Developing Replication Plugins for Drizzle

Jay Pipes
jaypipes@gmail.com
http://joinfu.com

Padraig O'Sullivan
osullivan.padraig@gmail.com
http://posulliv.com

what we'll cover today

- Contributing to Drizzle
- Overview of Drizzle's architecture
- Code walkthrough of Drizzle plugin basics
- Overview of Drizzle's replication system
- Understanding Google Protobuffers
- The Transaction message
- In depth walkthrough of the filtered replicator
- In-depth walkthrough of the transaction log
- The future, your ideas, making an impact

Drizzle is a Community

Being a Drizzler

Some things to remember...

- No blame
- No shame
- Be open and transparent
- Learn something from someone? Pass it on...
  - By adding to the wiki (http://drizzle.org/wiki/)
  - By sharing it with another contributor
  - By blogging about it
  - By posting what you learn to the mailing list
- *There is no such thing as a silly question*

NO TROLLS.

Managing Your Code

Launchpad and BZR

Launchpad.net

- The Drizzle community focal-point
  - http://launchpad.net/drizzle
- Join the drizzle-developers team:
  - http://launchpad.net/~drizzle-developers
  - Once on the team, you'll be able to push BZR branches to the main Drizzle code repository

Launchpad.net

- Code management
- Task (blueprint) management
- Bug reporting
- Translations (Rosetta)
- FAQ functionality
  - http://www.joinfu.com/2008/08/a-contributors-guide-to-launchpadnet-part-1-getting-started/
  - http://www.joinfu.com/2008/08/a-contributors-guide-to-launchpadnet-part-2-code-management/

Understanding how BZR isn't SVN

- Drizzle developers use BZR for source control
- It's a *distributed* version control system
- It's NOT subversion, and takes some getting used to
  - But it's easy to use once you get used to it ;)
- Remember, there is no ~~spoon~~ “central sources”
- Code lives in branches
- Branches live in a repository

Creating a local BZR branch

- You create a branch on your local workstation by *branching* an existing branch:

bzr branch lp:drizzle working

- What does the above do?
  - Creates a local (to your workstation) branch called *working* which is *derived* from the development series' default branch on Launchpad.net
  - FYI: development series default branch is called *trunk*
  - FYI: there is another series on Launchpad.net called *stage*. We push code to *stage* before it goes into *trunk*.

Making code changes

- You make changes to your local branch with an editor, just like any other source control system
- If you add a new file to the source code, you must tell BZR that you've done so:

```
bzr add drizzled/my_new_file.cc
```

- The above would tell bzr to add the file *my_new_file.cc* in the *drizzled* directory to source control

Committing your changes

- When done making changes, commit them:

bzr commit

- The above will commit your changes to source control and open up your default editor so that you can type a comment describing your changes
- When you save and close your editor, a *changeset* will be produced and saved by BZR

More on committing

- When you bzr commit, you are committing your changes *locally*
  - You'll learn how to push those changes shortly...
- You can automatically add a comment to your commit (and not open an editor) with the -m option:

bzr commit -m "Small changes to XXX"

Best Practice #1

- Be as descriptive as possible for your commit comments
  - Allows others to better understand your code
  - They allow you to have a decent history of why you made certain changes
- Good comment:
  - “Fix issue where xyz struct on little-endian machines was incorrectly stored to disk. Fixes Bug #221333”
- Bad comment:
  - “Fixes endian”

Publishing your branch

- Must be a member of the Drizzle Developers team
- You will *push* your branch up to Launchpad:

```
bzr push lp:~$user/drizzle/$branchname
```

- Where $user is your username *on Launchpad.net*
- Example of me pushing a branch called "timezones"

```
bzr push lp:~jaypipes/drizzle/timezones
```

Taking a look at a branch

- Once a branch is pushed to Launchpad.net, you can give someone a link to it:
  - http://code.launchpad.net/~$user/drizzle/$branchname
- Or...someone else can branch your published branch! Your friend does:

```
bzr branch lp:~$user/drizzle/$branchname
```

- And branches your code...

