

Writing Storage Engines

Brian Aker, 2007

A couple of thoughts...

- Start from an example engine.
- Make your engine pluggable from the beginning.
- Automate your testing.
- When in doubt, see how another engine handles the query!

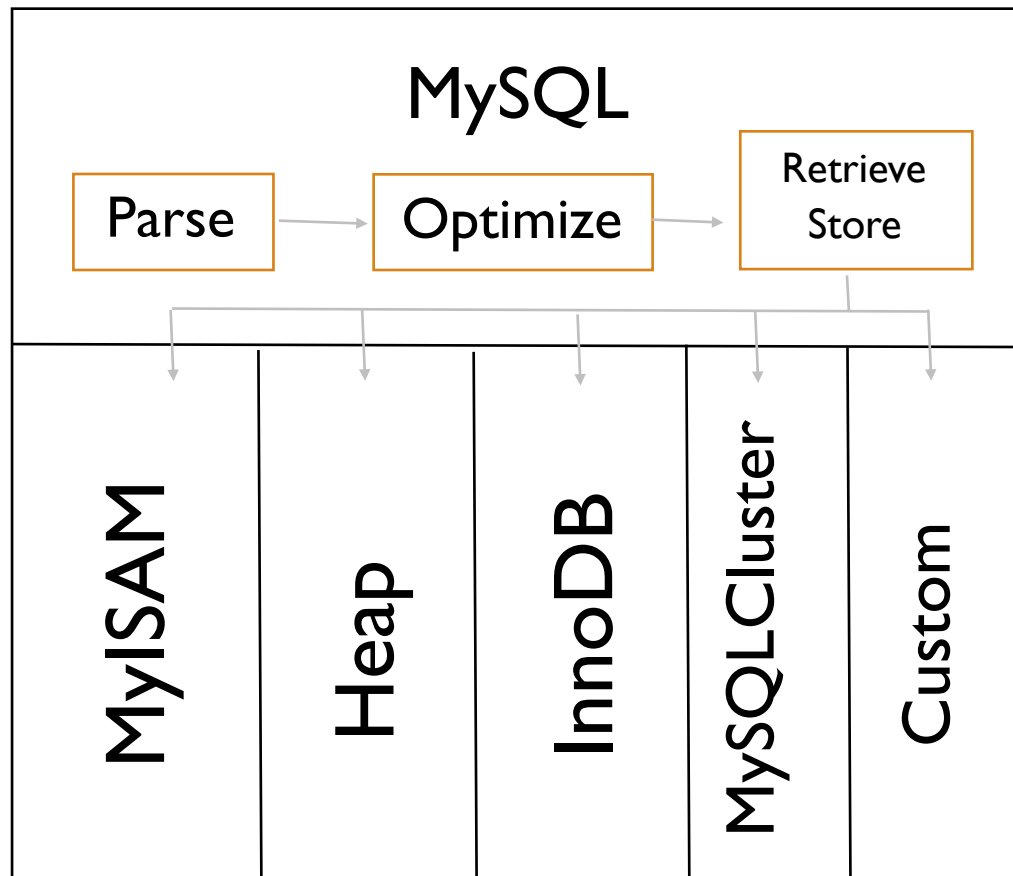
Server Desing

- Written in C, C++, ASM
- Majority (80%) in C, client library, network communications, I/O, storage engines
- Parse tree and optimizer implemented C++
- String functionality in ASM on platforms that require it for performance.

Process Model

- Multithreaded
- 1 Connection == 1 Thread
- Minimum overhead per client
 - Network buffers (configurable, limits max query size)
 - Thread Stack (e.g. 128-256K on Linux)
 - read_buffer_size
 - read_rnd_buffer_size
 - sort_buffer_size
 - tmp_table_size
- Deployments with 1000s of concurrent connections not uncommon

Basic Architecture



**MySQL Database
Management Level**

**Table Handler /
Storage Engine
Level**

There are a lot of engines...

- Memory
- Archive
- MyISAM
- Innodb
- SolidDB
- PBXT
- AWS
- ha_memcache
- myhttp
- Cluster
- Infobright
- Nitro
- Merge
- CSV
- Federated
- ODBCFC
- Falcon
- Blackhole...

