mi_rnext_same.c – same as mi_rnext.c, but abort if the key changes
- mi_rprev.c – read previous row with same key as previously-read row
- mi_rrnd.c – read a row based on position
- mi_rsame.c – find current row using positional read or key-based read
- mi_rsamepos.c – positional read
- mi_scan.c – read all rows sequentially
- mi_search.c – key-handling functions
- mi_static.c – static variables (very short)
- mi_statrec.c – functions to handle fixed-length records
- mi_test1.c – testing basic functions
- mi_test2.c – testing database and storing results

- mi_test3.c – testing locking
- mi_unique.c – functions to check if a row is unique
- mi_update.c – update an existing row
- mi_write.c – insert a new row

14.1.18 myisammrg

`MyISAM` Merge table handler.

As with other table handlers, you'll find that the '`*.c`' files in the '`myissammrg`' directory have counterparts in the '`myisam`' directory. In fact, this general description of a `myisammrg` program is almost always true: The '`myisammrg`' function checks an argument, the '`myisammrg`' function formulates an expression for passing to a '`myisam`' function, the '`myisammrg`' calls a '`myisam`' function, the '`myisammrg`' function returns.

These are the 21 files in the '`myisammrg`' directory, with notes about the `myisam` functions or programs they're connected with:

- myrg_close.c – mi_close.c
- myrg_create.c – mi_create.c
- myrg_delete.c – mi_delete.c / delete last-read record
- myrg_extra.c – mi_extra.c / "extra functions we want to do ..."
- myrg_info.c – mi_info.c / display information about a mymerge file
- myrg_locking.c – mi_locking.c / lock databases
- myrg_open.c – mi_open.c / open a `MyISAM MERGE` table
- myrg_panic.c – mi_panic.c / close in a hurry
- myrg_queue.c – read record based on a key
- myrg_range.c – mi_range.c / find records in a range
- myrg_rfirst.c – mi_rfirst.c / read first record according to specific key
- myrg_rkey.c – mi_rkey.c / read record based on a key
- myrg_rlast.c – mi_rlast.c / read last row with same key as previous read
- myrg_rnext.c – mi_rnext.c / read next row with same key as previous read
- myrg_rnext_same.c – mi_rnext_same.c / read next row with same key
- myrg_rprev.c – mi_rprev.c / read previous row with same key
- myrg_rrnd.c – mi_rrnd.c / read record with random access
- myrg_rsame.c – mi_rsame.c / call mi_rsame function, see \myisam\mi_rsame.c
- myrg_static.c – mi_static.c / static variable declaration
- myrg_update.c – mi_update.c / call mi_update function, see \myisam\mi_update.c
- myrg_write.c – mi_write.c / call mi_write function, see \myisam\mi_write.c

14.1.19 mysql-test

A test suite for mysqld.

The directory has a '`README`' file which explains how to run the tests, how to make new tests (in files with the filename extension '`*.test`'), and how to report errors.

There are four subdirectories:

- \misc – contains one minor Perl program

- \r – contains *.result, i.e. "what happened" files and *.required, i.e. "what should happen" file
- \std_data – contains standard data for input to tests
- \t – contains tests

There are 186 '*.test' files in the '\t' subdirectory. Primarily these are SQL scripts which try out a feature, output a result, and compare the result with what's required. Some samples of what the test files check are: latin1_de comparisons, date additions, the `HAVING` clause, outer joins, openSSL, load data, logging, truncate, and `UNION`.

There are other tests in these directories:

- sql-bench
- tests

14.1.20 mysys

MySQL system library. Low level routines for file access and so on.

There are 115 *.c programs in this directory:

- array.c – Dynamic array handling
- charset.c – Using dynamic character sets, set default character set, ...
- charset2html.c – Check what character set a browser is using
- checksum.c – Calculate checksum for a memory block, used for pack_isam
- default.c – Find defaults from *.cnf or *.ini files
- errors.c – English text of global errors
- hash.c – Hash search/compare/free functions "for saving keys"
- list.c – Double-linked lists
- make-conf.c – "Make a charset .conf file out of a ctype-charset.c file"
- md5.c – MD5 ("Message Digest 5") algorithm from RSA Data Security
- mf_brkhant.c – Prevent user from doing a Break during critical execution (not used in MySQL; can be used by standalone `MyISAM` applications)
- mf_cache.c – "Open a temporary file and cache it with io_cache"
- mf_dirname.c – Parse/convert directory names
- mf_fn_ext.c – Get filename extension
- mf_format.c – Format a filename
- mf_getdate.c – Get date, return in yyyy-mm-dd hh:mm:ss format
- mf_iocache.c – Cached read/write of files in fixed-size units
- mf_iocache2.c – Continuation of mf_iocache.c
- mf_keycache.c – Key block caching for certain file types
- mf_loadpath.c – Return full path name (no ..\ stuff)
- mf_pack.c – Packing/unpacking directory names for create purposes
- mf_path.c – Determine where a program can find its files
- mf_qsort.c – Quicksort
- mf_qsort2.c – Quicksort, part 2 (allows the passing of an extra argument to the sort-compare routine)
- mf_radix.c – Radix sort
- mf_same.c – Determine whether filenames are the same

- mf_sort.c – Sort with choice of Quicksort or Radix sort
- mf_soundex.c – Soundex algorithm derived from EDN Nov. 14, 1985 (pg. 36)
- mf_strip.c – Strip trail spaces from a string
- mf_tempdir.c – Initialize/find/free temporary directory
- mf_tempfile.c – Create a temporary file
- mf_unixpath.c – Convert filename to UNIX-style filename
- mf_util.c – Routines, #ifdef'd, which may be missing on some machines
- mf_wcomp.c – Comparisons with wildcards
- mf_wfile.c – Finding files with wildcards
- mulalloc.c – Malloc many pointers at the same time
- my_aes.c – AES encryption
- my_alarm.c – Set a variable value when an alarm is received
- my_alloc.c – malloc of results which will be freed simultaneously
- my_append.c – one file to another
- my_bit.c – smallest X where 2^X >= value, maybe useful for divisions
- my_bitmap.c – Handle uchar arrays as large bitmaps
- my_chsize.c – Truncate file if shorter, else fill with a filler character
- my_clock.c – Time-of-day ("clock()") function, with OS-dependent #ifdef's
- my_compress.c – Compress packet (see also description of \zlib directory)
- my_copy.c – Copy files
- my_create.c – Create file
- my_delete.c – Delete file
- my_div.c – Get file's name
- my_dup.c – Open a duplicated file
- my_error.c – Return formatted error to user
- my_fopen.c – File open
- my_fstream.c – Streaming file read/write
- my_getwd.c – Get working directory
- my_gethostbyname.c – Thread-safe version of standard net gethostbyname() func
- my_getopt.c – Find out what options are in effect
- my_handler.c – Compare two keys in various possible formats
- my_init.c – Initialize variables and functions in the mysys library
- my_lib.c – Compare/convert directory names and file names
- my_lock.c – Lock part of a file
- my_lockmem.c – "Allocate a block of locked memory"
- my_lread.c – Read a specified number of bytes from a file into memory
- my_lwrite.c – Write a specified number of bytes from memory into a file
- my_malloc.c – Malloc (memory allocate) and dup functions
- my_messnc.c – Put out a message on stderr with "no curses"
- my_mkdir.c – Make directory
- my_net.c – Thread-safe version of net inet_ntoa function
- my_netware.c – Functions used only with the Novell Netware version of MySQL
- my_once.c – Allocation / duplication for "things we don't need to free"