Proposing your branch for merging

- What good is your code if it lives all by itself?
- Get your code reviewed and merged into the “mainline”
- You must request your branch to be merged
- Go to your branch on Launchpad.net:
  - http://code.launchpad.net/~$user/drizzle/$branchname

Best Practice #2

- Launchpad Blueprints are a way to track progress on tasks you work on
- Create detailed blueprints for stuff you work on and you can:
  - Assign the blueprint to yourself
  - Link your branch to the blueprint
  - Track progress of your work on a task
  - Create dependencies (and visualize them)
  - Request mentoring on your task
  - Offer mentoring to someone else!

Inside the Code

Overview of the Drizzle Code Base

directory organization

- /client
  - Client programs (drizzle.cc, drizzledump.cc etc)
- /config
  - Scripts such as autorun.sh for the build process
- /extra
  - Contains my_print_defaults.cc
  - Will be going away this summer (yeah! \o/)
- /gnulib
  - Portability headers

directory organization (cont'd)

- /support-files
  - Various utility scripts
- /tests
  - Unit and functional test cases and suites
  - As a contributor, you will want to familiarize yourself with this directory! :)
- /drizzled
  - ALL kernel code
  - Optimizer, parser, runtime, plugin *APIs*

/drizzled directory

- /drizzled/memory
  - Legacy memory allocation
  - Will be a day of days when it is removed
- /drizzled/internal
  - MySQL portability/system library
  - Many things removed from original MySQL mysys library
  - You should take care when using any function in here
    - Check for a standard library prototype first!

/drizzled (kernel code)

- /drizzled/atomic
  - Portable C++ atomic<> implementation
- /drizzled/message
  - Google Protobuffer proto definitions
- /drizzled/utf8
  - C++ UTF8 thin library
- /drizzled/util
  - Bits and pieces of utility code

/drizzled (cont'd)

- /drizzled/plugin
  - Plugin base interface class definitions
- /drizzled/item
  - Item derived classes
- /drizzled/field
  - Field storage classes
- /drizzled/function
  - Built-in SQL functions

/drizzled (cont'd)

- /drizzled/optimizer
  - Most optimizer code, range operations, aggregation
- /drizzled/statement
  - SQL Statement classes
  - e.g. statement::Insert
- /drizzled/algorithm
  - crc32, sha1, etc..

/plugin (module code)

- Lots of plugin examples and default implementations
  - Authentication
  - Data Dictionaries (TableFunction)
  - Replicators
  - Transaction log
  - Logging
  - Session scheduling
  - Pluggable functions
  - Storage engines

libdrizzle

- BSD licensed, written in pure C by Eric Day
- Client/server communication protocol
- Clean, stack-based approach
  - http://launchpad.net/libdrizzle
- Requirement for developing Drizzle:

```
bzr branch lp:libdrizzle libdrizzle
cd libdrizzle; ./config/autorun.sh; ./configure
make && make check
sudo make install
```