First Let us Talk about a st_plugin

```
mysql_declare_plugin(memcache)
{
    MYSQL_STORAGE_ENGINE_PLUGIN,
    &memcache_storage_engine,
    "MEMCACHE",
    "Brian Aker, Tangent Org",
    "Simple Interface for working with memcache as a storage engine",
    PLUGIN_LICENSE_GPL,
    memcache_init_func, /* Plugin Init */
    memcache_done_func, /* Plugin Deinit */
    0x0005,
    NULL,                /* status variables */
    NULL,                /* system variables */
    NULL,                /* config options */
},
```

Simple Handlerton

```
static int example_init_func(void *p)
{
    DEBUG_ENTER("example_init_func");
    if (!example_init)
    {
        example_hnton= (handlerton *)p;
        example_init= 1;
        VOID(pthread_mutex_init(&example_mutex,MY_MUTEX_INIT_FAST));
        (void) hash_init(&example_open_tables,system_charset_info,32,0,0,
            (hash_get_key) example_get_key,0,0);

        example_hnton->state= SHOW_OPTION_YES;
        example_hnton->db_type= DB_TYPE_EXAMPLE_DB;
        example_hnton->create= example_create_handler;
        example_hnton->flags= HTON_CAN_RECREATE;
    }
    DEBUG_RETURN(0);
}
```

A Little More Complex

```
federated_hton->db_type= DB_TYPE_FEDERATED_DB;  
federated_hton->commit= federated_commit;  
federated_hton->rollback= federated_rollback;  
federated_hton->create= federated_create_handler;  
federated_hton->panic= federated_db_end;  
federated_hton->flags= HTON_ALTER_NOT_SUPPORTED;
```

(If you want more complex, look at `ha_ndbcluster.cc`)

The handler

(think of it as a cursor)

- Implemented in handler.h
- Has both table level operations and cursor operations.
- There is more to it than can be covered in three hours (not surprising)

What about defining features?

- `tables_flags()`
- `index_flags()`
- `handlerton flags`

Handlerton Flags

- HTON_CLOSE_CURSORS_AT_COMMIT
- HTON_ALTER_NOT_SUPPORTED
- HTON_HIDDEN
- HTON_NOT_USER_SELECTABLE
- HTON_NO_PARTITION

Table Flags

- HA_NO_TRANSACTIONS
- HA_TABLE_SCAN_ON_INDEX
- HA_NO_BLOBS
- HA_CAN_INDEX_BLOBS
- HA_CAN_FULLTEXT
- HA_FILE_BASED

Index Flags

- HA_READ_NEXT
- HA_READ_PREV
- HA_READ_ORDER

So I want to know more!

The best solution is to use the code. Read, and reread periodically the handler.h found in the sql/ directory.

Table Object Methods

- `base_ext()`
- `create()`
- `delete_table()`
- `rename_table()`
- `optimize()`, `repair()`, etc...

bas_ext()

```
static const char *ha_myisam_exts[] = {  
    ".MYI",  
    ".MYD",  
    NullS  
};  
  
const char **ha_myisam::bas_ext() const  
{  
    return ha_myisam_exts;  
}
```

create()

```
int ha_federated::create(const char *name, TABLE *table_arg,
                        HA_CREATE_INFO *create_info)
{
    int retval;
    FEDERATED_SHARE tmp_share; // Only a temporary share, to test the url
    DEBUG_ENTER("ha_federated::create");

    if (!(retval= parse_url(&tmp_share, table_arg, 1)))
        retval= check_foreign_data_source(&tmp_share, 1);

    my_free((gptr) tmp_share.scheme, MYF(MY_ALLOW_ZERO_PTR));
    DEBUG_RETURN(retval);
}
```

delete_table()

(not everyone has one)

```
int ha_myisam::delete_table(const char *name)
{
    return mi_delete_table(name);
}
```

rename_table()

```
int ha_heap::rename_table(const char * from, const char *  
to)  
{  
    return heap_rename(from,to);  
}
```