- my_open.c – Open a file
- my_os2cond.c – OS2-specific: "A simple implementation of posix conditions"
- my_os2dirsrch.c – OS2-specific: Emulate a Win32 directory search
- my_os2dlfcn.c – OS2-specific: Emulate UNIX dynamic loading
- my_os2file64.c – OS2-specific: For File64bit setting
- my_os2mutex.c – OS2-specific: For mutex handling
- my_os2thread.c – OS2-specific: For thread handling
- my_os2tls.c – OS2-specific: For thread-local storage
- my_port.c – OS/machine-dependent porting functions, e.g. AIX-specific my_ulonglong2double()
- my_pread.c – Read a specified number of bytes from a file
- my_pthread.c – A wrapper for thread-handling functions in different OSs
- my_quick.c – Read/write (labeled a "quicker" interface, perhaps obsolete)
- my_read.c – Read a specified number of bytes from a file, possibly retry
- my_realloc.c – Reallocate memory allocated with my_alloc.c (probably)
- my_redel.c – Rename and delete file
- my_rename.c – Rename without delete
- my_seek.c – Seek, i.e. point to a spot within a file
- my_semaphore.c – Semaphore routines, for use on OS that doesn't support them
- my_sleep.c – Wait n microseconds
- my_static.c – Static variables used by the mysys library
- my_symlink.c – Read a symbolic link (symlinks are a UNIX thing, I guess)
- my_symlink2.c – Part 2 of my_symlink.c
- my_tempnam.c – Obsolete temporary-filename routine used by ISAM table handler
- my_thr_init.c – initialize/allocate "all mysys & debug thread variables"
- my_wincond.c – Windows-specific: emulate Posix conditions
- my_winsem.c – Windows-specific: emulate Posix threads
- my_winthread.c – Windows-specific: emulate Posix threads
- my_write.c – Write a specified number of bytes to a file
- ptr_cmp.c – Point to an optimal byte-comparison function
- queues.c – Handle priority queues as in Robert Sedgewick's book
- raid2.c – RAID support (the true implementation is in raid.cc)
- rijndael.c – "Optimized ANSI C code for the Rijndael cipher (now AES")
- safemalloc.c – A version of the standard malloc() with safety checking
- sha1.c – Implementation of Secure Hashing Algorithm 1
- string.c – Initialize/append/free dynamically-sized strings; see also sql_string.cc in the /sql directory
- testhash.c – Standalone program: test the hash library routines
- test_charset.c – Standalone program: display character set information
- test_dir.c – Standalone program: placeholder for "test all functions" idea
- test_fn.c – Standalone program: apparently tests a function
- test_xml.c – Standalone program: test XML routines
- thr_alarm.c – Thread alarms and signal handling

- thr_lock.c – "Read and write locks for Posix threads"
- thr_mutex.c – A wrapper for mutex functions
- thr_rwlock.c – Synchronizes the readers' thread locks with the writer's lock
- tree.c – Initialize/search/free binary trees
- typelib.c – Find a string in a set of strings; returns the offset to the string found

You can find documentation for the main functions in these files elsewhere in this document. For example, the main functions in 'my_getwd.c' are described thus:

```
"int my_getwd _A((string buf, uint size, myf MyFlags));
     int my_setwd _A((const char *dir, myf MyFlags));
     Get and set working directory."
```

14.1.21 netware

Files related to the Novell NetWare version of MySQL.

There are 39 files on this directory. Most have filename extensions of '*.def', '*.sql', or '*.c'.

The twenty-five '*.def' files are all from Novell Inc. They contain import or export symbols. ('.def' is a common filename extension for "definition".)

The two '*.sql' files are short scripts of SQL statements used in testing.

These are the five *.c files, all from Novell Inc.:

- libmysqlmain.c – Only one function: init_available_charsets()
- my_manage.c – Standalone management utility
- mysql_install_db.c – Compare \scripts\mysql_install_db.sh
- mysql_test_run.c – Short test program
- mysqld_safe.c – Compare \scripts\mysqld_safe.sh

Perhaps the most important file is:

- netware.patch – NetWare-specific build instructions and switches (compare \mysql-4.1\ltmain.sh)

For instructions about basic installation, see "Deployment Guide For NetWare AMP" at: http://developer.novell.com/ndk/whitepapers/namp.htm

14.1.22 NEW-RPMS

Directory to place RPMs while making a distribution.

This directory is not part of the Windows distribution. It is a temporary directory used during RPM builds with Linux distributions.

14.1.23 os2

Routines for working with the OS2 operating system.

The files in this directory are the product of the efforts of three people from outside MySQL: Yuri Dario, Timo Maier, and John M Alfredsson. There are no '.C' program files in this directory.

The contents of \os2 are:

- A Readme.Txt file
- An \include subdirectory containing .h files which are for OS/2 only
- Files used in the build process (configuration, switches, and one .obj)

The README file refers to MySQL version 3.23, which suggests that there have been no updates for MySQL 4.0 for this section.

14.1.24 pstack

Process stack display (not currently used).

This is a set of publicly-available debugging aids which all do pretty well the same thing: display the contents of the stack, along with symbolic information, for a running process. There are versions for various object file formats (such as ELF and IEEE-695). Most of the programs are copyrighted by the Free Software Foundation and are marked as "part of GNU Binutils".

In other words, the pstack files are not really part of the MySQL library. They are merely useful when you re-program some MYSQL code and it crashes.

14.1.25 regex

Henry Spencer's Regular Expression library for support of REGEXP function.

This is the copyrighted product of Henry Spencer from the University of Toronto. It's a fairly-well-known implementation of the requirements of POSIX 1003.2 Section 2.8. The library is bundled with Apache and is the default implementation for regular-expression handling in BSD Unix. MySQL's Monty Widenius has made minor changes in three programs (debug.c, engine.c, regexec.c) but this is not a MySQL package. MySQL calls it only in order to support two MySQL functions: REGEXP and RLIKE.

Some of Mr Spencer's documentation for the regex library can be found in the README and WHATSNEW files.

One MySQL program which uses regex is \cmd-line-utils\libedit\search.c

This program calls the 'regcomp' function, which is the entry point in \regex\regexp.c.

14.1.26 SCCS

Source Code Control System (not part of source distribution).

You will see this directory if and only if you used BitKeeper for downloading the source. The files here are for BitKeeper administration and are not of interest to application programmers.

14.1.27 scripts

SQL batches, e.g. mysqlbug and mysql_install_db.

The '`*.sh`' filename extension stands for "shell script". Linux programmers use it where Windows programmers would use a '`*.bat`' (batch filename extension).

The '`*.sh`' files on this directory are:

- fill_help_tables.sh – Create help-information tables and insert
- make_binary_distribution.sh – Get configure information, make, produce tar
- msql2mysql.sh – Convert (partly) mSQL programs and scripts to MySQL
- mysqlbug.sh – Create a bug report and mail it
- mysqld_multi.sh – Start/stop any number of mysqld instances
- mysqld_safe-watch.sh – Start/restart in safe mode
- mysqld_safe.sh – Start/restart in safe mode
- mysqldumpslow.sh – Parse and summarize the slow query log
- mysqlhotcopy.sh – Hot backup
- mysql_config.sh – Get configuration information that might be needed to compile a client

- mysql_convert_table_format.sh – Conversion, e.g. from ISAM to MyISAM
- mysql_explain_log.sh – Put a log (made with --log) into a MySQL table
- mysql_find_rows.sh – Search for queries containing <regexp>
- mysql_fix_extensions.sh – Renames some file extensions, not recommended
- mysql_fix_privilege_tables.sh – Fix mysql.user etc. when upgrading. Can be safely run during any upgrade to get the newest MySQL privilege tables
- mysql_install_db.sh – Create privilege tables and func table
- mysql_secure_installation.sh – Disallow remote root login, eliminate test, etc.
- mysql_setpermission.sh – Aid to add users or databases, sets privileges
- mysql_tableinfo.sh – Puts info re MySQL tables into a MySQL table
- mysql_zap.sh – Kill processes that match pattern

14.1.28 sql

Programs for handling SQL commands. The "core" of MySQL.