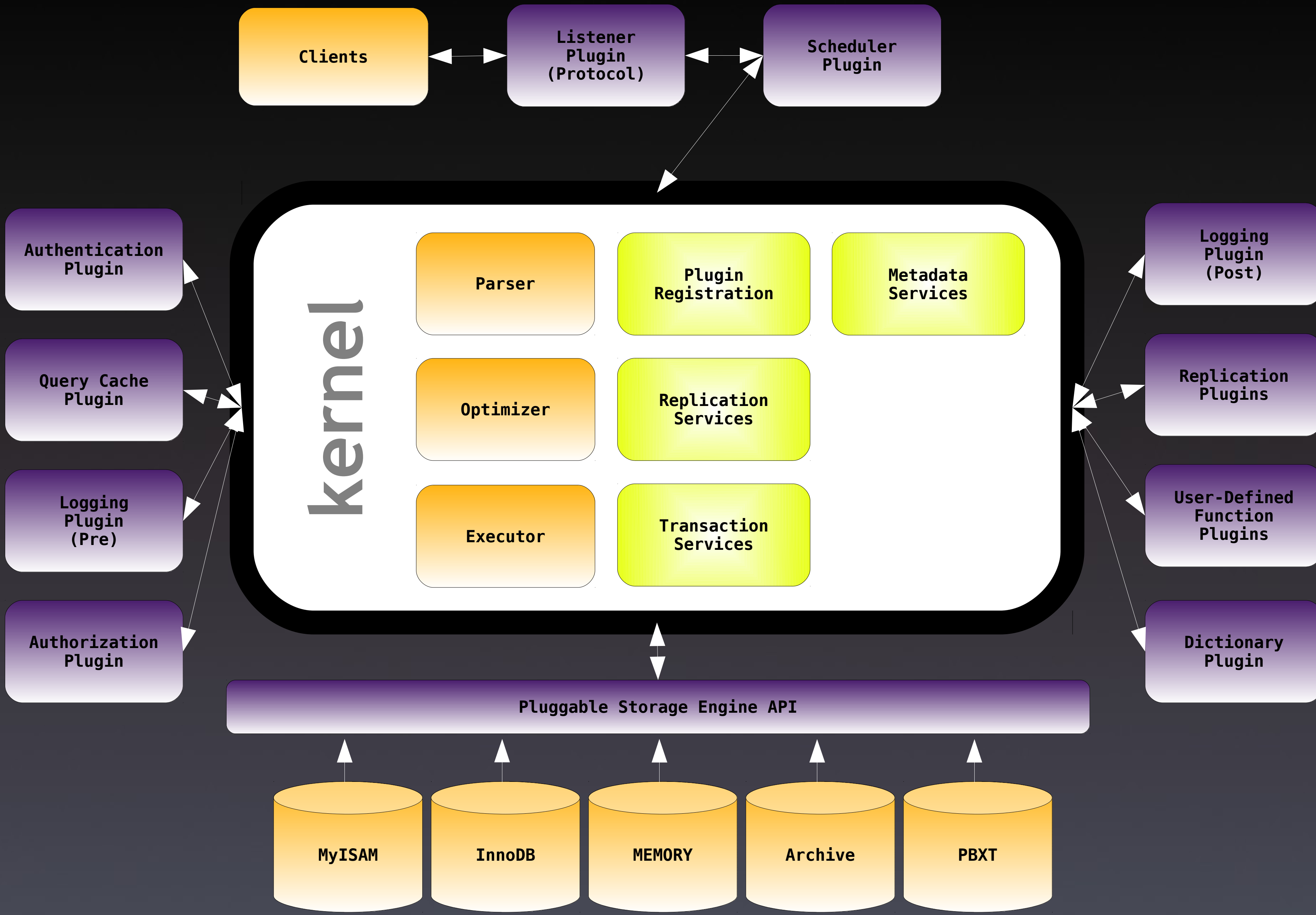
Overview of Drizzle's Architecture

drizzle's system architecture

- “Microkernel” design means most features are built as plugins
  - Authentication, replication, logging, information schema, storage engine, etc
  - The kernel is really just the parser, optimizer, and runtime
- We are C++, not C+
- We use open source libraries as much as possible
  - STL, gettext, Boost, pcre, GPB, etc
  - Don't reinvent the wheel

drizzle's system architecture

- No single “right way” of implementing something
  - Your solution may be great for your environment, but not good for others
  - And that's fine – it's what the plugin system is all about
- We focus on the APIs so you can focus on the implementation
- Drizzle is just one part of a large ecosystem
  - Web servers, caching layers, authentication systems


Clients
Listener Plugin (Protocol)
Scheduler Plugin
Authentication Plugin
Query Cache Plugin
Logging Plugin (Pre)
Authorization Plugin
kernel
Parser
Plugin Registration
Metadata Services
Optimizer
Replication Services
Executor
Transaction Services
Logging Plugin (Post)
Replication Plugins
User-Defined Function Plugins
Dictionary Plugin
Pluggable Storage Engine API
MyISAM
InnoDB
MEMORY
Archive
PBXT

ignore the kernel

- You should be able to ignore the kernel as a "black box"
- Plugin developers should focus on their plugin or module and not change anything in the kernel
- If you need to meddle with or change something in the kernel, it is a sign of a bad interface
  - And you should file a bug! :)

