MySQL Internals Manual

MySQL Internals Manual

Abstract

This is the MySQL Internals Manual.

Document generated on: 2006-10-18 (revision: 3666)

Table of Contents

List of Figures

Preface

This is a manual about MySQL internals. MySQL development personnel change it on an occasional basis. We don't guarantee that it's all true or up-to-date. We do hope it will show you how MySQL's programmers work, and how MySQL's server works as a result.

Chapter 1. A Guided Tour Of The MySQL Source Code

Hi, welcome to chapter 1. What we're about to do in this chapter is pick up the latest copy of the MySQL source code off the Internet. Then we'll get a list of the directories and comment on why they're there. Next we'll open up some of the files that are vital to MySQL's working, and comment on specific lines in the source code. We'll close off with a few pictures of file formats.

1.1. BitKeeper

We want to download the latest, the very latest, version of the MySQL server. So we won't click “Downloads” on the MySQL Developer Zone page — that's usually a few weeks old. Instead we'll get BitKeeper (tm), which is a revision control package, vaguely like CVS or Perforce. This is what MySQL's developers use every day, so what we download with BitKeeper is usually less than a day old. If you've ever submitted a bug report and gotten the response “thanks, we fixed the bug in the source code repository” that means you can get the fixed version with BitKeeper.

First, log on to www.bitkeeper.com and register. One way is:

- Click “Downloads” which takes you to the downloads page http://bitkeeper.com/Products.BK_Pro.Evaluation.html
- Click on “Software download form” which takes you to http://www.bitmover.com/cgi-bin/license.cgi
- Fill in all the fields, including the Platform name, which was “Linux/x86 ...” for us, but you'll be okay with other choices. Like MySQL, BitKeeper is available on many platforms, so the details will vary here.
- Submit the form.

Then you'll have to wait until BitKeeper mails you some more instructions, which are actually quite simple. There is no fee to use BitKeeper for downloads of open-source code repositories like MySQL's.

After you have BitKeeper, you'll be able to clone. That is, you'll be able to get a copy of the source code, using a statement that looks like this:

```
shell> bk clone
```

... that is,

```
bk clone <MySQL machine:/directory name> <your directory name>
```

... that is,

- Start a shell
- On the shell, make a directory named (say) mysql-5.0:

  ```
  shell> mkdir $HOME/mysql-5.0
  shell>cd $HOME
  shell> bk clone bk://mysql.bkbits.net/mysql-5.0 mysql-5.0
  ```

(The $HOME directory is usually your personal area that you're allowed to write to. If that's not the case, replace $HOME with your personal choice whenever it appears.)

There is a lot of code, so the first time you do this the download will take over an hour. That's if you're lucky. Glitches can occur, for example the 'bk' command fails due to a firewall restriction.

If you're glitch-prone, you'll need to read the manual: Section 2.8.3, Installing from the Development Source Tree.

On later occasions, you'll be doing what's called a “bk pull” instead of a “bk clone”, and it will go faster. Typically a “bk pull” takes 10 minutes and is glitch-free.

Directories, Alphabetical Order

After bk clone finishes and says “Clone completed successfully” you'll have 40 new sets of files on your computer, as you'll be able to see with `ls` or `dir`.

```
BUILD
BitKeeper
Docs
NEW-RPMS
SCCS
SSL
VC++Files
bdb
client
cmd-line-utils
config
dbug
extra
heap
include
innobase
libmysql
libmysql_r
libmysqld
man
myisam
myisammrg
mysql-test
mysys
ndb
netware
os2
pstack
regex
scripts
server-tools
sql
sql-bench
sql-common
strings
support-files
tests
tools
vio
zlib
```

These will all be installed as directories below the $HOME directory. At first all these directory names might intimidate you, and that's natural. After all, MySQL is a big project. But we're here to show you that there's order in this apparent chaos.

1.2. The Major Directories

1. `BUILD`
2. `client`

3. `Docs`
4. `myisam`
5. `mysys`
6. `sql`
7. `vio`

The orderly approach is to look first at the most important directories, then we'll look at the whole list in our second pass. So, first, let's look at what you'll find in just seven of the directories: BUILD, client, Docs, myisam, mysys, sql, and vio.

1.2.1. Major Directories: BUILD

The first major directory we'll look at is BUILD. It actually has very little in it, but it's useful, because one of the first things you might want to do with the source code is: compile and link it.

The example command line that we could use is

```
BUILD/compile-pentium-debug --prefix=$HOME/mysql-bin
```

It invokes a batch file in the BUILD directory. When it's done, you'll have an executable MySQL server and client.

Or, um, well, maybe you won't. Sometimes people have trouble with this step because there's something missing in their operating system version, or whatever. Don't worry, it really does work, and there are people around who might help you if you have trouble with this step. Search for "build" in the archives of lists.mysql.com.

We, when we're done building, tend to install it with the following sequence:

```
shell> make
shell> make install
shell> $HOME/mysql-bin/bin/mysql_install_db\
 --basedir=$HOME/mysql-bin\
 --datadir=$HOME/mysql-bin/var
```

This puts the new MySQL installation files on

```
shell> $HOME/mysql-bin/libexec      -- for the server
shell> $HOME/mysql-bin/bin          -- for the mysql client
shell> $HOME/mysql-bin/var          -- for the databases
```

1.2.1.1. gdb (GNU debugger)

Once you've got something that runs, you can put a debugger on it. We recommend use of the GNU debugger

```
http://www.gnu.org/software/gdb/documentation/
```

And many developers use the graphical debugger tool DDD - Data Display Debugger

```
http://www.gnu.org/software/ddd/manual/
```

These are free and common, they're probably on your Linux system already.

There are debuggers for Windows and other operating systems, of course — don't feel left out just because we're mentioning a Linux tool name! But it happens that we do a lot of things with Linux ourselves, so we happen to know what to say. To debug the mysqld server, say:

```
shell> ddd --gdb --args \
     $HOME/mysql-bin/libexec/mysqld \
     --basedir=$HOME/mysql-bin \
     --datadir=$HOME/mysql-bin/var\
     --skip-networking
```

From this point on, it may be tempting to follow along through the rest of the "guided tour" by setting breakpoints, displaying the contents of variables, and watching what happens after starting a client from another shell. That would be more fun. But it would require a detour, to discuss how to use the debugger. So we'll plow forward, the dull way, noting what's in the directories and using a text editor to note what's in the individual files.

1.2.1.2. Running a Test with the Debugger

To run a test named `some.test` with the debugger in embedded mode you could do this:

1. Run `libmysqld/examples/test_run --gdb some.test`. This creates a `libmysqld/examples/test-gdbinit` file which contains the required parameters for `mysqltest`.
2. Make a copy of the `test-gdbinit` file (call it, for example, `some-gdbinit`). The `test-gdbinit` file will be removed after `test-run --gdb` has finished.
3. Load `libmysqld/examples/mysqltest_embedded` into your favorite debugger, for example: `gdb mysqltest_embedded`.
4. In the debugger, for example in `gdb`, do: `--sou some-gdbinit`

Now `some.test` is running, and you can see if it's passing or not.

If you just want to debug some queries with the embedded server (not the test), it's easier to just run `libmysqld/examples/mysql`. It's the embedded server-based clone of the usual `mysql` tool, and works fine under `gdb` or whatever your favorite debugger is.

1.2.2. Major Directories: client

The next major directory is mysql-5.0/client.

```
size   name           comment
----   ----           -------
100034 mysql.cc     "The MySQL command tool"
 36913 mysqladmin.c maintenance of MYSQL databases
 22829 mysqlshow.c  show databases, tables, or columns
+ 12 more .c and .cc programs
```

It has the source code of many of your familiar favorites, like mysql, which everybody has used to connect to the MySQL server at one time or another. There are other utilities too — in fact, you'll find the source of most client-side programs here. There are also programs for checking the password, and for testing that basic functions — such as threading or access via SSL — are possible.

You'll notice, by the way, that we're concentrating on the files that have extension of ".c" or ".cc". By now it's obvious that C is our principal language although there are some utilities written in Perl as well.

1.2.3. Major Directories: Docs

The next major directory is mysql-5.0/Docs.

With the BitKeeper downloads, /Docs is nearly empty. Binary and source distributions include some pre-formatted documentation files, such as the MySQL Reference manual in Info format (for Unix) or CHM format (for Windows). The `mysqldoc` documentation repository is available separately from http://dev.mysql.com/tech-resources/sources.html. If you have Subversion, you can check out a copy of the repository with this command:

```
svn checkout http://svn.mysql.com/svnpublic/mysqldoc/
```

Some important files in the `mysqldoc` repository are:

```
name                             comment
----                             -------
.../sample-data/world/world.sql script to make 'world' database
.../internals/internals.xml     internals manual
.../refman/*.xml                reference manual
+ several more .xml files
```

Our documents are written in XML using DocBook. The DocBook format is becoming popular among publishers, and of course there's lots of general documentation for it, for example at http://www.docbook.org/. For our immediate purpose, the most interesting directory might be the `internals` directory, because it contains the source for the Internals Manual that you're reading now.

At this moment, the Internals Manual has more than 100 pages of information, including some details about the formats of MySQL files that you won't find anywhere else, and a complete description of the message formats that the client and server use to communicate. Although it's rough and may contain errors and is obsolete in parts, it is a document that you must read to truly understand the workings of MySQL.

1.2.4. Major Directories: myisam

The next major directory is labelled myisam. We will begin by mentioning that myisam is one of what we call the MySQL storage engine directories.

```
The MySQL storage engine directories:
heap           -- also known as 'memory'
innodb         -- maintained by Innobase Oy
myisam         -- see next section!
ndb            -- ndb cluster
```

For example the heap directory contains the source files for the heap storage engine and the ndb directory contains the source files for the ndb storage engine.

But the files in those directories are mostly analogues of what's in the myisam directory, and the myisam directory is sort of a 'template'.

On the myisam directory, you'll find the programs that do file I/O. Notice that the file names begin with the letters mi, by the way. That stands for MyISAM, and most of the important files in this directory start with mi.

File handling programs on mysql-5.0/myisam:

```
size   name           comment
----   ----           -------
 40301 mi_open.c      for opening
  3593 mi_close.c     for closing
  1951 mi_rename.c    for renaming
+ more mi_*.c programs
```

Row handling programs on mysql-5.0/myisam:

```
size   name           comment
----   ----           -------
 29064 mi_delete.c    for deleting
  2562 mi_delete_all.c for deleting all
  6797 mi_update.c    for updating
 32613 mi_write.c     for inserting
+ more mi_*.c programs
```

Drilling down a bit, you'll also find programs in the myisam directory that handle deleting, updating, and inserting of rows. The only one that's a little hard to find is the program for inserting rows, which we've called mi_write.c instead of mi_insert.c.

Key handling programs on mysql-5.0/myisam:

```
size   name           comment
----   ----           -------
  4668 mi_rkey.c      for random key searches
  3646 mi_rnext.c     for next-key searches
 15440 mi_key.c       for managing keys
+ more mi_*.c programs
```

The final notable group of files in the myisam directory is the group that handles keys in indexes.

To sum up: (1) The myisam directory is where you'll find programs for handling files, rows, and keys. You won't find programs for handling columns — we'll get to them a bit later. (2) The myisam directory is just one of the handler directories. The programs in the other storage engine directories fulfill about the same functions.

1.2.5. Major Directories: mysys

The next major directory is labelled mysys, which stands for MySQL System Library. This is the toolbox directory, for example it has low level routines for file access. The .c files in mysys have procedures and functions that are handy for calling by main programs, for example by the programs in the myisam directory. There are 115 .c files in mysys, so we only can note a sampling.

Sampling of programs on mysql-5.0/mysys

```
size   name           comment
----   ----           -------
 17684 charset.c      character sets
  6165 mf_qsort.c     quicksort
  5609 mf__tempfile.c temporary files
+ 112 more *.c programs
```

Example one: with charset.c routines, you can change the character set.

Example two: mf_qsort.c contains our quicksort package.

Example three: mf_tempfile.c has what's needed for maintaining MySQL's temporary files.

You can see from these examples that mysys is a hodgepodge. That's why we went to the trouble of producing extra documentation in this document to help you analyze mysys's contents.

1.2.6. Major Directories: sql

The next major directory is mysql-5.0/sql. If you remember your manual, you know that you must pronounce this: ess queue ell.

The "parser" programs on mysql-5.0/sql:

```
size   name           comment
----   ----           -------
 51326 sql_lex.cc     lexer
230026 sql_yacc.yy    parser
+ many more *.cc programs
```

This is where we keep the parser. In other words, programs like sql_lex.cc and sql_yacc.yy are responsible for figuring out what's in an SQL command, and deciding what to do about it.

The "handler" programs on mysql-5.0/sql:

```
size   name             comment
----   ----             -------
 79798 ha_berkeley.cc   bdb
 56687 ha_federated.cc  federated (sql/med)
 61033 ha_heap.cc       heap (memory)
214046 ha_innodb.cc     innodb
 47361 ha_myisam.cc     myisam
 14727 ha_myisammrg.cc  merge
215091 ha_ndbcluster.cc ndb
```

This is also where we keep the handler programs. Now, you'll recall that the storage engine itself, for example myisam, is a separate directory. But here in the sql directory, we have programs which are responsible for determining which handler to call, formatting appropriate arguments, and checking results. In other words, the programs that begin with the letters ha are the handler interface programs, and there's one for each storage engine.

The "statement" routines in mysql-5.0/sql:

```
size   name           comment
----   ----           -------
 24212 sql_delete.cc  'delete ...' statement
  1217 sql_do.cc      'do ...'
 22362 sql_help.cc    'help ...'
 75331 sql_insert.cc  'insert ...'
430486 sql_select.cc  'select ...'
130861 sql_show.cc    'show ...'
 42346 sql_update.cc  'update ...'
+ many more sql_*.cc programs
```

Also in the sql directory, you'll find individual programs for handling each of the syntactical components of an SQL statement. These programs tend to have names beginning with sql_. So for the SELECT statement, check out sql_select.cc.

Thus, there are "statement" routines like sql_delete.c, sql_load.c, and sql_help.c, which take care of the DELETE, LOAD DATA, and HELP statements. The file names are hints about the SQL statements involved.

The "statement function" routines in mysql-5.0/sql:

```
size   name           comment
----   ----           -------
 19906 sql_string.cc  strings
  6152 sql_olap.cc    olap (rollup)
 14241 sql_udf.cc     user-defined functions
 17669 sql_union.cc   unions
```

Then there are the routines for components of statements, such as strings, or online analytical processing which at this moment just means ROLLUP, or user-defined functions, or the UNION operator.

1.2.7. Major Directories: vio

The final major directory that we'll highlight is labelled vio, for "virtual I/O".

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.

That wraps up our quick look at the seven major directories. Just one summary chart remains to do.

1.3. The Flow

This is a diagram of the flow.

```
User enters "INSERT" statement     /* client */
|
|
Message goes over TCP/IP line      /* vio, various */
|
|
Server parses statement            /* sql */
|
|
Server calls low-level functions   /* myisam */
|
|
Handler stores in file             /* mysys */
```

The diagram is very simplified — it's a caricature that distorts important things, but remember that we've only discussed seven major directories so far: Docs, BUILD, and the five that you see here.

The flow works like this:

First, the client routines get an SQL statement from a user, allowing edit, performing initial checks, and so on.

Then, via the vio routines, the somewhat-massaged statement goes off to the server.

Next, the sql routines handle the parsing and call what's necessary for each individual part of the statement. Along the way, the sql routines will be calling the low level mysys routines frequently.

Finally, one of the ha (handler) programs in the sql directory will dispatch to an appropriate handler for storage. In this case we've assumed, as before, that the handler is myisam — so a myisam-directory program is involved. Specifically, that program is mi_write.c, as we mentioned earlier.

Simple, eh?

1.4. The Open-source Directories

We're now getting into the directories which aren't "major". Starting with:

```
dbug
pstack
regex
strings
zlib
```

Now it's time to reveal a startling fact, which is — we didn't write all of the source code in all of the source code directories all by ourselves. This list is, in a sense, a tribute to the idea of open source.

There's dbug, which is Fred Fish's debug library.

There's pstack, which displays the process stack.

There's regex, which is what we use for our regular expressions function.

There's strings, the meaning of which is obvious.

There's zlib, which is for Zempel-Liv compression.

All of the programs in these directories were supplied by others, as open source code. We didn't just take them, of course. MySQL has checked and changed what's in these directories. But we acknowledge with thanks that they're the products of other projects, and other people's labor, and we only regret that we won't have time to note all the contributed or publicly available components of MySQL, in this manual.

1.5. The Internal and External Storage Engine Directories

Continuing with our extract from the directory list ...

```
bdb                           /* external */
heap
innobase                      /* external */
myisam
myisammrg
ndb
```

Let's go through the idea of storage engines once more, this time with a list of all the storage engines, both the ones that we produce, and the ones that others produce. We've already mentioned the internal ones — so now we'll remark on the directories of the two common external storage engines — BDB and innobase.

The BDB, or Berkeley Database, handler, is strictly the product of Sleepycat software. Sleepycat has a web page at sleepycat.com, which contains, among other things, documentation for their product. So you can download Sleepycat's own documentation of the source code in the BDB directory.

As for the innobase handler, which many of you probably use, you'll be happy to know that the comments in the files are reasonably clear (the InnoBase Oy people are pretty strict about comments). There are two chapters about it in this document.

1.6. The "OS Specific" Directories

```
netware
NEW-RPMS
os2
VC++Files
```

A few words are in order about the directories that contain files which relate to a particular environment that MySQL can run in.

The netware directory contains a set of files for interfacing with netware, and anyone who has an involvement with NetWare knows that we're allied with them, and so this is one of the directories that represents the joint enterprise.

The NEW-RPMS directory (empty at time of writing) is for Linux, and the os2 directory is for OS/2.

Finally, the VC++Files directory is for Windows. We've found that the majority of Windows programmers who download and build MySQL from source use Microsoft Visual C. In the VC++Files directory you will find a nearly complete replication of what's in all the other directories that we've discussed, except that the .c files are modified to account for the quirks of Microsoft tools.

Without endorsing by particular names, we should note that other compilers from other manufacturers also work.

1.7. Odds and Ends

Finally, for the sake of completeness, we'll put up a list of the rest of the directories — those that we haven't had occasion to mention till now.

```
Source Code Administration Directories:
BitKeeper
SCCS

Common .h Files:
include

GNU Readline library and related:
cmd-line-utils

Stand-alone Utility & Test Programs:
extra
mysql-test
repl-tests
support-files
tests
tools
```

You don't have to worry about the administration directories since they're not part of what you build.

You probably won't have to worry about the stand-alone programs either, since you just use them, you don't need to remake them.

There's an include directory that you SHOULD have a look at, because the common header files for programs from several directories are in here.

Finally, there are stand-alone utility and test programs. Strictly speaking they're not part of the "source code". But it's probably reassuring to know that there's a test suite, for instance. Part of the quality-assurance process is to run the scripts in the test suite before releasing.

And those are the last. We've now traipsed through every significant directory created during your download of the MySQL source package.

1.8. A Chunk of Code in `/sql/sql_update.cc`

Now, having finished with our bird's eye view of the source code from the air, let's take the perspective of the worms on the ground. (Which is another name for MySQL's developer staff — turn on laugh track here.)

```
int mysql_update(THD *thd, ...)
{
   ...
   if ((lock_tables(thd, table_list)))
     DBUG_RETURN(1); ...
   ...
   init_read_record(&info,thd,table,select,0,1); ...
   while (!(error=info.read_record(&info)) && !thd->killed)
   {
      ...
     if (!(error=table->file->update_row((byte*) table->record[1]),
                                          (byte*) table->record[0])))
        updated++;
      ...
      if (table->triggers)
        table->triggers->process_triggers(thd, TRG_EVENT_UPDATE, TRG_ACTION_AFTER);
      ...
     }
   ...
   if (updated && (error <= 0 || !transactional_table))
   {
     mysql_bin_log.write(&qinfo) && transactional_table);
     ...
}
```

Here's a snippet of code from a .c file in the sql directory, specifically from sql_update.cc, which — as we mentioned earlier -- is invoked when there's an UPDATE statement to process.

The entire routine has many error checks with handlers for improbable solutions, and showing multiple screens would be tedious, so we've truncated the code a lot. Where you see an ellipsis (three dots in a row), that means "and so on".

So, what do we learn from this snippet of code? In the first place, we see that it's fairly conventional C code. A brace causes an indentation, instructions tend to be compact with few unnecessary spaces, and comments are sparse.

Abbreviations are common, for example thd stands for thread, you just have to get used to them. Typically a structure will be in a separate .h file.

Routine names are sometimes long enough that they explain themselves. For example, you can probably guess that this routine is opening and locking, allocating memory in a cache, initializing a process for reading records, reading records in a loop until the thread is killed or there are no more to read, storing a modified record for the table, and — after the loop is through — possibly writing to the log. Incidentally, a transactional table is usually a BDB or an InnoDB table.

Obviously we've picked out what's easy to follow, and we're not pretending it's all smooth sailing. But this is actual code and you can check it out yourself.

1.9. The Skeleton Of The Server Code

And now we're going to walk through something harder, namely the server.

WARNING WARNING WARNING: code changes constantly, so names and parameters may have changed by the time you read this.

Important programs we'll be walking through:

```
/sql/mysqld.cc
/sql/sql_parse.cc
/sql/sql_prepare.cc
/sql/sql_insert.cc
/sql/ha_myisam.cc
/myisam/mi_write.c
```

This is not as simple as what we've just done. In fact we'll need multiple pages to walk through this one, and that's despite our use of truncation and condensation again. But the server is important, and if you can grasp what we're doing with it, you'll have grasped the essence of what the MySQL source code is all about.

We'll mostly be looking at programs in the sql directory, which is where mysqld and most of the programs for the SQL engine code are stored.

Our objective is to follow the server from the time it starts up, through a single INSERT statement that it receives from a client, to the point where it finally performs the low level write in the MyISAM file.

Walking Through The Server Code: /sql/mysqld.cc

```
  int main(int argc, char **argv)
  {
    _cust_check_startup();
    (void) thr_setconcurrency(concurrency);
    init_ssl();
    server_init();                            // 'bind' + 'listen'
    init_server_components();
    start_signal_handler();
    acl_init((THD *)0, opt_noacl);
    init_slave();
    create_shutdown_thread();
    create_maintenance_thread();
    handle_connections_sockets(0);             // !
    DBUG_PRINT("quit",("Exiting main thread"));
    exit(0);
  }
```

Here is where it all starts, in the main function of mysqld.cc.

Notice that we show a directory name and program name just above this snippet. We will do the same for all the snippets in this series.

By glancing at this snippet for a few seconds, you will probably see that the main function is doing some initial checks on startup, is initializing some components, is calling a function named handle_connections_sockets, and then is exiting. It's possible that acl stands for "access control" and it's interesting that DBUG_PRINT is something from Fred Fish's debug library, which we've mentioned before. But we must not digress.

In fact there are 150 code lines in the main function, and we're only showing 13 code lines. That will give you an idea of how much we are shaving and pruning. We threw away the error checks, the side paths, the optional code, and the variables. But we did not change what was left. You will be able to find these lines if you take an editor to the mysqld.cc program, and the same applies for all the other routines in the snippets in this series.

The one thing you won't see in the actual source code is the little marker "// !". This marker will always be on the line of the function that will be the subject of the next snippet. In this case, it means that the next snippet will show the handle_connection_sockets function. To prove that, let's go to the next snippet.

Walking Through The Server Code: /sql/mysqld.cc

```
  handle_connections_sockets (arg __attribute__((unused))
  {
     if (ip_sock != INVALID_SOCKET)
     {
       FD_SET(ip_sock,&clientFDs);
       DBUG_PRINT("general",("Waiting for connections."));
       while (!abort_loop)
       {
         new_sock = accept(sock, my_reinterpret_cast(struct sockaddr*)
           (&cAddr),
            &length);
         thd= new THD;
         if (sock == unix_sock)
         thd->host=(char*) localhost;
         create_new_thread(thd);            // !
         }
```

Inside handle_connections_sockets you'll see the hallmarks of a classic client/server architecture. In a classic client/server, the server has a main thread which is always listening for incoming requests from new clients. Once it receives such a request, it assigns resources which will be exclusive to that client. In particular, the main thread will spawn a new thread just to handle the connection. Then the main server will loop and listen for new connections — but we will leave it and follow the new thread.

As well as the sockets code that we chose to display here, there are several variants of this thread loop, because clients can choose to connect in other ways, for example with named pipes or with shared memory. But the important item to note from this section is that the server is spawning new threads.

Walking Through The Server Code: /sql/mysqld.cc

```
  create_new_thread(THD *thd)
  {
    pthread_mutex_lock(&LOCK_thread_count);
    pthread_create(&thd->real_id,&connection_attrib,
        handle_one_connection,                        // !
        (void*) thd));
    pthread_mutex_unlock(&LOCK_thread_count);
  }
```

Here is a close look at the routine that spawns the new thread. The noticeable detail is that, as you can see, it uses a mutex or mutual exclusion object. MySQL has a great variety of mutexes that it uses to keep actions of all the threads from conflicting with each other.

Walking Through The Server Code: /sql/sql_parse.cc

```
handle_one_connection(THD *thd)
  {
    init_sql_alloc(&thd->mem_root, MEM_ROOT_BLOCK_SIZE, MEM_ROOT_PREALLOC);
    while (!net->error && net->vio != 0 && !thd->killed)
    {
      if (do_command(thd))            // !
        break;
    }
    close_connection(net);
    end_thread(thd,1);
    packet=(char*) net->read_pos;
```

With this snippet, we've wandered out of mysqld.cc. Now, we're in the sql_parse program, still in the sql directory. This is where the session's big loop is.

The loop repeatedly gets and does commands. When it ends, the connection closes. At that point, the thread will end and the resources for it will be deallocated.

But we're more interested in what happens inside the loop, when we call the do_command function.

```
Graphic:

   client           <===== MESSAGE ====>       server
                    <======PACKETS ====>

   Example:
   INSERT INTO Table1 VALUES (1);
```

To put it graphically, at this point there is a long-lasting connection between the client and one server thread. Message packets will go back and forth between them through this connection. For today's tour, let's assume that the client passes the INSERT statement shown on the Graphic, for the server to process.

Walking Through The Server Code: /sql/sql_parse.cc

```
bool do_command(THD *thd)
{
  net_new_transaction(net);
  packet_length=my_net_read(net);
  packet=(char*) net->read_pos;
  command = (enum enum_server_command) (uchar) packet[0];
  dispatch_command(command,thd, packet+1, (uint) packet_length);
// !
}
```

You've probably noticed by now that whenever we call a lower-level routine, we pass an argument named thd, which is an abbreviation for the word thread (we think). This is the essential context which we must never lose.

The my_net_read function is in another program called net_serv.cc. It gets a packet from the client, uncompresses it, and strips the header.

Once that's done, we've got a multi-byte variable named packet which contains what the client has sent. The first byte is important because it contains a code identifying the type of message.

We'll pass that and the rest of the packet on to the dispatch_command function.

Walking Through The Server Code: /sql/sql_parse.cc

```
bool dispatch_command(enum enum_server_command command, THD *thd,
       char* packet, uint packet_length)
{
  switch (command) {
    case COM_INIT_DB:          ...
    case COM_REGISTER_SLAVE:   ...
    case COM_TABLE_DUMP:       ...
    case COM_CHANGE_USER:      ...
    case COM_EXECUTE:
         mysql_stmt_execute(thd,packet);
    case COM_LONG_DATA:        ...
    case COM_PREPARE:
         mysql_stmt_prepare(thd, packet, packet_length);   // !
    /* and so on for 18 other cases */
    default:
     send_error(thd, ER_UNKNOWN_COM_ERROR);
     break;
    }
```

And here's just part of a very large switch statement in sql_parse.cc. The snippet doesn't have room to show the rest, but you'll see when you look at the dispatch_command function that there are more case statements after the ones that you see here.

There will be — we're going into list mode now and just reciting the rest of the items in the switch statement — code for prepare, close statement, query, quit, create database, drop database, dump binary log, refresh, statistics, get process info, kill process, sleep, connect, and several minor commands. This is the big junction.

We have cut out the code for all of the cases except for two, COM_EXECUTE and COM_PREPARE.

Walking Through The Server Code: /sql/sql_prepare.cc

We are not going to follow what happens with COM_PREPARE. Instead, we are going to follow the code after COM_EXECUTE. But we'll have to digress from our main line for a moment and explain what the prepare does.

```
"Prepare:
Parse the query
Allocate a new statement, keep it in 'thd->prepared statements' pool
Return to client the total number of parameters and result-set
metadata information (if any)"
```

The prepare is the step that must happen before execute happens. It consists of checking for syntax errors, looking up any tables and columns referenced in the statement, and setting up tables for the execute to use. Once a prepare is done, an execute can be done multiple times without having to go through the syntax checking and table lookups again.

Since we're not going to walk through the COM_PREPARE code, we decided not to show its code at this point. Instead, we have cut and pasted some code comments that describe prepare. All we're illustrating here is that there are comments in the code, so you will have aid when you look harder at the prepare code.

Walking Through The Server Code: /sql/sql_parse.cc

```
  bool dispatch_command(enum enum_server_command command, THD *thd,
       char* packet, uint packet_length)
  {
  switch (command) {
    case COM_INIT_DB:          ...
    case COM_REGISTER_SLAVE:   ...
    case COM_TABLE_DUMP:       ...
    case COM_CHANGE_USER:      ...
    case COM_EXECUTE:
         mysql_stmt_execute(thd,packet);                   // !
    case COM_LONG_DATA:        ...
    case COM_PREPARE:
         mysql_stmt_prepare(thd, packet, packet_length);
    /* and so on for 18 other cases */
    default:
     send_error(thd, ER_UNKNOWN_COM_ERROR);
     break;
    }
```

Let's return to the grand central junction again in sql_parse.cc for a moment. The thing to note on this snippet is that the line which we're really going to follow is what happens for COM_EXECUTE.

Walking Through The Server Code: /sql/sql_prepare.cc

```
  void mysql_stmt_execute(THD *thd, char *packet)
  {
    if (!(stmt=find_prepared_statement(thd, stmt_id, "execute")))
    {
      send_error(thd);
      DBUG_VOID_RETURN;
    }
    init_stmt_execute(stmt);
    mysql_execute_command(thd);           // !
  }
```

In this case, the line that we're following is the line that executes a statement.

Notice how we keep carrying the THD thread and the packet along with us, and notice that we expect to find a prepared statement waiting for us, since this is the execute phase. Notice as well that we continue to sprinkle error-related functions that begin with the letters DBUG, for use by the debug library. Finally, notice that the identifier "stmt" is the same name that ODBC uses for the equivalent object. We try to use standard names when they fit.

Walking Through The Server Code: /sql/sql_parse.cc

```
  void mysql_execute_command(THD *thd)
       switch (lex->sql_command) {
       case SQLCOM_SELECT: ...
       case SQLCOM_SHOW_ERRORS: ...
       case SQLCOM_CREATE_TABLE: ...
       case SQLCOM_UPDATE: ...
       case SQLCOM_INSERT: ...                   // !
       case SQLCOM_DELETE: ...
       case SQLCOM_DROP_TABLE: ...
       }
```

In the mysql_execute_command function. we encounter another junction. One of the items in the switch statement is named SQLCOM_INSERT.

Walking Through The Server Code: /sql/sql_parse.cc

```
case SQLCOM_INSERT:
{
  my_bool update=(lex->value_list.elements ? UPDATE_ACL : 0);
  ulong privilege= (lex->duplicates == DUP_REPLACE ?
                    INSERT_ACL | DELETE_ACL : INSERT_ACL | update);
  if (check_access(thd,privilege,tables->db,&tables->grant.privilege))
    goto error;
  if (grant_option && check_grant(thd,privilege,tables))
    goto error;
  if (select_lex->item_list.elements != lex->value_list.elements)
  {
    send_error(thd,ER_WRONG_VALUE_COUNT);
    DBUG_VOID_RETURN;
  }
  res = mysql_insert(thd,tables,lex->field_list,lex->many_values,
                     select_lex->item_list, lex->value_list,
                     (update ? DUP_UPDATE : lex->duplicates));
// !
  if (thd->net.report_error)
    res= -1;
  break;
}
```

For this snippet, we've blown up the code around the SQLCOM_INSERT case in the mysql_execute_command function. The first thing to do is check whether the user has the appropriate privileges for doing an INSERT into the table, and this is the place where the server checks for that, by calling the check_access and check_grant functions. It would be tempting to follow those functions, but those are side paths. Instead, we'll follow the path where the work is going on.

Walking Through The Server Code: Navigation Aid

Some program names in the /sql directory:

```
Program Name           SQL statement type
------------           ------------------
sql_delete.cc          DELETE
sql_do.cc              DO
sql_handler.cc         HANDLER
sql_help.cc            HELP
sql_insert.cc          INSERT            // !
sql_load.cc            LOAD
sql_rename.cc          RENAME
sql_select.cc          SELECT
sql_show.cc            SHOW
sql_update.cc          UPDATE
```

Question: Where will mysql_insert() be?

The line that we're following will take us next to a routine named mysql_insert. Sometimes it's difficult to guess what program a routine will be in, because MySQL has no consistent naming convention. However, here is one aid to navigation that works for some statement types. In the sql directory, the names of some programs correspond to statement types. This happens to be the case for INSERT, for instance. So the mysql_insert program will be in the program sql_insert.cc. But there's no reliable rule.

(Let's add here a few sentences about the tags 'ctags' program. When an editor supports ctags (and the list is long, but vi and emacs of course are there), the function definition is one key press away - no guessing involved. In the above case, a vim user could press ^] on mysql_insert name and vim would open sql_insert.cc and position the curson on the first line of the mysql_insert() function. The tags help can be indispensable in everyday work.)

Walking Through The Server Code: /sql/sql_insert.cc

```
  int mysql_insert(THD *thd,TABLE_LIST *table_list, List<Item> &fields,
        List<List_item> &values_list,enum_duplicates duplic)
  {
    table = open_ltable(thd,table_list,lock_type);
    if (check_insert_fields(thd,table,fields,*values,1) ||
      setup_tables(table_list) ||
      setup_fields(thd,table_list,*values,0,0,0))
      goto abort;
    fill_record(table->field,*values);
    error=write_record(table,&info);                 // !
    query_cache_invalidate3(thd, table_list, 1);
    if (transactional_table)
      error=ha_autocommit_or_rollback(thd,error);
    query_cache_invalidate3(thd, table_list, 1);
    mysql_unlock_tables(thd, thd->lock);
    }
```

For the mysql_insert routine, we're just going to read what's in the snippet. What we're trying to do here is highlight the fact that the function names and variable names are nearly English.

Okay, we start by opening a table. Then, if a check of the fields in the INSERT fails, or if an attempt to set up the tables fails, or if an attempt to set up the fields fails, we'll abort.

Next, we'll fill the record buffer with values. Then we'll write the record. Then we'll invalidate the query cache. Remember, by the way, that MySQL stores frequently-used select statements and result sets in memory as an optimization, but once the insert succeeds the stored sets are invalid. Finally, we'll unlock the tables.

Walking Through The Server Code: /sql/sql_insert.cc

```
  int write_record(TABLE *table,COPY_INFO *info)
  {
    table->file->write_row(table->record[0];           // !
  }
```

You can see from our marker that we're going to follow the line that contains the words 'write row'. But this is not an ordinary function call, so people who are just reading the code without the aid of a debugger can easily miss what the next point is in the line of execution here. The fact is, 'write_row' can take us to one of several different places.