These are the '.c' and '.cc' files in the 'sql' directory:

- convert.cc – convert tables between different character sets
- derror.cc – read language-dependent message file
- des_key_file.cc – load DES keys from plaintext file
- field.cc – "implement classes defined in 'field.h'" (long); defines all storage methods MySQL uses to store field information into records that are then passed to handlers
- field_conv.cc – functions to copy data between fields
- filesort.cc – sort a result set, using memory or temporary files
- frm_crypt.cc – contains only one short function: 'get_crypt_for_frm'
- gen_lex_hash.cc – Knuth's algorithm from Vol 3 Sorting and Searching, Chapter 6.3; used to search for SQL keywords in a query
- gstream.cc – GTextReadStream, used to read GIS objects
- handler.cc – handler-calling functions
- hash_filo.cc – static-sized hash tables, used to store info like hostname -> ip tables in a FIFO manner
- ha_berkeley.cc – Handler: BDB
- ha_heap.cc – Handler: Heap
- ha_innodb.cc – Handler: InnoDB
- ha_isam.cc – Handler: ISAM
- ha_isammrg.cc – Handler: (ISAM MERGE)
- ha_myisam.cc – Handler: MyISAM
- ha_myisammrg.cc – Handler: (MyISAM MERGE)
- hostname.cc – Given IP, return hostname
- init.cc – Init and dummy functions for interface with unireg
- item.cc – Item functions
- item_buff.cc – Buffers to save and compare item values
- item_cmpfunc.cc – Definition of all compare functions
- item_create.cc – Create an item. Used by 'lex.h'.
- item_func.cc – Numerical functions

- item_row.cc – Row items for comparing rows and for IN on rows
- item_sum.cc – Set functions (SUM(), AVG(), etc.)
- item_strfunc.cc – String functions
- item_subselect.cc – Item subquery
- item_timefunc.cc – Date/time functions, e.g. week of year
- item_uniq.cc – Empty file, here for compatibility reasons
- key.cc – Functions to create keys from records and compare a key to a key in a record
- lock.cc – Locks
- log.cc – Logs
- log_event.cc – Log event (a binary log consists of a stream of log events)
- matherr.c – Handling overflow, underflow, etc.
- mf_iocache.cc – Caching of (sequential) reads and writes
- mini_client.cc – Client included in server for server-server messaging; used by the replication code
- mysqld.cc – Source for mysqld.exe; includes the main() program that starts mysqld, handling of signals and connections
- my_lock.c – Lock part of a file (like '/mysys/my_lock.c', but with timeout handling for threads)
- net_serv.cc – Read/write of packets on a network socket
- nt_servc.cc – Initialize/register/remove an NT service
- opt_ft.cc – Create a FT or QUICK RANGE based on a key (very short)
- opt_range.cc – Range of keys
- opt_sum.cc – Optimize functions in presence of (implied) GROUP BY
- password.c – Password checking
- procedure.cc – Procedure interface, as used in SELECT * FROM Table_name PROCEDURE ANALYSE()
- protocol.cc – Low level functions for PACKING data that is sent to client; actual sending done with 'net_serv.cc'
- records.cc – Functions for easy reading of records, possible through a cache
- repl_failsafe.cc – Replication fail-save (not yet implemented)
- set_var.cc – Set and retrieve MySQL user variables
- slave.cc – Procedures for a slave in a master/slave (replication) relation
- spatial.cc – Geometry stuff (lines, points, etc.)
- sql_acl.cc – Functions related to ACL security; checks, stores, retrieves, and deletes MySQL user level privileges
- sql_analyse.cc – Implements the PROCEDURE ANALYSE(), which analyzes a query result and returns the 'optimal' data type for each result column
- sql_base.cc – Basic functions needed by many modules, like opening and closing tables with table cache management
- sql_cache.cc – SQL query cache, with long comments about how caching works
- sql_class.cc – SQL class; implements the SQL base classes, of which THD (THREAD object) is the most important
- sql_crypt.cc – Encode / decode, very short
- sql_db.cc – Create / drop database

- sql_delete.cc – The `DELETE` statement
- sql_derived.cc – Derived tables, with long comments
- sql_do.cc – The `DO` statement
- sql_error.cc – Errors and warnings
- sql_handler.cc – Implements the `HANDLER` interface, which gives direct access to rows in `MyISAM` and `InnoDB`
- sql_help.cc – The `HELP` statement
- sql_insert.cc – The `INSERT` statement
- sql_lex.cc – Does lexical analysis of a query; i.e. breaks a query string into pieces and determines the basic type (number, string, keyword, etc.) of each piece
- sql_list.cc – Only list_node_end_of_list, short (the rest of the list class is implemented in '`sql_list.h`')
- sql_load.cc – The `LOAD DATA` statement
- sql_map.cc – Memory-mapped files (not yet in use)
- sql_manager.cc – Maintenance tasks, e.g. flushing the buffers periodically; used with `BDB` table logs
- sql_olap.cc – `ROLLUP`
- sql_parse.cc – Parse an SQL statement; do initial checks and then jump to the function that should execute the statement
- sql_prepare.cc – Prepare an SQL statement, or use a prepared statement
- sql_repl.cc – Replication
- sql_rename.cc – Rename table
- sql_select.cc – Select and join optimization
- sql_show.cc – The `SHOW` statement
- sql_string.cc – String functions: alloc, realloc, copy, convert, etc.
- sql_table.cc – The `DROP TABLE` and `ALTER TABLE` statements
- sql_test.cc – Some debugging information
- sql_udf.cc – User-defined functions
- sql_union.cc – The `UNION` operator
- sql_update.cc – The `UPDATE` statement
- stacktrace.c – Display stack trace (Linux/Intel only)
- table.cc – Table metadata retrieval; read the table definition from a '`.frm`' file and store it in a TABLE object
- thr_malloc.cc – Thread-safe interface to '`/mysys/my_alloc.c`'
- time.cc – Date and time functions
- udf_example.cc – Example file of user-defined functions
- uniques.cc – Function to handle quick removal of duplicates
- unireg.cc – Create a unireg form file (.frm) from a `FIELD` and field-info struct

14.1.29 sql-bench

The MySQL Benchmarks.

This directory has the programs and input files which MySQL uses for its comparisons of MySQL, PostgreSQL, mSQL, Solid, etc. Since MySQL publishes the comparative results, it's only right that it should make available all the material necessary to reproduce all the tests.

There are five subdirectories and sub-subdirectories:

- \Comments – Comments about results from tests of Access, Adabas, etc.
- \Data\ATIS – '.txt' files containing input data for the "ATIS" tests
- \Data\Wisconsin – '.txt' files containing input data for the "Wisconsin" tests
- \Results – old test results
- \Results-win32 – old test results from Windows 32-bit tests

There are twenty-four '*.sh' (shell script) files, which involve Perl programs.

There are three '*.bat' (batch) files.

There is one README file and one TODO file.

14.1.30 SSL

Secure Sockets Layer; includes an example certification one can use test an SSL (secure) database connection.

This isn't a code directory. It contains a short note from Tonu Samuel (the NOTES file) and seven '*.pem' files. PEM stands for "Privacy Enhanced Mail" and is an Internet standard for adding security to electronic mail. Finally, there are two short scripts for running clients and servers over SSL connections.

14.1.31 strings

The string library.

Many of the files in this subdirectory are equivalent to well-known functions that appear in most C string libraries. For those, there is documentation available in most compiler handbooks.

On the other hand, some of the files are MySQL additions or improvements. Often the MySQL changes are attempts to optimize the standard libraries. It doesn't seem that anyone tried to optimize for recent Pentium class processors, though.