Walkthrough of Drizzle
Plugin Basics

plugin/module development basics

- A working C++ development environment
  - http://www.joinfu.com/2008/08/getting-a-working-c-c-plusplus-development-environment-for-developing-drizzle/
- A module in Drizzle is a set of source files in **`/plugin/`** that implements some functionality
  - For instance **`/plugin/transaction_log/*`** contains all files for the Transaction Log module
- Each module must have a **`plugin.ini`** file
  - The fabulous work by Monty Taylor on the Pandora build system automates most work for you

plugin/module development basics

- A module contains one or more implementations of a plugin class
- A plugin class is any class interface declared in **`/drizzled/plugin/`**
  - For instance, the header file **`/drizzled/plugin/transaction_applier.h`** declares the interface for the **`plugin::TransactionApplier`** API
  - The header files contain documentation for the plugin interfaces
  - You can also see documentation on the drizzle.org website: **`http://drizzle.org/doxygen/`**

the plugin.ini

- A description file for the plugin
- Read during compilation and Pandora build system creates appropriate linkage for you
- Required fields:
  - headers= <list of all header files in module>
  - sources= <list of all source files in module>
  - title= <name of the module/plugin>
  - description= <decription for the module>

from plugin.ini to data dictionary

```
[plugin]
title=Filtered Replicator
author=Padraig O Sullivan
version=0.2
license=PLUGIN_LICENSE_GPL
description=
  A simple filtered replicator which allows a user to filter out events based on a schema or
table name
load_by_default=yes
sources=filtered_replicator.cc
headers=filtered_replicator.h
```

```
drizzle> SELECT * FROM DATA_DICTIONARY.MODULES
    -> WHERE MODULE_NAME LIKE 'FILTERED%'\G
*************************** 1. row ***************************
       MODULE_NAME: filtered_replicator
    MODULE_VERSION: 0.2
     MODULE_AUTHOR: Padraig O'Sullivan
        IS_BUILTIN: FALSE
    MODULE_LIBRARY: filtered_replicator
MODULE_DESCRIPTION: Filtered Replicator
    MODULE_LICENSE: GPL

drizzle> SELECT * FROM DATA_DICTIONARY.PLUGINS
    -> WHERE PLUGIN_NAME LIKE 'FILTERED%'\G
*************************** 1. row ***************************
PLUGIN_NAME: filtered_replicator
PLUGIN_TYPE: TransactionReplicator
  IS_ACTIVE: TRUE
MODULE_NAME: filtered_replicator
```

module initialization

- Recommend placing module-level variables and routines in **`/plugin/$module/module.cc`**
- Required: an initialization function taking a reference to the **`plugin::Context`** object for your module as its only parameter
  - Typically named **`init()`**
- Optional: module-level system variables
- Required: **`DECLARE_PLUGIN($init, $vars)`** macro inside above source file

module initialization example

```
static DefaultReplicator *default_replicator= NULL; /* The singleton replicator */

static int init(plugin::Context &context)
{
  default_replicator= new DefaultReplicator("default_replicator");
  context.add(default_replicator);
  return 0;
}

DRIZZLE_PLUGIN(init,  NULL);
```

what are plugin hooks?

- Places in the source code that notify plugins about certain events are called *plugin hooks*
- During the course of a query's execution, many plugin hooks can be called
- The subclass of plugin::Plugin determines on which events a plugin is notified and what gets passed as a state parameter to the plugin during notification
- These plugin hooks define the plugin's *API*

Example: plugin::Authentication

```
class Authentication : public Plugin
{
public:
  explicit Authentication(std::string name_arg)
    : Plugin(name_arg, "Authentication")
  {}
  virtual ~Authentication() {}

  virtual bool authenticate(const SecurityContext &sctx,
                            const std::string &passwd)= 0;

  static bool isAuthenticated(const SecurityContext &sctx,
                              const std::string &password);
};
```