Insert!

```
T@3 :|||||>ha_archive::open
T@3 :|||||>ha_archive::get_share
T@3 :|||||<ha_archive::get_share
T@3 :|||||>ha_archive::create_record_buffer
T@3 :|||||<ha_archive::create_record_buffer
T@3 :|||||<ha_archive::open
T@3 :||||>ha_archive::start_bulk_insert
T@3 :||||<ha_archive::start_bulk_insert
T@3 :||||>ha_archive::write_row
T@3 :||||>ha_archive::pack_row
T@3 :||||<ha_archive::pack_row
T@3 :||||<ha_archive::write_row
T@3 :||||>ha_archive::end_bulk_insert
T@3 :||||<ha_archive::end_bulk_insert
```

write_row()

```
int ha_tina::write_row(byte * buf)
{
    int size;
    DEBUG_ENTER("ha_tina::write_row");

    if (share->crashed)
        DEBUG_RETURN(HA_ERR_CRASHED_ON_USAGE);

    ha_statistic_increment(&SSV::ha_write_count);

    if (table->timestamp_field_type & TIMESTAMP_AUTO_SET_ON_INSERT)
        table->timestamp_field->set_time();

    size= encode_quote(buf);
    stats.records++;
    DEBUG_RETURN(0);
}
```

Read (Scan)

```
T@3 : ||||| >ha_archive::info
T@3 : ||||| <ha_archive::info
T@3 : ||||| | >ha_archive::rnd_init
T@3 : ||||| | <ha_archive::rnd_init
T@3 : ||||| >ha_archive::rnd_next
T@3 : ||||| ||| >ha_archive::unpack_row
T@3 : ||||| ||| <ha_archive::unpack_row
T@3 : ||||| | <ha_archive::get_row
T@3 : ||||| <ha_archive::rnd_next
T@3 : ||||| >ha_archive::rnd_next
T@3 : ||||| <ha_archive::rnd_next
```

info()

```
int ha_archive::info(uint flag)
{
    DEBUG_ENTER("ha_archive::info");
    stats.records= share->rows_recorded;
    stats.deleted= 0;
    /* Costs quite a bit more to get all information */
    if (flag & HA_STATUS_TIME)
    {
        MY_STAT file_stat; // Stat information for the data file
        VOID(my_stat(share->data_file_name, &file_stat, MYF(MY_WME)));
        stats.mean_rec_length= table->s->reclength + buffer.allocated_length();
        stats.data_file_length= file_stat.st_size;
        stats.create_time= file_stat.st_ctime;
        stats.update_time= file_stat.st_mtime;
        stats.max_data_file_length= share->rows_recorded * stats.mean_rec_length;
    }
    stats.delete_length= 0;
    stats.index_file_length=0;
    if (flag & HA_STATUS_AUTO)
    {
        azflush(&archive, Z_SYNC_FLUSH);
        stats.auto_increment_value= archive.auto_increment;
    }
    DEBUG_RETURN(0);
}
```

rnd_init()

```
int
ha_innobase::rnd_init(
/*=====*/
/* out: 0 or error number */
bool scan) /* in: ??????? */
{
    int err;

    row_prebuilt_t* prebuilt = (row_prebuilt_t*) innobase_prebuilt;
    if (prebuilt->clust_index_was_generated) {
        err = change_active_index(MAX_KEY);
    } else {
        err = change_active_index(primary_key);
    }

    start_of_scan = 1;

    return(err);
}
```

rnd_next()

```
int
ha_innobase::rnd_next(
/*=====*/
    /* out: 0, HA_ERR_END_OF_FILE, or error number */
    mysql_byte* buf)/* in/out: returns the row in this buffer,
    in MySQL format */
{
    int    error;

    DEBUG_ENTER("rnd_next");
    statistic_increment(current_thd->status_var.ha_read_rnd_next_count,
        &LOCK_status);

    if (start_of_scan) {
        error = index_first(buf);
        if (error == HA_ERR_KEY_NOT_FOUND) {
            error = HA_ERR_END_OF_FILE;
        }
        start_of_scan = 0;
    } else {
        error = general_fetch(buf, ROW_SEL_NEXT, 0);
    }

    DEBUG_RETURN(error);
}
```