The .C files are:

- atof.c – ascii-to-float, MySQL version
- bchange.c – short replacement routine written by Monty Widenius in 1987
- bcmp.c – binary compare, rarely used
- bcopy-duff.c – block copy: attempt to copy memory blocks faster than cmemcpy
- bfill.c – byte fill, to fill a buffer with (length) copies of a byte
- bmove.c – block move
- bmove512.c – "should be the fastest way to move a multiple of 512 bytes"
- bmove_upp.c – bmove.c variant, starting with last byte
- bzero.c – something like bfill with an argument of 0
- conf_to_src.c – reading a configuration file
- ctype*.c – string handling programs for each char type MySQL handles
- do_ctype.c – display case-conversion and sort-conversion tables
- int2str.c – integer-to-string
- is_prefix.c – checks whether string1 starts with string2
- llstr.c – convert long long to temporary-buffer string, return pointer
- longlong2str.c – ditto, but to argument-buffer
- memcmp.c – memory compare
- memset.c – memory set

- my_vsnprintf.c – variant of printf
- r_strinstr.c – see if one string is within another
- str2int.c – convert string to integer
- strappend.c – fill up a string to n characters
- strcat.c – concatenate strings
- strcend.c – point to where a character C occurs within str, or NULL
- strchr.c – point to first place in string where character occurs
- strcmp.c – compare two strings
- strcont.c – point to where any one of a set of characters appears
- strend.c – point to the '\0' byte which terminates str
- strfill.c – fill a string with n copies of a byte
- strinstr.c – find string within string
- strlen.c – return length of string in bytes
- strmake.c – create new string from old string with fixed length, append end \0 if needed
- strmov.c – move source to dest and return pointer to end
- strnlen.c – return min(length of string, n)
- strnmov.c – move source to dest for source size, or for n bytes
- strrchr.c – find a character within string, searching from end
- strstr.c – find an instance of pattern within source
- strto.c – string to long, to long long, to unsigned long, etc.
- strtol.c – string to long
- strtoll.c – string to long long
- strtoul.c – string to unsigned long
- strtoull.c – string to unsigned long long
- strxmov.c – move a series of concatenated source strings to dest
- strxnmov.c – like strxmov.c but with a maximum length n
- str_test.c – test of all the string functions encoded in assembler
- udiv.c – unsigned long divide, for operating systems that don't support these
- xml.c – read and parse XML strings; used to read character definition information stored in /sql/share/charsets

There are also four .ASM files – macros.asm, ptr_cmp.asm, strings.asm, and strxmov.asm – which can replace some of the C-program functions. But again, they look like optimizations for old members of the Intel processor family.

14.1.32 support-files

Files used to build MySQL on different systems.

The files here are for building ("making") MySQL given a package manager, compiler, linker, and other build tools. The support files provide instructions and switches for the build processes. They include example my.cnf files one can use as a default setup for MySQL.

14.1.33 tests

Tests in Perl and in C.

The files in this directory are test programs that can be used as a base to write a program to simulate problems in MySQL in various scenarios: forks, locks, big records, exporting, truncating, and so on. Some examples are:

- connect_test.c – test that a connect is possible
- insert_test.c – test that an insert is possible
- list_test.c – test that a select is possible
- select_test.c – test that a select is possible
- showdb_test.c – test that a show-databases is possible
- ssl_test.c – test that SSL is possible
- thread_test.c – test that threading is possible

14.1.34 tools

Tools – well, actually, one tool.

The only file is:

- mysqlmanager.c – A "server management daemon" by Sasha Pachev. This is a tool under development and is not yet useful. Related to fail-safe replication.

14.1.35 VC++Files

Visual C++ Files.

Includes this entire directory, repeated for VC++ (Windows) use.

VC++Files includes a complete environment to compile MySQL with the VC++ compiler. To use it, just copy the files on this directory; the make_win_src_distribution.sh script uses these files to create a Windows source installation.

This directory has subdirectories which are copies of the main directories. For example, there is a subdirectory \VC++Files\heap, which has the Microsoft developer studio project file to compile \heap with VC++. So for a description of the files in \VC++Files\heap, see the description of the files in \heap. The same applies for almost all of VC++Files's subdirectories (bdb, client, isam, libmysql, etc.). The difference is that the \VC++Files variants are specifically for compilation with Microsoft Visual C++ in 32-bit Windows environments.

In addition to the "subdirectories which are duplicates of directories", VC++Files contains these subdirectories, which are not duplicates:

- comp_err – (nearly empty)
- contrib – (nearly empty)
- InstallShield – script files
- isamchk – (nearly empty)
- libmysqltest – one small non-MySQL test program: mytest.c
- myisamchk – (nearly empty)
- myisamlog – (nearly empty)
- myisammrg – (nearly empty)
- mysqlbinlog – (nearly empty)
- mysqlmanager – MFC foundation class files created by AppWizard

- mysqlserver – (nearly empty)
- mysqlshutdown – one short program, mysqlshutdown.c
- mysqlwatch.c – Windows service initialization and monitoring
- my_print_defaults – (nearly empty)
- pack_isam – (nearly empty)
- perror – (nearly empty)
- prepare – (nearly empty)
- replace – (nearly empty)
- SCCS – source code control system
- test1 – tests connecting via X threads
- thr_insert_test – (nearly empty)
- thr_test – one short program used to test for memory-allocation bug
- winmysqladmin – the winmysqladmin.exe source

The "nearly empty" subdirectories noted above (e.g. comp_err and isamchk) are needed because VC++ requires one directory per project (i.e. executable). We are trying to keep to the MySQL standard source layout and compile only to different directories.

14.1.36 vio

Virtual I/O Library.

The VIO routines are wrappers for the various network I/O calls that happen with different protocols. The idea is that in the main modules one won't have to write separate bits of code for each protocol. Thus vio's purpose is somewhat like the purpose of Microsoft's winsock library.

The underlying protocols at this moment are: TCP/IP, Named Pipes (for WindowsNT), Shared Memory, and Secure Sockets (SSL).

The C programs are:

- test-ssl.c – Short standalone test program: SSL
- test-sslclient.c – Short standalone test program: clients
- test-sslserver.c – Short standalone test program: server
- vio.c – Declarations + open/close functions
- viosocket.c – Send/retrieve functions
- viossl.c – SSL variations for the above
- viosslfactories.c – Certification / Verification
- viotest.cc – Short standalone test program: general
- viotest-ssl.c – Short standalone test program: SSL
- viotest-sslconnect.cc – Short standalone test program: SSL connect

The older functions – raw_net_read, raw_net_write – are now obsolete.

14.1.37 zlib

Data compression library, used on Windows.

zlib is a data compression library used to support the compressed protocol and the COMPRESS/UNCOMPRESS functions under Windows. On Unix, MySQL uses the system libgz.a library for this purpose.

Zlib – which presumably stands for "Zip Library" – is not a MySQL package. It was produced by the GNU Zip (gzip.org) people. Zlib is a variation of the famous "Lempel-Ziv" method, which is also used by "Zip". The method for reducing the size of any arbitrary string of bytes is as follows:

- Find a substring which occurs twice in the string.
- Replace the second occurrence of the substring with (a) a pointer to the first occurrence, plus (b) an indication of the length of the first occurrence.

There is a full description of the library's functions in the gzip manual at `http://www.gzip.org/zlib/manual.html`. There is therefore no need to list the modules in this document.

The MySQL program \mysys\my_compress.c uses zlib for packet compression. The client sends messages to the server which are compressed by zlib. See also: '`\sql\net_serv.cc`'.