- `authenticate()` is the pure virtual method that an implementing class should complete
- `isAuthenticated()` is the plugin hook that is called by the kernel to determine authorization

example plugin hook

```
class AuthenticateBy : public unary_function<plugin::Authentication *, bool>
{
...
  inline result_type operator()(argument_type auth)
  {
    return auth->authenticate(sctx, password);
  }
};

bool plugin::Authentication::isAuthenticated(const SecurityContext &sctx,
                                             const string &password)
{
...
  /* Use find_if instead of foreach so that we can collect return codes */
  vector<plugin::Authentication *>::iterator iter=
    find_if(all_authentication.begin(), all_authentication.end(),
            AuthenticateBy(sctx, password));
...
  if (iter == all_authentication.end())
  {
    my_error(ER_ACCESS_DENIED_ERROR, MYF(0),
             sctx.getUser().c_str(),
             sctx.getIp().c_str(),
             password.empty() ? ER(ER_NO) : ER(ER_YES));
    return false;
  }
  return true;
}
```

testing your plugin

- No plugin should be without corresponding test cases
- Luckily, again because of the work of Monty Taylor, your plugin can easily hook into the Drizzle testing system
- Create a `tests/` directory in your plugin's directory, containing a `t/` and an `r/` subdirectory (for “test” and “result”)

creating test cases

- Your plugin will most likely not be set to load by default
- To activate your plugin, you need to start the server during your tests with:

`--plugin-add=$module`

- To automatically have the server started with command-line options by the Drizzle test suite, create a file called `$testname-master.opt` and place it along with your test case in your `/plugin/$module/tests/t/` directory

running your test cases

- Simply run the test-run.pl script with your suite:

```
jpipes@serialcoder:~/repos/drizzle/trunk$ cd tests/
jpipes@serialcoder:~/repos/drizzle/trunk/tests$ ./test-run --suite=transaction_log
Drizzle Version 2010.04.1439
...
================================================================================
DEFAULT STORAGE ENGINE: innodb
TEST                                                        RESULT    TIME (ms)
--------------------------------------------------------------------------------

transaction_log.alter                                       [ pass ]    1025
transaction_log.auto_commit                                 [ pass ]     650
transaction_log.blob                                        [ pass ]     661
transaction_log.create_select                               [ pass ]     688
transaction_log.create_table                                [ pass ]     413
transaction_log.delete                                      [ pass ]    1744
transaction_log.filtered_replicator                         [ pass ]    6132
...
transaction_log.schema                                      [ pass ]     137
transaction_log.select_for_update                           [ pass ]    6496
transaction_log.slap                                        [ pass ]   42522
transaction_log.sync_method_every_write                     [ pass ]      23
transaction_log.temp_tables                                 [ pass ]     549
transaction_log.truncate                                    [ pass ]     441
transaction_log.truncate_log                                [ pass ]     390
transaction_log.udf_print_transaction_message               [ pass ]     408
transaction_log.update                                      [ pass ]    1916
--------------------------------------------------------------------------------
Stopping All Servers
All 28 tests were successful.
```