Walking Through The Server Code: /sql/handler.h

```
  /* The handler for a table type.
     Will be included in the TABLE structure */

  handler(TABLE *table_arg) :
table(table_arg),active_index(MAX_REF_PARTS),
    ref(0),ref_length(sizeof(my_off_t)),
block_size(0),records(0),deleted(0),
    data_file_length(0), max_data_file_length(0),
index_file_length(0),
    delete_length(0), auto_increment_value(0), raid_type(0),
    key_used_on_scan(MAX_KEY),
    create_time(0), check_time(0), update_time(0), mean_rec_length(0),
    ft_handler(0)
    {}
  ...
  virtual int write_row(byte * buf)=0;
```

To see what the write_row statement is doing, we'll have to look at one of the include files. In handler.h on the sql directory, we find that write_row is associated with a handler for a table. This definition is telling us that the address in write_row will vary — it gets filled in at run time. In fact, there are several possible addresses.

There is one address for each handler. In our case, since we're using the default values, the value at this point will be the address of write_row in the MyISAM handler program.

Walking Through The Server Code: /sql/ha_myisam.cc

```
int ha_myisam::write_row(byte * buf)
{
  statistic_increment(ha_write_count,&LOCK_status);
   /* If we have a timestamp column, update it to the current time */
   if (table->time_stamp)
    update_timestamp(buf+table->time_stamp-1);
   /*
  If we have an auto_increment column and we are writing a changed row
    or a new row, then update the auto_increment value in the record.
  */
  if (table->next_number_field && buf == table->record[0])
    update_auto_increment();
  return mi_write(file,buf);     // !
}
```

And that brings us to write_row in the ha_myisam.cc program. Remember we told you that these programs beginning with the letters ha are interfaces to handlers, and this one is the interface to the myisam handler. We have at last reached the point where we're ready to call something in the handler package.

Walking Through The Server Code: /myisam/mi_write.c

```
int mi_write(MI_INFO *info, byte *record)
{
  _mi_readinfo(info,F_WRLCK,1);
  _mi_mark_file_changed(info);
  /* Calculate and check all unique constraints */
  for (i=0 ; i < share->state.header.uniques ; i++)
  {
    mi_check_unique(info,share->uniqueinfo+i,record,
      mi_unique_hash(share->uniqueinfo+i,record),
      HA_OFFSET_ERROR);
  }

  ... to be continued in next snippet
```

Notice that at this point there is no more referencing of tables, the comments are about files and index keys. We have reached the bottom level at last. Notice as well that we are now in a C program, not a C++ program.

In this first half of the mi_write function, we see a call which is clearly commented. This is where checking happens for uniqueness (not the UNIQUE constraint, but an internal matter).

Walking Through The Server Code: /myisam/mi_write.c

```
 ... continued from previous snippet

  /* Write all keys to indextree */
  for (i=0 ; i < share->base.keys ; i++)
  {
    share->keyinfo[i].ck_insert(info,i,buff,
      _mi_make_key(info,i,buff,record,filepos)
  }
  (*share->write_record)(info,record);
  if (share->base.auto_key)
    update_auto_increment(info,record);
}
```

In this second half of the mi_write function, we see another clear comment, to the effect that this is where the new keys are made for any indexed columns. Then we see the culmination of all that the last 20 snippets have been preparing, the moment we've all been waiting for, the writing of the record.

And, since the object of the INSERT statement is ultimately to cause a write to a record in a file, that's that. The server has done the job.

Walking Through The Server Code: Stack Trace

```
main in /sql/mysqld.cc
handle_connections_sockets in /sql/mysqld.cc
create_new_thread in /sql/mysqld.cc
handle_one_connection in /sql/sql_parse.cc
do_command in /sql/sql_parse.cc
dispatch_command in /sql/sql_parse.cc
mysql_stmt_execute in /sql/sql_prepare.cc
mysql_execute_command in /sql/sql_parse.cc
mysql_insert in /sql/mysql_insert.cc
write_record in /sql/mysql_insert.cc
ha_myisam::write_row in /sql/ha_myisam.cc
mi_write in /myisam/mi_write.c
```

And now here's a look at what's above us on the stack, or at least an idea of how we got here. We started with the main program in mysqld.cc. We proceeded through the creation of a thread for the client, the several junction processes that determined where we're heading, the parsing and initial execution of an SQL statement, the decision to invoke the MyISAM handler, and the writing of the row. We ended in a low level place, where we're calling the routines that write to the file. That's about as low as we should go today.

The server program would, of course, continue by returning several times in a row, sending a packet to the client saying "Okay", and ending up back in the loop inside the handle_one_connection function.

We, instead, will pause for a moment in awe at the amount of code we've just flitted past. And that will end our walk through the server code.

```
Graphic: A Chunk of MyISAM File

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');

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

Continuing with our worm's-eye view, let's glance at the structure of a record in a MyISAM file.

The SQL statements on this graphic show a table definition and some insert statements that we used to populate the table.

The final line on the graphic is a hexadecimal dump display of the two records that we ended up with, as taken from the MyISAM file for Table1.

The thing to notice here is that the records are stored compactly. There is one byte at the start of each record — F1 for the first record and F5 for the second record — which contains a bit list.

When a bit is on, that means its corresponding field is NULL. That's why the second row, which has a NULL in the second column, or field, has a different header byte from the first row.

Complications are possible, but a simple record really does look this simple.

```
Graphic: A Chunk of InnoDB File

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'
```

If, on the other hand, you look at an InnoDB file, you'll find that it's got more complexities in the storage. The details are elsewhere in this document. But here's an introductory look.

The header here begins with offsets — unlike MyISAM, which has no offsets. So you'd have to go through column 1 before getting to column 2.

Then there is a fixed header — the extra bytes.

Then comes the record proper. The first fields of a typical record contain information that the user won't see, such as a row ID, a transaction ID, and a rollback pointer. This part would look different if the user had defined a primary key during the CREATE TABLE statement.

And finally there are the column contents — the string of Ps at the end of the snippet here. You can see that InnoDB does more administrating.

There's been a recent change for InnoDB; what you see above is from a database made before version 5.0.

```
Graphic: A Packet

Header
Number Of Rows
ID
Status
Length
Message Content
```

Our final worm's-eye look at a physical structure will be a look at packets.

By packet, we mean: what's the format of a message that the client sends over the tcp/ip line to the server — and what does the server send back?

Here we're not displaying a dump. If you want to see hexadecimal dumps of the contents of packets, this document is full of them. We're just going to note that a typical message will have a header, an identifier, and a length, followed by the message contents.

Admittedly this isn't following a standard like ISO's RDA or IBM's DRDA, but it's documented so if you want to go out and write your own type 4 JDBC driver, you've got what you need here. (Subject to license restrictions, of course.) But a word of advice on that last point: it's already been done. Mark Matthews wrote it originally, it's all in "MySQL Connector/J".

1.10. Recap

Okay, let's back up and restate. In this walkthrough, we've told you four main things.

One: How to get the MySQL source.

Two: What's in each directory in the source.

Three: The main sequence, as one walks through the server code.

Four: What physical structures look like.

We worked hard to make a description of the MySQL source that is simple, without distorting. If you were able to follow all that we've said, then that's wonderful, congratulations. If you ended up thinking that MySQL is really simple, well that's not what we wanted to convey, but we think you'll be disabused of that notion when you have a look at the code yourself.

Chapter 2. Coding Guidelines

This chapter shows the guidelines that MySQL's developers follow when writing new code. Consistent style is important for us, because everyone must know what to expect. For example, after we become accustomed to seeing that everything inside an “if” is indented two spaces, we can glance at a listing and understand what's nested within what. Writing non-conforming code can be bad. For example, if we want to find where assignments are made to variable “mutex_count”, we might search for “mutex_count=” with an editor and miss assignments that look like “mutex_count =” with a space before the equal sign (which is non-conforming). Knowing our rules, you'll find it easier to read our code, and when you you decide to contribute (which we hope you'll consider!) we'll find it easier to read your code.

The chapter covers the following topics:

- C/C++ coding guidelines for MySQL Server
- `configure` support for plugins

2.1. C/C++ Coding Guidelines

This section covers guidelines for C/C++ code for the MySQL server. The guidelines do not necessarily apply for other projects such as MySQL Connector/J or MaxDB.

Indentation and Spacing

- Use spaces for space. Do not use tabs (\t). See the editor configuration tips at the end of this section for instructions on configuring a vim or emacs editor to use spaces instead of tabs.
- Use line feed (\n) for line breaks. Do not use carriage return + line feed (\r\n); that can cause problems for other users and for builds. This rule is particularly important if you use a Windows editor.
- To begin indenting, add two spaces. To end indenting, subtract two spaces. For example:

```
{
  code, code, code
  {
    code, code, code
  }
}
```

- The maximum line width is 80 characters. If you are writing a longer line, try to break it at a logical point and continue on the next line with the same indenting. Use of backslash is okay; however, multi-line literals might cause less confusion if they are defined before the function start.
- You may use empty lines (two line breaks in a row) wherever it seems helpful for readability. But never use two or more empty lines in a row. The only exception is after a function definition (see below).
- To separate two functions, use three line breaks (two empty lines). To separate a list of variable declarations from executable statements, use two line breaks (one empty line). For example:

```
int function_1()
{
  int i;
  int j;

  function0();
}


int function2()
{
  return;
}
```

- Matching '`{}`' (left and right braces) should be in the same column, that is, the closing '}' should be directly below the opening '{'. Do not put any non-space characters on the same line as a brace, not even a comment. Indent within braces. Exception 1: after `switch` there is a different rule, see below. Exception 2: if there is nothing between two braces, i.e. '{}', they should appear together. For example:

```
if (code, code, code)
{
  code, code, code;
}
for (code, code, code)
{}
```

- After `switch` use a brace on the same line, and do not indent the next line. For example:

```
switch (condition) {
code
code
code
}
```

- You may align variable declarations like this:

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

- When assigning to a variable, put zero spaces after the target variable name, then the assignment operator ('=' '+=' etc.), then space(s). For single assignments, there should be only one space after the equal sign. For multiple assignments, add additional spaces so that the source values line up. For example:

```
a/= b;
return_value= my_function(arg1);
...
int x=          27;
int new_var=    18;
```

  Align assignments from one structure to another, like this:

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

- Put separate statements on separate lines. This applies for both variable declarations and executable statements. For example, this is wrong:

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

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

  This is right:

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

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

- Put spaces both before and after binary comparison operators ('>', '==', '>=', etc.), binary arithmetic operators ('+' etc.), and binary Boolean operators ('||' etc.). Do not put spaces around unary operators like '!' or '++'. Do not put spaces around [de-]referencing operators like '->' or '[]'. Do not put space after '*' when '*' introduces a pointer. Do not put spaces after '('. Put one space after ')' if it ends a condition, but not if it ends a list of function arguments. For example:

```
int *var;

if ((x == y + 2) && !param->is_signed)
  function_call();
```

- When a function has multiple arguments separated by commas (`,`), put one space after each comma. For example:

```
ln= mysql_bin_log.generate_name(opt_bin_logname, "-bin", 1, buf);
```

- Put one space after a keyword which introduces a condition, such as `if` or `for` or `while`.
- After `if` or `else` or `while`, when there is only one instruction after the condition, braces are not necessary and the instruction goes on the next line, indented.

```
if (sig != MYSQL_KILL_SIGNAL && sig != 0)
  unireg_abort(1);
else
  unireg_end();
while (*val && my_isspace(mysqld_charset, *val))
  *val++;
```

- In function declarations and invocations: there is no space between function name and '('; there is no space or line break between '(' and the first argument; if the arguments do not fit on one line then align them. Examples:

```
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));
```

```
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 may but don't have to split `Class_name::method_name` into two lines.)

When arguments do not fit on one line, consider renaming them.

- 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 a constructor can fit into one line:

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

Naming Conventions

- For identifiers formed from multiple words, separate each component with underscore rather than capitalization. Thus, use `my_var` instead of `myVar` or `MyVar`.
- Avoid capitalization except for class names; class names should begin with a capital letter.

  ```
  class Item;
  class Query_arena;
  class Log_event;
  ```

- Avoid function names, structure elements, or variables that begin or end with '`_`'.
- 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: put the '/*' and '*/' on their own lines, put the '*/' below the '/*', put a line break and a two-space indent after the '/*', do not use additional asterisks on the left of the comment.

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

  ```

  /* ********* This comment is bad. It's indented incorrectly, it has
  ```

```
*            additional asterisks. Don't write this way.
*  *********/
```

- When writing single-line comments, the '/*' and '*/" are on the same line. For example:

```
/* We must check if stack_size = 0 as Solaris 2.9 can return 0 here */
```

- For a short comment at the end of a line, you may use either /* ... */ or a `//` double slash. In C files or in header files used by C files, avoid `//` comments.
- Align short side `//` or /* ... */ comments by 48 column (start the comment in column 49).

```
{
  qc*= 2;                                      /* double the estimation */
}
```

- When commenting members of a structure or a class , align comments by 48th column. If a comment doesn't fit into one line, move it to a separate line. Do not create multiline comments aligned by 48th column.

```
struct st_mysql_stmt
{
...
  MYSQL_ROWS     *data_cursor;         /* current row in cached result */
  /* copy of mysql->affected_rows after statement execution */
  my_ulonglong   affected_rows;
  my_ulonglong   insert_id;            /* copy of mysql->insert_id */
  /*
    mysql_stmt_fetch() calls this function to fetch one row (it's different
    for buffered, unbuffered and cursor fetch).
  */
  int            (*read_row_func)(struct st_mysql_stmt *stmt,
...
};
```

- 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.

  Use the following example as a template for function comments:

```
/*
  Initialize SHA1Context

  SYNOPSIS
    sha1_reset()
      context in/out     The context to reset.

  DESCRIPTION
    This function will initialize the SHA1Context in preparation
    for computing a new SHA1 message digest.

  RETURN VALUE
    SHA_SUCCESS          ok
    != SHA_SUCCESS       sha Error Code.
*/

int sha1_reset(SHA1_CONTEXT *context)
{
 ...
```

  Additional sections can be used: WARNING, NOTES, TODO, SEE ALSO, ERRORS, REFERENCED BY.

- All comments should be in English.
- Put two line breaks (one empty line) between a function comment and its description.

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

int my_function()
```

General Development Guidelines

- We use BitKeeper [http://www.bitkeeper.com/] 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.

  It is okay to use inline functions are which satisfy most of the following requirements:

  - The function is very short (just a few lines).
  - The function is used in a speed critical place and is executed over and over again.
  - The function is handling the normal case, not some extra functionallity that most users will not use.
  - The function is rarely called. (This restriction must be followed unless the function translates to fewer than 16 assembler instructions.)
  - The compiler can do additional optimizations with inlining and the resulting function will be only a fraction of size 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.
- If you have an 'if' statement that ends with a 'goto' or 'return' you should NOT have an else statement:

```
if (a == b)
  return 5;
else return 6;

->

if (a == b)
  return 5;
return 6;
```

- 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++. Always use a pointer in-

stead!

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.

- Do not use the `%p` marker of `printf()` (`fprintf()`, `vprintf()`, etc) because it leads to different outputs (for example on some Linux and Mac OS X the output starts with `0x` while it does not on some Solaris). Use `0x%lx` instead, even if it causes a truncation on some 64-bit platforms. Being sure that there is always `0x` enables us to quickly identify pointer values in the DBUG trace.

- Relying on loop counter variables being local to the loop body if declared in the `for` statement is not portable. Some compilers still don't implement this ANSI C++ specification. The symptom of such use is an error like this:

```
c-1101 CC: ERROR File = listener.cc, Line = 187
  "i" has already been declared in the current scope.

    for (int i= 0; i < num_sockets; i++)
```

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.

2.2. `configure` Support for Plugins

Beginning with MySQL 5.1, the server supports a plugin architecture for loading plugins. For example, several storage engines have been converted to plugins, and they can be selected or disabled at configuration time by means of command-line options to `configure`.

This section describes the command-line options that are used to control which plugins get built, and the autotools macros that enable plugin configuration support to be described.

2.2.1. `configure` Command-Line Plugin Options

Several `configure` options apply to plugin selection and building. You can build a plugin as static (compiled into the server) or dynamic (built as a dynamic library that must be installed using the `INSTALL PLUGIN` statement before it can be used). Some plugins might not support static or dynamic build.

`configure --help` shows the following information pertaining to plugins:

- The plugin-related options
- The names of all available plugins
- For each plugin, a description of its purpose, which build types it supports (static or dynamic), and which plugin groups it is a part of.

The following `configure` options are used to select or disable plugins:

```
--with-plugins=PLUGIN[,PLUGIN]...
--with-plugins=GROUP
--with-plugin-PLUGIN
--without-plugin-PLUGIN
```

`PLUGIN` is an individual plugin name such as `csv` or `archive`.

As shorthand, *`GROUP`* is a configuration group name such as `none` (select no plugins), `all` (select all plugins), or `max` (select all plugins used in a `mysqld-max` server).

`--with-plugins` can take a list of one or more plugin names separated by commas, or a plugin group name. The named plugins are configured to be built as static plugins.

`--with-plugin-PLUGIN` configures the given plugin to be built as a static plugin.

`--without-plugin-PLUGIN` disables the given plugin from being built.

If a plugin is named both with a `--with` and `--without` option, the result is undefined.

For any plugin that is not explicitly selected or disabled, it is selected to be built dynamically if it supports dynamic build, and not built if it does not support dynamic build. (Thus, in the case that no plugin options are given, all plugins that support dynamic build are selected to be built as dynamic plugins. Plugins that do not support dynamic build are not built.)

2.2.2. Autotools Plugin Macros

The following macros enable plugin support in the autotools configuration files.

- Declaring a plugin:

```
MYSQL_PLUGIN(name, long-name, description [,configlist])
```

Each plugin is required to have `MYSQL_PLUGIN()` declared first. *`configlist`* is an optional argument that is a comma-separated list of configurations of which the module is a member.

Example:

```
MYSQL_PLUGIN(ftexample, [Simple Parser], [Simple full-text parser plugin])
```

- Declaring a storage engine plugin:

```
MYSQL_STORAGE_ENGINE(name, legacy-opt, long-name, description [,configlist])
```

This is a simple utility macro that calls `MYSQL_PLUGIN`. It performs the bare basics required to declare a storage engine plugin and provides support for handling the legacy `configure` command-line options. If *`legacy-opt`* is not specified, it will default to `--with-`*`name`*`-storage-engine`. Set the `legacy-opt` value to `no` if you do not want to handle any legacy option.

This macro is roughly equivalent to:

```
MYSQL_PLUGIN(name, long-name, description)
MYSQL_PLUGIN_DEFINE(name, WITH_NAME_STORAGE_ENGINE)
```

Example:

```
MYSQL_STORAGE_ENGINE(berkeley,  berkeley-db, [BerkeleyDB Storage Engine],
      [Transactional Tables using BerkeleyDB], [max,max-no-ndb])
```

- Declaring a C preprocessor variable:

```
MYSQL_PLUGIN_DEFINE(name, define-name)
```

When a plugin will be included in a static build, this will set a preprocessor variable to 1. These preprocessor variables are defined in `config.h`.

Example:

```
MYSQL_PLUGIN_DEFINE(innobase, WITH_INNOBASE_STORAGE_ENGINE)
```

- Declaring a source directory for a plugin:

```
MYSQL_PLUGIN_DIRECTORY(name, dir-name)
```

Includes the specified directory into the build. If a file named `configure` is detected in the directory, it will be executed as part of the `configure` build otherwise it is assumed that there is a `Makefile` to be built in that directory. Currently, there is only support for plugin directories to be specified in the `storage/` and `plugin/` subdirectories.

Example:

```
MYSQL_PLUGIN_DIRECTORY(archive, [storage/archive])
```

- Declaring a static library name for a plugin:

```
MYSQL_PLUGIN_STATIC(name, lib-name)
```

Sets the `configure` substitution `@plugin_*name*_static_target@` to the supplied library name if the plugin is a static build. It also adds the library to the list of libraries to be linked into `mysqld`. It may either be just the name of the library (where, if there is a directory specified, the directory will be prepended for the link) or another `make` variable or substitution (in which case, it will be passed through as is).

Example:

```
MYSQL_PLUGIN_STATIC(archive, [libarchive.a])
MYSQL_PLUGIN_STATIC(berkeley, [[\$(bdb_libs_with_path)]])
```

- Declaring a dynamic library name for a plugin:

  ```
  MYSQL_PLUGIN_DYNAMIC(name, dso-name)
  ```

  Sets the `configure` substitution `@plugin_*name*_shared_target@` to the supplied dynamic shared object library name if the module is a dynamic build.

  Example:

  ```
  MYSQL_PLUGIN_DYNAMIC(archive, [ha_archive.la])
  ```

- Declaring a plugin as a mandatory module:

  ```
  MYSQL_PLUGIN_MANDATORY(name)
  ```

  Mandatory plugins cannot be disabled.

  Example:

  ```
  MYSQL_PLUGIN_MANDATORY(myisam)
  ```

- Declaring a plugin as disabled:

  ```
  MYSQL_PLUGIN_DISABLED(name)
  ```

  A disabled plugin will not be included in any build. If the plugin has been marked as `MANDATORY`, it will result in an `autoconf` error.

- Declaring additional plugin `configure` actions:

  ```
  MYSQL_PLUGIN_ACTIONS(name, configure-actions)
  ```

  This is useful if there are additional `configure` actions required for a plugin. The *`configure-actions`* argument may either be the name of an `autoconf` macro or more `autoconf` script.

  Example:

  ```
  MYSQL_PLUGIN_ACTIONS(ndbcluster,[MYSQL_SETUP_NDBCLUSTER])
  ```

- Declaring plugin dependencies:

  ```
  MYSQL_PLUGIN_DEPENDS(name, dependencies)
  ```

  Declares all plugins, in a comma-separated list, that are required for the named plugin to be built. If the named plugin is selected, it will in turn enable all its depenencies. All plugins listed as a dependency must already have been declared with `MYSQL_PLUGIN()`.

Example:

```
MYSQL_PLUGIN_DEPENDS(ndbcluster, [partition])
```

- Performing the magic:

  ```
  MYSQL_CONFIGURE_PLUGINS(default-names)
  ```

  Actually performs the task of generating the shell scripts for `configure` based upon the declarations made previously. It emits the shell code neccessary to check the options and sets the variables accordingly.

  Example:

  ```
  MYSQL_CONFIGURE_PLUGINS([none])
  ```

Plugin-related `configure` errors:

- When any plugin macro is called before `MYSQL_PLUGIN()` is declared for that plugin, `configure` aborts with an error.
- When any of the plugins specified in the dependency list don't exist, `configure` aborts with an error.
- When a mandatory plugin is specified in `--without-plugin-PLUGIN`, `configure` aborts with an error.
- When a disabled plugin is specified in `--with-modules=...` or `--with-plugin=PLUGIN`, `configure` reports an error.
- When an optional plugin that may only be built dynamically is specified in `--with-plugins=...` or `--with-plugin-PLUGIN`, `configure` emits a warning and continues to configure the plugin for dynamic build.
- When an optional plugin that may only be built statically is specified neither in `--with-plugins=...` nor `--without-plugin-PLUGIN`, `configure` emits a warning but should proceed anyway.

Avoiding `configure.in` changes:

- If a plugin source (which is located in a subdirectory of the `storage/` or `plugin/` directory) contains a `plug.in` file (for example, `storage/example/plug.in`), this file will be included as a part of `configure.in`. This way, `configure.in` does not need to be modified to add a new plugin to the build.
- A `plug.in` file may contain everything, particularly all `MYSQL_PLUGIN_xxx` macros as just described. The `plug.in` file does not need to specify `MYSQL_PLUGIN_DIRECTORY`; it is set automatically to the directory of the `plug.in` file.

Chapter 3. The Optimizer

This chapter describes the operation of the MySQL Query optimizer, which is used to determine the most efficient means for executing queries.

3.1. Code and Concepts

This section discusses key optimizer concepts, terminology, and how these are reflected in the MySQL server source code.

3.1.1. Definitions

This description uses a narrow definition: The *optimizer* 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 file and routine, such as "*See*: `/sql/select_cc`, `optimize_cond()`".

A *transformation* occurs when one query is changed into another query which delivers the same result. For example, a query could be changed from

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

to

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

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

3.1.2. The Optimizer Code

This diagram shows the structure of the function `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 applicable to all query types.

The `optimize_cond()` and `opt_sum_query()` routines perform 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.

3.2. Primary Optimizations

This section discusses the most important optimizations performed by the server.

3.2.1. Optimizing Constant Relations

3.2.1.1. 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()`.

3.2.1.2. Eliminating “Dead” Code

A transformation takes place for conditions that are always true, for example:

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

In this case, the first condition is removed, leaving

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

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

A transformation also takes place for conditions that are always false. For example, consider this `WHERE` clause:

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

Since the parenthesized part is always false, it is removed, reducing this expression to

```
WHERE s1 = 7
```

In some cases, where the `WHERE` clause represents an impossible condition, the optimizer might eliminate it completely. Consider the following:

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

Because it is never possible for this condition to be true, the `EXPLAIN` statement will show the words `Impossible WHERE`. Informally, we at MySQL say that 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 `Impossible WHERE` situations — there are too many possibilities in this regard. 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.

3.2.1.3. Folding of Constants

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.

3.2.1.4. 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()`.

3.2.2. Optimizing Joins

This section discusses the various methods used to optimize joins.

3.2.2.1. Determining the 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`: a unique or primary index with an equality relation
- `ref`: an index with an equality relation, where the index value cannot be `NULL`
- `ref_or_null`: an index with an equality relation, where it is possible for the index value to be `NULL`
- `range`: an index with a relation such as `BETWEEN`, `IN`, `>=`, `LIKE`, and so on.
- `index`: a sequential scan on an index
- `ALL`: a sequential scan of the entire 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 Plan: Read every row in the table. (This is called a *sequential scan* of `Table1` or simply *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 why it is often good to choose `indexed_column` as the driver.

3.2.2.2. Joins and Access Methods

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 such possible 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 that 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 performs 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 `LEFT JOIN` and `RIGHT JOIN` Optimization [http://dev.mysql.com/doc/refman/5.1/en/left-join-optimization.html].

3.2.2.3. 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()`.

3.2.2.4. 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 simply 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.

3.2.2.5. The `Index Merge` Join Type

3.2.2.5.1. 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.

3.2.2.5.2. 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.

3.2.2.5.3. The `range` Optimizer

For `range` 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 bound-

ary values.

2. is for a `SEL_ARG` object with condition on next key component.

3. is for a `SEL_ARG` object with an interval on the same field as this `SEL_ARG` object. Intervals between the current and “left” objects are disjoint and `left_sel_arg_cond.sup_val <= inf_val`.

4. is for a `SEL_ARG` object with an interval on the same field as this `SEL_ARG` object. Intervals between the current and “right” objects 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.

3.2.2.5.4. 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.

3.2.3. Transpositions

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 Section 3.2.1.4, “Constants and Constant Tables”.)

3.2.3.1. `AND` Relations

An `AND`ed search has the form `condition1 AND condition2`, as in this example:

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

Here, the optimizer's decision process can be described as follows:

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 Section 3.2.2.1, “Determining the 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.

   The optimizer can also choose to perform an `index_merge` index intersection, as described in Index Merge Optimization [http://dev.mysql.com/doc/refman/5.1/en/index-merge-optimization.html].

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.

3.2.3.2. `OR` Relations

An `OR`ed search has the form `condition1 OR condition2`, as in this example:

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

Here the optimizer's decision is to use a sequential scan.

There is also an option to use index merge under such circumstances. See Section 3.2.2.5.2, “Index Merge Optimizer” and Index Merge Optimization [http://dev.mysql.com/doc/refman/5.1/en/index-merge-optimization.html], for more information.

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.

3.2.3.3. `UNION` Queries

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.

3.2.3.4. `NOT` (`<>`) Relations

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 for both the previous cases.

3.2.4. `ORDER BY` Clauses

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 Section 3.2.2.4, “The `index` Join Type”.) 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()`.

`ORDER BY` Optimization [http://dev.mysql.com/doc/refman/5.1/en/order-by-optimization.html], provides a description of the internal sort procedure which we will not repeat here, but urge you to read, because it describes how the buffering and the quicksort mechanisms operate.

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

3.2.5. `GROUP BY` and Related Conditions

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().

3.3. Other Optimizations

In this section, we discuss other, more specialized optimizations performed in the MySQL server.

3.3.1. NULLs Filtering for ref and eq_ref Access

This section discusses the NULLs filtering optimization used for ref and eq_ref joins.

3.3.1.1. Early NULLs Filtering

Suppose we have a join order such as this one:

```
...,  tblX, ..., tblY, ...
```

Suppose further that table tblY is accessed via ref or eq_ref access on

```
tblY.key_column = tblX.column
```

or, in the case of ref access using multiple key parts, via

```
... AND tblY.key_partN = tblX.column AND ...
```

where tblX.*column* can be NULL. Here the early NULLs filtering for ref (or eq_ref) access is applied. We make the following inference:

```
(tblY.key_partN = tblX.column)  =>  (tblX.column IS NOT NULL)
```

The original equality can be checked only after we've read the current rows of both tables `tblX` and `tblY`. The `IS NOT NULL` predicate can be checked after we've read the current row of table `tblX`. If there are any tables in the join order between `tblX` and `tblY`, the added `IS NOT NULL` check will allow us to skip accessing those tables.

This feature is implemented in these places in the server code:

- The `ref` analyzer (contained in such functions as `update_ref_and_keys()`) detects and marks equalities like that shown above by setting `KEY_FIELD::null_rejecting=TRUE`.
- After the join order has been choosen, `add_not_null_conds()` adds appropriate `IS NOT NULL` predicates to the conditions of the appropriate tables.

It is possible to add `IS NOT NULL` predicates for all equalities that could be used for `ref` access (and not for those that are actually used). However, this is currently not done.

3.3.1.2. Late `NULL`s Filtering

Suppose we have a query plan with table `tblX` being accessed via the `ref` access method:

```
tblX.key_part1 = expr1 AND tblX.key_part2 = expr2 AND ...
```

Before performing an index lookup, we determine whether any of the `expri` values is `NULL`. If it is, we don't perform the lookup, but rather immediately return that the matching tuple is not found.

This optimization reuses the `null_rejecting` attribute produced by the early `NULL`s filtering code (see Section 3.3.1.1, “Early `NULL`s Filtering”). The check itself is located in the function `join_read_always_key()`.

3.3.2. Partitioning-Related Optimizations

This section discussions optimizations relating to MySQL Partitioning. See Partitioning [http://dev.mysql.com/doc/refman/5.1/en/partitioning.html] for general information about the partitioning implementation in MySQL 5.1 and later.

3.3.2.1. Partition pruning

The operation of *partition pruning* is defined as follows:

“Given a query over partitioned table, match the table DDL against any `WHERE` or `ON` clauses, and find the minimal set of partitions that must be accessed to resolve the query.”

The set of partitions thus obtained (hereafter referred to as “used”) can be smaller then the set of all table partitions. Partitions that did not get into this set (that is, those that were pruned away) will not be accessed at all: this is how query execution is made faster.

Non-Transactional Table Engines. With non-transactional tables such as `MyISAM`, locks are placed on entire partitioned table. It is theoretically possible to use partition pruning to improve concurrency by placing locks only on partitions that are actually used, but this is currently not implemented.

Partition pruning doesn't depend on what table engine is used. Therefore its implementation is a part of the MySQL Query Optimizer. The next few sections provide a detailed description of partition pruning.

3.3.2.1.1. Partition Pruning Overview

Partition pruning is performed using the following steps:

1. Analyze the `WHERE` clause and construct an *interval graph* describing the results of this analysis.
2. Walk the graph, and find sets of partitions (or subpartitions, if necessary) to be used for each interval in the graph.
3. Construct a set of partitions used for the entire query.

The description represented by the interval graph is structured in a "bottom-up" fashion. In the discussion that follows, we first define the term *partitioning interval*, then describe how partitioning interval are combined to make an interval graph, and then describe the graph "walking" process.

3.3.2.1.2. Partitioning Intervals

3.3.2.1.2.1. Single-Point Intervals

Let's start from simplest cases. Suppose that we have a partitioned table with *`N`* columns, using partitioning type *`p_type`* and the partitioning function *`p_func`*, represented like this:

```
CREATE TABLE t (columns)
PARTITION BY p_type(p_func(col1, col2,... colN)...);
```

Suppose also that we have a `WHERE` clause of the form

```
WHERE t.col1=const1 AND t.col2=const2 AND ... t.colN=constN
```

We can calculate `p_func(const1, const2 ... constN)` and discover which partition can contain records matching the `WHERE` clause. Note that this process works for all partitioning types and all partitioning functions.

Note

This process works only if the `WHERE` clause is of the exact form given above — that is, each column in the table must be tested for equality with some arbitrary constant (not necessarily the same constant for each column). For example, if `col1=const1` were missing from the example `WHERE` clause, then we would not be able to calculate the partitioning function value and so would be unable to restrict the set of partitions to those actually used.

3.3.2.1.2.2. Interval Walking

Let a partitioned table `t` be defined with a set of column definitions *`columns`*, a partitioning type *`p_type`* using a partitioning function *`p_func`* taking an integer column *`int_col`*, as shown here:

```
CREATE TABLE t (columns)
PARTITION BY
p_type(p_func(int_col))
...
```

Now suppose that we have a query whose `WHERE` clause is of the form

```
WHERE const1 <= int_col <= const2
```

We can reduce this case to a number of cases of single-point intervals by converting the `WHERE` clause into the following relation:

```
int_field=const1       OR
int_field=const1 + 1   OR
int_field=const1 + 2   OR
...                    OR
int_field=const2
```

In the source code this conversion is referred to as *interval walking*. Walking over short intervals is not very expensive, since we can reduce the number of partitions to scan to a small number. However, walking over long intervals may not be very efficient — there will be lots of numbers to examine, and we are very likely to out that all partitions need to be scanned.

The threshold for interval walking is determined by

```
#define MAX_RANGE_TO_WALK=10
```

> **Note**
>
> The logic of the previous example also applies for a relation such as this one:
>
> ```
> const1 >= int_col >= const2
> ```

3.3.2.1.2.3. Interval mapping

Let a partitioned table `t` be defined as follows:

```
CREATE TABLE t (columns)
PARTITION BY RANGE|LIST(unary_ascending_function(column))
```

Suppose we have a query on table `t` whose `WHERE` clause is of one of the forms shown here:

- `const1 <= t.column <= const2`
- `t.column <= const2`
- `const1 <= t.column`

Since the partitioning function is ascending, the following relationship holds:

```
const1 <= t.col <= const2

  => p_func(const1) <=

p_func(t.column) <= p_func(const2)
```

Using `A` and `B` to denote the leftmost and rightmost parts of this relation, we can rewrite it like this:

```
A <= p_func(t.column) <= B
```

> **Note**
>
> In this instance, the interval is closed and has two bounds. However, similar inferences can be performed for other kinds of intervals.

For RANGE partitioning, each partition occupies one interval on the partition function value axis, and the intervals are disjoint, as ahown here:

```
                          p0     p1      p2
  table partitions    ------x------x--------x-------->

  search interval     ----x==============x----------->
                          A              B
```

A partition needs to be accessed if and only if its interval has a non-empty intersection with the search interval [*A*, *B*].

For LIST partitioning, each partition covers a set of points on the partition function value axis. Points produced by various partitions may be interleaved, as shown here:

```
                        p0  p1   p2   p1   p1   p0
table partitions    --+---+----+----+----+----+---->

search interval     ----x===================x------>
                        A                   B
```

A partition needs to be accessed if it has at least one point in the interval [*A*, *B*]. The set of partitions used can be determined by running from *A* to *B* and collecting partitions that have their points within this range.

3.3.2.1.3. Subpartitioning Intervals

In the previous sections we've described ways to infer the set of used partitions from "elementary" WHERE clauses. Everything said there about partitions also applies to subpartitions (with exception that subpartitioning by RANGE or LIST is currently not possible).

Since each partition is subpartitioned in the same way, we'll find which subpartitions should be accessed within each partition.

3.3.2.1.4. From WHERE Clauses to Intervals

Previous sections deal with inferring the set of partitions used from WHERE clauses that represent partitioning or subpartitioning intervals. Now we look at how MySQL extracts intervals from arbitrary WHERE clauses.

The extraction process uses the *Range Analyzer* — a part of the MySQL optimizer that produces plans for the range access method. This is because the tasks are similar. In both cases we have a WHERE clause as input: the range access method needs index ranges (that is, intervals) to scan; partition pruning module needs partitioning intervals so that it can determine which partitions should be used.

For range access, the Range Analyzer is invoked with the WHERE clause and descriptions of table indexes. Each index is described by an ordered list of the columns which it covers:

```
(keypart1, keypart2, ..., keypartN)
```

For partition pruning, Range Analyzer is invoked with the WHERE clause and a list of table columns used by the partitioning and subpartitioning functions:

```
(part_col1, part_col2, ... part_colN,
subpart_col1, subpart_col2, ... subpart_colM)
```

The result of the Range Analyzer's work is known as a *SEL_ARG graph*. This is a complex (and not yet fully documented) structure, which we will not attempt to describe here. What's important for the current discussion is that we can walk over it and collect partitioning and subpartitioning intervals.

The following example illustrates the structure and the walking process. Suppose a table `t` is partitioned as follows:

```
CREATE TABLE t (..., pf INT, sp1 CHAR(5), sp2 INT, ... )
  PARTITION BY LIST (pf)
  SUBPARTITION BY HASH(sp1, sp2) (
    PARTITION p0 VALUES IN (1),
    PARTITION p1 VALUES IN (2),
    PARTITION p2 VALUES IN (3),
    PARTITION p3 VALUES IN (4),
    PARTITION p4 VALUES IN (5),
  );
```

Now suppose that a query on table `t` has a highly complex `WHERE` clause, such as this one:

```
pf=1 AND (sp1='foo' AND sp2 IN (40,50))

OR

(pf1=3 OR pf1=4) AND sp1='bar' AND sp2=33

OR

((pf=3 OR pf=4) AND sp1=5)

OR

p=8
```

The `SEL_ARG` graph for this is shown here:

```
(root)
   |               :
   | Partitioning  :         Sub-partitioning
   |               :
   |               :
   |               :
   |   +------+    :     +-----------+   +--------+
   \---| pf=1 |----:-----| sp1='foo' |---| sp2=40 |
       +------+    :     +-----------+   +--------+
          |        :                         |
          |        :                     +--------+
          |        :                     | sp2=50 |
          |        :                     +--------+
          |        :
          |        :
       +------+    :     +-----------+   +--------+
       | pf=3 |----:--+--| sp1='bar' |---| sp2=33 |
       +------+    :  |  +-----------+   +--------+
          |        :  |
       +------+    :  |
       | pf=4 |----:--+
       +------+    :
          |        :
          |        :
       +------+    :     +-----------+
       | pf=8 |----:-----| sp1='baz' |
       +------+    :     +-----------+
```

In the previous diagram, vertical edges (`|`) represent `OR` and the horizontal ones (`-`) represent `AND` (the line with both horizontal and vertical segments also represents `AND`).

The partition-pruning code walks the graph top to bottom and from left to right, making these inferences:

1. Start with an empty set of used partitions at the topmost and leftmost interval.

2.
    a. Perform interval analysis for `pf=1`; find a corresponding set of partitions *`P0`*; move right.

    b. Move right again, to `sp2=40`.

    c. Analyze the interval `sp1='foo' AND sp2=40` interval; find that it covers rows in some subpartition *`SP1`*. Make first inference: within each partition making up set *`P0`*, mark subpartition *`SP1`* as “used”.

    d. Move down to `sp2=50`.

    e. Analyze the interval `sp1='foo' AND sp2=50`, finding that it covers rows in some subpartition *`SP2`*. Make another inference: within each partition of set *`P0`*, mark subpartition *`SP2`* as used.

    f. Move back to `pf=1`, and then down to `pf=3`.

3.
    a. Perform interval analysis for `pf=3`; find a corresponding set of partitions *`P1`*; move right.

    b. Move right again, to `sp2=33`.

    c. Analyze the interval `sp1='foo' AND sp2=33`, find that it covers rows in a subpartition *`SP3`*. Make another inference: within each partition from set *`P1`*, mark subpartition *`SP3`* as “used”.

    d. Move back to `pf=3`, then down to `pf=4`.

4.
    a. Perform interval analysis for `pf=4`; find a corresponding set of partitions *`P2`*; move right.

    b. Perform moves and inferences analogous to what we did to the right of `pf=3`. There is some potential inefficiency due to the fact that that we will analyze the interval for `sp1='foo' AND sp2=33` again, but this should not have much impact on overall performance.

    c. Move back to `pf=3`, then down to `pf=8`.

5.
    a. Perform interval analysis for `pf=8`; find a corresponding set of partitions *`P3`*, move right.

    b. Now we've arrived at `sp1='baz'`, and find that we can't move any further to the right and can't construct a subpartitioning interval. We remember this, and move back to `pf=8`.

    c. In the previous step we could not limit the set of subpartitions, so we make this inference: for every partition in set *`P3`*, assume that all subpartitions are active, and mark them as such.

6. Try to move down from `pf=8`; find that there is nothing there; this completes the graph analysis.

Note

In certain cases the result of the `RANGE` optimizer will be several `SEL_ARG` graphs that are to be combined using `OR` or `AND` operators. This happens for `WHERE` clauses which either are very complicated or do not allow for the construction of a single list of intervals. In such cases, the partition pruning code takes apprpriate action, an example being this query:

```
SELECT * FROM t1 WHERE partition_id=10 OR subpartition_id=20
```

No single list of intervals can be constructed in this instance, but the partition pruning code correctly infers that the set of partitions used is a union of:

1. All subpartitions within the partition containing rows with partition_id=10; and

   a subpartition containing rows with subpartition_id=20 within each partition.

3.3.2.1.5. Partition Pruning in the Source Code

Here is a short walkthrough of what is where in the code:

- sql/opt_range.cc:

  This file contains the implementation of what is described in Section 3.3.2.1.4, “From WHERE Clauses to Intervals”. The entry point is the function prune_partitions().

  There are also detailed code-level comments about partition pruning; search for PartitionPruningModule to find the starting point.

- sql/partition_info.h:

```
class partition_info {
  ...
  /*
    Bitmap of used (i.e. not pruned away) partitions. This is where result
    of partition pruning is stored.
  */
  MY_BITMAP used_partitions;

  /*
    "virtual function" pointers to functions that perform interval analysis
    on this partitioned table (used by the code in opt_range.cc)
  */
  get_partitions_in_range_iter get_part_iter_for_interval;
  get_partitions_in_range_iter get_subpart_iter_for_interval;

};
```

- sql/sql_partition.cc:

  This file contains the functions implementing all types of partitioning interval analysis.

3.3.2.2. Partition selection

If a partitioned table is accessed in a series of index lookups (that is, using the ref, eq_ref, or ref_or_null access methods), MySQL checks to see whether it needs to make index lookups in all partitions or that it can limit access to a particular partition. This is performed for each index lookup.

Consider this example:

```
CREATE TABLE t1 (a INT, b INT);

INSERT INTO t1 VALUES (1,1),(2,2),(3,3);

CREATE TABLE t2 (
    keypart1 INT,
    keypart2 INT,
    KEY(keypart1, keypart2)
)
PARTITION BY HASH(keypart2);

INSERT INTO t2 VALUES (1,1),(2,2),(3,3);
```

The query

```
SELECT * FROM t1, t2
    WHERE  t2.keypart1=t1.a
    AND    t2.keypart2=t1.b;
```

is executed using this algorithm:

```
(for each record in t1:)
{
  t2->index_read({current-value-of(t1.a), current-value-of(t1.b)});
  while( t2->index_next_same() )
    pass row combination to query output;
}
```

In the `index_read()` call, the partitioned table handler will discover that the value of all partitioning columns (in this case, the single column `b`) is fixed, and find a single partition to access. If this partition was pruned away, then no partitions will be accessed at all.

Chapter 4. Important Algorithms and Structures

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

4.1. The `Item` Class

To us, the word *Item* means more than just “thingamabob”; it is a technical term with a precise definition in the context of our source code. `Item` is a class. Each instance of the `Item` class has:

- an analogue in the SQL language
- a value
- a data type descriptor

All of the following SQL “thingamabobs” are modeled in the `Item` class:

- literals
- column references
- session or global variables
- procedure variables
- parameters
- SQL functions (not a surprise since SQL functions have data types and return values).

In the *function* category we include operators such as `+` and `||`, because operators are merely functions that return values. We also include operators such as `=` and `LIKE`, which are operators that return boolean values. Consider the following statement:

```
SELECT UPPER(column1) FROM t WHERE column2 = @x;
```

For this statement, MySQL will need to store a list of items for the select list ('column1' column reference and UPPER function), and a list of items for the WHERE clause ('column2' column reference and '@x' variable and '=' operator).

Terminology: an Item instance in a MySQL program roughly corresponds to a "site", which according to the standard_SQL definition is "a place that holds an instance of a value of a specified data type", Another word that you'll see often in MySQL code is "field", which means column reference, and the Item_field subclass is generally for column values that occur for the intersection of a row and column in a table.

MySQL's Item class is defined in .../sql/item.h, and its subclasses are defined in .../sql/item*.h (that is, in item.h, item_cmpfunc.h, item_func.h, item_geofunc.h, item_row.h, item_strfunc.h, item_subselect.h, item_sum.h, item_timefunc.h). Page-width limitations prevent us from displaying the whole tree, but these are the main Item subclasses, and the subclasses of the subclasses:

```
Item_ident (Item_field, Item_ref)
Item_null
Item_num (Item_int, Item_real)
Item_param
```

```
Item_string (Item_static_string_func, Item_datetime, Item_empty_string)
Item_hex_string (Item_bin_string)
Item_result_field (all "item_func.h" "item_subselect.h" "item_sub.h" classes)
Item_copy_string
Item_cache (Item_cache_int, Item_cache_real, Item_cache_str, Item_cache_row)
Item_type_holder
Item_row
```

There's no formal classification of subclasses, but the main distinctions are by use (field, parameter, function) and by data type (num, string).

So, how does MySQL use items? You'll find that nearly every .cc program in the /sql directory makes some use of the Item class and its subclasses, so this list of programs is only partial and very general:

```
sql_parse.cc:     Makes new items in add_field_to_list()
item_sum.cc:      Uses item_func subclasses for COUNT, AVG, SUM
item_buff.cc:     Where buffers for item values can be stored
item_cmpfunc.cc:  Comparison functions with item_func subclasses
item_create.cc    For creating items that the lex might use
item_subselect.cc Subqueries are another type of function
mysqld.cc:        When main() ends, it uses clean_up() for items
opt_range.cc:     Uses field, compare-condition, and value subclasses
procedure.cc:     Notice Procedure * has a pointer to an item list
protocol.cc:      Uses send_fields() to pass item values back to users
sys_var.cc:       System variables have Item associations too
sql_base.cc:      Thread-specific Item searchers like find_field_in_table()
sql_class.cc:     Look at cleanup_after_query()
sql_delete.cc     This (like sql_insert.cc etc.) has field references
sql_error.cc      Has one of many examples of SHOW's use of items
sql_lex.cc:       Notice "add...to_list" functions
sql_select.cc     The largest program that uses items, apparently
udf_example.cc    The comments in this program are extensive
```

Whenever there's a need for an SQL operation that assigns, compares, aggregates, accepts, sends, or validates a site, you'll find a MySQL use of Item and its subclasses.

4.2. 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.)

4.3. 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.

4.4. 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.

4.5. 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.)

4.6. 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.

4.7. Full-text Search

MySQL uses Ranking with Vector Spaces for ordinary full-text queries.

Rank, also known as relevance rank, also known as relevance measure, is a number that tells us how good a match is.

Vector Space, which MySQL sometimes calls "natural language", is a well-known system based on a metaphor of lines that stretch in different dimensions (one dimension per term) for varying distances (one distance unit per occurrence of term). The value of thinking of it this way is: once you realize that term occurrences are lines in a multi-dimensional space, you can apply basic trigonometry to calculate "distances", and those distances are equatable with similarity measurements. A comprehensible discussion of vector space technology is here: http://www.miislita.com/term-vector/term-vector-1.html. And a text which partly inspired our original developer is here:
ftp://ftp.cs.cornell.edu/pub/smart/smart.11.0.tar.Z ("SMART").

But let's try to describe the classic formula:

```
w = tf * idf
```

This means "weight equals term frequency times inverse of document frequency", or "increase weight for number of times term appears in one document, decrease weight for number of documents the term appears in". (For historical rasons we're using the word "weight" instead of "distance", and we're using the information-retrieval word "document" throughout; when you see it, think of "the indexed part of the row".)

For example: if "rain" appears three times in row #5, weight goes up; but if "rain" also appears in 1000 other documents, weight goes down.

MySQL uses a variant of the classic formula, and adds on some calculations for "the normalization factor". In the end, MySQL's formula looks something like:

```
w = (log(dtf)+1)/sumdtf * U/(1+0.0115*U) * log((N-nf)/nf)
```

Where:

```
dtf     is the number of times the term appears in the document
sumdtf  is the sum of (log(dtf)+1)'s for all terms in the same document
U       is the number of Unique terms in the document
N       is the total number of documents
nf      is the number of documents that contain the term
```

The formula has three parts: base part, normalization factor, global multiplier.

The base part is the left of the formula, "(log(dtf)+1)/sumdtf".

The normalization factor is the middle part of the formula. The idea of normalization is: if a document is shorter than average length then weight goes up, if it's average length then weight stays the same, if it's longer than average length then weight goes down. We're using a pivoted unique normalization factor. For the theory and justification, see the paper "Pivoted Document Length Normalization" by Amit Singhal and Chris Buckley and Mandar Mitra ACM SIGIR'96, 21-29, 1996: http://ir.iit.edu/~dagr/cs529/files/handouts/singhal96pivoted.pdf. The word "unique" here means that our measure of document length is based on the unique terms in the document. We chose 0.0115 as the pivot value, it's PIVOT_VAL in the MySQL source code header file myisam/ftdefs.h.

If we multiply the base part times the normalization factor, we have the term weight. The term weight is what MySQL stores in the index.

The global multiplier is the final part of the formula. In the classic Vector Space formula, the final part would be the inverse document frequency, or simply

```
log(N/nf)
```

We have replaced it with

```
log((N-nf)/nf)
```

This variant is more often used in "probabilistic" formulas. Such formulas try to make a better guess of the probability that a term will be relevant. To go back to the old system, look in myisam/ftdefs.h for "#define GWS_IN_USE GWS_PROB" (i.e. global weights by probability) and change it to "#define GWS_IN_USE GWS_IDF" (i.e. global weights by inverse document frequency).

Then, when retrieving, the rank is the product of the weight and the frequency of the word in the query:

```
R = w * qf;
```

Where:

```
w      is the weight (as always)
qf     is the number of times the term appears in the query
```

In vector-space speak, the similarity is the product of the vectors.

And R is the floating-point number that you see if you say: SELECT MATCH(...) AGAINST (...) FROM t.

To sum it up, w, which stands for weight, goes up if the term occurs more often in a row, goes down if the term occurs in many rows, goes up / down depending whether the number of unique words in a row is fewer / more than average. Then R, which stands for either Rank or Relevance, is w times the frequency of the term in the AGAINST expression.

The Simplest Possible Example

First, make a fulltext index. Follow the instructions in the "MySQL Full-Text Functions" section of the MySQL Reference Manual. Succinctly, the statements are:

```
CREATE TABLE articles (
       id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
       title VARCHAR(200),
       body TEXT,
       FULLTEXT (title,body)
     );
INSERT INTO articles (title,body) VALUES
     ('MySQL Tutorial','DBMS stands for DataBase ...'),
     ('How To Use MySQL Well','After you went through a ...'),
     ('Optimizing MySQL','In this tutorial we will show ...'),
     ('1001 MySQL Tricks','1. Never run mysqld as root. 2. ...'),
     ('MySQL vs. YourSQL','In the following database comparison ...'),
     ('MySQL Security','When configured properly, MySQL ...');
```

Now, let's look at the index.

There's a utility for looking at the fulltext index keys and their weights. The source code is myisam/myisam_ftdump.c, and the executable comes with the binary distribution. So, if exedir is where the executable is, and datadir is the directory name that you get with "SHOW VARIABLES LIKE 'datadir%'", and dbname is the name of the database that contains the articles table, then this works:

```
>/exedir/myisam_ftdump /datadir/dbname/articles 1 -d
       b8            0.9456265 1001
       f8            0.9560229 comparison
      140            0.8148246 configured
        0            0.9456265 database
       f8            0.9560229 database
        0            0.9456265 dbms
        0            0.9456265 mysql
       38            0.9886308 mysql
       78            0.9560229 mysql
       b8            0.9456265 mysql
       f8            0.9560229 mysql
      140            1.3796179 mysql
       b8            0.9456265 mysqld
       78            0.9560229 optimizing
      140            0.8148246 properly
       b8            0.9456265 root
      140            0.8148246 security
       78            0.9560229 show
        0            0.9456265 stands
       b8            0.9456265 tricks
        0            0.9456265 tutorial
       78            0.9560229 tutorial
       f8            0.9560229 yoursql
```

Let's see how one of these numbers relates to the formula.

The term 'tutorial' appears in document 0. The full document is "MySQL Tutorial / DBMS stands for DataBase ...". The word "tutorial" appears once in the document, so dtf = 1. The word "for" is a stop-word, so there are only 5 unique terms in the document ("mysql", "tutorial", "dbms", "stands", "database"), so U = 5. Each of these terms appears once in the document, so sumdtf is the sum of log(1)+1, five times. So, taking the first two parts of the formula (the term weight), we have:

```
(log(dtf)+1)/sumdtf * U/(1+0.0115*U)
```

which is

```
(log(1)+1)/( (log(1)+1)*5) * 5/(1+0.0115*5)
```

which is

```
0.9456265
```

which is what myisam_ftdump says. So the term weight looks good.

Now, what about the global multiplier? Well, myisam_ftdump could calculate it, but you'll see it with the mysql client. The total number of rows in the articles table is 6, so N = 6. And "tutorial" occurs in two rows, in row 0 and in row 78, so nf = 2. So, taking the final (global multiplier) part of the formula, we have:

```
log((N-nf)/nf)
```

which is

```
log((6-2)/2)
```

which is

```
0.6931472
```

So what would we get for row 0 with a search for 'tutorial'? Well, first we want w, so: Multiply the term weight of tutorial (which is 0.9456265) times the global multiplier (which is 0.6931472). Then we want R, so: Multiply w times the number of times that the word 'tutorial' appears in the search (which is 1). In other words, R = 0.9456265 * 0.6931472 * 1. Here's the proof:

```
mysql> select round(0.9456265 * 0.6931472 * 1, 7) as R;
+-----------+
| R         |
+-----------+
| 0.6554583 |
+-----------+
1 row in set (0.00 sec)
```

```
mysql> select round(match(title,body) against ('tutorial'), 7) as R
    -> from articles limit 1;
+-----------+
| R         |
+-----------+
| 0.6554583 |
+-----------+
1 row in set (0.00 sec)
```

You'll need memory

The MySQL experience is that many users appreciate the full-text precision or recall, that is, the rows that MySQL returns are relevant and the rows that MySQL misses are rare, in the judgment of some real people. That means that the weighting formula is probably justifiable for most occasions. Since it's the product of lengthy academic research, that's understandable.

On the other hand, there are occasional complaints about speed. Here, the tricky part is that the formula depends on global factors -- specifically N (the number of documents) and nf (the number of documents that contain the term). Every time that insert/update/delete occurs for any row in the table, these global weight factors change for all rows in the table.

If MySQL was a search engine and there was no need to update in real time, this tricky part wouldn't matter. With occasional batch runs that redo the whole index, the global factors can be stored in the index. Search speed declines as the number of rows increases, but search engines work.

However, MySQL is a DBMS. So when updates happen, users expect the results to be visible immediately. It would take too long to replace the weights for all keys in the fulltext index, for every single update/insert/delete. So MySQL only stores the local factors in the index. The global factors are more dynamic. So MySQL stores an in-memory binary tree of the keys. Using this tree, MySQL can calculate the count of matching rows with reasonable speed. But speed declines logarithmically as the number of terms increases.

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.

4.8. `FLOAT` and `DOUBLE` data types and their representation.

The MySQL Reference Manual has a discussion of floating-point numbers in Section 11.2 Numeric Types, including details about the storage. Let us now take up the story from where the MySQL Reference Manual leaves off.

The following discussion concentrates on the case where no display width and decimals are given. This means that `FLOAT` is stored as whatever the C type `float` is and `REAL` or `DOUBLE [PRECISION]` is stored as whatever the C type `double` is. The field length is selected by the MySQL code.

This document was created when Bug#4457 [http://bugs.mysql.com/4457] (Different results in SQL-Statements for the same record) was fixed at the end of August 2004. Until then there was some confusion in the double-to-string conversion at different places in the code.

The bugfix for Bug#4937 [http://bugs.mysql.com/4937] (`INSERT + SELECT + UNION ALL + DATE to VARCHAR(8)` conversion problem) produced a conversion function which was a promising approach to the conversion problems. Unfortunately it was only used for direct field conversions and not for function results etc. It did not take small numbers (absolute value less than 1) and negative numbers into account. It did not take the limited precision of `float` and `double` data types into account. The bugfix was developed in two steps: The first attempt looked like this (in principle):

```
length= sprintf(buff, "%.*g", field_length, nr);
if (length > field_length)
  length= sprintf(buff, "%.*g", field_length-5, nr);
```

If the `libc` conversion produces too many characters, the precision is reduced by the space required for the scientific notation (1.234e+05). Thus the `printf()` conversion is forced to switch to the scientific notation, since the value would not fit otherwise. Or, if it was scientific already, the precision is reduced and also uses less space. I left out some important stuff around limit checking just to show the idea. This simple algorithm should work quite well in most cases, but has been discarded for the sake of performance. The double call to the slow `printf()` conversion `%g` didn't seem reasonable, though it would only be used for extreme values and small fields. During my explorations of the code I didn't find places where `float` or `double` were to be converted into small fields. Remeber that I talk only of conversions where field length and precision are not given. In this case a sufficient field length is selected at several places, except for a bug where it was selected wrongly. If a field length is given, a different conversion is used anyway. But since the code is quite complex, I don't claim to grasp it in full, and therefore may be in error. So let us look further:

The second attempt to fix the bug looked like this:

```
bool use_scientific_notation=TRUE;
if (field_length < 32 && nr > 1)
{
  double e[]={1, 1e1, 1e2, 1e4, 1e8, 1e16 }, p=1;
  for (int i=sizeof(e), j=1<<i-- ; j; i--,  j>>=1 )
  {
    if (field_length & j)
      p*=e[i];
  }
  use_scientific_notation=(p < nr);
}
length= sprintf(buff, "%.*g", use_scientific_notation ?
                              field_length-5 : field_length, nr);
```

Here we evaluate if the string representation of a given number fits into field_length characters. If not, we reduce the precision to make it fit. Again, I left out important details. For example, the evaluation is done only once per field for the sake of performance. The downside here is the unconditional reduction of precision for field length > 31 (which doesn't really matter), for negative numbers and for small numbers (absolute value less than 1).

Both algorithms do not take the limited precision of `float` and `double` values into account. This could lead to conversions with ridiculous bogus precision output. For example a value of 0.7 converted with `%.30g` will give a lot of digits, which pretend to tell about deviations from the value 0.7 and are completely absurd: 0.699999988079071044921875. To understand more about the `%g` conversion, I quote from a comment introduced in the source at the beginning of bugfixing #4937 (this comment was removed because it mainly describes, how the `printf()` conversion works, but I think it's valuable enough to include it here):

```
/*
  Let's try to pretty print a floating point number. Here we use
  '%-*.*g' conversion string:
    '-' stands for right-padding with spaces, if such padding will take
  place
    '*' is a placeholder for the first argument, field_length, and
  signifies minimal width of result string. If result is less than
  field length it will be space-padded. Note, however, that we'll not
  pass spaces to Field_string::store(const char *, ...), due to
  strcend in the next line.
    '.*' is a placeholder for DBL_DIG and defines maximum number of
  significant digits in the result string. DBL_DIG is a hardware
  specific C define for maximum number of decimal digits of a floating
  point number, such that rounding to hardware floating point
  representation and back to decimal will not lead to loss of
  precision. That is: if DBL_DIG is 15, number 123456789111315 can be
  represented as double without precision loss.  As one can judge from
  this description, choosing DBL_DIG here is questionable, especially
  because it introduces a system dependency.
    'g' means that conversion will use [-]ddd.ddd (conventional) style,
  and fall back to [-]d.ddde[+|i]ddd (scientific) style if there is not
  enough space for all digits.
  Maximum length of result string (not counting spaces) is (I guess)
  DBL_DIG + 8, where 8 is 1 for sign, 1 for decimal point, 1 for
  exponent sign, 1 for exponent, and 4 for exponent value.
  XXX: why do we use space-padding and trim spaces in the next line?
*/
sprintf(to,"%-*.*g",(int) field_length,DBL_DIG,nr);
to=strcend(to,' ');
```

There is one small misapprehension in the comment. `%g` does not switch to scientific notation when there is 'not enough space for all digits'. As the commentator says, the field length gives the minimal output length. `printf()` happily outputs more characters if required to produce a result with 'precision' digits. In fact it switches to scientific when the value can no longer be represented by 'precision' digits in conventional notation. The man page says "Style e is used if the exponent from its conversion is less than -4 or greater than or equal to the precision." In explanation, a precision of 3 digits can print a value of 345 in conventional notation, but 3456 needs scientific notation, as it would require 4 digits (a precision of 4) in conventional notation. Thus, it is printed as 3.46e+03 (rounded).

Since we don't want spaces in the output, we should not give a field length, but always use `"%.*g"`. However, the precision matters, as seen above. It is worth its own paragraph.

Since MySQL uses the machine-dependent binary representation of `float` and `double` to store values in the database, we have to care about these. Today, most systems use the IEEE standard 754 for binary floating-point arithmetic. It describes a representation for single precision numbers as 1 bit for sign, 8 bits for biased exponent and 23 bits for fraction and for double precision numbers as 1-bit sign, 11-bit biased exponent and 52-bit fraction. However, we can not rely on the fact that every system uses this representation. Luckily, the ISO C standard requires the standard C library to have a header `float.h` that describes some details of the floating point representation on a machine. The comment above describes the value `DBL_DIG`. There is an equivalent value `FLT_DIG` for the C data type `float`.

So, whenever we print a floating-point value, we must not specify a precision above `DBL_DIG` or `FLT_DIG` respectively. Otherwise we produce a bogus precision, which is wrong. For the honor of the writer of the first attempt above, I must say that his complete algorithm took `DBL_DIG` into account, if however only for the second call to `sprintf()`. But `FLT_DIG` has never been accounted for. At the conversion section of the code, it was not even known whether the value came from a `float` or `double` field.

My attempt to solve the problems tries to take all this into account. I tried to concentrate all `float`/`double`-to-string conversions in one function, and to bring the knowledge about `float` versus `double` to this function wherever it is called. This solution managed to keep the test suite happy while solving the new problem of Bug#4457 [http://bugs.mysql.com/4457]. Luckily the first problem was not big, as the test cases have been very carefully selected, so that they succeed as long as the machine uses

IEEE 754.

Nevertheless, the function is still not perfect. It is not possible to guess how many sigificant digits a number has. Given that, it is not simple to tell how long the resulting string would be. This applies to numbers with an absolute value smaller then 1. There are probably ways to figure this out, but I doubt that we would win in terms of performance over the simple solution of the first attempt, and besides we might cause new bugs. The compromise taken here is to accept that the resulting string may exceed the destination field length by five characters in the worst case.

```
if (nr < 0.0)
{
  abs_nr= -nr;
  extra_space= 1;
}
else
{
  abs_nr= nr;
  extra_space= 0;
}
precision= is_float ? FLT_DIG : DBL_DIG;
if (precision > field_length)
  precision= field_length;

if (! initialized)
{
  /* Better switch to scientific too early than too late. */
  double mult;
  mult= 1e0;
  for (length= 0; length < DBL_DIG; length++)
    mult/= 1e1;
  mult= 1e1 - mult;

  double val;
  val= 1.0;
  for (int idx= 0; idx < DBL_DIG+1; idx++)
  {
    DBUG_PRINT("info",("double_to_string_conv: big[%d] %.*g",
                       idx, DBL_DIG+3, val));
    big_number[idx]= val;
    val*= mult;
  }
  small_number[0]= 1e0;
  small_number[1]= 1e0;
  small_number[2]= 1e0;
  small_number[3]= 1e-1;
  small_number[4]= 1e-2;
  small_number[5]= 1e-3;
  small_number[6]= 1e-4;
  /* %g switches to scientific when exponent < -4. */
  for (int idx= 7; idx < DBL_DIG+1; idx++)
    small_number[idx]= 1e-4;
  initialized= TRUE;
}

use_scientific_notation= (abs_nr != 0.0) &&
                         ((abs_nr >   big_number[precision]) ||
                          (abs_nr < small_number[precision]));

if (use_scientific_notation)
{
  if (((nr >= 0.0) && ((nr >= 1e+100) || (nr <= 1e-100))) ||
      ((nr < 0.0) && ((nr <= -1e+100) || (nr >= -1e-100))))
    extra_space+= 6; /* .e+100 or .e-100 */
  else
    extra_space+= 5; /* .e+99  or .e-99 */
}

if (field_length < extra_space)
  precision= 0;
else if (precision > (field_length - extra_space))
  precision= field_length - extra_space;

length= sprintf(buff, "%.*g", precision, nr);
```

This solution takes performance into account by initializing the limiting numbers arrays only once into static space. It copes with negative numbers and tries to decide even over small numbers. The latter has only small implications, as the prefix 0.000 is exactly the same size as the postfix e-100. But knowing if scientific notation will be selected by `sprintf()` allows for saving one digit when the exponent is larger than -100.

The calculations for the big number array are less precise than in the second attempt, but faster. The precision is sufficient for the guess whether `sprintf()` uses scientific notation. There may be number to field length combinations which exploit the gap, but these won't emerge anyway as I found no situation where this function is called with small field lengths. Remember again that it is not called with user-supplied field lengths.

However in the current stable releases (including gamma) we have some places where the field length is too small by one character. Thus, the precision is sometimes one digit smaller than `DBL_DIG` would allow for. Consequently, we cannot use the simple algorithm in the stable releases. There is a chance of doing it in a development release, though.

Addendum:

There turned out to be a new solution to the "big number array" problem. We have a statically initialized array `log_10`, which holds the necessary values. But I did not check whether these values are safe. Even if computed by the compiler, they could carry values slightly above the decimal powers, which would be bad. In this case we needed to initialize by 9.99999999e+xxx, where the number of nines is equal to `DBL_DIG`. This must be protected by `#if DBL_DIG == yy`, so that a new `DBL_DIG` on a new platform is detected. And the array is of limited length. We must at least protect it by a `DBUG_ASSERT(sizeof(log_10)/sizeof(log_10[0]) > DBL_DIG)`.

But all of this is probably completely unneccessary, since we are only speaking of ceses where no user-supplied field length is given. So MySQL selects the field length on its own. So it is totally possible, indeed highly desirable, that MySQL selects a field length, which allows for a maximum of precision for all possible values. And these are `DBL_DIG+7` or `FLT_DIG+6` respectively as far as IEEE 754 is used. In this case we can have values of about +/-1e-307 to +/-1e+308 for `double` and +/-1e-37 to +/-1e+38 for `float`. That is, for example -1.<DBL_DIG-1 digits>e+100. For cases where a precision above IEEE 754 is possible, we may need +8 instead. We can detect this with `#if DBL_MAX_10_EXP >= 1000`. So using a field length of `DBL_DIG+8` in all cases should be sufficient for a simple `sprintf(buff, "%.*g", DBL_DIG, nr)` or `sprintf(buff, "%.*g", FLT_DIG, nr)`, respectively. To be safe, we should not use the machine dependent constants everywhere, but instead concentrate them into definitions like these:

```
#if (DBL_MAX_10_EXP > 9999) || (DBL_MIN_10_EXP < -9999)
#  error "Need new definition for UNSPECIFIED_DOUBLE_FIELD_LENGTH"
#elif (DBL_MAX_10_EXP > 999) || (DBL_MIN_10_EXP < -999)
#  define UNSPECIFIED_DOUBLE_FIELD_LENGTH (DBL_DIG+8)
#else
#  define UNSPECIFIED_DOUBLE_FIELD_LENGTH (DBL_DIG+7)
#endif

#if (FLT_MAX_10_EXP > 999) || (FLT_MIN_10_EXP < -999)
#error "Need new definition for UNSPECIFIED_FLOAT_FIELD_LENGTH"
#elif (FLT_MAX_10_EXP > 99) || (FLT_MIN_10_EXP < -99)
#  define UNSPECIFIED_FLOAT_FIELD_LENGTH (FLT_DIG+7)
#else
#  define UNSPECIFIED_FLOAT_FIELD_LENGTH (FLT_DIG+6)
#endif
```

These definitions should be used wherever an item or field of type `float` or `double` without an explicit field length specification is encountered. We have to propagate these lengths though all derived items and fields and we have to select the maximum of all field lengths wherever in two or more of them are used in an expression or a function.

We need to treat the precision (`DBL_DIG`/`FLT_DIG`) similarly, but have to select the minimum in expressions or functions.

4.9. Threads

Threads in mysqld can run at four different priorities, defined in mysql_priv.h:

```
      #define INTERRUPT_PRIOR 10
      #define CONNECT_PRIOR    9
      #define WAIT_PRIOR    8
      #define QUERY_PRIOR    6
```

Some threads try to set their priority; others don't. These calls are passed along to pthread_setschedparam() if the native threading library implements it.

The different threads are:

- The main thread. Runs at CONNECT_PRIOR priority. Calls thr_setconcurrency() if it is available at compile time; this call is generally assumed to exist only on Solaris, its value should reflect the number of physical CPUs.
- The "bootstrap" thread. See handle_bootstrap() in sql_parse.cc. The mysql_install_db script starts a server with an option telling it to start this thread and read commands in from a file. Used to initialize the grant tables. Runs once and then exits.
- The "maintenance" thread. See sql_manager_cc. Like the old "sync" daemon in unix, this thread occasionally flushes myisam tables to disk. InnoDB has a separate maintenance thread, but BDB also uses this one to occasionally call berkeley_cleanup_log_files(). Begins at startup and persists until shutdown.
- The "handle TCP/IP sockets" thread. See handle_connections_sockets() in mysqld.cc. Loop with a select() function call, to handle incoming connections.
- The "handle named pipes" thread. Only on Windows.
- The "handle shared memory connections" thread. Only on Windows.
- Signal handler ("interrupt") thread. See signal_hand() in mysqld.cc. Runs at INTERRUPT_PRIOR priority. Sets up to receive signals, and then handles them as they come in. Begins at server startup and persists until shutdown.
- The "shutdown" thread. See kill_server() in mysqld.cc. Created by the signal handling thread. Closes all connections with close_connections(), the ends.
- Active and cached per-connection threads. See handle_one_connection() in sql_parse.cc. These can run at QUERY_PRIOR priority or WAIT_PRIOR priority depending on what they are doing.
- The "delayed" thread. See handle_delayed_insert() in sql_insert.cc. Used for MyISAM's delayed inserts.
- The two slave threads, in slave.cc. One thread connects to the master and handles network IO. The other reads queries from the relay log and executes them.

In InnoDB, all thread management is handled through os/os0thread.c InnoDB's threads are:

- The I/O handler threads, See io_handler_thread().
- Two "watchmen" threads: srv_lock_timeout_and_monitor_thread(), and srv_error_monitor_thread().
- The master thread "which does purge and other utility operations", See srv_master_thread().

InnoDB's internal os_thread_set_priority() function implements three priorities (Background, normal, and high) but only on windows. The function is a no-op on unix.

4.10. Character/Collation Sets

Character sets are used by MySQL when storing information, both to ensure that the information is stored (and returned) in the correct format, but also for the purposes of collation and sorting. Each character set supports one or more collations, and so these are collectively known as `Collation Sets`, rather than character sets.

Character sets are recorded against individual tables and returned as part of the field data. For example, the `MYSQL_FIELD` data type definition includes the field `charsetnr`:

```
typedef struct st_mysql_field {
  char *name;                 /* Name of column */
  char *org_name;             /* Original column name, if an alias */
  char *table;                /* Table of column if column was a field */
  char *org_table;            /* Org table name, if table was an alias */
  char *db;                   /* Database for table */
  char *catalog;              /* Catalog for table */
  char *def;                  /* Default value (set by mysql_list_fields) */
  unsigned long length;       /* Width of column (create length) */
  unsigned long max_length;   /* Max width for selected set */
  unsigned int name_length;
  unsigned int org_name_length;
  unsigned int table_length;
  unsigned int org_table_length;
  unsigned int db_length;
  unsigned int catalog_length;
  unsigned int def_length;
  unsigned int flags;         /* Div flags */
  unsigned int decimals;      /* Number of decimals in field */
  unsigned int charsetnr;     /* Character set */
  enum enum_field_types type; /* Type of field. See mysql_com.h for types */
} MYSQL_FIELD;
```

Character set and collation informaiton are specific to a server version and installation, and are generated automatically from the `sql/share/charsets/Index.xml` file in the source distribution.

You can obtain a list of the available character sets configured within a server by running `SHOW COLLATION`, or by running a query on the `INFORMATION_SCHEMA.COLLATION` table. A sample of the information from that table has been provided here for reference.

Collation Id	Charset	Collation	Default	Sortlen
64	`armscii8`	`armscii8_bin`		1
32	`armscii8`	`armscii8_general_ci`	Yes	1
65	`ascii`	`ascii_bin`		1
11	`ascii`	`ascii_general_ci`	Yes	1
84	`big5`	`big5_bin`		1
1	`big5`	`big5_chinese_ci`	Yes	1
63	`binary`	`binary`	Yes	1
66	`cp1250`	`cp1250_bin`		1
44	`cp1250`	`cp1250_croatian_ci`		1
34	`cp1250`	`cp1250_czech_cs`		2
26	`cp1250`	`cp1250_general_ci`	Yes	1
50	`cp1251`	`cp1251_bin`		1
14	`cp1251`	`cp1251_bulgarian_ci`		1

52	cp1251	cp1251_general_cs		1
23	cp1251	cp1251_ukrainian_ci		1
51	cp1251	cp1251_general_ci	Yes	1
67	cp1256	cp1256_bin		1
57	cp1256	cp1256_general_ci	Yes	1
58	cp1257	cp1257_bin		1
29	cp1257	cp1257_lithuanian_ci		1
59	cp1257	cp1257_general_ci	Yes	1
80	cp850	cp850_bin		1
4	cp850	cp850_general_ci	Yes	1
81	cp852	cp852_bin		1
40	cp852	cp852_general_ci	Yes	1
68	cp866	cp866_bin		1
36	cp866	cp866_general_ci	Yes	1
96	cp932	cp932_bin		1
95	cp932	cp932_japanese_ci	Yes	1
69	dec8	dec8_bin		1
3	dec8	dec8_swedish_ci	Yes	1
98	eucjpms	eucjpms_bin		1
97	eucjpms	eucjpms_japanese_ci	Yes	1
85	euckr	euckr_bin		1
19	euckr	euckr_korean_ci	Yes	1
86	gb2312	gb2312_bin		1
24	gb2312	gb2312_chinese_ci	Yes	1
87	gbk	gbk_bin		1
28	gbk	gbk_chinese_ci	Yes	1
93	geostd8	geostd8_bin		1
92	geostd8	geostd8_general_ci	Yes	1
70	greek	greek_bin		1
25	greek	greek_general_ci	Yes	1
71	hebrew	hebrew_bin		1
16	hebrew	hebrew_general_ci	Yes	1
72	hp8	hp8_bin		1
6	hp8	hp8_english_ci	Yes	1
73	keybcs2	keybcs2_bin		1
37	keybcs2	keybcs2_general_ci	Yes	1
74	koi8r	koi8r_bin		1
7	koi8r	koi8r_general_ci	Yes	1
75	koi8u	koi8u_bin		1
22	koi8u	koi8u_general_ci	Yes	1
47	latin1	latin1_bin		1
15	latin1	latin1_danish_ci		1

48	latin1	latin1_general_ci		1
49	latin1	latin1_general_cs		1
5	latin1	latin1_german1_ci		1
31	latin1	latin1_german2_ci		2
94	latin1	latin1_spanish_ci		1
8	latin1	latin1_swedish_ci	Yes	1
77	latin2	latin2_bin		1
27	latin2	latin2_croatian_ci		1
2	latin2	latin2_czech_cs		4
21	latin2	latin2_hungarian_ci		1
9	latin2	latin2_general_ci	Yes	1
78	latin5	latin5_bin		1
30	latin5	latin5_turkish_ci	Yes	1
79	latin7	latin7_bin		1
20	latin7	latin7_estonian_cs		1
42	latin7	latin7_general_cs		1
41	latin7	latin7_general_ci	Yes	1
43	macce	macce_bin		1
38	macce	macce_general_ci	Yes	1
53	macroman	macroman_bin		1
39	macroman	macroman_general_ci	Yes	1
88	sjis	sjis_bin		1
13	sjis	sjis_japanese_ci	Yes	1
82	swe7	swe7_bin		1
10	swe7	swe7_swedish_ci	Yes	1
89	tis620	tis620_bin		1
18	tis620	tis620_thai_ci	Yes	4
90	ucs2	ucs2_bin		1
138	ucs2	ucs2_czech_ci		8
139	ucs2	ucs2_danish_ci		8
145	ucs2	ucs2_esperanto_ci		8
134	ucs2	ucs2_estonian_ci		8
146	ucs2	ucs2_hungarian_ci		8
129	ucs2	ucs2_icelandic_ci		8
130	ucs2	ucs2_latvian_ci		8
140	ucs2	ucs2_lithuanian_ci		8
144	ucs2	ucs2_persian_ci		8
133	ucs2	ucs2_polish_ci		8
131	ucs2	ucs2_romanian_ci		8
143	ucs2	ucs2_roman_ci		8
141	ucs2	ucs2_slovak_ci		8
132	ucs2	ucs2_slovenian_ci		8

142	ucs2	ucs2_spanish2_ci		8
135	ucs2	ucs2_spanish_ci		8
136	ucs2	ucs2_swedish_ci		8
137	ucs2	ucs2_turkish_ci		8
128	ucs2	ucs2_unicode_ci		8
35	ucs2	ucs2_general_ci	Yes	1
91	ujis	ujis_bin		1
12	ujis	ujis_japanese_ci	Yes	1
83	utf8	utf8_bin		1
202	utf8	utf8_czech_ci		8
203	utf8	utf8_danish_ci		8
209	utf8	utf8_esperanto_ci		8
198	utf8	utf8_estonian_ci		8
210	utf8	utf8_hungarian_ci		8
193	utf8	utf8_icelandic_ci		8
194	utf8	utf8_latvian_ci		8
204	utf8	utf8_lithuanian_ci		8
208	utf8	utf8_persian_ci		8
197	utf8	utf8_polish_ci		8
195	utf8	utf8_romanian_ci		8
207	utf8	utf8_roman_ci		8
205	utf8	utf8_slovak_ci		8
196	utf8	utf8_slovenian_ci		8
206	utf8	utf8_spanish2_ci		8
199	utf8	utf8_spanish_ci		8
200	utf8	utf8_swedish_ci		8
201	utf8	utf8_turkish_ci		8
192	utf8	utf8_unicode_ci		8
33	utf8	utf8_general_ci	Yes	1

Note that it is the collation ID, not the character set ID, that is used to identify the unique combination of character set and collation. Thus, when requesting character set information using one of the character set functions in `mysys/charset.c`, such as `get_charset()`, different IDs may return the same base character set, but a different collation set.

The following functions provide an internal interface to the collation and character set information, enabling you to access the information by name or ID:

```
static uint get_collation_number_internal(const char *name)
uint get_collation_number(const char *name)
uint get_charset_number(const char *charset_name, uint cs_flags)
const char *get_charset_name(uint charset_number)
static CHARSET_INFO *get_internal_charset(uint cs_number, myf flags)
CHARSET_INFO *get_charset(uint cs_number, myf flags)
CHARSET_INFO *get_charset_by_name(const char *cs_name, myf flags)
CHARSET_INFO *get_charset_by_csname(const char *cs_name,
                                    uint cs_flags,
                                    myf flags)
```

The table below details the functions, the key argument that is supplied, and the return value.

Function	Supplied Argument	Return Value
`get_collation_number_internal()`	Collation name	Collation ID
`get_collation_number()`	Collation name	Collation ID
`get_charset_number()`	Character set name	Collation ID
`get_charset_name()`	Collation ID	Charactet set name
`get_internal_charset()`	Collation ID	Character datatype
`get_charset()`	Collation ID	Character datatype

An example of using the collation/character set functions is available in the `extras/charset2html.c`, which outputs an HTML version of the internal collation set table.

4.11. Error flags and functions

The following flags can be examined or set to alter the behavior during error handling:

- `thd->net.report_error`

  `thd->net.report_error` is set in `my_message_sql()` if the error message was registered. (`my_message_sql()` is called by `my_error()`, `my_printf_error()`, `my_message()`).

- `thd->query_error`

  Like `net.report_error`, but is always set to 1 in `my_message_sql()` if error was not caught by an error handler. Used by replication to see if a query generated any kind of errors.

- `thd->no_warnings_for_error`

  Normally an error also generates a warning. The warning can be disabled by setting `thd->no_warnings_for_error`. (This allows one to catch all error messages generated by a statement)

- `thd->lex->current_select->no_error`

  This is set to in case likes `INSERT IGNORE ... SELECT`. In this case we ignore all not fatal errors generated by the select.

- `thd->is_fatal_error`

  Set this if we should abort the current statement (and any multi-line statements) because something went fatally wrong. (for example, astored procedure should be able to catch this). This is reset by `mysql_reset_thd_for_next_command()`.

- `thd->abort_on_warning`

  Strict mode flag, which means that we should abort the statement if we get a warning. In the `field::store` function this changes the warning level from `WARN` to `ERROR`. In other cases, this flag is mostly tested with `thd->really_abort_on_warning()` to ensure we don't abort in the middle of an update with not transactional tables.

- `thd->count_cuted_fields`

If set, we generate warning for field conversations (normal case for `INSERT`/`UPDATE`/`DELETE`). This is mainly set to 0 when doing internal copying of data between fields and we don't want to generate any conversion errors at any level.

- `thd->killed`

  Set in case of error in connection protocol or in case of 'kill'. In this case we should abort the query and kill the connection.

Error functions

- `thd->really_abort_on_warning()`

  This function returns 1 if a warning should be converted to an error, like in strict mode when all tables are transactional. The conversion is handled in `sql_error.cc::push_warning()`.

- `thd->fatal_error()`

  Should be called if we want to abort the current statement and any multi-line statement.

- `thd->clear_error()`

  Resets `thd->net.report_error` and `thd->query_error`.

4.12. 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 My-Flags));`

  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.

4.13. Bitmaps

Inside the mysys directory is a file named my_bitmap.c. It contains functions for manipulating bitmaps. Specifically there are functions for setup or teardown (bitmap_init, bitmap_free), for setting and clearing individual bits or whole sections of the bitmap (bitmap_set_bit, bitmap_fast_test_and_set, bitmap_clear_all, bitmap_set_all, bitmap_set_prefix, bitmap_set_above), and for performing comparisons and set operations on two bitmaps (bitmap_cmp, bitmap_intersect, bitmap_subtract, bitmap_union). Bitmaps are useful, so the functions are called from several places (opt_range.cc, slave.cc, mysqld.c, sql_insert.cc, log_event.cc, sql_show.cc) and we're expecting to make more use of them in the next version of MySQL, MySQL 5.1.

There are a few warnings and limitations that apply for the present bitmap implementation. First: the allocation is an integral number of bytes, and it is not possible to determine whether the last few bits are meaningful. Second: the whole bitmap might have to be protected by a mutex for manipulations; this is settable by passing appropriate flag values. Third: the bitmap is allocated with a 'uint' size, which means that ordinarily it can't have more than 2^32 bytes. Fourth: when unioning two bitmaps, they must be of the same size.

Chapter 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 `SELECT`s (`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 `SELECT`s. 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 `UNION`s).
- 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` (`subse-`

lect_single_select_engine and subselect_union_engine, both are inherited from subselect_engine).

The engine will be created at the time Item_subselect 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_subselect will never call the cleanup() procedure for JOIN. Every JOIN::cleanup will call cleanup() for inner JOINs. 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 JOIN::exec(), 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 subquery `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?

Chapter 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_in_optimizer|
                        +-----------------+
                                 |
           +---------------------+------------+
           |                                  |
```

```
+-----------------------+             +-----------------+
|   <left_expression>   |             |Item_in_subselect|
|                       |             +-----------------+
+-----------------------+                      |
|<left_expression cache>|          +-----------+-----------+
|                       |          |                       |
+-----------------------+          |                       |
           ^                 +----------+        +--------------------+
           +<<<<<<<<<<<<<<<<<| 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.

Chapter 7. MySQL Client/Server Protocol

7.1. Licensing Notice

The MySQL Protocol is proprietary.

The MySQL Protocol is part of the MySQL Database Management System. As such, it falls under the provisions of the GNU Public License (GPL). A copy of the GNU Public License is available on MySQL's web site, and in the product download.

Because this is a GPL protocol, any product which uses it to connect to a MySQL server, or to emulate a MySQL server, or to interpose between any client and server which uses the protocol, or for any similar purpose, is also bound by the GPL. Therefore if you use this description to write a program, you must release your program as GPL. Contact MySQL AB if you need clarification of these terms or if you need to ask about alternative arrangements.

7.2. Organization

The topic is: the contents of logical packets in MySQL version 5.0 client/server communication.

The description is of logical packets. There will be only passing mention of non-logical considerations, such as physical packets, transport, buffering, and compression. If you are interested in those topics, you may wish to consult another document: "MySQL Client - Server Protocol Documentation" in the file `net_doc.txt` in the `internals` directory of the `mysqldoc` MySQL documentation repository.

The description is of the version-5.0 protocol at the time of writing. Most of the examples show version-4.1 tests, which is okay because the changes from version-4.1 to version-5.0 were small.

A typical description of a packet will include:

"Bytes and Names". This is intended as a quick summary of the lengths and identifiers for every field in the packet, in order of appearance. The "Bytes" column contains the length in bytes. The Names column contains names which are taken from the MySQL source code whenever possible. If the version-4.0 and version-4.1 formats differ significantly, we will show both formats.

Descriptions for each field. This contains text notes about the usage and possible contents.

(If necessary) notes about alternative terms. Naming in this document is not authoritative and you will often see different words used for the same things, in other documents.

(If necessary) references to program or header files in the MySQL source code. An example of such a reference is: sql/protocol.cc net_store_length() which means "in the sql subdirectory, in the protocol.cc file, the function named net_store_length".

An Example. All examples have three columns:

```
-- the field name
-- a hexadecimal dump
-- an ascii dump, if the field has character data
```

All spaces and carriage returns in the hexadecimal dump are there for formatting purposes only.

In the later sections, related to prepared statements, the notes should be considered unreliable and there are no examples.

7.3. Elements

Null-Terminated String: used for some variable-length character strings. The value '\0' (sometimes written 0x00) denotes the end of the string.

Length Coded Binary: a variable-length number. To compute the value of a Length Coded Binary, one must examine the value of its first byte.

```
  Value Of     # Of Bytes  Description
  First Byte   Following
  ----------   ----------- -----------
  0-250        0           = value of first byte
  251          0           column value = NULL
                           only appropriate in a Row Data Packet
  252          2           = value of following 16-bit word
  253          4           = value of following 32-bit word
  254          8           = value of following 64-bit word
```

Thus the length of a Length Coded Binary, including the first byte, will vary from 1 to 9 bytes. The relevant MySQL source program is sql/protocol.cc net_store_length().

All numbers are stored with least significants byte first. All numbers are unsigned.

Length Coded String: a variable-length string. Used instead of Null-Terminated String, especially for character strings which might contain '\0' or might be very long. The first part of a Length Coded String is a Length Coded Binary number (the length); the second part of a Length Coded String is the actual data. An example of a short Length Coded String is these three hexadecimal bytes: 02 61 62, which means "length = 2, contents = 'ab'".

7.4. The Packet Header

```
Bytes                 Name
-----                 ----
3                     Packet Length
1                     Packet Number

Packet Length: The length, in bytes, of the packet
               that follows the Packet Header. There
               may be some special values in the most
               significant byte. Since 2**24 = 16MB,
               the maximum packet length is 16MB.

Packet Number: A serial number which can be used to
               ensure that all packets are present
               and in order. The first packet of a
               client query will
               have Packet Number = 0. Thus, when a
               new SQL statement starts, the packet
               number is re-initialised.
```

The Packet Header will not be shown in the descriptions of packets that follow this section. Think of it as always there. But logically, it "precedes the packet" rather than "is included in the packet".

Alternative terms: Packet Length is also called "packetsize". Packet Number is also called "Packet no".

```
Relevant MySQL Source Code:
include/global.h int3store()
sql/net_serv.cc my_net_write(), net_flush(), net_write_command(), my_net_read()
```

7.5. Packet Types

This is what happens in a typical session:

```
The Handshake (when client connects):
  Server Sends To Client: Handshake Initialisation Packet

  Client Sends To Server: Client Authentication Packet

  Server Sends To Client: OK Packet, or Error Packet

The Commands (for every action the client wants the server to do):

  Client Sends To Server: Command Packet

  Server Sends To Client: OK Packet, or Error Packet, or Result Set Packet
```

In the rest of this chapter, you will find a description for each packet type, in separate sections.

Alternative terms: The Handshake is also called "client login" or "login procedure" or "connecting".

7.6. Handshake Initialization Packet

From server to client during initial handshake.

```
Bytes                        Name
-----                        ----
1                            protocol_version
n (Null-Terminated String)   server_version
4                            thread_id
8                            scramble_buff
1                            (filler) always 0x00
2                            server_capabilities
1                            server_language
2                            server_status
13                           (filler) always 0x00 ...

protocol_version:    The server takes this from PROTOCOL_VERSION
                     in /include/mysql_version.h. Example value = 10.

server_version:      The server takes this from MYSQL_SERVER_VERSION
                     in /include/mysql_version.h. Example value = "4.1.1-alpha".

thread_number:       ID of the server thread for this connection.

scramble_buff:       The password mechanism uses this.
                     (See "Password functions" section elsewhere in this document.)

server_capabilities: CLIENT_XXX options. The possible flag values at time of
writing (taken from  include/mysql_com.h):
 CLIENT_LONG_PASSWORD 1 /* new more secure passwords */
 CLIENT_FOUND_ROWS 2 /* Found instead of affected rows */
 CLIENT_LONG_FLAG 4 /* Get all column flags */
 CLIENT_CONNECT_WITH_DB 8 /* One can specify db on connect */
 CLIENT_NO_SCHEMA 16 /* Don't allow database.table.column */
 CLIENT_COMPRESS         32 /* Can use compression protocol */
 CLIENT_ODBC         64 /* Odbc client */
 CLIENT_LOCAL_FILES 128 /* Can use LOAD DATA LOCAL */
 CLIENT_IGNORE_SPACE 256 /* Ignore spaces before '(' */
 CLIENT_PROTOCOL_41 512 /* New 4.1 protocol */
 CLIENT_INTERACTIVE 1024 /* This is an interactive client */
 CLIENT_SSL              2048 /* Switch to SSL after handshake */
 CLIENT_IGNORE_SIGPIPE   4096    /* IGNORE sigpipes */
 CLIENT_TRANSACTIONS 8192 /* Client knows about transactions */
 CLIENT_RESERVED         16384   /* Old flag for 4.1 protocol  */
 CLIENT_SECURE_CONNECTION 32768  /* New 4.1 authentication */
 CLIENT_MULTI_STATEMENTS 65536   /* Enable/disable multi-stmt support */
 CLIENT_MULTI_RESULTS    131072  /* Enable/disable multi-results */

server_language:     current server character set number

server_status:       SERVER_STATUS_xxx flags: e.g. SERVER_STATUS_AUTOCOMMIT
```

Alternative terms: Handshake Initialization Packet is also called "greeting package". protocolversion is also called "Prot. version". server_version is also called "Server Version String". thread_number is also called "Thread Number". current server charset number is also called "charset_no". scramble_buff is also called "crypt seed". server_status is also called "SERVER_STATUS_xxx flags" or "Server status variables".

```
Example Handshake Initialization Packet
                    Hexadecimal                ASCII
                    -----------                -----
protocol_version    0a                         .
server_version      34 2e 31 2e 31 2d 71 6c    4.1.1-al
                    70 68 61 2d 64 65 62 75    pha-debu
                    67 00                      g.
thread_number       01 00 00 00                ....
scramble_buff       3a 23 3d 4b 43 4a 2e 43    ........
(filler)            00                         .
server_capabilities 2c 82                      ..
server_language     08                         .
server_status       02 00                      ..
(filler)            00 00 00 00 00 00 00 00    ........
                    00 00 00 00 00
```

In the example, the server is telling the client that its server_capabilities include CLIENT_MULTI_RESULTS, CLIENT_SSL, CLIENT_COMPRESS, CLIENT_CONNECT_WITH_DB, CLIENT_FOUND_ROWS.

7.7. Client Authentication Packet

From client to server during initial handshake.

```
VERSION 4.0
Bytes                        Name
-----                        ----
2                            client_flags
3                            max_packet_size
n  (Null-Terminated String)  user
8                            scramble_buff
1                            (filler) always 0x00

VERSION 4.1
Bytes                        Name
-----                        ----
4                            client_flags
4                            max_packet_size
1                            charset_number
23                           (filler) always 0x00...
n  (Null-Terminated String)  user
8                            scramble_buff
1                            (filler) always 0x00
n (Null-Terminated String)   databasename

client_flags:            CLIENT_xxx options. The list of possible flag
                         values is in the description of the Handshake
                         Initialisation Packet, for server_capabilities.
                         For some of the bits, the server passed "what
                         it's capable of". The client leaves some of the
                         bits on, adds others, and passes back to the server.
                         One important flag is: whether compression is desired.

max_packet_size:         the maximum number of bytes in a packet for the client

charset_number:          in the same domain as the server_language field that
                         the server passes in the Handshake Initialisation packet.

user:                    identification

scramble_buff:           the password, after encrypting using the scramble_buff
                         contents passed by the server (see "Password functions"
                         section elsewhere in this document)

databasename:            name of schema to use initially
```

The scramble_buff and databasename fields are optional.

Alternative terms: "Client authentication packet" is sometimes called "client auth response" or "client auth packet". "Scramble_buff" is sometimes called "crypted password".

```
Relevant MySQL Source Code:
- On the client side: libmysql/libmysql.c::mysql_real_connect().
- On the server side: sql/sql_parse.cc::check_connections()
```

```
Example Client Authentication Packet
                    Hexadecimal                ASCII
                    -----------                -----
client_flags        85 a6 03 00                ....
max_packet_size     00 00 00 01                ....
charset_number      08                         .
(filler)            00 00 00 00 00 00 00 00    ........
                    00 00 00 00 00 00 00 00    ........
                    00 00 00 00 00 00 00       .......
user                70 67 75 6c 75 74 7a 61    pgulutza
                    6e 00                      n.
```

7.8. Password functions

The Server Initialization Packet and the Client Authentication Packet both have an 8-byte field, scramble_buff. The value in this field is used for password authentication. It works thus:

```
  The server sends a random string to the client, in scramble_buff.
  The client encrypts the scramble_buff value using the password that the user
  enters. This happens in sql/password.c:scramble() function.
  The client sends the encrypted scramble_buff value to the server.
  The server encrypts the original random string using a value in the mysql
  database, mysql.user.Password.
  The server compares its encryted random string to what the client sent
  in scramble_buff.
  If they are the same, the password is okay.
```

```
Relevant MySQL Source Code:
libmysql/password.c comments at start of file.
```

7.9. Command Packet

From client to server whenever the client wants the server to do something.

```
Bytes                        Name
-----                        ----
1                            command
n                            arg

command:      The most common value is 03 COM_QUERY, because
              INSERT UPDATE DELETE SELECT etc. have this code.
              The possible values at time of writing (taken
              from /include/mysql_com.h for enum_server_command) are:

              #      Name                Associated client function
              -      ----                --------------------------

              0x00   COM_SLEEP           (default, e.g. SHOW PROCESSLIST)
              0x01   COM_QUIT            mysql_close
              0x02   COM_INIT_DB         mysql_select_db
              0x03   COM_QUERY           mysql_real_query
              0x04   COM_FIELD_LIST      mysql_list_fields
              0x05   COM_CREATE_DB       mysql_create_db
              0x06   COM_DROP_DB         mysql_drop_db
              0x07   COM_REFRESH         mysql_refresh
              0x08   COM_SHUTDOWN
              0x09   COM_STATISTICS      mysql_stat
              0x0a   COM_PROCESS_INFO    mysql_list_processes
              0x0b   COM_CONNECT         (during authentication handshake)
              0x0c   COM_PROCESS_KILL    mysql_kill
              0x0d   COM_DEBUG
              0x0e   COM_PING            mysql_ping
              0x0f   COM_TIME            (special value for slow logs?)
              0x10   COM_DELAYED_INSERT
              0x11   COM_CHANGE_USER     mysql_change_user
              0x12   COM_BINLOG_DUMP     (used by slave server / mysqlbinlog)
              0x13   COM_TABLE_DUMP      (used by slave server to get master table)
              0x14   COM_CONNECT_OUT     (used by slave to log connection to master)
              0x15   COM_REGISTER_SLAVE  (reports slave location to master)
              0x16   COM_STMT_PREPARE    see description of Prepare Packet
              0x17   COM_STMT_EXECUTE    see description of Execute Packet
              0x18   COM_STMT_SEND_LONG_DATA       see description of Long Data Packet
              0x19   COM_STMT_CLOSE      new, for closing statement
              0x1a   COM_STMT_RESET
              0x1b   COM_SET_OPTION
              0x1c   COM_STMT_FETCH

arg:           The text of the command is just the way the user typed it, there is no processing
               by the client (except removal of the final ';').
               This field is not a null-terminated string; however,
               the size can be calculated from the packet size,
               and the MySQL client appends '\0' when receiving.
```

```
Relevant MySQL source code:
sql-common/client.c cli_advanced_command(), mysql_send_query().
libmysql/libmysql.c mysql_real_query(), simple_command(), net_field_length().
```

```
Example Command Packet
                    Hexadecimal                ASCII
                    -----------                -----
command             02                         .
arg                 74 65 73 74                test
```

In the example, the value 02 in the command field stands for COM_INIT_DB. This is the packet that the client puts together for "use test;".

7.10. Types Of Result Packets

A "result packet" is a packet that goes from the server to the client in response to a Client Authentication Packet or Command Packet. To distinguish between the types of result packets, a client must look at the first byte in the packet. We will call this byte "field_count" in the description of each individual package, although it goes by several names.

```
Type Of Result Packet       Hexadecimal Value Of First Byte (field_count)
---------------------       ---------------------------------------------

OK Packet                   00
Error Packet                ff
Result Set Packet           1-250 (first byte of Length-Coded Binary)
Field Packet                1-250 ("")
Row Data Packet             1-250 ("")
EOF Packet                  fe
```

7.11. OK Packet

From server to client in response to command, if no error and no result set.

```
VERSION 4.0
Bytes                       Name
-----                       ----
1   (Length Coded Binary)   field_count, always = 0
1-9 (Length Coded Binary)   affected_rows
1-9 (Length Coded Binary)   insert_id
2                           server_status
n   (Length Coded String)   message

VERSION 4.1
Bytes                       Name
-----                       ----
1   (Length Coded Binary)   field_count, always = 0
1-9 (Length Coded Binary)   affected_rows
```

```
1-9 (Length Coded Binary)   insert_id
2                           server_status
2                           warning_count
n   (Length Coded String)   message

field_count:     always = 0

affected_rows:   = number of rows affected by INSERT/UPDATE/DELETE

insert_id:       This will have the "last INSERT id", that is, the value
                 that an auto_increment column received in the last INSERT.
                 = 0 if "last INSERT id" was not changed by the command.

server_status:   = 0, usually. The client can use this to check if the
                 command was inside a transaction.

warning_count:   number of warnings

message:         For example, after a multi-line INSERT, message might be
                 "Records: 3 Duplicates: 0 Warnings: 0"
```

The message field is optional.

Alternative terms: OK Packet is also known as "okay packet" or "ok packet" or "OK-Packet". field_count is also known as "number of rows" or "marker for ok packet". message is also known as "Messagetext". OK Packets (and result set packets) are also called "Result packets".

```
Relevant files in MySQL source:
(client) sql/client.c mysql_read_query_result()
(server) sql/protocol.cc send_ok()
```

```
Example OK Packet
                    Hexadecimal                ASCII
                    -----------                -----
field_count         00                         .
affected_rows       01                         .
insert_id           00                         .
server_status       02 00                      ..
warning_count       00 00                      ..
```

In the example, the optional message field is missing (the client can determine this by examining the packet length). This is a packet that the server returns after a successful INSERT of a single row that contains no auto_increment columns.

7.12. Error Packet

From server to client in response to command, if error.

```
VERSION 4.0
Bytes                       Name
-----                       ----
1                           field_count, always = 0xff
2                           errno
n                           message

VERSION 4.1
Bytes                       Name
-----                       ----
1                           field_count, always = 0xff
2                           errno
1                           (sqlstate marker), always '#'
5                           sqlstate
n                           message

field_count:       Always 0xff (255 decimal).

errno:             The possible values are listed in the manual, and in
                   the MySQL source code file /include/mysqld_error.h.

(sqlstate marker): This is always '#'. It is necessary for distinguishing
                   version-4.1 messages.
```

```
sqlstate:          The server translates errno values to sqlstate values
                   with a function named mysql_errno_to_sqlstate(). The
                   possible values are listed in the manual, and in the
                   MySQL source code file file /include/sql_state.h.

message:           The error message is a string which ends at the end of
                   the packet, that is, its length can be determined from
                   the packet header. The MySQL client (in the my_net_read()
                   function) always adds '\0' to a packet, so the message
                   may appear to be a Null-Terminated String.
                   Expect the message to be between 0 and 512 bytes long.
```

Alternative terms: field_count is also known as "Status code" or "Error Packet marker". errno is also known as "Error Number" or "Error Code".

Relevant files in MySQL source: (client) client.c net_safe_read() (server) sql/protocol.cc send_error()

```
Example of Error Packet
                    Hexadecimal                ASCII
                    -----------                -----
field_count         ff                         .
errno               1b 04                      ..
(sqlstate marker)   23                         #
sqlstate            34 32 53 30 32             42S02
message             55 63 6b 6e 6f 77 6e 20    Unknown
                    74 61 62 6c 6c 65 20 27    table '
                    71 27                      q'
```

7.13. Result Set Header Packet

From server to client after command, if no error and result set -- that is, if the command was a query which returned a result set.

The Result Set Header Packet is the first of several, possibly many, packets that the server sends for result sets. The order of packets for a result set is:

```
  (Result Set Header Packet)  the number of columns
  (Field Packets)             column descriptors
  (EOF Packet)                marker: end of Field Packets
  (Row Data Packets)          row contents
  (End Packet)                marker: end of Data Packets
```

```
Bytes                        Name
-----                        ----
1-9   (Length-Coded-Binary)  field_count
1-9   (Length-Coded-Binary)  extra

field_count: See the section "Types Of Result Packets"
             to see how one can distinguish the
             first byte of field_count from the first
             byte of an OK Packet, or other packet types.

extra:       For example, SHOW COLUMNS uses this to send
             the number of rows in the table.
```

The "extra" field is optional and never appears for ordinary result sets.

Alternative terms: a Result Set Packet is also called "a result packet for a command returning rows" or "a field description packet".

```
Relevant MySQL source code:
libmysql/libmysql.c (client):
  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.
See also my_net_write() which describes local data loading.
```

```
Example of Result Set Header Packet
                    Hexadecimal                ASCII
                    -----------                -----
field_count         03                         .
```

In the example, we se what the packet would contain after "SELECT * FROM t7" if table t7 has 3 columns.

7.14. Field Packet

From Server To Client, part of Result Set Packets. One for each column in the result set. Thus, if the value of field_columns in the Result Set Header Packet is 3, then the Field Packet occurs 3 times.

```
VERSION 4.0
Bytes                      Name
-----                      ----
n (Length Coded String)    table
n (Length Coded String)    name
4 (Length Coded Binary)    length
2 (Length Coded Binary)    type
2 (Length Coded Binary)    flags
1                          decimals
n (Length Coded Binary)    default

VERSION 4.1
Bytes                      Name
-----                      ----
n (Length Coded String)    catalog
n (Length Coded String)    db
n (Length Coded String)    table
n (Length Coded String)    org_table
n (Length Coded String)    name
n (Length Coded String)    org_name
1                          (filler)
2                          charsetnr
4                          length
1                          type
2                          flags
1                          decimals
2                          (filler), always 0x00
n (Length Coded Binary)    default
```

In practice, since identifiers are almost always 250 bytes or shorter, the Length Coded Strings look like: (1 byte for length of data) (data)

```
catalog:                 Catalog, for version 5.0 use. The value now is "std".
db:                      Database identifier, also known as schema name.
table:                   Table identifier, after AS clause (if any).
org_table:               Original table identifier, before AS clause (if any).
name:                    Column identifier, after AS clause (if any).
org_name:                Column identifier, before AS clause (if any).
charsetnr:               Character set number.
length:                  Length of column, according to the definition.
                         Also known as "display length". The value given
                         here may be larger than the actual length, for
                         example an instance of a VARCHAR(2) column may
                         have only 1 character in it.
type:                    The code for the column's data type. Also known as
                         "enum_field_type". The possible values at time of
                         writing (taken from  include/mysql_com.h), in hexadecimal:
                         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
                         0x0f   FIELD_TYPE_VARCHAR (new in MySQL 5.0)
                         0x10   FIELD_TYPE_BIT (new in MySQL 5.0)
                         0xf6   FIELD_TYPE_NEWDECIMAL (new in MYSQL 5.0)
                         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

flags:                   The possible flag values at time of
                         writing (taken from  include/mysql_com.h), in hexadecimal:
                         0001 NOT_NULL_FLAG
                         0002 PRI_KEY_FLAG
                         0004 UNIQUE_KEY_FLAG
                         0008 MULTIPLE_KEY_FLAG
                         0010 BLOB_FLAG
                         0020 UNSIGNED_FLAG
                         0040 ZEROFILL_FLAG
                         0080 BINARY_FLAG
                         0100 ENUM_FLAG
                         0200 AUTO_INCREMENT_FLAG
                         0400 TIMESTAMP_FLAG
                         0800 SET_FLAG

decimals:                The number of positions after the decimal
                         point if the type is DECIMAL or NUMERIC.
                         Also known as "scale".
default:                 For table definitions. Doesn't occur for
                         normal result sets. See mysql_list_fields().
```

Alternative Terms: Field Packets are also called "Header Info Packets" or "field descriptor packets" (that's a better term but it's rarely used). In non-MySQL contexts Field Packets are more commonly known as "Result Set Metadata".