15 Annotated List of Files in the InnoDB Source Code Distribution

ERRATUM BY HEIKKI TUURI (START)

Errata about InnoDB row locks:

```
  #define LOCK_S  4 /* shared */
  #define LOCK_X  5 /* exclusive */
...
/* Waiting lock flag */
  #define LOCK_WAIT 256
/* this wait bit should be so high that it can be ORed to the lock
mode and type; when this bit is set, it means that the lock has not
yet been granted, it is just waiting for its turn in the wait queue */
...
/* Precise modes */
  #define LOCK_ORDINARY 0
/* this flag denotes an ordinary next-key lock in contrast to LOCK_GAP
or LOCK_REC_NOT_GAP */
  #define LOCK_GAP 512
/* this gap bit should be so high that it can be ORed to the other
flags; when this bit is set, it means that the lock holds only on the
gap before the record; for instance, an x-lock on the gap does not
give permission to modify the record on which the bit is set; locks of
this type are created when records are removed from the index chain of
records */
  #define LOCK_REC_NOT_GAP 1024
/* this bit means that the lock is only on the index record and does
NOT block inserts to the gap before the index record; this is used in
the case when we retrieve a record with a unique key, and is also used
in locking plain SELECTs (not part of UPDATE or DELETE) when the user
has set the READ COMMITTED isolation level */
  #define LOCK_INSERT_INTENTION 2048
/* this bit is set when we place a waiting gap type record lock
request in order to let an insert of an index record to wait until
there are no conflicting locks by other transactions on the gap; note
that this flag remains set when the waiting lock is granted, or if the
lock is inherited to a neighboring record */
```

ERRATUM BY HEIKKI TUURI (END)

The InnoDB source files are the best place to look for information about internals of the file structure that MySQLites can optionally use for transaction support. But when you first look at all the subdirectories and file names you'll wonder: Where Do I Start? It can be daunting.

Well, I've been through that phase, so I'll pass on what I had to learn on the first day that I looked at InnoDB source files. I am very sure that this will help you grasp, in overview, the organization of InnoDB modules. I'm also going to add comments about what is going on – which you should mistrust! These comments are reasonable working hypotheses; nevertheless, they have not been subjected to expert peer review.

Here's how I'm going to organize the discussion. I'll take each of the 32 InnoDB subdirectories that come with the MySQL 4.0 source code in '\mysql\innobase' (on my Windows directory). The format of each section will be like this every time:

\subdirectory-name (LONGER EXPLANATORY NAME)

File Name	What Name Stands For	Size	Comment Inside File
file-name	my-own-guess	in-bytes	from-the-file-itself

...
My-Comments

For example:

```
"
\ha (HASHING)
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  ha0ha.c     Hashing/Hashing        7,452   Hash table with external chains

  Comments about hashing will be here.
"
```

The "Comment Inside File" column is a direct copy from the first /* comment */ line inside the file. All other comments are mine. After I've discussed each directory, I'll finish with some notes about naming conventions and a short list of URLs that you can use for further reference.

Now let's begin.

```
\ha (HASHING)
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  ha0ha.c     Hashing / Hashing      7,452   Hash table with external chains

I'll hold my comments until the next section, \hash (HASHING).

\hash (HASHING)
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  hash0hash.c Hashing / Hashing      3,257   Simple hash table utility

The two C programs in the \ha and \hashing directories -- ha0ha.c and
hash0hash.c -- both refer to a "hash table" but hash0hash.c is
specialized, it is mostly about accessing points in the table under
mutex control.

When a "database" is so small that InnoDB can load it all into memory
at once, it's more efficient to access it via a hash table. After all,
no disk i/o can be saved by using an index lookup, if there's no disk.

\os (OPERATING SYSTEM)
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  os0shm.c    OS / Shared Memory     3,150   To shared memory primitives
  os0file.c   OS / File             64,412   To i/o primitives
  os0thread.c OS / Thread            6,827   To thread control primitives
  os0proc.c   OS / Process           3,700   To process control primitives
  os0sync.c   OS / Synchronization  10,208   To synchronization primitives

This is a group of utilities that other modules may call whenever they
```

want to use an operating-system resource. For example, in os0file.c there is a public InnoDB function named os_file_create_simple(), which simply calls the Windows-API function CreateFile. Naturally the contents of this group are somewhat different for other operating systems.

The "Shared Memory" functions in os0shm.c are only called from the communications program com0shm.c (see \com COMMUNICATIONS). The i/o and thread-control primitives are called extensively. The word "synchronization" in this context refers to the mutex-create and mutex-wait functionality.

\ut (UTILITIES)

File Name	What Name Stands For	Size	Comment Inside File
ut0ut.c	Utilities / Utilities	7,041	Various utilities
ut0byte.c	Utilities / Debug	1,856	Byte utilities
ut0rnd.c	Utilities / Random	1,475	Random numbers and hashing
ut0mem.c	Utilities / Memory	5,530	Memory primitives
ut0dbg.c	Utilities / Debug	642	Debug utilities

The two functions in ut0byte.c are just for lower/upper case conversion and comparison. The single function in ut0rnd.c is for finding a prime slightly greater than the given argument, which is useful for hash functions, but unrelated to randomness. The functions in ut0mem.c are wrappers for "malloc" and "free" calls -- for the real "memory" module see section \mem (MEMORY). Finally, the functions in ut0ut.c are a miscellany that didn't fit better elsewhere: get_high_bytes, clock, time, difftime, get_year_month_day, and "sprintf" for various diagnostic purposes.

In short: the \ut group is trivial.

\buf (BUFFERING)

File Name	What Name Stands For	Size	Comment Inside File
buf0buf.c	Buffering / Buffering	53,246	The database buffer buf_pool
buf0flu.c	Buffering / Flush	23,711	... flush algorithm
buf0lru.c	/ least-recently-used	20,245	... replacement algorithm
buf0rea.c	Buffering / read	17,399	... read

There is a separate file group (\mem MEMORY) which handles memory requests in general.A "buffer" usually has a more specific definition, as a memory area which contains copies of pages that ordinarily are in the main data file. The "buffer pool" is the set of all buffers (there are lots of them because InnoDB doesn't depend on the OS's caching to make things faster).

The pool size is fixed (at the time of this writing) but the rest of the buffering architecture is sophisticated, involving a host of control structures. In general: when InnoDB needs to access a new page it looks first in the buffer pool; InnoDB reads from disk to a new buffer when the page isn't there; InnoDB chucks old buffers (basing

its decision on a conventional Least-Recently-Used algorithm) when it has to make space for a new buffer.

There are routines for checking a page's validity, and for read-ahead. An example of "read-ahead" use: if a sequential scan is going on, then a DBMS can read more than one page at a time, which is efficient because reading 32,768 bytes (two pages) takes less than twice as long as reading 16,384 bytes (one page).

\btr (B-TREE)

File Name	What Name Stands For	Size	Comment Inside File
btr0btr.c	B-tree / B-tree	74,255	B-tree
btr0cur.c	B-tree / Cursor	94,950	index tree cursor
btr0sea.c	B-tree / Search	36,580	index tree adaptive search
btr0pcur.c	B-tree / persistent cursor	14,548	index tree persistent cursor

If you total up the sizes of the C files, you'll see that \btr is the second-largest file group in InnoDB. This is understandable because maintaining a B-tree is a relatively complex task. Luckily, there has been a lot of work done to describe efficient management of B-tree and B+-tree structures, much of it open-source or public-domain, since their original invention over thirty years ago.

InnoDB likes to put everything in B-trees. This is what I'd call a "distinguishing characteristic" because in all the major DBMSs (like IBM DB2, Microsoft SQL Server, and Oracle), the main or default or classic structure is the heap-and-index. In InnoDB the main structure is just the index. To put it another way: InnoDB keeps the rows in the leaf node of the index, rather than in a separate file. Compare Oracle's Index Organized Tables, and Microsoft SQL Server's Clustered Indexes.

This, by the way, has some consequences. For example, you may as well have a primary key since otherwise InnoDB will make one anyway. And that primary key should be the shortest of the candidate keys, since InnoDB
will use it as a pointer if there are secondary indexes.

Most importantly, it means that rows have no fixed address. Therefore the routines for managing file pages should be good. We'll see about that when we look at the \row (ROW) program group later.