update_row()

```
T@3 :||| |>ha_tina::info
T@3 :||| |<ha_tina::info
T@3 :||| |>ha_tina::rnd_init
T@3 :||| |<ha_tina::rnd_init
T@3 :||| |>ha_tina::extra
T@3 :||| |<ha_tina::extra
T@3 :||| |>ha_tina::rnd_next
T@3 :||| |<ha_tina::rnd_next
T@3 :||| |>ha_tina::rnd_next
T@3 :||| |<ha_tina::rnd_next
T@3 :||| |>ha_tina::update_row
T@3 :||| |<ha_tina::update_row
T@3 :||| |>ha_tina::rnd_next
T@3 :||| |<ha_tina::rnd_next
T@3 :||| |>ha_tina::update_row
T@3 :||| |<ha_tina::update_row
T@3 :||| |>ha_tina::rnd_next
T@3 :||| |<ha_tina::rnd_next
T@3 :||| |>ha_tina::extra
T@3 :||| |<ha_tina::extra
T@3 :||| |>ha_tina::rnd_end
T@3 :||| |>ha_tina::write_meta_file
T@3 :||| |<ha_tina::write_meta_file
T@3 :||| |<ha_tina::rnd_end
```



Update occurs here

update_row()

```
int ha_tina::update_row(const byte * old_data, byte * new_data)
{
    int size;
    DEBUG_ENTER("ha_tina::update_row");

    ha_statistic_increment(&SSV::ha_read_rnd_next_count);

    if (table->timestamp_field_type & TIMESTAMP_AUTO_SET_ON_UPDATE)
        table->timestamp_field->set_time();

    size= encode_quote(new_data);

    if (chain_append())
        DEBUG_RETURN(-1);

    DEBUG_RETURN(0);
}
```

delete_row()

```
T@3 :| | | | >ha_tina::info
T@3 :| | | | <ha_tina::info
T@3 :| | | | | >ha_tina::rnd_init
T@3 :| | | | | <ha_tina::rnd_init
T@3 :| | | | | >ha_tina::extra
T@3 :| | | | | <ha_tina::extra
T@3 :| | | | | >ha_tina::rnd_next
T@3 :| | | | | <ha_tina::rnd_next
T@3 :| | | | | >ha_tina::rnd_next
T@3 :| | | | | <ha_tina::rnd_next
T@3 :| | | | | >ha_tina::delete_row
T@3 :| | | | | <ha_tina::delete_row
T@3 :| | | | | >ha_tina::rnd_next
T@3 :| | | | | <ha_tina::rnd_next
T@3 :| | | | | >ha_tina::rnd_next
T@3 :| | | | | <ha_tina::rnd_next
T@3 :| | | | | >ha_tina::extra
T@3 :| | | | | <ha_tina::extra
T@3 :| | | | | >ha_tina::rnd_end
T@3 :| | | | | >ha_tina::write_meta_file
T@3 :| | | | | <ha_tina::write_meta_file
T@3 :| | | | | <ha_tina::rnd_end
```



delete
occurs
here

delete_row()

```
int ha_tina::delete_row(const byte * buf)
{
    DEBUG_ENTER("ha_tina::delete_row");
    ha_statistic_increment(&SSV::ha_delete_count);

    if (chain_append())
        DEBUG_RETURN(-1);

    stats.records--;

    /* DELETE should never happen on the log table */
    DEBUG_ASSERT(!share->is_log_table);

    DEBUG_RETURN(0);
}
```

index_read()

```
int ha_memcache::index_read(byte *buf, const byte *key,  
                             uint key_len, enum ha_rkey_function find_flag)  
{  
    DEBUG_ENTER("ha_memcache::index_read");  
    unsigned int rc= 0;  
    rc= find_row(buf, key, key_len);  
    DEBUG_RETURN(rc);  
}
```

index_next()

```
int index_next(byte * buf)
{
    return HA_ERR_END_OF_FILE;
}
```