```
Relevant MySQL source code:
(client) client/client.c unpack_fields().
(server) sql/sql_base.cc send_fields().
```

```
Example of Field Packet
                    Hexadecimal                ASCII
                    -----------                -----
catalog             03 73 74 64                .std
db                  03 64 62 31                .db1
table               02 54 37                   .T7
org_table           02 74 37                   .t7
name                02 53 31                   .S1
org_name            02 73 31                   .s1
(filler)            0c                         .
charsetnr           08 00                      ..
length              01 00 00 00                ....
type                fe                         .
flags               00 00                      ..
decimals            00                         .
(filler)            00 00                      ..
```

In the example, we see what the server returns for "SELECT s1 AS S1 FROM t7 AS T7" where column s1 is defined as CHAR(1).

7.15. EOF Packet

From Server To Client, at the end of a series of Field Packets, and at the end of a series of Data Packets. With prepared statements, EOF Packet can also end parameter information, which we'll describe later.

```
VERSION 4.0
Bytes                 Name
```

```
-----                 ----
1                     field_count, always = 0xfe

VERSION 4.1
Bytes                 Name
-----                 ----
1                     field_count, always = 0xfe
2                     warning_count
2                     Status Flags

field_count:          The value is always 0xfe (decimal 254).
                      However ... recall (from the
                      section "Elements", above) that the value 254 can begin
                      a Length-Encoded-Binary value which contains an 8-byte
                      integer. So, to ensure that a packet is really an EOF
                      Packet: (a) check that first byte in packet = 0xfe, (b)
                      check that size of packet < 9.

warning_count:        Number of warnings. Sent after all data has been sent
                      to the client.

server_status:        Contains flags like SERVER_STATUS_MORE_RESULTS.
```

Alternative terms: EOF Packet is also known as "Last Data Packet" or "End Packet".

```
Relevant MySQL source code:
(server) protocol.cc send_eof()
```

```
Example of EOF Packet
                    Hexadecimal                ASCII
                    -----------                -----
field_count         fe                         .
warning_count       00 00                      ..
server_status       00 00                      ..
```

7.16. Row Data Packet

From server to client. One packet for each row in the result set.

```
Bytes                   Name
-----                   ----
n (Length Coded String) (column value)
...

(column value):       The data in the column, as a character string.
                      If a column is defined as non-character, the
                      server converts the value into a character
                      before sending it. Since the value is a Length
                      Coded String, a NULL can be represented with a
                      single byte containing 251(see the description
                      of Length Coded Strings in section "Elements" above).
```

The (column value) fields occur multiple times. All (column value) fields are in one packet. There is no space between each (column value).

Alternative Terms: Row Data Packets are also called "Row Packets" or "Data Packets".

```
Relevant MySQL source code:
(client) client/client.c read_rows
```

```
Example of Row Data Packet
                    Hexadecimal                ASCII
                    -----------                -----
(first column)      01 58                      .X
(second column)     02 35 35                   .55
```

In the example, we see what the packet contains after a SELECT from a table defined as "(s1 CHAR, s2

INTEGER)" and containing one row where s1='X' and s2=55.

7.17. Row Data Packet: Binary (Tentative Description)

From server to client, or from client to server (if the client has a prepared statement, the "result set" packet format is used for transferring parameter descriptors and parameter data).

Recall that in the description of Row Data we said that: "If a column is defined as non-character, the server converts the value into a character before sending it." That doesn't have to be true. If it isn't true, it's a Row Data Packet: Binary.

```
Bytes                   Name
-----                   ----
1                       Null Bit Map with first two bits = 01
n (Length Coded String) (column value)
...

Bytes                   Name
-----                   ----
?                       Packet Header
1                       Null Bit Map with first two bits = 01

Null Bit Map: The most significant 2 bits are reserved. Since
              there is always one bit on and one bit off, this can't be
              confused with the first byte of an Error Packet (255), the
              first byte of a Last Data Packet (254), or the first byte of
              an OK Packet (0).

(column value): The column order and organization are the same as for
                conventional Row Data Packets. The difference is that
                each column value is sent just as it is stored. It's now up
                to the client to convert numbers to strings if that's desirable.
                For a description of column storage, see "Physical Attributes Of
                Columns" elsewhere in this document.
```

Only non-zero parameters are passed.

Because no conversion takes place, fixed-length data items are as described in the "Physical Attributes of Columns" section: one byte for TINYINT, two bytes for FLOAT, four bytes for FLOAT, etc. Strings will appear as packed-string-length plus string value. DATETIME, DATE and TIME will be 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 + 0-11    Length + sign (0 = pos, 1= neg), 4 byte days,
                             1 byte HHMMDD, 4 byte billionth of a second
```

Alternative Terms: Row Data Packet: Binary is also called "Binary result set packet".

Except for the different way of signalling NULLs, the server/client parameter interaction here proceeds the say way that the server sends result set data to the client. Since the data is not sent as a string, the length and meaning depend on the data type. The client must make appropriate conversions given its knowledge of the data type.

7.18. Prepared Statement Initialization Packet (Tentative Description)

From server to client, when a statement is being sent with the COM_PREPARE command.

```
Bytes               Name
-----               ----
```

```
1                    field_count
4                    statement_handler_id
2                    columns
2                    parameters

field_count:          Always = 0, as with OK Packet.

statement_handler_id: ID of statement handler.

columns:              Number of columns in result set.

parameters:           Number of parameters in query.
```

Alternative terms: statement_handler_id is called "statement handle" or "hstmt" everywhere but at MySQL. Prepraed statement initialization packet is also called "prepared statement init packet".

7.19. OK for Prepared Statement Initialization Packet (Tentative Description)

From server to client, in response to prepared statement initialization packet.

```
Bytes               Name
-----               ----
1                   0 - marker for OK packet
4                   statement_handler_id
2                   number of columns in result set
2                   number of parameters in query
```

7.20. Parameter Packet (Tentative Description)

From server to client, for prepared statements which contain parameters.

The Parameter Packets follow a Prepared Statement Initialization Packet which has a positive value in the parameters field.

```
Bytes                    Name
-----                    ----
2                        type
2                        flags
1                        decimals
4                        length

type:                 Same as for type field in a Field Packet.

flags:                Same as for flags field in a Field Packet.

decimals:             Same as for decimals field in a Field Packet.

length:               Same as for length field in a Field Packet.
```

Notice the similarity to a Field Packet.

The parameter data will be sent in a packet with the same format as Row Data Packet: Binary.

7.21. Long Data Packet (Tentative Description)

From client to server, for long parameter values.

```
Bytes                    Name
-----                    ----
4                        statement_handler_id
2                        parameter_number
2                        type
n                        data
```

```
statement_handler_id:     ID of statement handler

parameter_number:         Parameter number.

type:                     Parameter data type. Not used at time of writing.

data:                     Value of parameter, as a string.
                          The length of the data can be computed from
                          the packet length
```

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 one big string.

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

The server will not send an @code{ok} or @code{error} packet in response to this. If there is an error (for example the string is too big), one will see the error when calling "execute".

```
Relevant MySQL Source Code:
(server) mysql_send_long_data
```

7.22. Execute Packet (Tentative Description)

From client to server, to execute a prepared statement.

```
Bytes                Name
-----                ----
1                    code
4                    statement_id
1                    flags
4                    iteration_count
(param_count+7)/8    null_bit_map
1                    new_parameter_bound_flag
n                    type

code:          always COM_EXECUTE

statement_id:  statemeent identifier

flags:         reserved for future use.
               in MySQL 4.0, always 0.
               used in MySQL 5.0 for "open cursor".

iteration_count: reserved for future use. in MySQL 4.1, always 1.

null_bit_map:  A map of bits which are null / not null, one for
               each parameter. The first two bits are reserved so as to
               avoid confusion with other packet types. For example, if
               the first parameter (parameter 0) is NULL, then
               the least significant bit in null_bit_map will be 1.

new_parameter_bound_flag:   Contains 1 if this is the first time
                            that "execute" has been called, or if
                            the parameter has been rebound.

type:                       Occurs once for each parameter that
                            is described as NOT NULL. The high
                            15 bits of the low-byte-first number
                            are the type, then there is one bit which
                            is set to on for "unsigned". This is not
                            used with mysql_send_long_data().
```

The Execute Packet is also known as "COM_EXECUTE Packet".

In response to an Execute Packet, the server should send back one of: an OK Packet, an Error Packet, or a series of Result Set Packets in which all the Row Data Packets are binary.

Relevant MySQL Source Code: libmysql/libmysql.c cli_read_prepare_result()

7.23. Compression

This chapter does not discuss compression, but you should be aware of its existence.

Compression is of one or more logical packets. The packet_number field that is in each packet header is an aid for keeping track.

The opposite of "compressed" is "raw".

Compression is used if both client and server support zlib compression, and the client requests compression.

A compressed packet header is: packet length (3 bytes), packet number (1 byte), and Uncompressed Packet Length (3 bytes). The Uncompressed Packet Length is the number of bytes in the original, uncompressed packet. If this is zero then the data is not compressed.

When the compressed protocol is in use (that is, when the client has requested it by setting the flag bit in the Client Authentication Packet and the server has accepted it), either the client or the server may compress packets. However, compression will not occur if the compressed length is greater than the original length. Thus, some packets will be compressed while other packets are not compressed.

Chapter 8. Prepared Statements and Stored Routines

Let us start with a general description of the MySQL statement processing workflow in order to provide the reader with understanding of the problem of reexecution and vocabulary for the following sections.

Conventional statements, that is, SQL queries sent in `COM_QUERY` protocol packet, are the only statements present in MySQL server prior to version 4.1. Execution of such statements is performed in a batch mode, one query processed by the server at a time. The original implementation is streamlined for this mode and has a single global connection state `THD` shared among all operational steps.

When executing a query in conventional mode, the server sequentially parses its text, acquires table level locks, analyzes the parsed tree, builds an execution plan, executes the built plan and releases the locks.

Memory for parsing is allocated using block allocator `MEM_ROOT` in 4k chunks and freed once in the end of execution. Memory for execution is allocated in the memory root of the parsed tree, as well as in the system heap, and in some cases in local "memory roots" of execution modules.

The role of the parser is to create a set of objects to represent the query. E.g. for a `SELECT` statement, this set includes a list of Item's for `SELECT` list, a list of tables (`TABLE_LIST` object for each table) for `FROM` clause, and a tree of Item's for `WHERE` clause.

During context analysis phase, links are established from the parsed tree to the physical objects of the database, such as open tables and table columns. A physical table is represented by a heir of class handler that corresponds to the storage engine the table belongs to, and is saved in `TABLE_LIST::file`.

When context analysis is done, the query optimizer is run. It performs two major tasks:

- Query transformation — a transformation of the parsed tree to an equivalent one, which is simpler and more efficient to execute.
- Creation of an execution plan, including evaluation of an order of joins and initialization of methods to access the used tables. At this step parts of the execution plan are attached to the parsed tree.

Finally, the query is passed to the execution runtime — an interpreter that operates with and modifies both the parsed tree and the execution plan in order to execute the query.

It should be noted that the overall procedure is infamous for breaking borders between abstraction layers. For example, MySQL performs [sub]query transformation during context analysis; moreover, most parts of the code rely on the fact that `THD` is processing only one statement at a time.

8.1. Statement Re-execution Requirements

Features of MySQL 4.1 and 5.0 put a new demand on the execution process: prepared statements and stored routines need to reuse the same parsed tree to execute a query many times.

So far no easy mechanism that would allow query reexecution using the conventional query processing code has been found. For instance, copying of the parsed tree before each reexecution is not simple to implement as a parsed tree, which can contain instances of more than 300 different classes, has a lot of cross-references between its objects.

The present solution introduces a concept of change management for the changes of the parsed tree and is largely a unification of numerous fixes of bugs in reexecution. The solution has two aspects.

The first one is that modifications of the parsed tree are tracked and a way to restore the tree to a state that allows reexecution is introduced.

The second aspect is that a dedicated block allocator (memory root) is used to store the parsed tree, and the memory allocated in this memory root is freed only when the parsed tree is destroyed. Later this memory root will be denoted as the permanent memory root of a statement.

In order to properly restore the parsed tree to a usable state, all modifications of the tree are classified as destructive or non-destructive and an appropriate action is taken for every type of modification.

A non-destructive modification does not depend on actual values of prepared statement placeholders or contents of the tables used in a query. Such modification is [and should be, for future changes] made only once and the memory for it is allocated in the permanent memory root of the statement.

As a result, the modified parsed tree remains usable.

Examples of non-destructive and useful modifications of the parsed tree are:

- `WHERE`/`ON` clause flattening
- `NOT` elimination
- `LEFT JOIN` elimination, when it can be done based on the constants explicitly specified in the query

The rest of modifications are destructive, generally because they are based on actual contents of tables or placeholders.

Examples of destructive modifications are:

- Equality propagation
- Sorting of members of `IN` array for quick evaluation of `IN` expression.

Destructive modifications are (and should be for all future changes) allocated in a memory root dedicated to execution, are registered in `THD::change_list` and rolled back in the end of each execution. Later the memory root dedicated to execution of a statement will be denoted as the runtime memory root of the statement. Because allocations are done indirectly via `THD::mem_root`, `THD::mem_root` at any given moment of time can point either to the permanent or to the runtime memory root of the statement. Consequently, `THD::mem_root` and `THD::free_list` can be denoted as 'currently active arena' of THD.

8.2. Preparation of a Prepared Statement

As mentioned above, `THD` is currently a required argument and the runtime context for every function in the server. Therefore, in order to call the parser and allocate memory in the statement memory root we perform several save-restore steps with `THD::mem_root` and `THD::free_list` (the active arena of `THD`).

1. In order to parse a statement, we save the currently active arena of `THD` and assign its members from the permanent arena of the statement. This is achieved by calling `THD::set_and_backup_active_arena`. This way `alloc_query` and `yyparse` operate on the permanent arena.

2. We don't want the garbage which is created during statement validation to be left in the permanent arena of the statement. For that, after parse but before validation of the statement, we restore the THD arena saved in (1). In other words, we use the arena of `THD` that was active when `Prepared_statement::prepare` was invoked as the runtime arena of the statement when it is validated.

3. Statement validation is performed in function `check_prepared_statement()`. This function will subsequently call `st_select_lex_unit::prepare()` and `setup_fields()` for the main LEX unit, create `JOINs` for every unit, and call `JOIN::prepare` for every join (`JOINs` in MySQL represents a part of the execution plan). Our prepared statement engine does not save the execution plan in a prepared statement for reuse, and ideally we should not create it at prepare stage. However, currently there is no other way to validate a statement except to call `JOIN::prepare` for all its units.

4. During validation we may perform a transformation of the parsed tree. In a clean implementation this would belong to a separate step, but in our case the majority of the server runtime was not refactored to support reexecution of statements, and a permanent transformation of the parsed tree can happen at any moment during validation. Such transformations **absolutely must** use the permanent arena of the prepared statement. To make this arena accessible, we save a pointer to it in `thd->stmt_arena` before calling `check_prepared_statement`.

   Later, whenever we need to perform a permanent transformation, we first call `THD::activate_stmt_arena_if_needed` to make the permanent arena active, transform the tree, and restore the runtime arena.

5. Some parts of the execution do not distinguish between preparation of a prepared statement and its execution and perform destructive optimizations of the parsed tree even during validation. These changes of the parsed tree are recorded in `THD::change_list using` method `THD::register_item_tree_change`.

6. After the validation is done, we rollback the changes registered in `THD::change_list` and free new items and other memory allocated by destructive transformations.

8.3. Execution of a Prepared Statement

In order to call mysql_execute_command (the function that executes a statement) for a prepared statement and not damage its parse tree, we backup and restore the active Query_arena of THD.

- We don't want the garbage created during execution to be left in the permanent arena of the statement. To ensure that, every statement is executed in the runtime arena of `THD`. In other words, the arena which was active when `mysql_stmt_execute` was called is used as the runtime arena of the statement during its execution.

- Before calling `mysql_stmt_execute`, we `allocate thd->query` with parameter markers ('?') replaced with their values: the new query is allocated in the runtime arena. We'll need this query for general, binary, error and slow logs.

- The execution plan created at prepare stage is not saved (see Section 8.2, “Preparation of a Prepared Statement”), and at execute we simply create a new set of JOINs and then prepare and optimize it. During the first execution of the prepared statement the server may perform non-destructive transformations of statement's parsed tree: normally that would belong to a separate step executed at statement prepare, but once again, this haven't been done in 4.1 or 5.0. Such transformations **absolutely must** use the permanent arena of the prepared statement (saved in `thd->stmt_arena`). Whenever we need to perform a permanent transformation, we first call `THD::activate_stmt_arena_if_needed` to make the permanent arena active, transform the tree, and restore the runtime arena. To avoid double transformations in such cases, we track current state of the parsed tree in `Query_arena::state`.

This state may be one of the following:

- `INITIALIZED` — we're in statement `PREPARE`.
- `INITIALIZED_FOR_SP` — we're in first execution of a stored procedure statement.
- `PREPARED` — we're in first execution of a prepared statement.
- `EXECUTED` — we're in a subsequent execution of a prepared statement or a stored procedure statement.
- `CONVENTIONAL_EXECUTION` — we're executing a pre-4.1 query.

One can use helper methods of `Query_arena` to check this state (`is_conventional_execution()`, `is_stmt_prepare()`, `is_stmt_execute()`, `is_stmt_prepare_or_first_sp_execute()`).

Additionally, `st_select_lex_unit::first_execution` contains a flag for the state of each subquery in a complex statement. A separate variable is needed because not all subqueries may get executed during the first execution of a statement.

- Some optimizations damage the parsed tree, e.g. replace leafs and subtrees of items with other items or leave item objects cluttered with runtime data. To allow re-execution of a prepared statement the following mechanisms are currently employed:

  1. A hierarchy of `Item::cleanup()` and `st_select_lex::cleanup()` methods to restore the parsed tree to the condition of right-after-parse. These cleanups are called in `Prepared_statement::cleanup_stmt()` after the statement has been executed.

  2. In order to roll back destructive transformations of the parsed tree, every replacement of one item with another is registered in `THD::change_list` by using `THD::change_item_tree()`. In the end of execution all such changes are rolled back in reverse order.

     Example:

     ```
     if (!(fld= new Item_field(from_field)))
     goto error;
     thd->change_item_tree(reference, fld);
     ```

     If a transformation is a non-destructive, it should not be registered, but performed only once in the permanent memory root. Additionally, be careful to not supply a pointer to stack as the first argument of `change_item_tree()`; that will lead to stack corruption when a tree is restored.

  3. `AND`/`OR` subtrees of `WHERE` and `ON` clauses are created anew for each execution. It was easier to implement in 4.1, and the approach with change record list used in (b) could not have been used for `AND`/`OR` transformations, because these transformations not only replace one item with another, but also can remove a complete subtree. Leafs of `AND`/`OR` subtrees are not copied by this mechanism because currently they are not damaged by the transformation. For details, see `Item::copy_andor_structure()`.

  4. **No** other mechanism exists in the server at the moment to allow re-execution. If the code that you're adding transforms the parsed tree, you must use one of the mechanisms described above, or propose and implement a better approach.

- When execution is done, we rollback the damage of the parsed tree.

8.4. Execution of a Stored Procedure Statement

Execution of a stored procedure statement is similar to execution of a prepared statement. The few existing exceptions are described below.

During execution of a stored procedure, `THD::stmt_arena` points to the permanent query arena of the stored procedure. This arena happens to be also the permanent query arena of every instruction of the procedure, as the parser creates all instructions in the same arena. More generally, `THD::stmt_arena` is always set and always points to the permanent arena of a statement. If the statement is a conventional query, then the permanent arena simply points to the runtime arena of the query.

An own runtime memory root is set up for execution of every stored procedure statement and freed in the end of execution. This is a necessary measure to avoid memory leaks if a stored procedure statement is executed in a loop.

With regard to the transformations and restoration of the parsed tree, execution of a stored procedure statement follows the path of execution of a prepared statement, with the exception that there is no separate prepare step. `THD::is_first_sp_execute()` is used to determine whether it's the first execution, and in this case non-destructive permanent transformations of the parsed tree are made in the permanent memory root of the statement that is currently being executed.

During subsequent executions no non-destructive transformations are performed, while all destructive ones are rolled back in the end of execution using the same algorithm as in prepared statements.

Chapter 9. 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.

9.1. 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.

9.2. 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-debug-log created 03
# at 79
#030710 21:55:59 server id 1  log_pos 79        Query   thread_id=2     exec_time=16    error_code=0
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     error_code=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     error_code=0
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     error_code=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     error_code=0
SET TIMESTAMP=1057867044;
create table u(a int primary key, b int, key(b), foreign key (b) references u(a)) type=innodb;
# at 470
```

```
#030710 21:57:52 server id 1  log_pos 470       Query   thread_id=2     exec_time=0     error_code=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     error_code=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     error_code=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     error_code=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     error_code=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     error_code=0
SET TIMESTAMP=1057867164;
create table u(a int primary key, b int, key(b), foreign key (b) references v(c)) type=innodb;
# at 896
#030710 21:59:47 server id 1  log_pos 896       Query   thread_id=8     exec_time=0     error_code=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     error_code=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     error_code=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     error_code=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     error_code=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     error_code=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     error_code=0
SET TIMESTAMP=1057867232;
create table u(a int primary key, b int, key(b), foreign key (b) references v(c)) type=innodb;
# at 1602
#030710 22:00:44 server id 1  log_pos 1602      Query   thread_id=16    exec_time=0     error_code=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     error_code=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 9.3.1, “The Slave I/O Thread”.

- `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 9.7, “Replication Event Format in Detail”.

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.

9.3. 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)
{ ... }
```

9.3.1. 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

9.3.2. 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.

9.3.3. 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.

9.3.4. 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`).

9.4. How Replication Deals With...

This section describes how replication handles various problematic issues.

9.4.1. `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`.

9.4.2. 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`.

9.4.3. System Variables

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

9.4.4. 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).

9.4.5. 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.

9.4.6. 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.

9.4.7. 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.

9.4.8. `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.

9.5. 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.

9.6. 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, *MySQL Client/Server Protocol*. 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.

9.7. Replication Event Format in Detail

9.7.1. 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.

9.7.2. 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.

9.8. Plans

We have already made extensive changes to the above in MySQL 5.0. For an upcoming version we plan a new format where data is replicated by row instead of by statement. This makes it possible to replicate data from one MySQL Cluster to another. We also plan new functionality to handle replication connections from multiple masters into one slave.

Chapter 10. MyISAM Storage Engine

10.1. MyISAM Record Structure

10.1.1. 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.

10.1.2. 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'`)

  - In MySQL 4.1 the length is always 1 byte. In MySQL 5.0 the length may be either 1 byte (for up to 255) or 2 bytes (for 256 to 65535). Some further random notes about the new format: In old tables (from MySQL 4.1 and earlier), VARCHAR columns have type MYSQL_TYPE_VAR_STRING, which works exactly like a CHAR with the exception that if you do an `ALTER TABLE`, it's converted to a true VARCHAR (MYSQL_TYPE_VARCHAR). (This means that old tables will work as before for users.) ... Apart from the above case, there are no longer any automatic changes from CHAR to VARCHAR or from VARCHAR to CHAR. MySQL will remember the declared type and stick to it ... VARCHAR is implemented in `field.h` and `field.cc` through the new class `Field_varstring` ... MyISAM implements VARCHAR both for dynamic-length and fixed-length rows (as signaled with the ROW_FORMAT flag) ... VARCHAR now stores trailing spaces. (If they don't fit, that's an error in strict mode.) Trailing spaces are not significant in comparisons ... In `table->record`, the space is reserved for length (1 or 2 bytes) plus data ... The number of bytes used to store the length is in the field `Field_varchar->length_bytes`. Note that internally this can be 2 even if `Field_varchar->field_length` < 256 (for example, for a shortened key to a `varchar(256)`) ... There is a new macro, `HA_VARCHAR_PACKLENGTH(field_length)`, that can be used on `field->length` in write_row / read_row to check how many length bytes are used. (In this context we can't have a field_length < 256 with a 2-byte pack length) ... When creating a key for the handler, HA_KEYTYPE_VARTEXT1 and HA_KEYTYPE_BINARY1 are used for a key on a column that has a 1-byte length prefix and HA_KEYTYPE_VARTEXT2 and HA_KEYTYPE_BINARY2 for a column that has a 2-byte length prefix. (In the future we will probably delete HA_KEYTYPE_BINARY#, as this can be instead be done by just using the `binary` character set with HA_KEYTYPE_VARTEXT#.) ... When sending a key to the handler for `index_read()` or records_in_range, we always use a 2-byte length for the VARCHAR to make things simpler. (For version 5.1 we intend to change CHARs to also use a 2-byte length for these functions, as this will speed up and simplify the key handling code on the handler side.) ... The test case file `mysql-test/include/varchar.inc` should be included in the code that tests the handler. See `t/myisam.test` for how to use this. You should verify the result against the one in `mysql-test/t/myisam.result` to ensure that you get the correct results ... A client sees both the old and new `VARCHAR` type as `MYSQL_TYPE_VAR_STRING`. It will never (at least for 5.0) see `MYSQL_TYPE_VARCHAR`. This ensures that old clients will work as before ... If you run MySQL 5.0 with the `--new` option, MySQL will show old `VARCHAR` columns as `'CHAR'` in `SHOW CREATE TABLE`. (This is useful when testing whether a table is using the new `VARCHAR` type or not.)

- **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`

- MySQL 4.1 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'`)
- MySQL 5.0 Storage: high byte first, four-byte chunks. We call the four-byte chunks "*decimal* digits". Since 2**32 = 4294967296, one *decimal* digit can hold values up to 10**9 (999999999), and two *decimal* digits can hold values up to 10**18, and so on. There is an implied decimal point. Details are in /strings/decimal.c.
- Example: a MySQL 5.0 `DECIMAL(21,9)` column conaining `111222333444.555666777` looks like: `hexadecimal 80 6f 0d 40 8a 04 21 1e cd 59` -- (flag + '111', '222333444', '555666777').

`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.

10.1.3. 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.

10.2. 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 `my-isamdef.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 fields)+
                                                           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.

10.2.1. `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`.)

10.3. `MyISAM` Compressed Data File Layout

This chapter describes the layout for the data file of compressed `MyISAM` tables.

10.3.1. Huffman compression

`MyISAM` compression is based on Huffman compression. In his article from 1952 Huffman proved that his algorithm uses the least possible number of bits to encode a sequence of messages. The number of bits assigned to each message depends on its probability to appear in the sequence.

Huffman did not specify exactly, what those "messages" are. One could take all possible values - say of a table column - as "messages". But if there are too many of them, the code tables could become bigger than the uncompressed table. One would need to specify every possible value once and the code tree with its indexes and offsets. Not to forget the effort to step through big binary trees for every value and - on the encoding side - the comparison of each value against the already collected distinct values.

The usual way to define "Huffman messages" is to take the possible 256 values, which a byte can express, as the "messages". That way the code trees are of limited size. On the other hand, the theoretical maximum compression is 1:8 (12.5%) in this case.

10.3.2. The myisampack Program

myisampack tries both ways to compress the column values. When starting to analyze the existing uncompressed data, it collects distinct column values up to a limit of 8KB. If there are more, it falls back to byte value compression for this column.

This means also that myisampack may use different algorithms for different columns. Besides a couple of other tricks, myisampack determines for every column if distinct column value compression or byte value compression is better. After that it tries to combine byte value compression trees of different columns into one or more code trees. This means that finally we may have less code trees than columns. Therefore the column information in the file header contains the number of the code tree used for each column. Some columns might not need a code tree at all. This happens for columns which have the same value in all records.

10.3.3. Record and Blob Length Encoding

Since the compressed data file should be usable for read-only purposes by the MySQL database management system, every record starts on a byte boundary. Fore easier handling by the system, every record begins with a length information for the compressed record and a length information for the total size of all uncompressed blobs of this record. Both lengths are encoded in 1 to 5 bytes, depending on its value.

A length from 1 to 253 bytes is represented in one byte. A length of 254 to 65536 bytes (64KB) is represented by three bytes. The first contains the value 254 and the next two bytes contain the plain length. The low order byte goes first. A length of 65537 to 4294967296 bytes (4GB) is represented by five bytes. The first contains the value 255 and the next four bytes contain the plain length. The low order byte goes first.

The encoded compressed record length does not include these length bytes. It tells the number of bytes which follow behind the length bytes for this record.

10.3.4. Code Tree Representation

The code trees are binary trees. Every node has exactly two childs. The childs can be leafs or nodes. Each leaf contains one original, uncompressed value. The nodes do not contain values, but only pointers to the left and right child. The Huffman codes represent the navigation through the tree. Every left branch gets a 0 bit, every right branch gets a 1 bit.

The in-memory representation of the trees are two unsigned integers per node. Each describes either a leaf value or an offset (in unsigned integers) to the child node. To distinguish values from offsets, the 15th bit (decimal value 32768) is set together with offsets. This is safe as the size of the trees is limited by either having a maximum of 256 elements for byte value compression or 4096 elements for distinct column value compression.

The representaion of the trees in the compressed data file is almost the same. But instead of writing all bits of the unsigned integers, only as many bits are written as are required to represent the highest value or offset respectively. One more bit per value/offset is written in advance, to distinguish both. The number of bits required per value and per offset is computed in advance and part of the code tree description.

10.3.5. Usage of the Index File

While the header of the compressed data file contains a lot of information, there are still some things which need to be taken fron the index file. These are the number of columns of the table and the length of each column. The latter is required for columns with suppressed leading spaces or suppressed trailing spaces or zeros.

10.3.6. myisampack Tricks

As already mentioned, myisampack uses some tricks to decrease the amount of data to be encoded. These cope with leading and trailing spaces or zeros or with all blank or NULL fields.

I do not describe these in detail here. They do not materialize in the compressed data files other than the later mentioned field and pack types. They are however important to know for decoding the records.

10.3.7. Detailed Specification of the Decoding:

Below follows the detailed specification of the encoding:

Datafile fixed header (32 bytes):

```
4 byte  magic number
4 byte  total header length (fixed + column info + code trees)
4 byte  minimum packed record length
4 byte  maximum packed record length
4 byte  total number of elements in all code trees
4 byte  total number of bytes collected for distinct column values
2 byte  number of code trees
1 byte  maximum number of bytes required to represent record+blob lengths
1 byte  number of bytes required to represent the compressed data file length
4 byte  zeros
```

Column Information. For every column in the table:

```
5 bits  field type
    FIELD_NORMAL          0
    FIELD_SKIP_ENDSPACE   1
    FIELD_SKIP_PRESPACE   2
    FIELD_SKIP_ZERO       3
    FIELD_BLOB            4
    FIELD_CONSTANT        5
    FIELD_INTERVALL       6
    FIELD_ZERO            7
    FIELD_VARCHAR         8
    FIELD_CHECK           9

6 bits  pack type as a set of flags
    PACK_TYPE_SELECTED      1
    PACK_TYPE_SPACE_FIELDS  2
    PACK_TYPE_ZERO_FILL     4

5 bits  if pack type contains PACK_TYPE_ZERO_FILL
    minimum number of trailing zero bytes in this column
        else
    number of bits to encode the number of
    packed bytes in this column (length_bits)

x bits  number of the code tree used to encode this column
    x is the minimum number of bits required to represent the highest
    tree number.
```

Code Trees. For every tree:

```
1 bit   compression type
    0 = byte value compression
        8 bits  minimum byte value coded by this tree
        9 bits  number of byte values encoded by this tree
        5 bits  number of bits used to encode the byte values
        5 bits  number of bits used to encode offsets to next tree elements
    1 = distinct column value compression
        15 bits number of distinct column values encoded by this tree
        16 bits length of the buffer with all column values
        5 bits  number of bits used to encode the index of the column value
        5 bits  number of bits used to encode offsets to next tree elements
For each code tree element:
    1 bit   IS_OFFSET
    x bits  the announced number of bits for either a value or an offset
x bits  alignment to the next byte border
If compression by distinct column values:
    The number of 8-bit values that make up the column value buffer
```

Compressed Records. For every record:

```
1-5 bytes  length of the compressed record in bytes
1-5 bytes  total length of all expanded blobs of this record
For every column:
    If pack type includes PACK_TYPE_SPACE_FIELDS,
        1 bit   1 = spaces only, 0 = not only spaces
    In case the filed type is of:
        FIELD_SKIP_ZERO
            1 bit   1 = zeros only, 0 = not only zeros
                    In the latter case
                        x bits  the Huffman code for every byte
        FIELD_NORMAL
            x bits  the Huffman code for every byte
        FIELD_SKIP_ENDSPACE
            If pack type includes PACK_TYPE_SELECTED,
                1 bit   1 = more than min endspace, 0 = not more
                        In the former case
                            x bits  nr of extra spaces, x = length_bits
            else
                x bits  nr of extra spaces, x = length_bits
            x bits  the Huffman code for every byte
        FIELD_SKIP_PRESPACE
            If pack type includes PACK_TYPE_SELECTED,
                1 bit   1 = more than min prespace, 0 = not more
                        In the former case
                            x bits  nr of extra spaces, x = length_bits
            else
                x bits  nr of extra spaces, x = length_bits
            x bits  the Huffman code for every byte
        FIELD_CONSTANT or FIELD_ZERO or FIELD_CHECK
            nothing for these
        FIELD_INTERVALL
            x bits  the Huffman code for the buffer index of the column value
        FIELD_BLOB
            1 bit   1 = blob is empty, 0 = not empty
                    In the latter case
                        x bits  blob length, x = length_bits
                        x bits  the Huffman code for every byte
        FIELD_VARCHAR
            1 bit   1 = varchar is empty, 0 = not empty
                    In the latter case
                        x bits  blob length, x = length_bits
                        x bits  the Huffman code for every byte
    x bits  alignment to the next byte border
```

Chapter 11. InnoDB Storage Engine

11.1. 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.

11.1.1. 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.

11.1.1.1. 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`.

11.1.1.2. 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).

11.1.1.3. 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 NULLs take no space.

11.1.2. 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.

11.2. 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.

11.2.1. 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.

11.2.1.1. 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.