\com (COMMUNCATION)

File Name	What Name Stands For	Size	Comment Inside File
com0com.c	Communication	6,913	Communication primitives
com0shm.c	Communication / Shared Memory	24,633	... through shared memory

The communication primitives in com0com.c are said to be modelled after the ones in Microsoft's winsock library (the Windows Sockets

```
interface). The communication primitives in com0shm.c are at a
slightly lower level, and are called from the routines in com0com.c.

I was interested in seeing how InnoDB would handle inter-process
communication, since there are many options -- named pipes, TCP/IP,
Windows messaging, and Shared Memory being the main ones that come to
mind. It appears that InnoDB prefers Shared Memory. The main idea is:
there is an area of memory which two different processes (or threads,
of course) can both access. To communicate, a thread gets an
appropriate mutex, puts in a request, and waits for a response. Thread
interaction is also a subject for the os0thread.c program in another
program group, \os (OPERATING SYSTEM).
```

\dyn (DYNAMICALLY ALLOCATED ARRAY)

```
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  dyn0dyn.c   Dynamic / Dynamic        994   dynamically allocated array
```

```
There is a single function in the dyn0dyn.c program, for adding a
block to the dynamically allocated array. InnoDB might use the array
for managing concurrency between threads.

At the moment, the \dyn program group is trivial.
```

\fil (FILE)

```
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  fil0fil.c   File / File           39,725   The low-level file system
```

```
The reads and writes to the database files happen here, in
co-ordination with the low-level file i/o routines (see os0file.h in
the \os program group).

Briefly: a table's contents are in pages, which are in files, which
are in tablespaces. Files do not grow; instead one can add new files
to the tablespace. As we saw earlier (discussing the \btr program group)
the pages are nodes of B-trees. Since that's the case, new additions can
happen at various places in the logical file structure, not
necessarily at the end. Reads and writes are asynchronous, and go into
buffers, which are set up by routines in the \buf program group.
```

\fsp (FILE SPACE)

```
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  fsp0fsp.c   File Space Management 100,271  File space management
```

```
I would have thought that the \fil (FILE) and \fsp (FILE SPACE)
MANAGEMENT programs would fit together in the same program group;
however, I guess the InnoDB folk are splitters rather than lumpers.

It's in fsp0fsp.c that one finds some of the descriptions and comments
of extents, segments, and headers. For example, the "descriptor bitmap
```

of the pages in the extent" is in here, and you can find as well how the free-page list is maintained, what's in the bitmaps, and what various header fields' contents are.

\fut (FILE UTILITY)

File Name	What Name Stands For	Size	Comment Inside File
fut0fut.c	File Utility / Utility	293	File-based utilities
fut0lst.c	File Utility / List	14,129	File-based list utilities

Mainly these small programs affect only file-based lists, so maybe saying "File Utility" is too generic. The real work with data files goes on in the \fsp program group.

\log (LOGGING)

File Name	What Name Stands For	Size	Comment Inside File
log0log.c	Logging / Logging	77,834	Database log
log0recv.c	Logging / Recovery	80,701	Recovery

I've already written about the \log program group, so here's a link to my previous article: "How Logs work with MySQL and InnoDB": http://www.devarticles.com/art/1/181/2

\mem (MEMORY)

File Name	What Name Stands For	Size	Comment Inside File
mem0mem.c	Memory / Memory	9,971	The memory management
mem0dbg.c	Memory / Debug	21,297	... the debug code
mem0pool.c	Memory / Pool	16,293	... the lowest level

There is a long comment at the start of the mem0pool.c program, which explains what the memory-consumers are, and how InnoDB tries to satisfy them. The main thing to know is that there are really three pools: the buffer pool (see the \buf program group), the log pool (see the \log program group), and the common pool, which is where everything that's not in the buffer or log pools goes (for example the parsed SQL statements and the data dictionary cache).

\mtr (MINI-TRANSACTION)

File Name	What Name Stands For	Size	Comment Inside File
mtr0mtr.c	Mini-transaction /	12,433	Mini-transaction buffer
mtr0log.c	Mini-transaction / Log	8,180	... log routines

The mini-transaction routines are called from most of the other program groups. I'd describe this as a low-level utility set.

\que (QUERY GRAPH)

File Name	What Name Stands For	Size	Comment Inside File
que0que.c	Query Graph / Query	35,964	Query graph

```
The program que0que.c ostensibly is about the execution of stored
procedures which contain commit/rollback statements. I took it that
this has little importance for the average MySQL user.
```

\rem (RECORD MANAGER)

File Name	What Name Stands For	Size	Comment Inside File
rem0rec.c	Record Manager	14,961	Record Manager
rem0cmp.c	Record Manager / Comparison	25,263	Comparison services for records

```
There's an extensive comment near the start of rem0rec.c title
"Physical Record" and it's recommended reading. At some point you'll
ask what are all those bits that surround the data in the rows on a page,
and this is where you'll find the answer.
```

\row (ROW)

File Name	What Name Stands For	Size	Comment Inside File
row0row.c	Row / Row	16,764	General row routines
row0uins.c	Row / Undo Insert	7,199	Fresh insert undo
row0umod.c	Row / Undo Modify	17,147	Undo modify of a row
row0undo.c	Row / Undo	10,254	Row undo
row0vers.c	Row / Version	12,288	Row versions
row0mysql.c	Row / MySQL	63,556	Interface [to MySQL]
row0ins.c	Row / Insert	42,829	Insert into a table
row0sel.c	Row / Select	85,923	Select
row0upd.c	Row / Update	44,456	Update of a row
row0purge.c	Row / Purge	14,961	Purge obsolete records

```
Rows can be selected, inserted, updated/deleted, or purged (a
maintenance activity). These actions have ancillary actions, for
example after insert there can be an index-update test, but it seems
to me that sometimes the ancillary action has no MySQL equivalent (yet)
and so is inoperative.

Speaking of MySQL, notice that one of the larger programs in the \row
program group is the "interface between Innobase row operations and
MySQL" (row0mysql.c) -- information interchange happens at this level
because rows in InnoDB and in MySQL are analogous, something which
can't be said for pages and other levels.
```

\srv (Server)

File Name	What Name Stands For	Size	Comment Inside File
srv0srv.c	Server / Server	79,058	Server main program
srv0que.c	Server / Query	2,361	Server query execution
srv0start.c	Server / Start	34,586	Starts the server

```
This is where the server reads the initial configuration files, splits
up the threads, and gets going. There is a long comment deep in the
```

program (you might miss it at first glance) titled "IMPLEMENTATION OF THE SERVER MAIN PROGRAM" in which you'll find explanations about thread priority, and about what the responsibiities are for various thread types.

InnoDB has many threads, for example "user threads" (which wait for client requests and reply to them), "parallel communication threads" (which take part of a user thread's job if a query process can be split), "utility threads" (background priority), and a "master thread" (high priority, usually asleep).

\thr (Thread Local Storage)

```
File Name   What Name Stands For  Size     Comment Inside File
---------   --------------------  ------   -------------------
thr0loc.c   Thread / Local         5,261   The thread local storage
```

InnoDB doesn't use the Windows-API thread-local-storage functions, perhaps because they're not portable enough.

\trx (Transaction)

```
File Name   What Name Stands For  Size     Comment Inside File
---------   --------------------  ------   -------------------
trx0trx.c   Transaction /         37,447   The transaction
trx0purge.c Transaction / Purge   26,782   ... Purge old versions
trx0rec.c   Transaction / Record  36,525   ... Undo log record
trx0sys.c   Transaction / System  20,671   ... System
trx0rseg.c  / Rollback segment     6,214   ... Rollback segment
trx0undo.c  Transaction / Undo    46,595   ... Undo log
```

InnoDB's transaction management is supposedly "in the style of Oracle" and that's close to true but can mislead you.