Optimizer

- `info()`
- `scan_time()`
- `read_time()`
- `records_in_range()`
- pushdown conditions

info()

```
void ha_archive::info(uint flag)
{
  DEBUG_ENTER("ha_archive::info");
  /*
   This should be an accurate number now, though bulk and delayed inserts can
   cause the number to be inaccurate.
  */
  stats.records= share->rows_recorded;
  stats.deleted= 0;
  /* Costs quite a bit more to get all information */
  if (flag & HA_STATUS_TIME)
  {
    MY_STAT file_stat; // Stat information for the data file

    VOID(my_stat(share->data_file_name, &file_stat, MYF(MY_WME)));

    stats.mean_rec_length= table->s->reclength + buffer.allocated_length();
    stats.data_file_length= file_stat.st_size;
    stats.create_time= file_stat.st_ctime;
    stats.update_time= file_stat.st_mtime;
    stats.max_data_file_length= share->rows_recorded * stats.mean_rec_length;
  }
  stats.delete_length= 0;
  stats.index_file_length=0;

  if (flag & HA_STATUS_AUTO)
    stats.auto_increment_value= share->auto_increment_value;

  DEBUG_VOID_RETURN;
}
```

scan_time()

```
ha_innobase::scan_time()
/*=====*/
    /* out: estimated time measured in disk seeks */
{
    row_prebuilt_t* prebuilt      = (row_prebuilt_t*) innobase_prebuilt;

    /* Since MySQL seems to favor table scans too much over index
    searches, we pretend that a sequential read takes the same time
    as a random disk read, that is, we do not divide the following
    by 10, which would be physically realistic. */

    return((double) (prebuilt->table->stat_clustered_index_size));
}
```

read_time()

```
double
ha_innobase::read_time(
/*=====*/
    /* out: estimated time measured in disk seeks */
    uint  index, /* in: key number */
    uint  ranges, /* in: how many ranges */
    ha_rows rows) /* in: estimated number of rows in the ranges */
{
    ha_rows total_rows;
    double time_for_scan;

    if (index != table->s->primary_key) {
        /* Not clustered */
        return(handler::read_time(index, ranges, rows));
    }

    if (rows <= 2) {
        return((double) rows);
    }

    /* Assume that the read time is proportional to the scan time for all
    rows + at most one seek per range. */

    time_for_scan = scan_time();

    if ((total_rows = estimate_rows_upper_bound()) < rows) {
        return(time_for_scan);
    }

    return(ranges + (double) rows / (double) total_rows * time_for_scan);
}
```

records_in_range

```
ha_rows ha_heap::records_in_range(uint inx, key_range *min_key,
                                   key_range *max_key)
{
    KEY *key=table->key_info+inx;
    if (key->algorithm == HA_KEY_ALG_BTREE)
        return hp_rb_records_in_range(file, inx, min_key, max_key);

    if (!min_key || !max_key ||
        min_key->length != max_key->length ||
        min_key->length != key->key_length ||
        min_key->flag != HA_READ_KEY_EXACT ||
        max_key->flag != HA_READ_AFTER_KEY)
        return HA_POS_ERROR;           // Can only use exact keys

    if (stats.records <= 1)
        return stats.records;

    /* Assert that info() did run. We need current statistics here. */
    DEBUG_ASSERT(key_stat_version == file->s->key_stat_version);
    return key->rec_per_key[key->key_parts-1];
}
```

Locks and transactions anyone?