```
--------
- root -
--------
     |
----------------------
|                    |
|                    |
--------         --------
- leaf -  <-->   - leaf -
--------         --------
```

  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.

11.2.1.2. 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.

11.2.1.3. 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.

11.2.1.4. 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.

11.2.1.5. Free Space

I think it's clear what the Free Space part of a page is, from the discussion of other parts.

11.2.1.6. 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.)

11.2.1.7. 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.

11.2.2. 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.

11.2.3. 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.

Chapter 12. Writing a Custom Storage Engine

12.1. Introduction

With MySQL 5.1, MySQL AB has introduced a pluggable storage engine architecture that makes it possible to create new storage engines and add them to a running MySQL server without recompiling the server itself.

This architecture makes it easier to develop new storage engines for MySQL and deploy them.

This chapter is intended as a guide to assist you in developing a storage engine for the new pluggable storage engine architecture.

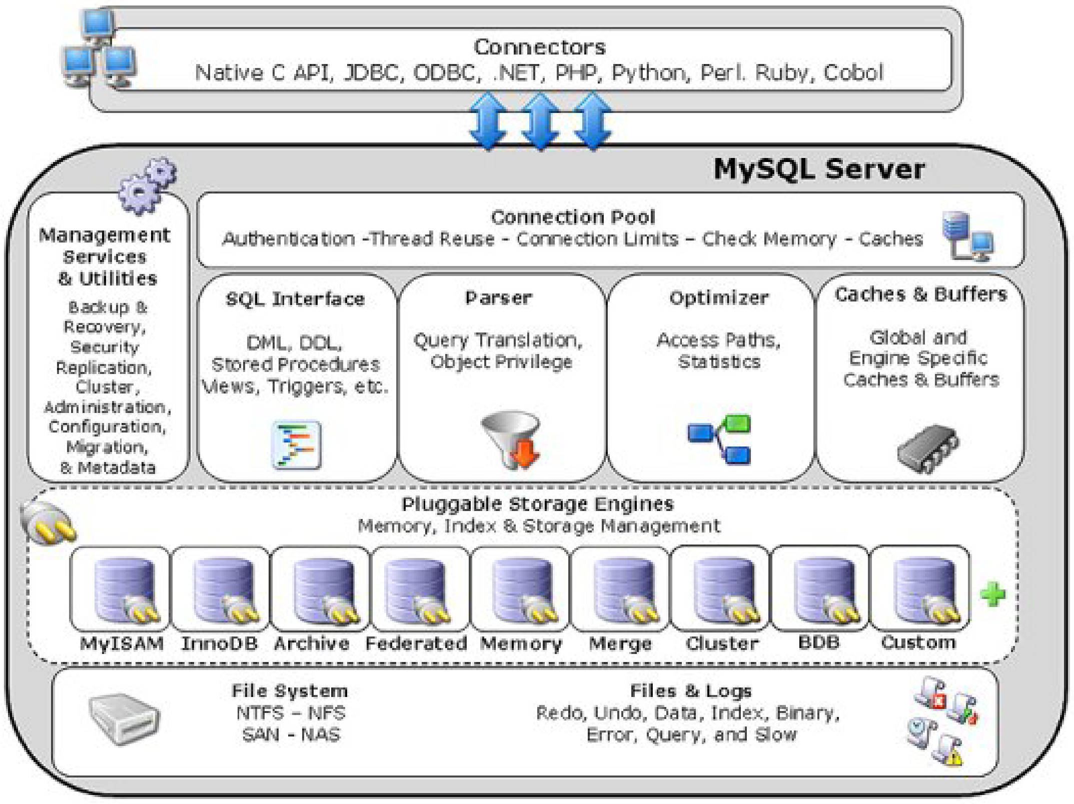
Additional resources

- A forum dedicated to custom storage engines is available at http://forums.mysql.com/list.php?94.

12.2. Overview

The MySQL server is built in a modular fashion:

Figure 12.1. MySQL architecture


Connectors
Native C API, JDBC, ODBC, .NET, PHP, Python, Perl. Ruby, Cobol
MySQL Server
Management Services & Utilities
Backup & Recovery, Security Replication, Cluster, Administration, Configuration, Migration, & Metadata
Connection Pool
Authentication -Thread Reuse - Connection Limits - Check Memory - Caches
SQL Interface
DML, DDL, Stored Procedures Views, Triggers, etc.
Parser
Query Translation, Object Privilege
Optimizer
Access Paths, Statistics
Caches & Buffers
Global and Engine Specific Caches & Buffers
Pluggable Storage Engines
Memory, Index & Storage Management
MyISAM
InnoDB
Archive
Federated
Memory
Merge
Cluster
BDB
Custom
File System
NTFS - NFS
SAN - NAS
Files & Logs
Redo, Undo, Data, Index, Binary, Error, Query, and Slow

The storage engines manage data storage and index management for MySQL. The MySQL server communicates with the storage engines through a defined API.

Each storage engine is a class with each instance of the class communicating with the MySQL server through a special `handler` interface.

Handlers are instanced on the basis of one handler for each thread that needs to work with a specific table. For example: If three connections all start working with the same table, three handler instances will need to be created.

Once a handler instance is created, the MySQL server issues commands to the handler to perform data storage and retrieval tasks such as opening a table, manipulating rows, and managing indexes.

Custom storage engines can be built in a progressive manner: Developers can start with a read-only storage engine and later add support for `INSERT`, `UPDATE`, and `DELETE` operations, and even later add support for indexing, transactions, and other advanced operations.

12.3. Creating Storage Engine Source Files

The easiest way to implement a new storage engine is to begin by copying and modifying the `EXAMPLE` storage engine. The files `ha_example.cc` and `ha_example.h` can be found in the `storage/example` directory of the MySQL 5.1 source tree. For instructions on how to obtain the 5.1 source tree, see MySQL Installation Using a Source Distribution [http://dev.mysql.com/doc/refman/5.1/en/installing-source.html].

When copying the files, change the names from `ha_example.cc` and `ha_example.h` to something appropriate to your storage engine, such as `ha_foo.cc` and `ha_foo.h`.

After you have copied and renamed the files you must replace all instances of `EXAMPLE` and `example` with the name of your storage engine. If you are familiar with `sed`, these steps can be done automatically (in this example, the name of your storage engine would be “FOO”):

```
sed s/EXAMPLE/FOO/g ha_example.h | sed s/example/foo/g ha_foo.h
sed s/EXAMPLE/FOO/g ha_example.cc | sed s/example/foo/g ha_foo.cc
```

12.4. Creating the `handlerton`

The `handlerton` (short for “handler singleton”) defines the storage engine and contains method pointers to those methods that apply to the storage engine as a whole, as opposed to methods that work on a per-table basis. Some examples of such methods include transaction methods to handle commits and rollbacks.

Here's an example from the `EXAMPLE` storage engine:

```
handlerton example_hton= {
  "EXAMPLE",
  SHOW_OPTION_YES,
  "Example storage engine",
  DB_TYPE_EXAMPLE_DB,
  NULL,    /* Initialize */
  0,       /* slot */
  0,       /* savepoint size. */
  NULL,    /* close_connection */
  NULL,    /* savepoint */
  NULL,    /* rollback to savepoint */
  NULL,    /* release savepoint */
  NULL,    /* commit */
  NULL,    /* rollback */
  NULL,    /* prepare */
  NULL,    /* recover */
  NULL,    /* commit_by_xid */
  NULL,    /* rollback_by_xid */
  NULL,    /* create_cursor_read_view */
  NULL,    /* set_cursor_read_view */
  NULL,    /* close_cursor_read_view */
  example_create_handler,    /* Create a new handler */
  NULL,    /* Drop a database */
  NULL,    /* Panic call */
  NULL,    /* Release temporary latches */
  NULL,    /* Update Statistics */
  NULL,    /* Start Consistent Snapshot */
  NULL,    /* Flush logs */
  NULL,    /* Show status */
  NULL,    /* Replication Report Sent Binlog */
  HTON_CAN_RECREATE
};
```

This is the definition of the `handlerton` from `handler.h`:

```
typedef struct
  {
    const char *name;
    SHOW_COMP_OPTION state;
    const char *comment;
    enum db_type db_type;
    bool (*init)();
    uint slot;
    uint savepoint_offset;
    int  (*close_connection)(THD *thd);
    int  (*savepoint_set)(THD *thd, void *sv);
    int  (*savepoint_rollback)(THD *thd, void *sv);
    int  (*savepoint_release)(THD *thd, void *sv);
    int  (*commit)(THD *thd, bool all);
    int  (*rollback)(THD *thd, bool all);
    int  (*prepare)(THD *thd, bool all);
    int  (*recover)(XID *xid_list, uint len);
    int  (*commit_by_xid)(XID *xid);
    int  (*rollback_by_xid)(XID *xid);
    void *(*create_cursor_read_view)();
    void (*set_cursor_read_view)(void *);
    void (*close_cursor_read_view)(void *);
    handler *(*create)(TABLE *table);
    void (*drop_database)(char* path);
    int (*panic)(enum ha_panic_function flag);
    int (*release_temporary_latches)(THD *thd);
    int (*update_statistics)();
    int (*start_consistent_snapshot)(THD *thd);
    bool (*flush_logs)();
    bool (*show_status)(THD *thd, stat_print_fn *print, enum ha_stat_type stat);
    int (*repl_report_sent_binlog)(THD *thd, char *log_file_name, my_off_t end_offset);
    uint32 flags;
  } handlerton;
```

There are a total of 30 handlerton elements, only a few of which are mandatory (specifically the first four elements and the `create()` method).

1. The name of the storage engine. This is the name that will be used when creating tables (`CREATE TABLE ... ENGINE = *FOO*;`).

2. The value to be displayed in the `status` field when a user issues the `SHOW STORAGE ENGINES` command.

3. The storage engine comment, a description of the storage engine displayed when using the `SHOW STORAGE ENGINES` command.

4. An integer that uniquely identifies the storage engine within the MySQL server. The constants used by the built-in storage engines are defined in the `handler.h` file. Custom engines should use `DB_TYPE_CUSTOM`.

5. A method pointer to the storage engine initializer. This method is only called once when the server starts to allow the storage engine class to perform any housekeeping that is necessary before hand-

lers are instanced.

6. The slot. Each storage engine has its own memory area (actually a pointer) in the `thd`, for storing per-connection information. It is accessed as `thd->ha_data[`*`foo`*`_hton.slot]`. The slot number is initialized by MySQL after *`foo`*`_init()` is called. For more information on the `thd`, see Section 12.16.3, “Implementing ROLLBACK”.

7. The savepoint offset. To store per-savepoint data the storage engine is provided with an area of a requested size (`0`, if no savepoint memory is necessary).

   The savepoint offset must be initialized statically to the size of the needed memory to store per-savepoint information. After *`foo`*`_init` it is changed to be an offset to the savepoint storage area and need not be used by the storage engine.

   For more information, see Section 12.16.5.1, “Specifying the Savepoint Offset”.

8. Used by transactional storage engines, clean up any memory allocated in their slot.

9. A method pointer to the handler's `savepoint_set()` method. This is used to create a savepoint and store it in memory of the requested size.

   For more information, see Section 12.16.5.2, “Implementing the `savepoint_set` Method”.

10. A method pointer to the handler's `rollback_to_savepoint()` method. This is used to return to a savepoint during a transaction. It's only populated for storage engines that support savepoints.

    For more information, see Section 12.16.5.3, “Implementing the `savepoint_rollback()` `Method`”.

11. A method pointer to the handler's `release_savepoint()` method. This is used to release the resources of a savepoint during a transaction. It's optionally populated for storage engines that support savepoints.

    For more information, see Section 12.16.5.4, “Implementing the `savepoint_release()` Method”.

12. A method pointer to the handler's `commit()` method. This is used to commit a transaction. It's only populated for storage engines that support transactions.

    For more information, see Section 12.16.4, “Implementing COMMIT”.

13. A method pointer to the handler's `rollback()` method. This is used to roll back a transaction. It's only populated for storage engines that support transactions.

    For more information, see Section 12.16.3, “Implementing ROLLBACK”.

14. Required for XA transactional storage engines. Prepare transaction for commit.

15. Required for XA transactional storage engines. Returns a list of transactions that are in the prepared state.

16. Required for XA transactional storage engines. Commit transaction identified by XID.

17. Required for XA transactional storage engines. Rollback transaction identified by XID.

18. Called when a cursor is created to allow the storage engine to create a consistent read view.

19. Called to switch to a specific consistent read view.

20. Called to close a specific read view.

21. *MANDATORY* - Construct and return a handler instance.

    For more information, see Section 12.5, “Handling Handler Instantiation”.

22. Used if the storage engine needs to perform special steps when a schema is dropped (such as in a storage engine that uses tablespaces).

23. Cleanup method called during server shutdown and crashes.

24. `InnoDB`-specific method.

25. `InnoDB`-specific method called at start of `SHOW ENGINE InnoDB STATUS`.

26. Method called to begin a consistent read.

27. Called to indicate that logs should be flushed to reliable storage.

28. Provides human readable status information on the storage engine for `SHOW ENGINE foo STATUS`.

29. `InnoDB`-specific method used for replication.

30. Handlerton flags that indicate the capabilities of the storage engine. Possible values are defined in `sql/handler.h` and copied here:

    ```
    #define HTON_NO_FLAGS                 0
    #define HTON_CLOSE_CURSORS_AT_COMMIT (1 << 0)
    #define HTON_ALTER_NOT_SUPPORTED     (1 << 1)
    #define HTON_CAN_RECREATE            (1 << 2)
    #define HTON_FLUSH_AFTER_RENAME      (1 << 3)
    #define HTON_NOT_USER_SELECTABLE     (1 << 4)
    ```

    `HTON_ALTER_NOT_SUPPORTED` is used to indicate that the storage engine cannot accept `ALTER TABLE` statements. The `FEDERATED` storage engine is an example.

    `HTON_FLUSH_AFTER_RENAME` indicates that `FLUSH LOGS` must be called after a table rename.

    `HTON_NOT_USER_SELECTABLE` indicates that the storage engine should not be shown when a user calls `SHOW STORAGE ENGINES`. Used for system storage engines such as the dummy storage engine for binary logs.

12.5. Handling Handler Instantiation

The first method call your storage engine needs to support is the call for a new handler instance.

Before the `handlerton` is defined in the storage engine source file, a method header for the instantiation method must be defined. Here is an example from the `CSV` engine:

```
static handler* tina_create_handler(TABLE *table);
```

As you can see, the method accepts a pointer to the table the handler is intended to manage, and returns a handler object.

After the method header is defined, the method is named with a method pointer in the `create()` `handlerton` element, identifying the method as being responsible for generating new handler instances.

Here is an example of the `MyISAM` storage engine's instantiation method:

```
static handler *myisam_create_handler(TABLE *table)
  {
    return new ha_myisam(table);
  }
```

This call then works in conjunction with the storage engine's constructor. Here is an example from the `FEDERATED` storage engine:

```
ha_federated::ha_federated(TABLE *table_arg)
  :handler(&federated_hton, table_arg),
  mysql(0), stored_result(0), scan_flag(0),
  ref_length(sizeof(MYSQL_ROW_OFFSET)), current_position(0)
  {}
```

And here's one more example from the `EXAMPLE` storage engine:

```
ha_example::ha_example(TABLE *table_arg)
  :handler(&example_hton, table_arg)
  {}
```

The additional elements in the `FEDERATED` example are extra initializations for the handler. The minimum implementation required is the `handler()` initialization shown in the `EXAMPLE` version.

12.6. Defining Filename Extensions

Storage engines are required to provide the MySQL server with a list of extensions used by the storage engine with regard to a given table, its data and indexes.

Extensions are expected in the form of a null-terminated string array. The following is the array used by the `CSV` engine:

```
static const char *ha_tina_exts[] = {
  ".CSV",
  NullS
};
```

This array is returned when the `bas_ext()` method is called:

```
const char **ha_tina::bas_ext() const
{
  return ha_tina_exts;
}
```

By providing extension information you can also omit implementing `DROP TABLE` functionality as the MySQL server will implement it for you by closing the table and deleting all files with the extensions you specify.

12.7. Creating Tables

Once a handler is instanced, the first operation that will likely be required is the creation of a table.

Your storage engine must implement the `create()` virtual method:

```
virtual int create(const char *name, TABLE *form, HA_CREATE_INFO *info)=0;
```

This method should create all necessary files but does not need to open the table. The MySQL server will call for the table to be opened later on.

The *name parameter is the name of the table. The *form parameter is a TABLE structure that defines the table and matches the contents of the *tablename*.frm file already created by the MySQL server. Storage engines must not modify the *tablename*.frm file.

The *info parameter is a structure containing information on the CREATE TABLE statement used to create the table. The structure is defined in handler.h and copied here for your convenience:

```
typedef struct st_ha_create_information
{
    CHARSET_INFO *table_charset, *default_table_charset;
    LEX_STRING connect_string;
    const char *comment,*password;
    const char *data_file_name, *index_file_name;
    const char *alias;
    ulonglong max_rows,min_rows;
    ulonglong auto_increment_value;
    ulong table_options;
    ulong avg_row_length;
    ulong raid_chunksize;
    ulong used_fields;
    SQL_LIST merge_list;
    enum db_type db_type;
    enum row_type row_type;
    uint null_bits;                       /* NULL bits at start of record */
    uint options;                         /* OR of HA_CREATE_ options */
    uint raid_type,raid_chunks;
    uint merge_insert_method;
    uint extra_size;                      /* length of extra data segment */
    bool table_existed;                /* 1 in create if table existed */
    bool frm_only;                        /* 1 if no ha_create_table() */
    bool varchar;                         /* 1 if table has a VARCHAR */
} HA_CREATE_INFO;
```

A basic storage engine can ignore the contents of *form and *info, as all that is really required is the creation and possibly the initialization of the data files used by the storage engine (assuming the storage engine is file-based).

For example, here is the implementation from the CSV storage engine:

```
int ha_tina::create(const char *name, TABLE *table_arg,
  HA_CREATE_INFO *create_info)
{
    char name_buff[FN_REFLEN];
    File create_file;
    DBUG_ENTER("ha_tina::create");

    if ((create_file= my_create(fn_format(name_buff, name, "", ".CSV",
          MY_REPLACE_EXT|MY_UNPACK_FILENAME),0,
          O_RDWR | O_TRUNC,MYF(MY_WME))) < 0)
    DBUG_RETURN(-1);

    my_close(create_file,MYF(0));

    DBUG_RETURN(0);
}
```

In the preceding example, the CSV engine does not refer at all to the *table_arg or *create_info parameters, but simply creates the required data files, closes them, and returns.

The my_create and my_close methods are helper methods defined in src/include/my_sys.h.

12.8. Opening a Table

Before any read or write operations are performed on a table, the MySQL server will call the handler::open() method to open the table data and index files (if they exist).

```
int open(const char *name, int mode, int test_if_locked);
```

The first parameter is the name of the table to be opened. The second parameter determines what file to open or what operation to take. The values are defined in `handler.h` and are copied here for your convenience:

```
O_RDONLY  -  Open read only
O_RDWR    -  Open read/write
```

The final option dictates whether the handler should check for a lock on the table before opening it. The following options are available:

```
#define HA_OPEN_ABORT_IF_LOCKED   0   /* default */
#define HA_OPEN_WAIT_IF_LOCKED    1
#define HA_OPEN_IGNORE_IF_LOCKED  2
#define HA_OPEN_TMP_TABLE         4   /* Table is a temp table */
#define HA_OPEN_DELAY_KEY_WRITE   8   /* Don't update index */
#define HA_OPEN_ABORT_IF_CRASHED  16
#define HA_OPEN_FOR_REPAIR        32  /* open even if crashed */
```

Typically your storage engine will need to implement some form of shared access control to prevent file corruption is a multi-threaded environment. For an example of how to implement file locking, see the `get_share()` and `free_share()` methods of `sql/examples/ha_tina.cc`.

12.9. Implementing Basic Table Scanning

The most basic storage engines implement read-only table scanning. Such engines might be used to support SQL queries of logs and other data files that are populated outside of MySQL.

The implementation of the methods in this section provide the first steps toward the creation of more advanced storage engines.

The following shows the method calls made during a nine-row table scan of the `CSV` engine:

```
ha_tina::store_lock
ha_tina::external_lock
ha_tina::info
ha_tina::rnd_init
ha_tina::extra - ENUM HA_EXTRA_CACHE   Cache record in HA_rrnd()
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::rnd_next
ha_tina::extra - ENUM HA_EXTRA_NO_CACHE   End caching of records (def)
ha_tina::external_lock
ha_tina::extra - ENUM HA_EXTRA_RESET   Reset database to after open
```

12.9.1. Implementing the `store_lock()` Method

The `store_lock()` method is called before any reading or writing is performed.

Before adding the lock into the table lock handler `mysqld` calls store lock with the requested locks. Store lock can modify the lock level, for example change blocking write lock to non-blocking, ignore the lock (if we don't want to use MySQL table locks at all) or add locks for many tables (like we do when we are using a MERGE handler).

When releasing locks, `store_lock()` is also called. In this case, one usually doesn't have to do anything.

If the argument of `store_lock` is `TL_IGNORE`, it means that MySQL requests the handler to store the same lock level as the last time.

The potential lock types are defined in `includes/thr_lock.h` and are copied here:

```
enum thr_lock_type
{
         TL_IGNORE=-1,
         TL_UNLOCK,                              /* UNLOCK ANY LOCK */
           TL_READ,                                /* Read lock */
           TL_READ_WITH_SHARED_LOCKS,
         TL_READ_HIGH_PRIORITY,      /* High prior. than TL_WRITE. Allow concurrent insert */
         TL_READ_NO_INSERT,                      /* READ, Don't allow concurrent insert */
         TL_WRITE_ALLOW_WRITE,                    /* Write lock, but allow other threads to read / writ
         TL_WRITE_ALLOW_READ,        /* Write lock, but allow other threads to read / write. */
         TL_WRITE_CONCURRENT_INSERT, /* WRITE lock used by concurrent insert. */
         TL_WRITE_DELAYED,                       /* Write used by INSERT DELAYED.  Allows READ locks */
         TL_WRITE_LOW_PRIORITY,      /* WRITE lock that has lower priority than TL_READ */
         TL_WRITE,                               /* Normal WRITE lock */
         TL_WRITE_ONLY               /* Abort new lock request with an error */
};
```

Actual lock handling will vary depending on your locking implementation and you may choose to implement some or none of the requested lock types, substituting your own methods as appropriate. The following is the minimal implementation, for a storage engine that does not need to downgrade locks:

```
THR_LOCK_DATA **ha_tina::store_lock(THD *thd,
                                     THR_LOCK_DATA **to,
                                     enum thr_lock_type lock_type)
{
   if (lock_type != TL_IGNORE && lock.type == TL_UNLOCK)
     lock.type=lock_type;
   *to++= &lock;
   return to;
}
```

See also `ha_myisammrg::store_lock()` for a more complex implementation.

12.9.2. Implementing the `external_lock()` Method

The `external_lock()` method is called at the start of a statement or when a `LOCK TABLES` statement is issued.

Examples of using `external_lock()` can be found in the `sql/ha_innodb.cc` file, but most storage engines can simply return `0`, as is the case with the `EXAMPLE` storage engine:

```
int ha_example::external_lock(THD *thd, int lock_type)
{
   DBUG_ENTER("ha_example::external_lock");
   DBUG_RETURN(0);
}
```

12.9.3. Implementing the `rnd_init()` Method

The method called before any table scan is the `rnd_init()` method. The `rnd_init()` method is used to prepare for a table scan, resetting counters and pointers to the start of the table.

The following example is from the `CSV` storage engine:

```
  int ha_tina::rnd_init(bool scan)
{
      DBUG_ENTER("ha_tina::rnd_init");

      current_position= next_position= 0;
      records= 0;
      chain_ptr= chain;

      DBUG_RETURN(0);
}
```

If the `scan` parameter is true, the MySQL server will perform a sequential table scan, if false the MySQL server will perform random reads by position.

12.9.4. Implementing the `info()` Method

Prior to commencing a table scan, the `info()` method is called to provide extra table information to the optimizer.

The information required by the optimizer is not given through return values but instead by populating certain properties of the storage engine class, which the optimizer reads after the `info()` call returns.

In addition to being used by the optimizer, many of the values set during a call to the `info()` method are also used for the `SHOW TABLE STATUS` statement.

The public properties are listed in full in `sql/handler.h`; several of the more common ones are copied here:

```
ulonglong data_file_length;          /* Length off data file */
ulonglong max_data_file_length; /* Length off data file */
ulonglong index_file_length;
ulonglong max_index_file_length;
ulonglong delete_length;          /* Free bytes */
ulonglong auto_increment_value;
ha_rows records;                  /* Records in table */
ha_rows deleted;                  /* Deleted records */
ulong raid_chunksize;
ulong mean_rec_length;          /* physical reclength */
time_t create_time;                  /* When table was created */
time_t check_time;
time_t update_time;
```

For the purposes of a table scan, the most important property is `records`, which indicates the number of records in the table. The optimizer will perform differently when the storage engine indicates that there are zero or one rows in the table than it will when there are two or more. For this reason it is important to return a value of two or greater when you do not actually know how many rows are in the table before you perform the table scan (such as in a situation where the data may be externally populated).

12.9.5. Implementing the `extra()` Method

Prior to some operations, the `extra()` method is called to provide extra hints to the storage engine on how to perform certain operations.

Implementation of the hints in the `extra` call is not mandatory, and most storage engines return `0`:

```
int ha_tina::extra(enum ha_extra_function operation)
{
   DBUG_ENTER("ha_tina::extra");
   DBUG_RETURN(0);
}
```

12.9.6. Implementing the `rnd_next()` Method

After the table is initialized, the MySQL server will call the handler's `rnd_next()` method once for every row to be scanned until the server's search condition is satisfied or an end of file is reached, in which case the handler returns HA_ERR_END_OF_FILE.

The rnd_next() method takes a single byte array parameter named *buf. The *buf parameter must be populated with the contents of the table row in the internal MySQL format.

The server uses three data formats: fixed-length rows, variable-length rows, and variable-length rows with BLOB pointers. In each format, the columns appear in the order in which they were defined by the CREATE TABLE statement. (The table definition is stored in the .frm file, and the optimizer and the handler are both able to access table metadata from the same source, its TABLE structure).

Each format begins with a “NULL bitmap” of one bit per nullable column. A table with as many as eight nullable columns will have a one-byte bitmap; a table with nine to sixteen nullable columns will have a two-byte bitmap, and so forth. One exception is fixed-width tables, which have an additional starting bit so that a table with eight nullable columns would have a two-byte bitmap.

After the NULL bitmap come the columns, one by one. Each column is of the size indicated in MySQL Data Types [http://dev.mysql.com/doc/refman/5.1/en/data-types.html]. In the server, column data types are defined in the sql/field.cc file. In the fixed length row format, the columns are simply laid out one by one. In a variable-length row, VARCHAR columns are coded as a one or two-byte length, followed by a string of characters. In a variable-length row with BLOB columns, each blob is represented by two parts: first an integer representing the actual size of the BLOB, and then a pointer to the BLOB in memory.

Examples of row conversion (or “packing”) can be found by starting at rnd_next() in any table handler. In ha_tina.cc, for example, the code in find_current_row() illustrates how the TABLE structure (pointed to by table) and a string object (named buffer) can be used to pack character data from a CSV file. Writing a row back to disk requires the opposite conversion, unpacking from the internal format.

The following example is from the CSV storage engine:

```
int ha_tina::rnd_next(byte *buf)
{
   DBUG_ENTER("ha_tina::rnd_next");

   statistic_increment(table->in_use->status_var.ha_read_rnd_next_count, &LOCK_status);

   current_position= next_position;
   if (!share->mapped_file)
     DBUG_RETURN(HA_ERR_END_OF_FILE);
   if (HA_ERR_END_OF_FILE == find_current_row(buf) )
     DBUG_RETURN(HA_ERR_END_OF_FILE);

   records++;
   DBUG_RETURN(0);
}
```

The conversion from the internal row format to CSV row format is performed in the find_current_row() method:

```
int ha_tina::find_current_row(byte *buf)
{
   byte *mapped_ptr= (byte *)share->mapped_file + current_position;
   byte *end_ptr;
   DBUG_ENTER("ha_tina::find_current_row");

   /* EOF should be counted as new line */
   if ((end_ptr=  find_eoln(share->mapped_file, current_position,
                            share->file_stat.st_size)) == 0)
     DBUG_RETURN(HA_ERR_END_OF_FILE);

   for (Field **field=table->field ; *field ; field++)
   {
     buffer.length(0);
     mapped_ptr++; // Increment past the first quote
     for(;mapped_ptr != end_ptr; mapped_ptr++)
     {
       // Need to convert line feeds!
       if (*mapped_ptr == '"' &&
           (((mapped_ptr[1] == ',') && (mapped_ptr[2] == '"')) ||
            (mapped_ptr == end_ptr -1 )))
       {
         mapped_ptr += 2; // Move past the , and the "
         break;
       }
       if (*mapped_ptr == '\\' && mapped_ptr != (end_ptr - 1))
       {
         mapped_ptr++;
         if (*mapped_ptr == 'r')
           buffer.append('\r');
         else if (*mapped_ptr == 'n' )
           buffer.append('\n');
         else if ((*mapped_ptr == '\\') || (*mapped_ptr == '"'))
           buffer.append(*mapped_ptr);
         else  /* This could only happed with an externally created file */
         {
           buffer.append('\\');
           buffer.append(*mapped_ptr);
         }
       }
       else
         buffer.append(*mapped_ptr);
     }
     (*field)->store(buffer.ptr(), buffer.length(), system_charset_info);
   }
   next_position= (end_ptr - share->mapped_file)+1;
   /* Maybe use \N for null? */
   memset(buf, 0, table->s->null_bytes); /* We do not implement nulls! */

   DBUG_RETURN(0);
}
```

12.10. Closing a Table

When the MySQL server is finished with a table, it will call the close() method to close file pointers and release any other resources.

Storage engines that use the shared access methods seen in the CSV engine and other example engines must remove themselves from the shared structure:

```
int ha_tina::close(void)
 {
   DBUG_ENTER("ha_tina::close");
   DBUG_RETURN(free_share(share));
 }
```

Storage engines using their own share management systems should use whatever methods are needed to remove the handler instance from the share for the table opened in their handler.

12.11. Adding Support for INSERT to a Storage Engine

Once you have read support in your storage engine, the next feature to implement is support for INSERT statements. With INSERT support in place, your storage engine can handle WORM (write once, read many) applications such as logging and archiving for later analysis.

All INSERT operations are handled through the write_row() method:

```
int ha_foo::write_row(byte *buf)
```

The *buf parameter contains the row to be inserted in the internal MySQL format. A basic storage engine could simply advance to the end of the data file and append the contents of the buffer directly (this would also make reading rows easier as you could read the row and pass it directly into the buffer parameter of the rnd_next() method.

The process for writing a row is the opposite of the process for reading one: take the data from the MySQL internal row format and write it to the data file. The following example is from the MyISAM storage engine:

```
int ha_myisam::write_row(byte * buf)
{
  statistic_increment(table->in_use->status_var.ha_write_count,&LOCK_status);

  /* If we have a timestamp column, update it to the current time */
  if (table->timestamp_field_type & TIMESTAMP_AUTO_SET_ON_INSERT)
    table->timestamp_field->set_time();

  /*
    If we have an auto_increment column and we are writing a changed row
    or a new row, then update the auto_increment value in the record.
  */
  if (table->next_number_field && buf == table->record[0])
    update_auto_increment();
  return mi_write(file,buf);
}
```

Three items of note in the preceding example include the updating of table statistics for writes, the setting of the timestamp prior to writing the row, and the updating of AUTO_INCREMENT values.

12.12. Adding Support for UPDATE to a Storage Engine

The MySQL server executes UPDATE statements by performing a (table/index/range/etc.) scan until it locates a row matching the WHERE clause of the UPDATE statement, then calling the update_row() method:

```
int ha_foo::update_row(const byte *old_data, byte *new_data)
```

The *old_data parameter contains the data that existed in the row prior to the update, while the *new_data parameter contains the new contents of the row (in the MySQL internal row format).

Performing an update will depend on row format and storage implementation. Some storage engines will replace data in-place, while other implementations delete the existing row and append the new row at the end of the data file.

Non-indexed storage engines can typically ignore the contents of the *old_data parameter and just deal with the *new_data buffer. Transactional engines may need to compare the buffers to determine what changes have been made for a later rollback.

If the table being updated contains timestamp columns, the updating of the timestamp will have to be managed in the update_row() call. The following example is from the CSV engine:

```
int ha_tina::update_row(const byte * old_data, byte * new_data)
{
   int size;
   DBUG_ENTER("ha_tina::update_row");

   statistic_increment(table->in_use->status_var.ha_read_rnd_next_count,
                       &LOCK_status);

   if (table->timestamp_field_type & TIMESTAMP_AUTO_SET_ON_UPDATE)
     table->timestamp_field->set_time();

   size= encode_quote(new_data);

   if (chain_append())
     DBUG_RETURN(-1);

   if (my_write(share->data_file, buffer.ptr(), size, MYF(MY_WME | MY_NABP)))
     DBUG_RETURN(-1);
   DBUG_RETURN(0);
}
```

Note the setting of the timestamp in the previous example.

12.13. Adding Support for `DELETE` to a Storage Engine

The MySQL server executes `DELETE` statements using the same approach as for `UPDATE` statements: It advances to the row to be deleted using the `rnd_next()` method and then calls the `delete_row()` method to delete the row:

```
int ha_foo::delete_row(const byte *buf)
```

The `*buf` parameter contains the contents of the row to be deleted. For non-indexed storage engines the parameter can be ignored, but transactional storage engines may need to store the deleted data for roll-back purposes.

The following example is from the `CSV` storage engine:

```
int ha_tina::delete_row(const byte * buf)
{
   DBUG_ENTER("ha_tina::delete_row");
   statistic_increment(table->in_use->status_var.ha_delete_count,
                       &LOCK_status);

   if (chain_append())
     DBUG_RETURN(-1);

   --records;

   DBUG_RETURN(0);
}
```

The steps of note in the preceding example are the update of the `delete_count` statistic and the re-cord count.

12.14. Supporting Non-Sequential Reads

In addition to table scanning, storage engines can implement methods for non-sequential reading. The MySQL server uses these methods for certain sort operations.

12.14.1. Implementing the `position()` Method

The `position()` method is called after every call to `rnd_next()` if the data needs to be reordered:

```
void ha_foo::position(const byte *record)
```

The contents of `*record` are up to you — whatever value you provide will be returned in a later call to retrieve the row. Most storage engines will store some form of offset or primary key value.

12.14.2. Implementing the `rnd_pos()` Method

The `rnd_pos()` method behaves in a similar fashion to the `rnd_next()` method but takes an addi-tional parameter:

```
int ha_foo::rnd_pos(byte * buf, byte *pos)
```

The `*pos` parameter contains positioning information previously recorded using the `position()` method.

A storage engine must locate the row specified by the position and return it through `*buf` in the internal

MySQL row format.

12.15. Supporting Indexing

Once basic read/write operations are implemented in a storage engine, the next stage is to add support for indexing. Without indexing, a storage engine's performance is quite limited.

This section documents the methods that must be implemented to add support for indexing to a storage engine.

12.15.1. Indexing Overview

Adding index support to a storage engine revolves around two tasks: providing information to the optimizer and implementing index-related methods. The information provided to the optimizer helps the optimizer to make better decisions about which index to use or even to skip using an index and instead perform a table scan.

The indexing methods either read rows that match a key, scan a set of rows by index order, or read information directly from the index.

The following example shows the method calls made during an `UPDATE` query that uses an index, such as `UPDATE foo SET ts = now() WHERE id = 1`:

```
ha_foo::index_init
ha_foo::index_read
ha_foo::index_read_idx
ha_foo::rnd_next
ha_foo::update_row
```

In addition to index reading methods, your storage engine must support the creation of new indexes and be able to keep table indexes up to date as rows are added, modified, and removed from tables.

12.15.2. Getting Index Information During `CREATE TABLE` Operations

It is preferable for storage engines that support indexing to read the index information provided during a `CREATE TABLE` operation and store it for future use. The reasoning behind this is that the index information is most readily available during table and index creation and is not as easily retrieved afterward.

The table index information is contained within the `key_info` structure of the `TABLE` argument of the `create()` method.

Within the `key_info` structure there is a `flag` that defines index behavior:

```
#define HA_NOSAME             1  /* Set if not duplicated records   */
#define HA_PACK_KEY           2  /* Pack string key to previous key */
#define HA_AUTO_KEY           16
#define HA_BINARY_PACK_KEY    32 /* Packing of all keys to prev key */
#define HA_FULLTEXT          128 /* For full-text search            */
#define HA_UNIQUE_CHECK      256 /* Check the key for uniqueness    */
#define HA_SPATIAL          1024 /* For spatial search              */
#define HA_NULL_ARE_EQUAL   2048 /* NULL in key are cmp as equal    */
#define HA_GENERATED_KEY    8192 /* Automatically generated key     */
```

In addition to the `flag`, there is an enumerator named `algorithm` that specifies the desired index type:

```
enum ha_key_alg {
  HA_KEY_ALG_UNDEF=     0,  /* Not specified (old file)     */
  HA_KEY_ALG_BTREE=     1,  /* B-tree, default one          */
  HA_KEY_ALG_RTREE=     2,  /* R-tree, for spatial searches */
  HA_KEY_ALG_HASH=      3,  /* HASH keys (HEAP tables)      */
  HA_KEY_ALG_FULLTEXT=  4   /* FULLTEXT (MyISAM tables)     */
};
```

In addition to the `flag` and `algorithm`, there is an array of `key_part` elements that describe the individual parts of a composite key.

The key parts define the field associated with the key part, whether the key should be packed, and the data type and length of the index part. See `ha_myisam.cc` for an example of how this information is parsed.

As an alternative, a storage engine can instead read index information from the `TABLE` structure of the handler during each operation.

12.15.3. Creating Index Keys

As part of every table-write operation (`INSERT`, `UPDATE`, `DELETE`), the storage engine is required to update its internal index information.

The method used to update indexes will vary from storage engine to storage engine, depending on the method used to store the index.

In general, the storage engine will have to use row information passed in methods such as `write_row()`, `delete_row()`, and `update_row()` in combination with index information for the table to determine what index data needs to be modified, and make the needed changes.

The method of associating an index with its row will depend on your storage approach. Current storage engines store the row offset.

12.15.4. Parsing Key Information

Many of the index methods pass a byte array named `*key` that identifies the index entry to be read in a standard format. Your storage engine will need to extract the information stored in the key and translate it into its internal index format to identify the row associated with the index.

The information in the key is obtained by iterating through the key, which is formatted the same as the definition in `table->key_info[`*`index`*`]->key_part[`*`part_num`*`]`.

12.15.5. Providing Index Information to the Optimizer

In order for indexing to be used effectively, storage engines need to provide the optimizer with information about the table and its indexes. This information is used to choose whether to use an index, and if so, which index to use.

12.15.5.1. Implementing the `info()` Method

The optimizer requests an update of table information by calling the `handler::info()` method. The `info()` method does not have a return value, instead it is expected that the storage engine will set a variety of public variables that the server will then read as needed. These values are also used to populate certain `SHOW` outputs such as `SHOW TABLE STATUS` and for queries of the `INFORMATION_SCHEMA`.

All variables are optional but should be filled if possible:

- `records` - The number of rows in the table. If you cannot provide an accurate number quickly you should set the value to be greater than 1 so that the optimizer does not perform optimizations for zero or one row tables.
- `deleted` - Number of deleted rows in table. Used to identify table fragmentation, where applicable.
- `data_file_length` - Size of the data file, in bytes. Helps optimizer calculate the cost of reads.
- `index_file_length` - Size of the index file, in bytes. Helps optimizer calculate the cost of reads.
- `mean_rec_length` - Average length of a single row, in bytes.
- `scan_time` - Cost in I/O seeks to perform a full table scan.
- `delete_length` -
- `check_time` -

When calculating values, speed is more important than accuracy, as there is no sense in taking a long time to give the optimizer clues as to what approach may be the fastest. Estimates within an order of magnitude are usually good enough.

12.15.5.2. Implementing the `records_in_range` Method

The `records_in_range()` method is called by the optimizer to assist in choosing which index on a table to use for a query or join. It is defined as follows:

```
ha_rows ha_foo::records_in_range(uint inx, key_range *min_key, key_range *max_key)
```

The `inx` parameter is the index to be checked. The `*min_key` parameter is the low end of the range while the `*max_key` parameter is the high end of the range.

`min_key.flag` can have one of the following values:

- `HA_READ_KEY_EXACT` - Include the key in the range
- `HA_READ_AFTER_KEY` - Don't include key in range

`max_key.flag` can have one of the following values:

- `HA_READ_BEFORE_KEY` - Don't include key in range
- `HA_READ_AFTER_KEY` - Include all 'end_key' values in the range

The following return values are allowed:

- `0` - There are no matching keys in the given range
- `number > 0` - There is approximately *`number`* matching rows in the range
- `HA_POS_ERROR` - Something is wrong with the index tree

When calculating values, speed is more important than accuracy.

12.15.6. Preparing for Index Use with index_init()

The index_init() method is called before an index is used to allow the storage engine to perform any necessary preparation or optimization:

```
int ha_foo::index_init(uint keynr, bool sorted)
```

Most storage engines do not need to make special preparations, in which case a default implementation will be used if the method is not explicitly implemented in the storage engine:

```
int handler::index_init(uint idx) { active_index=idx; return 0; }
```

12.15.7. Cleaning up with index_end()

The index_end() method is a counterpart to the index_init() method. The purpose of the index_end() method is to clean up any preparations made by the index_init() method.

If a storage engine does not implement index_init() it does not need to implement index_end().

12.15.8. Implementing the index_read() Method

The index_read() method is used to retrieve a row based on a key:

```
int ha_foo::index_read(byte * buf, const byte * key, uint key_len, enum ha_rkey_function find_flag)
```

The *buf parameter is a byte array that the storage engine populates with the row that matches the index key specified in *key. The key_len parameter indicates the prefix length when matching by prefix, and the find_flag parameter is an enumerator that dictates the search behavior to be used.

The index to be used is previously defined in the index_init() call and is stored in the active_index handler variable.

The following values are allowed for find_flag:

```
HA_READ_AFTER_KEY
HA_READ_BEFORE_KEY
HA_READ_KEY_EXACT
HA_READ_KEY_OR_NEXT
HA_READ_KEY_OR_PREV
HA_READ_PREFIX
HA_READ_PREFIX_LAST
HA_READ_PREFIX_LAST_OR_PREV
```

Storage engines must convert the *key parameter to a storage engine-specific format, use it to find the matching row according to the find_flag, and then populate *buf with the matching row in the MySQL internal row format. For more information on the internal row format, see Section 12.9.6, “Implementing the rnd_next() Method”.

In addition to returning a matching row, the storage engine must also set a cursor to support sequential index reads.

If the *key parameter is null the storage engine should read the first key in the index.

12.15.9. Implementing the index_read_idx() Method

The `index_read_idx()` method is identical to the `index_read()` with the exception that `index_read_idx()` accepts an additional `keynr` parameter:

```
int ha_foo::index_read_idx(byte * buf, uint keynr, const byte * key,
                           uint key_len, enum ha_rkey_function find_flag)
```

The `keynr` parameter specifies the index to be read, as opposed to `index_read` where the index is already set.

As with the `index_read()` method, the storage engine must return the row that matches the key according to the `find_flag` and set a cursor for future reads.

12.15.10. Implementing the `index_next()` Method

The `index_next()` method is used for index scanning:

```
int ha_foo::index_next(byte * buf)
```

The `*buf` parameter is populated with the row that corresponds to the next matching key value according to the internal cursor set by the storage engine during operations such as `index_read()` and `index_first()`.

12.15.11. Implementing the `index_prev()` Method

The `index_prev()` method is used for reverse index scanning:

```
        int ha_foo::index_prev(byte * buf)
```

The `*buf` parameter is populated with the row that corresponds to the previous matching key value according to the internal cursor set by the storage engine during operations such as `index_read()` and `index_last()`.

12.15.12. Implementing the `index_first()` Method

The `index_first()` method is used for index scanning:

```
        int ha_foo::index_first(byte * buf)
```

The `*buf` parameter is populated with the row that corresponds to the first key value in the index.

12.15.13. Implementing the `index_last()` Method

The `index_last()` method is used for reverse index scanning:

```
        int ha_foo::index_last(byte * buf)
```

The `*buf` parameter is populated with the row that corresponds to the last key value in the index.

12.16. Supporting Transactions

This section documents the methods that must be implemented to add support for transactions to a storage engine.

Please note that transaction management can be complicated and involve methods such as row versioning and redo logs, which is beyond the scope of this document. Instead coverage is limited to a description of required methods and not their implementation. For examples of implementation, please see `ha_innodb.cc`.

12.16.1. Transaction Overview

Transactions are not explicitly started on the storage engine level, but are instead implicitly started through calls to either `start_stmt()` or `external_lock()`. If the preceding methods are called and a transaction already exists the transaction is not replaced.

The storage engine stores transaction information in per-connection memory and also registers the transaction in the MySQL server to allow the server to later issue `COMMIT` and `ROLLBACK` operations.

As operations are performed the storage engine will have to implement some form of versioning or logging to permit a rollback of all operations executed within the transaction.

After work is completed, the MySQL server will call either the `commit()` method or the `rollback()` method defined in the storage engine's handlerton.

12.16.2. Starting a Transaction

A transaction is started by the storage engine in response to a call to either the `start_stmt()` or `external_lock()` methods.

If there is no active transaction, the storage engine must start a new transaction and register the transaction with the MySQL server so that `ROLLBACK` or `COMMIT` can later be called.

12.16.2.1. Starting a Transaction from a `start_stmt()` Call

The first method call that can start a transaction is the `start_stmt()` method.

The following example shows how a storage engine could register a transaction:

```
int my_handler::start_stmt(THD *thd, thr_lock_type lock_type)
{
  int error= 0;
  my_txn *txn= (my_txn *) thd->ha_data[my_handler_hton.slot];

  if (txn == NULL)
  {
    thd->ha_data[my_handler_hton.slot]= txn= new my_txn;
  }
  if (txn->stmt == NULL && !(error= txn->tx_begin()))
  {
    txn->stmt= txn->new_savepoint();
    trans_register_ha(thd, FALSE, &my_handler_hton);
  }
  return error;
}
```

`THD` is the current client connection. It holds state relevant data for the current client, such as identity, network connection and other per-connection data.

`thd->ha_data[my_handler_hton.slot]` is a pointer in `thd` to the connection-specific data of this storage engine. In this example we use it to store the transaction context.

An additional example of implementing `start_stmt()` can be found in `ha_innodb.cc`.

12.16.2.2. Starting a Transaction from a `external_lock()` Method

MySQL calls `handler::external_lock()` for every table it is going to use at the beginning of every statement. Thus, if a table is touched for the first time, it implicitly starts a transaction.

Note that because of pre-locking, all tables that can be potentially used between the beginning and the end of a statement are locked before the statement execution begins and `handler::external_lock()` is called for all these tables. That is, if an `INSERT` fires a trigger, which calls a stored procedure, that invokes a stored method, and so forth, all tables used in the trigger, stored procedure, method, etc., are locked in the beginning of the `INSERT`. Additionally, if there's a construct like

```
IF
.. use one table
ELSE
.. use another table
```

both tables will be locked.

Also, if a user calls `LOCK TABLES`, MySQL will call `handler::external_lock` only once. In this case, MySQL will call `handler::start_stmt()` at the beginning of the statement.

The following example shows how a storage engine can start a transaction and take locking requests into account:

```
int my_handler::external_lock(THD *thd, int lock_type)
{
  int error= 0;
  my_txn *txn= (my_txn *) thd->ha_data[my_handler_hton.slot];

  if (txn == NULL)
  {
    thd->ha_data[my_handler_hton.slot]= txn= new my_txn;
  }

  if (lock_type != F_UNLCK)
  {
    bool all_tx= 0;
    if (txn->lock_count == 0)
    {
      txn->lock_count= 1;
      txn->tx_isolation= thd->variables.tx_isolation;

      all_tx= test(thd->options & (OPTION_NOT_AUTOCOMMIT | OPTION_BEGIN | OPTION_TABLE_LOCK));
    }

    if (all_tx)
    {
      txn->tx_begin();
      trans_register_ha(thd, TRUE, &my_handler_hton);
    }
    else
    if (txn->stmt == 0)
    {
      txn->stmt= txn->new_savepoint();
      trans_register_ha(thd, FALSE, &my_handler_hton);
    }
  }
  else
  {
    if (txn->stmt != NULL)
    {
      /* Commit the transaction if we're in auto-commit mode */
      my_handler_commit(thd, FALSE);

      delete txn->stmt; // delete savepoint
      txn->stmt= NULL;
    }
  }

  return error;
}
```

Every storage engine must call `trans_register_ha()` every time it starts a transaction. The `trans_register_ha()` method registers a transaction with the MySQL server to allow for future `COMMIT` and `ROLLBACK` calls.

An additional example of implementing `external_lock()` can be found in `ha_innodb.cc`.

12.16.3. Implementing ROLLBACK

Of the two major transactional operations, `ROLLBACK` is the more complicated to implement. All operations that occurred during the transaction must be reversed so that all rows are unchanged from before the transaction began.

To support `ROLLBACK`, create a method that matches this definition:

```
int  (*rollback)(THD *thd, bool all);
```

The method name is then listed in the `rollback` (thirteenth) entry of the handlerton.

The `THD` parameter is used to identify the transaction that needs to be rolled back, while the `bool all` parameter indicates whether the entire transaction should be rolled back or just the last statement.

Details of implementing a `ROLLBACK` operation will vary by storage engine. Examples can be found in `ha_innodb.cc`.

12.16.4. Implementing COMMIT

During a commit operation, all changes made during a transaction are made permanent and a rollback operation is not possible after that. Depending on the transaction isolation used, this may be the first time such changes are visible to other threads.

To support `COMMIT`, create a method that matches this definition:

```
        int  (*commit)(THD *thd, bool all);
```

The method name is then listed in the `commit` (twelfth) entry of the handlerton.

The `THD` parameter is used to identify the transaction that needs to be committed, while the `bool all` parameter indicates if this is a full transaction commit or just the end of a statement that is part of the transaction.

Details of implementing a `COMMIT` operation will vary by storage engine. Examples can be found in `ha_innodb.cc`.

If the server is in auto-commit mode, the storage engine should automatically commit all read-only statements such as `SELECT`.

In a storage engine, "auto-committing" works by counting locks. Increment the count for every call to `external_lock()`, decrement when `external_lock()` is called with an argument of `F_UNLCK`. When the count drops to zero, trigger a commit.

12.16.5. Adding Support for Savepoints

First, the implementor should know how many bytes are required to store savepoint information. This should be a fixed size, preferably not large as the MySQL server will allocate space to store the savepoint for all storage engines with each named savepoint.

The implementor should store the data in the space preallocated by mysqld - and use the contents from the preallocated space for rollback or release savepoint operations.

When a `COMMIT` or `ROLLBACK` operation occurs (with `bool all` set to `true`), all savepoints are assumed to be released. If the storage engine allocates resources for savepoints, it should free them.

The following handlerton elements need to be implemented to support savepoints (elements 7,9,10,11):

```
uint savepoint_offset;
int  (*savepoint_set)(THD *thd, void *sv);
int  (*savepoint_rollback)(THD *thd, void *sv);
int  (*savepoint_release)(THD *thd, void *sv);
```

12.16.5.1. Specifying the Savepoint Offset

The seventh element of the handlerton is the `savepoint_offset`:

```
uint savepoint_offset;
```

The `savepoint_offset` must be initialized statically to the size of the needed memory to store per-savepoint information.

12.16.5.2. Implementing the `savepoint_set` Method

The `savepoint_set()` method is called whenever a user issues the `SAVEPOINT` statement:

```
int  (*savepoint_set)(THD *thd, void *sv);
```

The `*sv` parameter points to an uninitialized storage area of the size defined by `savepoint_offset`.

When `savepoint_set()` is called, the storage engine needs to store savepoint information into `sv` so that the server can later roll back the transaction to the savepoint or release the savepoint resources.

12.16.5.3. Implementing the `savepoint_rollback() Method`

The `savepoint_rollback()` method is called whenever a user issues the `ROLLBACK TO SAVEPOINT` statement:

```
int  (*savepoint_rollback) (THD *thd, void *sv);
```

The `*sv` parameter points to the storage area that was previously passed to the `savepoint_set()` method.

12.16.5.4. Implementing the `savepoint_release()` Method

The `savepoint_release()` method is called whenever a user issues the `RELEASE SAVEPOINT` statement:

```
int  (*savepoint_release) (THD *thd, void *sv);
```

The `*sv` parameter points to the storage area that was previously passed to the `savepoint_set()` method.

12.17. API Reference

12.17.1. bas_ext

Purpose

Defines the file extensions used by the storage engine.

Synopsis

```
virtual const char ** bas_ext ();
;
```

Description

This is the `bas_ext` method. It is called to provide the MySQL server with a list of file extensions used by the storage engine. The list returned is a null-terminated string array.

By providing a list of extensions, storage engines can in many cases omit the `delete_table()` method as the MySQL server will close all references to the table and delete all files with the specified extension.

Parameters

There are no parameters for this method.

Return Values

- Return value is a null-terminated string array of storage engine extensions. The following is an example from the `CSV` engine:

  ```
  static const char *ha_tina_exts[] =
   {
     ".CSV",
     NullS
   };
  ```

Usage

```
static const char *ha_tina_exts[] =
 {
   ".CSV",
   NullS
 };

const char **ha_tina::bas_ext() const
 {
   return ha_tina_exts;
 }
```

Default Implementation

```
static const char *ha_example_exts[] = {
   NullS
 };

const char **ha_example::bas_ext() const
 {
   return ha_example_exts;
 }
```

12.17.2. close

Purpose

Closes an open table.

Synopsis

```
virtual int close (void);
void ;
```

Description

This is the `close` method.

Closes a table. A good time to free any resources that we have allocated.

Called from sql_base.cc, sql_select.cc, and table.cc. In sql_select.cc it is only used to close up temporary tables or during the process where a temporary table is converted over to being a `MyISAM` table. For sql_base.cc look at close_data_tables().

Parameters

- `void`

Return Values

There are no return values.

Usage

Example from the `CSV` engine:

```
int ha_example::close(void)
{
  DBUG_ENTER("ha_example::close");
  DBUG_RETURN(free_share(share));
}
```

12.17.3. create

Purpose

Creates a new table.

Synopsis

```
virtual int create (name, form, info);
const char *name ;
TABLE *form ;
HA_CREATE_INFO *info ;
```

Description

This is the `create` method.

`create()` is called to create a table. The variable name will have the name of the table. When `create()` is called you do not need to open the table. Also, the `.frm` file will have already been created so `adjusting create_info` is not recommended.

Called from `handler.cc` by `ha_create_table()`.

Parameters

- `name`
- `form`
- `info`

Return Values

There are no return values.

Usage

Example from the `CSV` storage engine:

```
int ha_tina::create(const char *name, TABLE *table_arg,
                    HA_CREATE_INFO *create_info)
{
  char name_buff[FN_REFLEN];
  File create_file;
  DBUG_ENTER("ha_tina::create");

  if ((create_file= my_create(fn_format(name_buff, name, "", ".CSV",
                                        MY_REPLACE_EXT|MY_UNPACK_FILENAME),0,
                              O_RDWR | O_TRUNC,MYF(MY_WME))) < 0)
    DBUG_RETURN(-1);

  my_close(create_file,MYF(0));

  DBUG_RETURN(0);
}
```

12.17.4. delete_row

Purpose

Deletes a row.

Synopsis

```
virtual int delete_row (buf);
const byte *buf ;
```

Description

This is the `delete_row` method.

`buf` will contain a copy of the row to be deleted. The server will call this right after the current row has been called (from either a previous `rnd_next()` or index call). If you keep a pointer to the last row or can access a primary key it will make doing the deletion quite a bit easier. Keep in mind that the server

does not guarantee consecutive deletions. ORDER BY clauses can be used.

Called in sql_acl.cc and sql_udf.cc to manage internal table information. Called in sql_delete.cc, sql_insert.cc, and sql_select.cc. In sql_select it is used for removing duplicates, while in insert it is used for REPLACE calls.

Parameters

- buf

Return Values

There are no return values.

Usage

```

```

Default Implementation

```
{ return  HA_ERR_WRONG_COMMAND; }
```

12.17.5. delete_table

Purpose

Delete all files with extension from bas_ext().

Synopsis

```
virtual int delete_table (name);
const char *name ;
```

Description

This is the delete_table method.

Used to delete a table. By the time delete_table() has been called all opened references to this table will have been closed (and your globally shared references released). The variable name will be the name of the table. You will need to remove any files you have created at this point.

If you do not implement this, the default delete_table() is called from handler.cc, and it will delete all files with the file extensions returned by bas_ext(). We assume that the handler may return more extensions than were actually used for the file.

Called from handler.cc by delete_table and ha_create_table(). Only used during create if the table_flag HA_DROP_BEFORE_CREATE was specified for the storage engine.

Parameters

- name: Base name of table

Return Values

- `0` if we successfully deleted at least one file from `base_ext` and didn't get any other errors than `ENOENT`
- `#`: Error

Usage

Most storage engines can omit implementing this method.

12.17.6. external_lock

Purpose

Handles table locking for transactions.

Synopsis

```
virtual int external_lock (thd, lock_type);
THD *thd ;
intlock_type ;
```

Description

This is the `external_lock` method.

The "locking methods for mysql" section in `lock.cc` has additional comments on this topic that may be useful to read.

This creates a lock on the table. If you are implementing a storage engine that can handle transactions, look at `ha_innodb.cc` to see how you will want to go about doing this. Otherwise you should consider calling `flock()` here.

Called from `lock.cc` by `lock_external()` and `unlock_external()`. Also called from `sql_table.cc` by `copy_data_between_tables()`.

Parameters

- `thd`
- `lock_type`

Return Values

There are no return values.

Default Implementation

```
{ return 0; }
```

12.17.7. extra

Purpose

Passes hints from the server to the storage engine.

Synopsis

```
virtual int extra (operation);
enum ha_extra_functionoperation ;
```

Description

This is the `extra` method.

`extra()` is called whenever the server wishes to send a hint to the storage engine. The `MyISAM` engine implements the most hints. `ha_innodb.cc` has the most exhaustive list of these hints.

Parameters

- `operation`

Return Values

There are no return values.

Usage

Most storage engines will simply return `0`.

```
{ return 0; }
```

Default Implementation

By default your storage engine can opt to implement none of the hints.

```
{ return 0; }
```

12.17.8. index_end

Purpose

Indicates end of index scan, clean up any resources used.

Synopsis

```
virtual int index_end ();
;
```

Description

This is the `index_end` method. Generally it is used as a counterpart to the `index_init` method, cleaning up any resources allocated for index scanning.

Parameters

This method has no parameters.

Return Values

This method has no return values.

Usage

Clean up all resources allocated, return 0.

Default Implementation

```
          { active_index=MAX_KEY; return 0; }
```

12.17.9. index_first

Purpose

Retrieve first row in index and return.

Synopsis

```
virtual int index_first (buf);
byte *buf ;
```

Description

This is the `index_first` method.

`index_first()` asks for the first key in the index.

Called from opt_range.cc, opt_sum.cc, sql_handler.cc, and sql_select.cc.

Parameters

- `buf` - byte array to be populated with row.

Return Values

There are no return values.

Usage

Implementation depends on indexing method used.

Default Implementation

```
        { return  HA_ERR_WRONG_COMMAND; }
```

12.17.10. index_init

Purpose

Signals the storage engine that an index scan is about to occur. Storage engine should allocate any resources needed.

Synopsis

```
virtual int index_init (idx, sorted);
uint idx ;
bool sorted ;
```

Description

This is the `index_init` method. This method is called before an index scan, allowing the storage engine to allocate resources and make preparations.

Parameters

- `idx`
- `sorted`

Return Values

-

Usage

This method can typically just return 0 if there is no preparation needed.

Default Implementation

```
        { active_index=idx; return 0; }
```

12.17.11. index_last

Purpose

Return the last row in the index.

Synopsis

```
virtual int index_last (buf);
byte *buf ;
```

Description

This is the `index_last` method.

`index_last()` asks for the last key in the index.

Called from opt_range.cc, opt_sum.cc, sql_handler.cc, and sql_select.cc.

Parameters

- `buf` - byte array to be populated with matching row.

Return Values

This method has no return values.

Usage

Advance to last row in index and return row in buffer.

Default Implementation

```
          { return  HA_ERR_WRONG_COMMAND; }
```

12.17.12. index_next

Purpose

Return next row in index.

Synopsis

```
virtual int index_next (buf);
byte *buf ;
```

Description

This is the `index_next` method.

Used to read forward through the index.

Parameters

- `buf`

Return Values

This method has no return values.

Usage

Advance to next row in index using pointer or cursor, return row in buffer.

Default Implementation

```
        { return  HA_ERR_WRONG_COMMAND; }
```

12.17.13. index_prev

Purpose

Advance to previous row in index.

Synopsis

```
virtual int index_prev (buf);
byte *buf ;
```

Description

This is the `index_prev` method.

Used to read backward through the index.

Parameters

- `buf`

Return Values

This method has no return values.

Usage

Move to previous row in index, retun in buffer.

Default Implementation

```
        { return  HA_ERR_WRONG_COMMAND; }
```

12.17.14. index_read_idx

Purpose

Find a row based on a key and return.

Synopsis

```
virtual int index_read_idx (buf, index, key, key_len, find_flag);
byte *buf ;
```

```
uintindex ;
const byte *key ;
uintkey_len ;
enum ha_rkey_functionfind_flag ;
```

Description

This is the `index_read_idx` method.

Positions an index cursor to the index specified in key. Fetches the row if any. This is only used to read whole keys.

Parameters

- `buf`
- `index`
- `key`
- `key_len`
- `find_flag`

Return Values

This method has no return values.

Usage

Locate the row that matches the key passed and return it in the buffer provided.

12.17.15. index_read

Purpose

Find a row based on a key and return.

Synopsis

```
virtual int index_read (buf, key, key_len, find_flag);
byte *buf ;
const byte *key ;
uintkey_len ;
enum ha_rkey_functionfind_flag ;
```

Description

This is the `index_read` method.

Positions an index cursor to the index specified in the handle. Fetches the row if available. If the key value is null, begin at the first key of the index.

Parameters

- `buf`
- `key`
- `key_len`
- `find_flag`

Return Values

This method has no return values.

Usage

Default Implementation

```
          { return  HA_ERR_WRONG_COMMAND; }
```

12.17.16. info

Purpose

Prompts storage engine to report statistics.

Synopsis

```
virtual void info (uint);
uint ;
```

Description

This is the `info` method.

::info() is used to return information to the optimizer. Currently, this table handler doesn't implement most of the fields really needed. SHOW also makes use of this data Another note, you will probably want to have the following in your code: if (records < 2) records = 2; The reason is that the server will optimize for cases of only a single record. If in a table scan you don't know the number of records it will probably be better to set records to two so you can return as many records as you need. Along with records a few more variables you may wish to set are: records deleted data_file_length index_file_length delete_length check_time See public variables in handler.h for more information.

Called in: filesort.cc ha_heap.cc item_sum.cc opt_sum.cc sql_delete.cc sql_delete.cc sql_derived.cc sql_select.cc sql_select.cc sql_select.cc sql_select.cc sql_select.cc sql_show.cc sql_show.cc sql_show.cc sql_show.cc sql_table.cc sql_union.cc sql_update.cc

Parameters

- `uint`

Return Values

There are no return values.

Usage

This example is from the CSV storage engine:

```
void ha_tina::info(uint flag)
{
  DBUG_ENTER("ha_tina::info");
  /* This is a lie, but you don't want the optimizer to see zero or 1 */
  if (records < 2)
    records= 2;
  DBUG_VOID_RETURN;
}
```

12.17.17. open

Purpose

Opens a table.

Synopsis

```
virtual int open (name, mode, test_if_locked);
const char *name ;
intmode ;
uinttest_if_locked ;
```

Description

This is the `open` method.

Used for opening tables. The name will be the name of the file. A table is opened when it needs to be opened. For instance when a request comes in for a select on the table (tables are not open and closed for each request, they are cached).

Called from handler.cc by handler::ha_open(). The server opens all tables by calling ha_open() which then calls the handler specific open().

A handler object is opened as part of its initialization and before being used for normal queries (not before meta-data changes always.) If the object was opened it will also be closed before being deleted.

This is the `open` method. `open` is called to open a database table.

The first parameter is the name of the table to be opened. The second parameter determines what file to open or what operation to take. The values are defined in `handler.h` and are copied here for your convenience:

```
        #define HA_OPEN_KEYFILE           1
        #define HA_OPEN_RNDFILE           2
        #define HA_GET_INDEX             4
        #define HA_GET_INFO               8     /* do a ha_info() after open */
        #define HA_READ_ONLY             16   /* File opened as readonly */
        #define HA_TRY_READ_ONLY   32    /* Try readonly if can't open with read and write */
        #define HA_WAIT_IF_LOCKED   64   /* Wait if locked on open */
        #define HA_ABORT_IF_LOCKED 128   /* skip if locked on open.*/
        #define HA_BLOCK_LOCK             256   /* unlock when reading some records */
        #define HA_OPEN_TEMPORARY   512
```

The final option dictates whether the handler should check for a lock on the table before opening it.

Typically your storage engine will need to implement some form of shared access control to prevent file corruption is a multi-threaded environment. For an example of how to implement file locking, see the `get_share()` and `free_share()` methods of `sql/examples/ha_tina.cc`.

Parameters

- `name`
- `mode`
- `test_if_locked`

Return Values

There are no return values.

Usage

This example is from the `CSV` storage engine:

```
        int ha_tina::open(const char *name, int mode, uint test_if_locked)
        {
        DBUG_ENTER("ha_tina::open");

        if (!(share= get_share(name, table)))
        DBUG_RETURN(1);
        thr_lock_data_init(&share->lock,&lock,NULL);
        ref_length=sizeof(off_t);

        DBUG_RETURN(0);
        }
```

12.17.18. position

Purpose

Provide the MySQL server with position/offset information for last-read row.

Synopsis

```
virtual void position (record);
const byte *record ;
```

Description

This is the `position` method.

position() is called after each call to rnd_next() if the data needs to be ordered. You can do something like the following to store the position: my_store_ptr(ref, ref_length, current_position);

The server uses ref to store data. ref_length in the above case is the size needed to store current_position. ref is just a byte array that the server will maintain. If you are using offsets to mark rows, then current_position should be the offset. If it is a primary key, then it needs to be a primary key.

Called from filesort.cc, sql_select.cc, sql_delete.cc and sql_update.cc.

Parameters

- `record`

Return Values

This method has no return values.

Usage

Return offset or retrieval key information for last row.

12.17.19. records_in_range

Purpose

For the given range how many records are estimated to be in this range.

Synopsis

```
virtual ha_rows records_in_range (inx, min_key, max_key);
uintinx ;
key_range *min_key ;
key_range *max_key ;
```

Description

This is the `records_in_range` method.

Given a starting key, and an ending key estimate the number of rows that will exist between the two. end_key may be empty which in case determine if start_key matches any rows.

Used by optimizer to calculate cost of using a particular index.

Called from opt_range.cc by check_quick_keys().

Parameters

- `inx`
- `min_key`
- `max_key`

Return Values

Return the approxamite number of rows.

Usage

Determine an approxamite count of the rows between the key values and return.

Default Implementation

```
            { return (ha_rows) 10; }
```

12.17.20. rnd_init

Purpose

Initializes a handler for table scanning.

Synopsis

```
virtual int rnd_init (scan);
boolscan ;
```

Description

This is the `rnd_init` method.

rnd_init() is called when the system wants the storage engine to do a table scan.

Unlike index_init(), rnd_init() can be called two times without rnd_end() in between (it only makes sense if scan=1). then the second call should prepare for the new table scan (e.g if rnd_init allocates the cursor, second call should position it to the start of the table, no need to deallocate and allocate it again

Called from filesort.cc, records.cc, sql_handler.cc, sql_select.cc, sql_table.cc, and sql_update.cc.

Parameters

- `scan`

Return Values

There are no return values.

Usage

This example is from the `CSV` storage engine:

```
int ha_tina::rnd_init(bool scan)
{
  DBUG_ENTER("ha_tina::rnd_init");

  current_position= next_position= 0;
  records= 0;
  chain_ptr= chain;
  DBUG_RETURN(0);
}
```

12.17.21. rnd_next

Purpose

Reads the next row from a table and returns it to the server.

Synopsis

```
virtual int rnd_next (buf);
byte *buf ;
```

Description

This is the `rnd_next` method.

This is called for each row of the table scan. When you run out of records you should return HA_ERR_END_OF_FILE. Fill buff up with the row information. The Field structure for the table is the key to getting data into buf in a manner that will allow the server to understand it.

Called from filesort.cc, records.cc, sql_handler.cc, sql_select.cc, sql_table.cc, and sql_update.cc.

Parameters

- `buf`

Return Values

There are no return values.

Usage

This example is from the `ARCHIVE` storage engine:

```
int ha_archive::rnd_next(byte *buf)
{
  int rc;
  DBUG_ENTER("ha_archive::rnd_next");

  if (share->crashed)
      DBUG_RETURN(HA_ERR_CRASHED_ON_USAGE);

  if (!scan_rows)
    DBUG_RETURN(HA_ERR_END_OF_FILE);
  scan_rows--;

  statistic_increment(table->in_use->status_var.ha_read_rnd_next_count,
                      &LOCK_status);
    current_position= gztell(archive);
  rc= get_row(archive, buf);

  if (rc != HA_ERR_END_OF_FILE)
    records++;

  DBUG_RETURN(rc);
}
```

12.17.22. rnd_pos

Purpose

Return row based on position.

Synopsis

```
virtual int rnd_pos (buf, pos);
byte *buf ;
byte *pos ;
```

Description

This is the `rnd_pos` method.

Used for finding row previously marked with position. This is useful for large sorts.

This is like rnd_next, but you are given a position to use to determine the row. The position will be of the type that you stored in ref. You can use ha_get_ptr(pos,ref_length) to retrieve whatever key or position you saved when position() was called. Called from filesort.cc records.cc sql_insert.cc sql_select.cc sql_update.cc.

Parameters

- `buf`
- `pos`

Return Values

This method has no return values.

Usage

Locate row based on position value and return in buffer provided.

12.17.23. start_stmt

Purpose

Called at the beginning of a statement for transaction purposes.

Synopsis

```
virtual int start_stmt (thd, lock_type);
THD *thd ;
thr_lock_typelock_type ;
```

Description

This is the `start_stmt` method.

When table is locked a statement is started by calling start_stmt instead of external_lock

Parameters

- `thd`

- `lock_type`

Return Values

This method has no return values.

Usage

Make any preparations needed for a transaction start (if there is no current running transaction).

Default Implementation

```
            {return 0;}
```

12.17.24. store_lock

Purpose

Creates and releases table locks.

Synopsis

```
virtual THR_LOCK_DATA ** store_lock (thd, to, lock_type);
THD *thd ;
THR_LOCK_DATA **to ;
enum thr_lock_typelock_type ;
```

Description

This is the `store_lock` method.

The idea with handler::store_lock() is the following:

The statement decided which locks we should need for the table for updates/deletes/inserts we get WRITE locks, for SELECT... we get read locks.

Before adding the lock into the table lock handler `mysqld` calls store lock with the requested locks. Store lock can modify the lock level, e.g. change blocking write lock to non-blocking, ignore the lock (if we don't want to use MySQL table locks at all) or add locks for many tables (like we do when we are using a MERGE handler).

When releasing locks, store_lock() are also called. In this case one usually doesn't have to do anything.

If the argument of store_lock is TL_IGNORE, it means that MySQL requests the handler to store the same lock level as the last time.

Called from lock.cc by get_lock_data().

Parameters

- `thd`

- `to`
- `lock_type`

Return Values

There are no return values.

Usage

The following example is from the `ARCHIVE` storage engine:

```
/*
  Below is an example of how to setup row level locking.
*/
THR_LOCK_DATA **ha_archive::store_lock(THD *thd,
                                       THR_LOCK_DATA **to,
                                       enum thr_lock_type lock_type)
{
  if (lock_type == TL_WRITE_DELAYED)
    delayed_insert= TRUE;
  else
    delayed_insert= FALSE;

  if (lock_type != TL_IGNORE && lock.type == TL_UNLOCK)
  {
    /*
      Here is where we get into the guts of a row level lock.
      If TL_UNLOCK is set
      If we are not doing a LOCK TABLE or DISCARD/IMPORT
      TABLESPACE, then allow multiple writers
    */

    if ((lock_type >= TL_WRITE_CONCURRENT_INSERT &&
         lock_type <= TL_WRITE) && !thd->in_lock_tables
        && !thd->tablespace_op)
      lock_type = TL_WRITE_ALLOW_WRITE;

    /*
      In queries of type INSERT INTO t1 SELECT ... FROM t2 ...
      MySQL would use the lock TL_READ_NO_INSERT on t2, and that
      would conflict with TL_WRITE_ALLOW_WRITE, blocking all inserts
      to t2. Convert the lock to a normal read lock to allow
      concurrent inserts to t2.
    */

    if (lock_type == TL_READ_NO_INSERT && !thd->in_lock_tables)
      lock_type = TL_READ;

    lock.type=lock_type;
  }

  *to++= &lock;

  return to;
}
```

The following is the minimal implementation, for a storage engine that does not need to downgrade locks:

```
THR_LOCK_DATA **ha_tina::store_lock(THD *thd,
                                    THR_LOCK_DATA **to,
                                    enum thr_lock_type lock_type)
{
   /* Note that if the lock type is TL_IGNORE we don't update lock.type,
      preserving the previous lock level */
   if (lock_type != TL_IGNORE && lock.type == TL_UNLOCK)
     lock.type=lock_type;
    /* the heart of the store_lock() method and it's main purpose -
      storing the (possibly changed) lock level into the provided
      memory */
     *to++= &lock;
    return to;
}
```

See also `ha_myisammrg::store_lock()` for more complex implementation

12.17.25. update_row

Purpose

Updates the contents of an existing row.

Synopsis

```
virtual int update_row (old_data, new_data);
const byte *old_data ;
byte *new_data ;
```

Description

This is the `update_row` method.

old_data will have the previous row record in it, while new_data will have the newest data in it.

The server can do updates based on ordering if an ORDER BY clause was used. Consecutive ordering is not guaranteed.

Currently, new_data will not have an updated auto_increament record, or and updated timestamp field. You can do these for example by doing these: if (table->timestamp_field_type & TIMESTAMP_AUTO_SET_ON_UPDATE) table->timestamp_field->set_time(); if (table->next_number_field && record == table->record[0]) update_auto_increment();

Called from sql_select.cc, sql_acl.cc, sql_update.cc, and sql_insert.cc.

Parameters

- `old_data`
- `new_data`

Return Values

There are no return values.

Usage

```

```

Default Implementation

```
          { return  HA_ERR_WRONG_COMMAND; }
```

12.17.26. write_row

Purpose

Adds a new row to a table.

Synopsis

```
virtual int write_row (buf);
byte *buf ;
```

Description

This is the `write_row` method.

`write_row()` inserts a row. No `extra()` hint is given currently if a bulk load is happening. buf is a byte array of data with a size of table->s->reclength

You can use the field information to extract the data from the native byte array type. Example of this would be: for (Field **field=table->field ; *field ; field++) { ... }

BLOBs must be handled specially:

```
for (ptr= table->s->blob_field, end= ptr + table->s->blob_fields ; ptr != end ; ptr++)
  {
        char *data_ptr;
        uint32 size= ((Field_blob*)table->field[*ptr])->get_length();
        ((Field_blob*)table->field[*ptr])->get_ptr(&data_ptr);
        ...
  }
```

See ha_tina.cc for an example of extracting all of the data as strings.

See the note for `update_row()` on auto_increments and timestamps. This case also applied to write_row().

Called from item_sum.cc, item_sum.cc, sql_acl.cc, sql_insert.cc, sql_insert.cc, sql_select.cc, sql_table.cc, sql_udf.cc, and sql_update.cc.

Parameters

- `buf` byte array of data

Return Values

There are no return values.

Usage

```

```

Default Implementation

```
            { return  HA_ERR_WRONG_COMMAND; }
```

Chapter 13. Error Messages

This chapter describes how error messages are defined and how to add the capability of generating error messages to a table handler.

13.1. Adding New Error Messages to MySQL

The procedure for adding error messages depends on which version of MySQL you are using:

- Before MySQL 5.0.3, error messages are stored in `errmsg.txt` files in the language directories under `sql/share`. That is, the files have names like `czech/errmsg.txt`, `danish/errmsg.txt`, and so forth, and each one is language-specific. Each of these language-specific files must contain a line for each error message, so adding a new message involves adding a line to the `errmsg.txt` file for every language. The procedure involves adding the English message to the `english/errmsg.txt` file and running a script that adds the message to the other language-specific files. Translators may translate the message in other `errmsg.txt` files later.
- Beginning with MySQL 5.0.3, error messages are stored in a single `errmsg.txt` file in the `sql/share` directory, and it contains the error messages for all languages. The messages are grouped by error symbol. For each symbol, there must be an English message, and messages can be present for other languages as well. If there is no message for a given language, the English version is used.

For all versions, the `comp_err` program compiles the text error message file or files into language-specific `errmsg.sys` files that each are located in the appropriate language directory under `sql/share`. In MySQL 5.0.3 and up, `comp_err` also generates a number of header files in the `include` directory. The MySQL build process runs `comp_err` automatically.

Note: You should observe some general considerations regarding error messages that apply no matter your version of MySQL:

- The maximum error message length is `MYSQL_ERRMSG_SIZE` = 512. Ensure by using constructs such as `"%s-.64s"` that there are no buffer overflows!
- You should *never* add new parameters (such as %s) to existing error messages. Error messages are always supposed to be backward compatible. If a parameter is added, older servers will crash.

For versions of MySQL older than 5.0.3, use the following procedure to add new error messages:

1. Open the file `sql/share/english/errmsg.txt` in an editor.
2. Add new error messages at the end of this file. Each message should be on a separate line, and it must be quoted within double quote ('`"`') characters. By convention, every message line except the last should end with a comma ('`,`') following the second double quote.
3. For each new error message, add a `#define` line to the `include/mysqld_error.h` file before the last line (`#define ER_ERROR_MESSAGES`).
4. Adjust the value of `ER_ERROR_MESSAGES` to the new number of error messages.
5. Add the defined error symbols to `include/sql_state.h`. This file contains the SQL states for the error messages. If the new errors don't have SQL states, add a comment instead. 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`.

6. Go to the `sql` directory in a terminal window and type `./add_errmsg` *`N`*. This will copy the last *`N`* error messages from `share/english.txt` to all the other language files in `share/`.

7. Translate the error message for those languages that you know by editing the files `share/`*`language`*`/errmsg.txt`.

8. Make a full build (`configure` + `make`). A `make all` is insufficient to build the `sql/share/*/errmsg.sys` files.

For MySQL 5.0.3 and up, the procedure for adding error messages is less tedious. You need edit only a single message text file, and it's not necessary to edit `*.h` header files. Instead, `comp_err` generates the header files for you based on the contents of the message text file.

The `errmsg.txt` file begins with some lines that define general characteristics of error messages, followed by sections for particular messages. The following example shows a partial listing of an `errmsg.txt` file. (The `languages` line is wrapped here; it must be given all on one line.)

```
languages czech=cze latin2, danish=dan latin1, dutch=nla latin1,
english=eng latin1, estonian=est latin7, french=fre latin1, german=ger
latin1, greek=greek greek, hungarian=hun latin2, italian=ita latin1,
japanese=jpn ujis, japanese-sjis=jps sjis, korean=kor euckr,
norwegian-ny=norwegian-ny latin1, norwegian=nor latin1, polish=pol
latin2, portuguese=por latin1, romanian=rum latin2, russian=rus
koi8r, serbian=serbian cp1250, slovak=slo latin2, spanish=spa latin1,
swedish=swe latin1, ukrainian=ukr koi8u;

default-language eng

start-error-number 1000

ER_HASHCHK
        eng "hashchk"
ER_NISAMCHK
        eng "isamchk"
ER_NO
        cze "NE"
        dan "NEJ"
        nla "NEE"
        eng "NO"
        est "EI"
        ...
```

Indentation is significant. Unless otherwise specified, leading whitespace should not be used.

The “grammar” of the `errmsg.txt` file looks like this:

```
languages langspec [, langspec] ... ;

start-error-number number

default-language langcode

error-message-section
error-message-section
...
```

The `languages` line lists the languages for which language-specific `errmsg.sys` files should be generated. A language specification *`langspec`* in the `languages` line has this syntax:

```
langspec: langname=langcode langcharset
```

`langname` is the long language name, *`langcode`* is the short language code, and *`langcharset`* is the character set to use for error messages in the language.

The `default-language` line specifies the short language code for the default language. (If there is no translation into a given language for a given error message, the message from the default language will be used.)

The `start-error-number` line indicates the number to be assigned to the first error message. Messages that follow the first one are numbered consecutively from this value.

Each *`error-message-section`* begins with a line that lists an error (or warning) symbol, optionally followed by one or two SQLSTATE values. The error symbol must begin with `ER_` for an error or `WARN_` for a warning. Lines following the error symbol line provide language-specific error messages that correspond to the error symbol. Each message line consists of a tab, a short language code, a space, and the text of the error message within double quote ('`"`') characters. Presumably, there must be a message in the default language. There may be message translations for other languages. Order of message lines within a section does not matter. If no translation is given for a given language, the default language message will be used. The following example defines several language translations for the `ER_BAD_FIELD_ERROR` symbol:

```
ER_BAD_FIELD_ERROR 42S22 S0022
        dan "Ukendt kolonne '%-.64s' i tabel %s"
        nla "Onbekende kolom '%-.64s' in %s"
        eng "Unknown column '%-.64s' in '%-.64s'"
        est "Tundmatu tulp '%-.64s' '%-.64s'-s"
        fre "Champ '%-.64s' inconnu dans %s"
        ger "Unbekanntes Tabellenfeld '%-.64s' in %-.64s"
```

In the preceding example, two SQLSTATE values are given following the error symbol (`42S22`, `S0022`). Internally (in `sql/sql_state.c`), these are known as `odbc_state` and `jdbc_state`. Currently, only the first appears ever to be used.

Message strings for a given language must be written in the character set indicated for that language in the `languages` line. For example, the language information for Japanese in that line is `japanese=jpn ujis`, so messages with a language code of `jpn` must be written in the `ujis` character set. You might need to be careful about the editor you use for editing the `errmsg.txt` file. For example, there is a report that using `Emacs` will mangle the file, whereas `vi` will not.

Within a message string, C-style escape sequences are allowed:

```
\\ → \
\" → "
\n → newline
\N → N, where N is an octal number
\X → X, for any other X
```

A line beginning with a '`#`' character is taken as a comment. Comments and blank lines are ignored.

Use the following procedure to add new error messages:

1. To add a new language translation for an existing error message, find the section for the appropriate error symbol. Then add a new message line to the section. For example:

   Before:

   ```
   ER_UNKNOWN_COLLATION
           eng "Unknown collation: '%-.64s'"
           ger "Unbekannte Kollation: '%-.64s'"
           por "Collation desconhecida: '%-.64s'"
   ```

After (with a new Spanish translation):

```
ER_UNKNOWN_COLLATION
        eng "Unknown collation: '%-.64s'"
        ger "Unbekannte Kollation: '%-.64s'"
        por "Collation desconhecida: '%-.64s'"
        spa "Collation desconocida: '%-.64s'"
```

2. To add an entirely new error message, go to the end of the `errmsg.txt` file. Add a new error symbol line, followed by a message line for the default language, and message lines for any translations that you can supply.

3. Make a full build (`configure` + `make`). A `make all` is insufficient to build the `sql/share/*/errmsg.sys` files.

   `comp_err` will generate the `errmsg.sys` files, as well as the header files `mysqld_error.h`, `mysqld_ername.h`, and `sql_state.h` in the `include` directory.

Be aware that if you make a mistake editing a message text file, `comp_err` prints a cryptic error message and gives you no other feedback. For example, it does not print the input line number where it found a problem. It's up to you to figure this out and correct the file. Perhaps that is not a serious difficulty: `errmsg.txt` tends to grow by gradual accretion, so if an error occurs when `comp_err` processes it, the problem is likely due to whatever change you just made.

13.2. Adding Storage Engine Error Messages

To add error messages for table handlers, the following example may be helpful.

Purpose: Implement the `handler::get_error_message` function as `ha_federated::get_error_message` to return the handler-specific error message.

Example:

1. When an error occurs you return an error code. (It should not be in the range of those that `HA_ERR` uses, which currently is 120-159.)

2. When `handler::print_error` is called to convert the handler error code to a MySQL error code, it will enter the default label of the `switch(error)` statement:

```
handler.cc:1721
  default:
    {
      /* The error was "unknown" to this function.
     Ask handler if it has got a message for this error */
      bool temporary= FALSE;
      String str;
      temporary= get_error_message(error, &str);
      if (!str.is_empty())
      {
    const char* engine= table_type();
    if (temporary)
      my_error(ER_GET_TEMPORARY_ERRMSG, MYF(0), error, str.ptr(), engine);
    else
      my_error(ER_GET_ERRMSG, MYF(0), error, str.ptr(), engine);
      }
      else
    my_error(ER_GET_ERRNO,errflag,error);
      DBUG_VOID_RETURN;
    }
  }
```

3. Thus the `handler::get_error_message` is called and you can return the handler-specific error message, which is either a static error message that you retrieve from an error/string array, or a a dynamic one that you format when the error occurs.

When you have returned the error message it will be passed to MySQL and formatted as `Got error %d '%-.100s' from %s`. For example:

```
Got error 788  'Could not connect to remote server fed.bb.pl' from FEDERATED
```

The `Got error %d` part will be returned in the user's selected language, but the handler-specific one will use English (unless the handler supports returning the handler error message in the user's selected language).

Appendix A. 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.

A.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
- client — Client library
- cmd-line-utils — Command-line utilities (libedit and readline)
- config — Some files used during build
- dbug — Fred Fish's dbug library
- Docs — documentation files
- 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.)
- ndb — MySQL Cluster
- 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
- server-tools — instance manager
- sql — Programs for handling SQL commands; the "core" of MySQL
- sql-bench — The MySQL benchmarks
- sql-common — Some .c files related to sql directory
- 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

A.1.1. The `bdb` Directory

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.

A.1.2. The `BitKeeper` Directory

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.

A.1.3. The `BUILD` Directory

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

A.1.4. The `client` Directory

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.cc --- 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

A.1.5. The `config` Directory

Macros for use during build.

There is a single subdirectory: `\ac-macros`. All the files in it have the extension .m4, which is a normal expectation of the GNU autoconf tool.

A.1.6. The `cmd-line-utils` Directory

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
- unvis.c --- reverse effect of vis.c
- vi.c --- commands used when in the vi (editor) mode
- vis.c --- encode characters

A.1.7. The `dbug` Directory

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
- my_main.c --- MySQL-specific main.c variant
- sanity.c --- Declaration of a variable

A.1.8. The `Docs` Directory

With the BitKeeper downloads, /Docs is nearly empty. Binary and source distributions include some pre-formatted documentation files, such as the MySQL Reference manual in Info format (for Unix) or CHM format (for Windows).

A.1.9. The `extra` Directory

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:

- charset2html.c --- checks your browser's character set

- comp_err.c --- makes error-message files from a multi-language source
- 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

A.1.10. The `heap` Directory

The HEAP (MEMORY) 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\mi_log.c equivalent (to log statements).

A.1.11. The `include` Directory

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 55 *.h (header) files.

A.1.12. The `innobase` Directory

The Innobase (InnoDB) table handler.

A full description of these files can be found elsewhere in this document.

A.1.13. The `libmysql` Directory

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

A.1.14. The `libmysql_r` Directory

The MySQL Library, Part 2.

There is only one file here, used to build a thread-safe libmysql library:

- makefile.am

A.1.15. The `libmysqld` Directory

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_sql.c --- Emulate the vio directory's communication buffer

A.1.16. The `man` Directory

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.

A.1.17. The `myisam` Directory

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_keycache.c --- for handling key caches
- 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_preload.c --- preload indexes into key cache
- 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

A.1.18. The `myisammrg` Directory

`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 22 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

A.1.19. The `mysql-test` Directory

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
- \ndb --- for MySQL Cluster tsts
- \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 400 `*.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

A.1.20. The `mysys` Directory

MySQL system library. Low level routines for file access and so on.

There are 125 *.c programs in this directory:

- array.c --- Dynamic array handling
- charset.c --- Using dynamic character sets, set default character set, ...
- charset-def.c --- Inclcude character sets in the client using
- checksum.c --- Calculate checksum for a memory block, used for pack_isam
- default.c --- Find defaults from *.cnf or *.ini files
- default_modify.c --- edit option file
- 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_keycaches.c --- Handling of multiple key caches
- 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_access.c --- Check if file or path is accessible
- 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_crc32.c --- Include \zlib\crc32.c
- 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_file.c --- See how many open files we want
- my_fopen.c --- File open
- my_fstream.c --- Streaming file read/write
- my_gethostbyname.c --- Thread-safe version of standard net gethostbyname() func
- my_gethwaddr.c --- Get hardware address for an interface
- my_getopt.c --- Find out what options are in effect
- my_getsystime.c --- Time-of-day functions, portably
- my_getwd.c --- Get working directory
- my_handler.c --- Compare two keys in various possible formats
- my_init.c --- Initialize variables and functions in the mysys library
- my_largepage.c --- Gets the size of large pages from the OS
- 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_mmap.c --- Memory mapping
- 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_sync.c --- Sync data in file to disk
- my_thr_init.c --- initialize/allocate "all mysys & debug thread variables"
- my_wincond.c --- Windows-specific: emulate Posix conditions
- my_windac.c --- Windows NT/2000 discretionary access control functions
- 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."
```

A.1.21. The ndb Directory

The ndb (MySQL Cluster) source code.

MySQL's shared-nothing in-memory feature is practically a DBMS by itself. We generally use the term "ndb" when referring to the storage engine, and the term "MySQL Cluster" when referring to the combination of the storage engine and the rest of the MySQL facilities.

The sub-directories within ndb are:

- bin --- Two script files
- config --- Files needed for building
- demos --- Demonstration scripts

- docs --- A doxygen output and a .txt file
- home --- Some scripts and .pl files
- include --- The .h files
- lib --- empty
- ndbapi-examples --- Examples for the API
- src --- The .cpp files
- test --- Files for testing
- tools --- Programs for testing select, drop, and so on

A.1.22. The `netware` Directory

Files related to the Novell NetWare version of MySQL.

There are 43 files on this directory. Most have filename extensions of `*.def`, `*.sql`, or `*.c`.

The twenty-eight `*.def` files are all from Novell Inc. They contain import or export symbols. (`.def` is a common filename extension for "definition".)

The three `*.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 files are:

- netware/BUILD/*patch --- NetWare-specific build instructions and switches (compare the files in the BUILD directory)

For instructions about basic installation, see "Deployment Guide For NetWare AMP" at: http://developer.novell.com/ndk/whitepapers/namp.htm

A.1.23. The `NEW-RPMS` Directory

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. You only see it after you've done a "build".

A.1.24. The `os2` Directory

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.

A.1.25. The `pstack` Directory

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.

A.1.26. The `regex` Directory

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 WHATS-NEW 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.

A.1.27. The `SCCS` Directory

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.

A.1.28. The `scripts` Directory

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).

Some of 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

A.1.29. The `server-tools` Directory

The instance manager.

Quoting from the README file within this directory: "Instance Manager - manage MySQL instances locally and remotely. File description: mysqlmanager.cc - entry point to the manager, main, options.{h,cc} - handle startup options. manager.{h,cc} - manager process. mysql_connection.{h,cc} - handle one connection with mysql client. See also instance manager architecture description in mysqlmanager.cc.

A.1.30. The `sql` Directory

Programs for handling SQL commands. The "core" of MySQL.

These are the `.c` and `.cc` files in the `sql` directory:

- derror.cc --- read language-dependent message file
- des_key_file.cc --- load DES keys from plaintext file
- discover.cc --- Functions for discovery of frm file from handler
- 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_blackhole.cc --- Handler: Black Hole
- ha_federated.cc --- Handler: Federated
- ha_heap.cc --- Handler: Heap
- ha_innodb.cc --- Handler: InnoDB
- ha_myisam.cc --- Handler: MyISAM
- ha_myisammrg.cc --- Handler: (MyISAM MERGE)
- ha_ndbcluster.cc --- Handler: NDB
- 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_geofunc.cc --- Geometry functions
- item_row.cc --- Row items for comparing rows and for `IN` on rows
- item_strfunc.cc --- String functions
- item_subselect.cc --- Subqueries
- 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
- mysqld.cc --- Source for `mysqld.exe`; includes the `main()` program that starts `mysqld`, handling of signals and connections
- mf_decimal.cc --- New decimal and numeric code
- 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_range.cc --- Range of keys
- opt_sum.cc --- Optimize functions in presence of (implied) `GROUP BY`
- parse_file.cc --- Text .frm files management routines
- 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`

- protocol_cursor.cc --- Low level functions for storing data to be sent to the MySQL client
- 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
- sp.cc --- DB storage of stored procedures and functions
- sp_cache.cc --- For stored procedures
- sp_head.cc --- For stored procedures
- sp_pcontext.cc --- For stored procedures
- sp_rcontext.cc --- For stored procedures
- 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_client.cc --- A function called by my_net_init() to set some check variables
- 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_manager.cc --- Maintenance tasks, e.g. flushing the buffers periodically; used with `BDB` table logs
- sql_map.cc --- Memory-mapped files (not yet in use)
- 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_rename.cc --- Rename table
- sql_repl.cc --- Replication
- sql_select.cc --- Select and join optimization
- sql_show.cc --- The `SHOW` statement
- sql_state.c --- Functions to map mysqld errno to sqlstate
- 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_trigger.cc --- Triggers
- sql_udf.cc --- User-defined functions
- sql_union.cc --- The `UNION` operator
- sql_update.cc --- The `UPDATE` statement
- sql_view.cc --- Views
- stacktrace.c --- Display stack trace (Linux/Intel only)
- strfunc.cc --- String functions
- 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

A.1.31. The `sql-bench` Directory

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.

A.1.32. The `sql-common` Directory

Three files: client.c, my_time.c, pack.c. You will file symlinks to these files in other directories.

A.1.33. The `SSL` Directory

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.

A.1.34. The `strings` Directory

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:

- 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
- decimal.c --- for decimal and numeric conversions
- do_ctype.c --- display case-conversion and sort-conversion tables
- dump_map.c --- standalone file
- 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
- memcpy.c --- memory copy
- memset.c --- memory set
- my_strtoll10.c --- longlong2str for radix 10
- 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.
- strtod.c --- string to double
- 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
- uca-dump.c --- shows unicode collation algorithm dump
- udiv.c --- unsigned long divide, for operating systems that don't support these
- utr11-dump.c --- dump east Asian wide text file
- 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.

A.1.35. The `support-files` Directory

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.

A.1.36. The `tests` Directory

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

A.1.37. The `tools` Directory

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.

A.1.38. The `VC++Files` Directory

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.

A.1.39. The `vio` Directory

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.

A.1.40. The `zlib` Directory

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`.

Appendix B. InnoDB Source Code Distribution

The InnoDB source files are the best place to look for information about internals of the file structure that MySQLers 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 31 InnoDB subdirectories that come with the MySQL 5.0 source code in \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        8,145   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.

\btr (B-TREE)

```
  File Name   What Name Stands For         Size     Comment Inside File
  ---------   --------------------         ------   -------------------
  btr0btr.c   B-tree / B-tree              82,400   B-tree
  btr0cur.c   B-tree / Cursor             103,233   index tree cursor
  btr0sea.c   B-tree / Search              41,788   index tree adaptive search
  btr0pcur.c  B-tree / persistent cursor   16,720   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.

\buf (BUFFERING)

File Name	What Name Stands For	Size	Comment Inside File
buf0buf.c	Buffering / Buffering	65,582	The database buffer buf_pool
buf0flu.c	Buffering / Flush	29,583	... flush algorithm
buf0lru.c	/ least-recently-used	27,515	... replacement algorithm
buf0rea.c	Buffering / read	21,504	... 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 operating system'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).

\data (DATA)

File Name	What Name Stands For	Size	Comment Inside File
data0data.c	Data / Data	15,344	SQL data field and tuple
data0type.c	Data / Type	7,417	Data types

This is a collection of minor utility routines affecting rows.

\db (DATABASE)

There are no .c files in \db, just one .h file with some definitions for error codes.

\dict (DICTIONARY)

File Name	What Name Stands For	Size	Comment Inside File
dict0dict.c	Dictionary / Dictionary	114,263	Data dictionary system
dict0boot.c	Dictionary / boot	11,704	... booting
dict0crea.c	Dictionary / Create	37,278	... creation
dict0load.c	Dictionary / load	34,049	... load to memory cache
dict0mem.c	Dictionary / memory	7,470	... 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.

\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.

\eval (EVALUATING)

```
  File Name   What Name Stands For  Size    Comment Inside File
  ---------   --------------------  ------  -------------------
  eval0eval.c Evaluating/Evaluating 17,061  SQL evaluator
  eval0proc.c Evaluating/Procedures  5,001  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 MySQL handles stored procedures in its own way, so the eval0proc.c program is unimportant.

\fil (FILE)

```
  File Name   What Name Stands For  Size     Comment Inside File
  ---------   --------------------  ------   -------------------
  fil0fil.c   File / File          118,312   The low-level file system
```

The reads and writes to the database files happen here, in coordination with the low-level file i/o routines (see os0file.c 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 110,495  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   14,176   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.

\ha (HASHING)

```
 File Name   What Name Stands For  Size    Comment Inside File
 ---------   --------------------  ------  -------------------
 ha0ha.c     Hashing / Hashing      8,145  Hash table with external chains
 hash0hash.c Hashing / Hashing      3,283  Simple hash table utility
```

The two C programs in the \ha directory --- 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.

\ibuf (INSERT BUFFER)

```
 File Name   What Name Stands For  Size    Comment Inside File
 ---------   --------------------  ------  -------------------
 ibuf0ibuf.c Insert Buffer /       91,397  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.

\include (INCLUDE)

All .h and .ic files are in the \include directory. It's habitual to put comments along with the descriptions, so if (for example) you want to see comments about operating system file activity, the place to look is \include\os0file.h.

\lock (LOCKING)

```
 File Name   What Name Stands For  Size   Comment Inside File
 ---------   --------------------  ------ -------------------
 lock0lock.c Lock / Lock           139,207 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 my online article about the differences between Oracle-style and DB2/SQL-Server-style locks at: http://dbazine.com/gulutzan6.html

Now here is a notice from Heikki Tuuri of InnoDB. It concerns lock categories rather than lock0lock.c, but I place it in this section because this is the place that people are most likely to look for it.

Errata notice 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 */
```

Errata notice about InnoDB row locks ends.

\log (LOGGING)

```
  File Name   What Name Stands For  Size    Comment Inside File
  ---------   --------------------  ------  -------------------
  log0log.c   Logging / Logging     86,043  Database log
  log0recv.c  Logging / Recovery    91,352  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": ht-
tp://www.devarticles.com/c/a/MySQL/How-Logs-Work-On-MySQL-With-InnoDB-Tables

\mach (MACHINE FORMAT)

```
  File Name   What Name Stands For  Size   Comment Inside File
  ---------   --------------------  ------ -------------------
  mach0data.c Machine/Data           2,335 Utilities for converting
```

The mach0data.c program has two small routines for reading compressed ulints (unsigned long in-
tegers).

\mem (MEMORY)

```
  File Name   What Name Stands For  Size    Comment Inside File
  ---------   --------------------  ------  -------------------
  mem0mem.c   Memory / Memory       10,310  The memory management
  mem0dbg.c   Memory / Debug        22,054  ... the debug code
  mem0pool.c  Memory / Pool         16,511  ... 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 /    12,620  Mini-transaction buffer
  mtr0log.c   Mini-transaction / Log 8,090  ... 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.

\os (OPERATING SYSTEM)

File Name	What Name Stands For	Size	Comment Inside File
os0file.c	OS / File	104,081	To i/o primitives
os0thread.c	OS / Thread	7,754	To thread control primitives
os0proc.c	OS / Process	16,919	To process control primitives
os0sync.c	OS / Synchronization	14,256	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 call is within an "#ifdef __WIN__ ... #endif" block; the effective routines are somewhat different for other operating systems.

\page (PAGE)

File Name	What Name Stands For	Size	Comment Inside File
page0page.c	Page / Page	51,731	Index page routines
page0cur.c	Page / Cursor	38,127	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	45,376	SQL parser
pars0grm.c	Parsing/Grammar	62,685	A Bison parser
pars0opt.c	Parsing/Optimizer	31,268	Simple SQL Optimizer
pars0sym.c	Parsing/Symbol Table	5,239	SQL parser symbol table
lexyy.c	Parsing/Lexer	62,071	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` (also 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.

\que (QUERY GRAPH)

File Name	What Name Stands For	Size	Comment Inside File
que0que.c	Query Graph / Query	30,774	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.

\read (READ)

```
 File Name   What Name Stands For Size    Comment Inside File
 ---------   -------------------- ------  -------------------
 read0read.c Read / Read           9,935  Cursor read
```

The read0read.c program opens a "read view" of a query result, using some functions in the \trx program group.

\rem (RECORD MANAGER)

```
 File Name   What Name Stands For  Size     Comment Inside File
 ---------   --------------------  ------   -------------------
 rem0rec.c   Record Manager        38,573   Record Manager
 rem0cmp.c   Record Manager /      26,617   Comparison services for records
             Comparison
```

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             18,375   General row routines
 row0uins.c  Row / Undo Insert      6,799   Fresh insert undo
 row0umod.c  Row / Undo Modify     19,712   Undo modify of a row
 row0undo.c  Row / Undo            10,512   Row undo
 row0vers.c  Row / Version         14,385   Row versions
 row0mysql.c Row / MySQL          112,462   Interface [to MySQL]
 row0ins.c   Row / Insert          42,829   Insert into a table
 row0sel.c   Row / Select         111,719   Select
 row0upd.c   Row / Update          51,824   Update of a row
 row0purge.c Row / Purge           15,609   Purge obsolete records
```

Rows can be selected, inserted, updated/deleted, or purged (a maintenance activity). These actions cause following actions, for example after insert there can be an index-update test, but it seems to me that sometimes the following 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       75,633   Server main program
 srv0que.c   Server / Query         2,463   Server query execution
 srv0start.c Server / Start        50,154   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).

\sync (SYNCHRONIZATION)

File Name	What Name Stands For	Size	Comment Inside File
sync0sync.c	Synchronization /	37,940	Mutex, the basic sync primitive
sync0arr.c	... / array	26,455	Wait array used in primitives
sync0rw.c	... / read-write	22,846	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 sync0sync.c's helper file, \include\sync0sync.ic. This is the only program that contains any assembly code.

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.

\thr (Thread Local Storage)

File Name	What Name Stands For	Size	Comment Inside File
thr0loc.c	Thread / Local	5,334	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 /	50,480	The transaction
trx0purge.c	Transaction / Purge	29,133	... Purge old versions
trx0rec.c	Transaction / Record	37,346	... Undo log record
trx0roll.c	/ Rollback	31,448	... Rollback
trx0sys.c	Transaction / System	27,018	... System
trx0rseg.c	/ Rollback segment	6,445	... Rollback segment
trx0undo.c	Transaction / Undo	51,519	... 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	1,740	Sessions

One user can have multiple sessions (the session being all the things that happen between a connect and disconnect). This is where InnoDB used to track session IDs, and server/client messaging. It's another of those items which is usually MySQL's job, though. So now usr0sess.c merely closes.

\ut (UTILITIES)

```
 File Name   What Name Stands For  Size     Comment Inside File
 ---------   --------------------  ------   -------------------
 ut0ut.c     Utilities / Utilities  9,728   Various utilities
 ut0byte.c   Utilities / Debug        793   Byte utilities
 ut0rnd.c    Utilities / Random     1,474   Random numbers and hashing
 ut0mem.c    Utilities / Memory    10,358   Memory primitives
 ut0dbg.c    Utilities / Debug      2,579   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.

This is the end of the section-by-section account of InnoDB subdirectories.

Some Notes About Structures

InnoDB's job, as a storage engine for MySQL, is to provide: commit-rollback, crash recovery, row-level locking, and consistent non-blocking reads. How? With locks, a paged-file structure with buffer pooling, and undo/redo logs,

The locks are kept in bit maps in main memory. Thus InnoDB differs from Oracle in one respect: instead of storing lock information on the page as Oracle does with Interested Transaction Lists, InnoDB keeps it in a separate and more volatile structure. But both Oracle and InnoDB try to achieve a similar goal: "writers don't block readers". So a typical InnoDB row-read involves: (a) if the reading is for writing, then check if the row is locked and if so wait; (b) if according to the information in the row header the row has been changed by some newer ransaction, then get the older version from the log. We call the (b) part "versioning" because it means that a reader can get the older version of a row and thus will have a temporally consistent view of all rows.

The InnoDB workspace consists of: tablespace and log files. A tablespace consists of: segments, as many as necessary. A segment is usually a file, but might be a raw disk partition. A segment consists of: extents. An extent consists of: 64 pages. A page's length is always 16KB, for both data and index. A page consists of: a page header, and some rows. The page and row formats are the subjects of later chapters.

InnoDB keeps two logs, the redo log and the undo log.

The redo log is for re-doing data changes that had not been written to disk when a crash occurred. There is one redo log for the entire workspace, it contains multiple files (the number depends on innodb_log_files_in_group), it is circular (that is, after writing to the last file InnoDB starts again on the first file). The file header includes the last successful checkpoint. A redo log record's contents are: Page Number (4 bytes = page number within tablespace), Offset of change within page (2 bytes), Log Record Type (insert, update, delete, "fill space with blanks", etc.), and the changes on that page (only redo values, not old values).

The undo log is primarily for removing data changes that had been written to disk when a crash occurred, but should not have been written, because they were for uncommitted transactions. Sometimes InnoDB calls the undo log the "rollback segment". The undo log is inside the tablespace. The "insert" section of the undo log is needed only for transaction rollback and can be discarded at COMMIT time.

The "update/delete" section of the undo log is also useful for consistent reads, and can be discarded when InnoDB has ended all transactions that might need the undo log records to reconstruct earlier versions of rows. An undo log record's contents are: Primary Key Value (not a page number or physical address), Old Transaction ID (of the transaction that updated the row), and the changes (only old values).

COMMIT will write the contents of the log buffer to disk, and put undo log records in a history list. ROLLBACK will delete undo log records that are no longer needed. PURGE (an internal operation that occurs outside user control) will no-longer-necessary undo log records and, for data records that have been marked for deletion and are no longer necessary for consistent read, will remove the records. CHECKPOINT causes -- well, see the article "How Logs Work On MySQL With InnoDB Tables".

You might be able to find a slide show, "ACID Transactions in MySQL With InnoDB", via this page: http://dev.mysql.com/tech-resources/presentations/. If the links are broken, notify MySQL.

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/c/a/MySQL/How-Logs-Work-On-MySQL-With-InnoDB-Tables

- 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.

Index