- First: InnoDB uses rollback segments like Oracle8i does -- but Oracle9i uses a different name
- Second: InnoDB uses multi-versioning like Oracle does -- but I see nothing that looks like an Oracle ITL being stored in the InnoDB data pages.
- Third: InnoDB and Oracle both have short (back-to-statement-start) versioning for the READ COMMITTED isolation level and long (back-to-transaction-start) versioning for higher levels -- but InnoDB and Oracle have different "default" isolation levels.
- Finally: InnoDB's documentation says it has to lock "the gaps before index keys" to prevent phantoms -- but any Oracle user will tell you that phantoms are impossible anyway at the SERIALIZABLE isolation level, so key-locks are unnecessary.

The main idea, though, is that InnoDB has multi-versioning. So does Oracle. This is very different from the way that DB2 and SQL Server do things.

\usr (USER)

File Name	What Name Stands For	Size	Comment Inside File
usr0sess.c	User / Session	27,415	Sessions

One user can have multiple sessions (the session being all the things that happen betweeen a connect and disconnect). This is where InnoDB tracks session IDs, and server/client messaging. It's another of those items which is usually MySQL's job, though.

\data (DATA)

File Name	What Name Stands For	Size	Comment Inside File
data0data.c	Data / Data	26,002	SQL data field and tuple
data0type.c	Data / Type	2,122	Data types

This is a collection of minor utility routines affecting rows.

\dict (DICTIONARY)

File Name	What Name Stands For	Size	Comment Inside File
dict0dict.c	Dictionary / Dictionary	84,667	Data dictionary system
dict0boot.c	Dictionary / boot	12,134	... creation and booting
dict0load.c	Dictionary / load	26,546	... load to memory cache
dict0mem.c	Dictionary / memory	8,221	... memory object creation

The data dictionary (known in some circles as the catalog) has the metadata information about objects in the database -- column sizes, table names, and the like.

\eval (EVALUATING)

File Name	What Name Stands For	Size	Comment Inside File
eval0eval.c	Evaluating/Evaluating	15,682	SQL evaluator
eval0proc.c	Evaluating/Procedures	5,000	Executes SQL procedures

The evaluating step is a late part of the process of interpreting an SQL statement -- parsing has already occurred during \pars (PARSING).

The ability to execute SQL stored procedures is an InnoDB feature, but not a MySQL feature, so the eval0proc.c program is unimportant.

\ibuf (INSERT BUFFER)

File Name	What Name Stands For	Size	Comment Inside File
ibuf0ibuf.c	Insert Buffer /	69,884	Insert buffer

The words "Insert Buffer" mean not "buffer used for INSERT" but "insertion of a buffer into the buffer pool" (see the \buf BUFFER program group description). The matter is complex due to possibilities for deadlocks, a problem to which the comments in the ibuf0ibuf.c program devote considerable attention.

\mach (MACHINE FORMAT)

```
  File Name   What Name Stands For  Size    Comment Inside File
  ---------   --------------------  ------  -------------------
  mach0data.c Machine/Data           2,319  Utilities for converting
```

```
The mach0data.c program has two small routines for reading compressed
ulints (unsigned long integers).
```

\lock (LOCKING)

```
  File Name   What Name Stands For  Size    Comment Inside File
  ---------   --------------------  ------  -------------------
  lock0lock.c Lock / Lock           127,646 The transaction lock system
```

```
If you've used DB2 or SQL Server, you might think that locks have their
own in-memory table, that row locks might need occasional escalation to
table locks, and that there are three lock types: Shared, Update, Exclusive.

All those things are untrue with InnoDB! Locks are kept in the database
pages. A bunch of row locks can't be rolled together into a single table
lock. And most importantly there's only one lock type. I call this type
"Update" because it has the characteristics of DB2 / SQL Server Update
locks, that is, it blocks other updates but doesn't block reads.
Unfortunately, InnoDB comments refer to them as "x-locks" etc.

To sum it up: if your background is Oracle you won't find too much
surprising, but if your background is DB2 or SQL Server the locking
concepts and terminology will probably confuse you at first.

You can find an online article about the differences between
Oracle-style and DB2/SQL-Server-style locks at:
http://dbazine.com/gulutzan6.html
```

\odbc (ODBC)

```
  File Name   What Name Stands For Size    Comment Inside File
  ---------   --------------------  ------  -------------------
  odbc0odbc.c ODBC / ODBC           16,865  ODBC client library
```

```
The odbc0odbc.c program has a small selection of old ODBC-API
functions: SQLAllocEnv, SQLAllocConnect, SQLAllocStmt, SQLConnect,
SQLError, SQLPrepare, SQLBindParameter, SQLExecute.
```

\page (PAGE)

```
  File Name   What Name Stands For  Size    Comment Inside File
  ---------   --------------------  ------  -------------------
  page0page.c Page / Page           44,309  Index page routines
  page0cur.c  Page / Cursor         30,305  The page cursor
```

```
It's in the 'page0page.c' program that you'll learn as follows: index
pages start with a header, entries in the page are in order, at the
end of the page is a sparse "page directory" (what I would have called
a slot table) which makes binary searches easier.
```

```
Incidentally, the program comments refer to "a page size of 8 kB"
which seems obsolete. In 'univ.i' (a file containing universal
constants) the page size is now #defined as 16KB.
```

\pars (PARSING)

```
  File Name   What Name Stands For Size    Comment Inside File
  ---------   -------------------- ------  -------------------
  pars0pars.c Parsing/Parsing      49,947  SQL parser
  pars0grm.c  Parsing/Grammar      62,685  A Bison parser
  pars0opt.c  Parsing/Optimizer    30,809  Simple SQL Optimizer
  pars0sym.c  Parsing/Symbol Table  5,541  SQL parser symbol table
  lexyy.c     ?/Lexer              59,948  Lexical scanner

The job is to input a string containing an SQL statement and output an
in-memory parse tree. The EVALUATING (subdirectory \eval) programs
will use the tree.

As is common practice, the Bison and Flex tools were used -- 'pars0grm.c'
is what the Bison parser produced from an original file named
'pars0grm.y'
(not supplied), and 'lexyy.c' is what Flex produced.

Since InnoDB is a DBMS by itself, it's natural to find SQL parsing in
it. But in the MySQL/InnoDB combination, MySQL handles most of the
parsing. These files are unimportant.
```

\read (READ)

```
  File Name   What Name Stands For Size    Comment Inside File
  ---------   -------------------- ------  -------------------
  read0read.c Read / Read           6,244  Cursor read

The 'read0read.c' program opens a "read view" of a query result, using
some functions in the \trx program group.
```

\sync (SYNCHRONIZATION)

```
  File Name   What Name Stands For Size    Comment Inside File
  ---------   -------------------- ------  -------------------
  sync0sync.c Synchronization /    35,918  Mutex, the basic sync primitive
  sync0arr.c  ... / array          26,461  Wait array used in primitives
  sync0ipm.c  ... / interprocess    4,027  for interprocess sync
  sync0rw.c   ... / read-write     22,220  read-write lock for thread sync

A mutex (Mutual Exclusion) is an object which only one thread/process
can hold at a time. Any modern operating system API has some functions
for mutexes; however, as the comments in the sync0sync.c code indicate, it
can be faster to write one's own low-level mechanism. In fact the old
assembly-language XCHG trick is in here -- this is the only program
that contains any assembly code.
```

This is the end of the section-by-section account of InnoDB subdirectories.

A Note About File Naming

There appears to be a naming convention. The first letters of the file name are the same as the subdirectory name, then there is a '0' separator, then there is an individual name. For the main program in a subdirectory, the individual name may be a repeat of the subdirectory name. For example, there is a file named ha0ha.c (the first two letters ha mean "it's in in subdirectory ..\ha", the next letter 0 means "0 separator", the next two letters mean "this is the main ha program"). This naming convention is not strict, though: for example the file lexyy.c is in the \pars subdirectory.