- `external_lock()`
- `store_lock()`

store_lock()

```
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;  
}
```

store_lock()

```
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)  
    {  
        if ((lock_type >= TL_WRITE_CONCURRENT_INSERT &&  
            lock_type <= TL_WRITE) && !thd_in_lock_tables(thd)  
            && !thd_tablespace_op(thd))  
            lock_type = TL_WRITE_ALLOW_WRITE;  
        if (lock_type == TL_READ_NO_INSERT && !thd_in_lock_tables(thd))  
            lock_type = TL_READ;  
  
        lock.type=lock_type;  
    }  
  
    *to++= &lock;  
  
    return to;  
}
```

```

int ha_federated::external_lock(THD *thd, int lock_type)
{
    int error= 0;
    ha_federated *trx= (ha_federated *)thd->ha_data[ht->slot];
    DBUG_ENTER("ha_federated::external_lock");

    if (lock_type != F_UNLCK)
    {
        if (!(thd->options & (OPTION_NOT_AUTOCOMMIT | OPTION_BEGIN)))
        {
            error= connection_autocommit(TRUE);
            trans_register_ha(thd, FALSE, ht);
        }
        else
        {
            DBUG_PRINT("info",("not autocommit"));
            if (!trx)
            {
                error= connection_autocommit(FALSE);
                thd->ha_data[ht->slot]= this;
            }
            else
            {
                ha_federated *ptr;
                for (ptr= trx; ptr; ptr= ptr->trx_next)
                    if (ptr == this)
                        break;
                    else if (!ptr->trx_next)
                        ptr->trx_next= this;
            }
        }
    }
    DBUG_RETURN(0);
}

```

external_lock()

(yes, it is a historical, and now horrible name)

The rest of the commit?

```
federated_hton->commit= federated_commit;  
federated_hton->rollback= federated_rollback;  
federated_hton->create= federated_create_handler;  
federated_hton->panic= federated_db_end;  
federated_hton->flags= HTON_ALTER_NOT_SUPPORTED;
```

federated_commit()

```
static int federated_commit(handlerton *hton, THD *thd, bool all)
{
    int return_val= 0;
    ha_federated *trx= (ha_federated *)thd->ha_data[hton->slot];
    DBUG_ENTER("federated_commit");

    if (all)
    {
        int error= 0;
        ha_federated *ptr, *old= NULL;
        for (ptr= trx; ptr; old= ptr, ptr= ptr->trx_next)
        {
            if (old)
                old->trx_next= NULL;
            error= ptr->connection_commit();
            if (error && !return_val);
            return_val= error;
        }
        thd->ha_data[hton->slot]= NULL;
    }

    DBUG_PRINT("info", ("error val: %d", return_val));
    DBUG_RETURN(return_val);
}
```

federated_rollback()

```
static int federated_rollback(handlerton *hton, THD *thd, bool all)
{
    int return_val= 0;
    ha_federated *trx= (ha_federated *)thd->ha_data[hton->slot];
    DBUG_ENTER("federated_rollback");

    if (all)
    {
        int error= 0;
        ha_federated *ptr, *old= NULL;
        for (ptr= trx; ptr; old= ptr, ptr= ptr->trx_next)
        {
            if (old)
                old->trx_next= NULL;
            error= ptr->connection_rollback();
            if (error && !return_val)
                return_val= error;
        }
        thd->ha_data[hton->slot]= NULL;
    }

    DBUG_PRINT("info", ("error val: %d", return_val));
    DBUG_RETURN(return_val);
}
```

Any other interesting handlerton calls?

- `discover()`
- `find_files()`

```

int archive_discover(handlerton *hton, THD* thd, const char *db,
                    const char *name,
                    const void** frmblob,
                    uint* frmlen)
{
    azio_stream frm_stream;
    char az_file[FN_REFLen];
    char *frm_ptr;
    MY_STAT file_stat;

    fn_format(az_file, name, db, ARZ, MY_REPLACE_EXT | MY_UNPACK_FILENAME);

    if (!(my_stat(az_file, &file_stat, MYF(0))))
        goto err;

    if (!(azopen(&frm_stream, az_file, O_RDONLY|O_BINARY)))
    {
        if (errno == EROFS || errno == EACCES)
            DEBUG_RETURN(my_errno= errno);
        DEBUG_RETURN(HA_ERR_CRASHED_ON_USAGE);
    }

    if (frm_stream.frm_length == 0)
        goto err;

    frm_ptr= (char *)my_malloc(sizeof(char) * frm_stream.frm_length, MYF(0));
    azread_frm(&frm_stream, frm_ptr);
    azclose(&frm_stream);

    *frmlen= frm_stream.frm_length;
    *frmblob= frm_ptr;

    DEBUG_RETURN(0);
err:
    my_errno= 0;
    DEBUG_RETURN(1);
}

```

archive_discover()

What is an information schema?

```
mysql> use information_schema;  
mysql> show tables;
```

Information Plugin

```
{
  MYSQL_INFORMATION_SCHEMA_PLUGIN,
  &memcache_server_is,
  "memcache_servers",
  "Brian Aker",
  "Active Memached Servers.",
  PLUGIN_LICENSE_GPL,
  memcache_server_is_init, /* Plugin Init */
  memcache_server_is_deinit, /* Plugin Deinit */
  0x0005,
  NULL, /* status variables */
  NULL, /* system variables */
  NULL /* config options */
}
```

I_S init()

```
int memcache_server_is_init(void *p)
{
    DEBUG_ENTER("memcache_server_is_init");
    ST_SCHEMA_TABLE *schema= (ST_SCHEMA_TABLE *)p;

    struct st_plugin_int *plugin= (struct st_plugin_int *)p;

    schema->fields_info= memcache_server_field_info;
    schema->fill_table= fill_memcache_server_schema;

    DEBUG_RETURN(0);
}
```

I_S deinit()

```
int memcache_server_is_deinit(void *p)
{
    DEBUG_ENTER("zeroconf_is_deinit");
    DEBUG_RETURN(0);
}
```

field_info[]

```
ST_FIELD_INFO memcache_server_field_info[]=  
{  
    {"NAME", 120, MYSQL_TYPE_STRING, 0, 0, "Name"},  
    {"COUNT", 4, MYSQL_TYPE_LONG, 0, 0, "Count"},  
    {"CURRENT_ITEMS", 4, MYSQL_TYPE_LONG, 0, 0, "Current Items"},  
    {"TOTAL_ITEMS", 4, MYSQL_TYPE_LONG, 0, 0, "Total Items"},  
    {"BYTES", 4, MYSQL_TYPE_LONG, 0, 0, "Bytes"},  
    {"CURRENT_CONNECTIONS", 4, MYSQL_TYPE_LONG, 0, 0, "Current Connections"},  
    {"TOTAL_CONNECTIONS", 4, MYSQL_TYPE_LONG, 0, 0, "Total Connections"},  
    {"CONNECTION_STRUCTURES", 4, MYSQL_TYPE_LONG, 0, 0, "Connection Structure"},  
    {"GETS", 4, MYSQL_TYPE_LONG, 0, 0, "Gets"},  
    {"SETS", 4, MYSQL_TYPE_LONG, 0, 0, "Sets"},  
    {"HITS", 4, MYSQL_TYPE_LONG, 0, 0, "Hits"},  
    {"MISSES", 4, MYSQL_TYPE_LONG, 0, 0, "Misses"},  
    {"BYTES_READ", 4, MYSQL_TYPE_LONG, 0, 0, "Bytes Read"},  
    {"BYTES_WRITTEN", 4, MYSQL_TYPE_LONG, 0, 0, "Bytes Written"},  
    {"LIMIT_MAXBYTES", 4, MYSQL_TYPE_LONG, 0, 0, "Limit Maxbytes"},  
    {0, 0, MYSQL_TYPE_STRING, 0, 0, 0}  
};
```

fill_memcache_server_schema()

```
static int fill_memcache_server_schema(THD *thd, TABLE_LIST *tables, COND *cond)
{
    TABLE *table= (TABLE *) tables->table;
    DEBUG_ENTER("fill_memcache_server_schema");

    for (uint x= 0; x < memcache_server_hash.records; ++x)
    {
        table->field[0]->store(server->server_name,
                               server->server_name_length,
                               scs);
        table->field[1]->store(server->use_count);
        table->field[2]->store(stats->curr_items);
        table->field[3]->store(stats->total_items);
        table->field[4]->store(stats->bytes);
        table->field[5]->store(stats->curr_connections);
        schema_table_store_record(thd, table);
        pthread_mutex_unlock(&memcache_server_mutex);

        DEBUG_RETURN(rc);
    }
}
```

How to load this?

```
mysql> INSTALL PLUGIN memcache_servers SONAME  
'libmemcache_engine.so';
```

How about Testing

- `mysql-test`
- `mysqlslap`

What is missing?

- A lot!
- plug.in
- how to build as both a static and as a dynamic engine
- <http://hg.tangent.org/skeleton-mysql-engine>