A Note About Copyrights

Most of the files begin with a copyright notice or a creation date, for example "Created 10/25/1995 Heikki Tuuri". I don't know a great deal about the history of `InnoDB`, but found it interesting that most creation dates were between 1994 and 1998.

References

- Ryan Bannon, Alvin Chin, Faryaaz Kassam and Andrew Roszko. "InnoDB Concrete Architecture" `http://www.swen.uwaterloo.ca/~mrbannon/cs798/assignment_02/innodb.pdf`

  A student paper. It's an interesting attempt to figure out `InnoDB`'s architecture using tools, but I didn't end up using it for the specific purposes of this article.
- Peter Gulutzan. "How Logs Work With MySQL And InnoDB" `http://www.devarticles.com/art/1/181/2`
- Heikki Tuuri. "InnoDB Engine in MySQL-Max-3.23.54 / MySQL-4.0.9: The Up-to-Date Reference Manual of InnoDB" `http://www.innodb.com/ibman.html`

  This is the natural starting point for all InnoDB information. Mr Tuuri also appears frequently on MySQL forums.

Short Contents

1 Coding Guidelines 3
2 The Optimizer 9
3 Important Algorithms and Structures 21
4 Charsets and Related Issues 27
5 How MySQL Performs Different Selects 29
6 How MySQL Transforms Subqueries 35
7 MySQL Client/Server Protocol 41
8 Replication 65
9 `MyISAM` Record Structure 81
10 The '`.MYI`' file 87
11 `InnoDB` Record Structure 95
12 `InnoDB` Page Structure 99
13 Adding New Error Messages to MySQL 105
14 Annotated List of Files in the MySQL Source Code Distribution 107
15 Annotated List of Files in the `InnoDB` Source Code Distribution 129

Table of Contents

1 Coding Guidelines 3

2 The Optimizer 9
- 2.1 The `Index Merge` Join Type 18
  - 2.1.1 Overview 18
  - 2.1.2 Index Merge Optimizer 18
    - 2.1.2.1 Range Optimizer 18
    - 2.1.2.2 Index Merge Optimizer 19
  - 2.1.3 Row Retrieval Algorithm 20

3 Important Algorithms and Structures 21
- 3.1 How MySQL Does Sorting (`filesort`) 21
- 3.2 Bulk Insert 21
- 3.3 How MySQL Does Caching 22
- 3.4 How MySQL Uses the Join Buffer Cache 22
- 3.5 How MySQL Handles `FLUSH TABLES` 23
- 3.6 Full-text Search in MySQL 24
- 3.7 Functions in the `mysys` Library 24

4 Charsets and Related Issues 27
- 4.1 `CHARSET_INFO` Structure 27

5 How MySQL Performs Different Selects 29
- 5.1 Steps of Select Execution 29
- 5.2 `select_result` Class 29
- 5.3 `SIMPLE` or `PRIMARY SELECT` 29
- 5.4 Structure Of Complex Select 30
- 5.5 Non-Subquery `UNION` Execution 31
- 5.6 Derived Table Execution 31
- 5.7 Subqueries 32
- 5.8 Single Select Engine 32
- 5.9 Union Engine 32
- 5.10 Special Engines 33
- 5.11 Explain Execution 33

6 How MySQL Transforms Subqueries 35
- 6.1 `Item_in_subselect::select_transformer` 35
  - 6.1.1 Scalar `IN` Subquery 35
  - 6.1.2 Row `IN` Subquery 37
- 6.2 `Item_allany_subselect` 37
- 6.3 `Item_singlerow_subselect` 38

7 MySQL Client/Server Protocol 41
- 7.1 Raw Packet Without Compression 41
- 7.2 Raw Packet With Compression 41
- 7.3 Basic Packets 41
  - 7.3.1 OK Packet 41
  - 7.3.2 Error Packet 42
- 7.4 Communication 42
- 7.5 Fieldtype Codes 44
- 7.6 Functions Used to Implement the Protocol 44
- 7.7 Another Description of the Protocol 46
- 7.8 Changes to 4.0 Protocol in 4.1 59
- 7.9 4.1 Field Description Packet 60
- 7.10 4.1 Field Description Result Set 60
- 7.11 4.1 OK Packet 60
- 7.12 4.1 End Packet 61
- 7.13 4.1 Error Packet 61
- 7.14 4.1 Prepared Statement Init Packet 61
- 7.15 4.1 Long Data Handling 62
- 7.16 4.1 Execute 62
- 7.17 4.1 Binary Result Set 63

8 Replication 65
- 8.1 Main Code Files 65
- 8.2 The Binary Log 65
- 8.3 Replication Threads 68
  - 8.3.1 The Slave I/O Thread 68
  - 8.3.2 The Slave SQL Thread 69
  - 8.3.3 Why 2 Threads? 70
  - 8.3.4 The `Binlog Dump` Thread 70
- 8.4 How Replication Deals With... 72
  - 8.4.1 `auto_increment` Columns, `LAST_INSERT_ID()` 72
  - 8.4.2 User Variables (Since 4.1) 72
  - 8.4.3 System Variables 72
  - 8.4.4 Some Functions 72
  - 8.4.5 Non-repeatable UDF Functions 72
  - 8.4.6 Prepared Statements 72
  - 8.4.7 Temporary Tables 72
  - 8.4.8 `LOAD DATA [LOCAL] INFILE` (Since 4.0) 73
- 8.5 How a Slave Asks Its Master to Send Its Binary Log 73
- 8.6 Network Packets in Detail 74
- 8.7 Replication Event Format in Detail 74
  - 8.7.1 The Common Header 74
  - 8.7.2 The “Post-headers” (Event-specific Headers) 75
- 8.8 Plans for MySQL 5.0 79

9 `MyISAM` Record Structure 81
- 9.1 Introduction 81
- 9.2 Physical Attributes of Columns 82
- 9.3 Where to Look For More Information 85

10 The ‘`.MYI`’ file 87
- 10.1 `MyISAM` Files 93

11 InnoDB Record Structure 95
11.1 High-Altitude Picture 95
11.1.1 FIELD START OFFSETS 95
11.1.2 EXTRA BYTES 96
11.1.3 FIELD CONTENTS 96
11.2 Where to Look For More Information 98

12 InnoDB Page Structure 99
12.1 High-Altitude View 99
12.1.1 Fil Header 99
12.1.2 Page Header 100
12.1.3 The Infimum and Supremum Records 101
12.1.4 User Records 101
12.1.5 Free Space 101
12.1.6 Page Directory 101
12.1.7 Fil Trailer 102
12.2 Example 102
12.3 Where to Look For More Information 103

13 Adding New Error Messages to MySQL 105

14 Annotated List of Files in the MySQL Source Code Distribution 107
14.1 Directory Listing 107
14.1.1 bdb 108
14.1.2 BitKeeper 108
14.1.3 BUILD 108
14.1.4 Build-tools 108
14.1.5 client 108
14.1.6 cmd-line-utils 109
14.1.7 dbug 110
14.1.8 Docs 110
14.1.9 extra 111
14.1.10 heap 111
14.1.11 include 112
14.1.12 innobase 112
14.1.13 libmysql 112
14.1.14 libmysql_r 113
14.1.15 libmysqld 113
14.1.16 man 113
14.1.17 myisam 113
14.1.18 myisammrg 115
14.1.19 mysql-test 115
14.1.20 mysys 116
14.1.21 netware 119
14.1.22 NEW-RPMS 119
14.1.23 os2 119
14.1.24 pstack 119
14.1.25 regex 120
14.1.26 SCCS 120
14.1.27 scripts 120
14.1.28 sql 121

14.1.29 sql-bench 123
14.1.30 SSL 124
14.1.31 strings 124
14.1.32 support-files 125
14.1.33 tests 125
14.1.34 tools 126
14.1.35 VC++Files 126
14.1.36 vio 127
14.1.37 zlib 127

15 Annotated List of Files in the InnoDB Source Code Distribution 129