Overview of Drizzle's Replication System

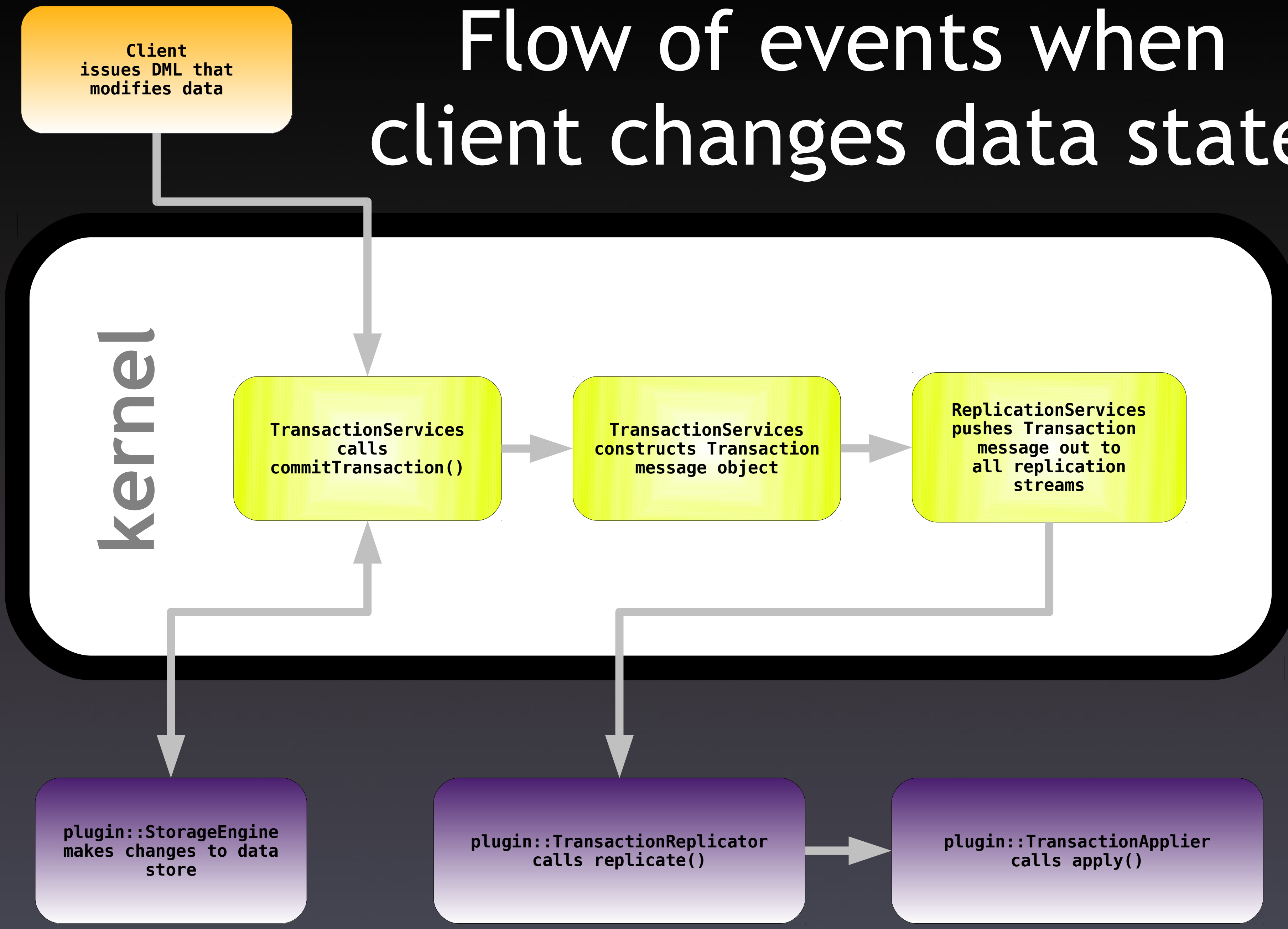
not in ~~Kansas~~ MySQL anymore

- Drizzle's replication system looks nothing like MySQL
- Drizzle is entirely row-based (yes even DDL)
- Forget the terms *master*, *slave*, and *binlog*
- We use the terms *publisher*, *subscriber*, *replicator* and *applier*
- We have a transaction log, but it is *not required* for replication
  - Drizzle's transaction log is a *module*
  - The transaction log module has example implementations of an *applier*

role of the kernel in replication

- *Marshall* all sources of and targets for replicated data
- *Construct* objects of type `message::Transaction` that represent the changes made in the server
- *Push* the Transaction messages out to the replication streams
- *Coordinate* requests from Subscribers with registered Publishers

Flow of events when client changes data state



what is a *replication stream*?

- A replication stream is the pair of a replicator and an applier
- Each applier must be matched with a replicator
  - Can be done via command-line arguments
  - Can be hard-coded
- To see the replication streams that are active, you can query **DATA_DICTIONARY.REPLICATION_STREAMS**:

```
drizzle> select * from data_dictionary.replication_streams;
+--------------------+-------------------------+
| REPLICATOR         | APPLIER                 |
+--------------------+-------------------------+
| default_replicator | transaction_log_applier |
+--------------------+-------------------------+
1 row in set (0 sec)
```

the Transaction message

- The Transaction message is the basic unit of work in the replication system
- Represents a set of changes that were made to a server
- Compressed binary format
- Google Protobuffer message

Understanding Google Protobuffers

protobuffers are XML on crack

- Google protobuffers
  - Compiler (`protoc`)
  - Library (`libprotobuf`)
- Compiler consumes a `.proto` file and produces source code files containing classes the represent your data
  - In a variety of programming languages
- Library contains routines and classes used in working with, serializing, and parsing protobuffer messages

http://code.google.com/apis/protocolbuffers/docs/overview.html

The .proto file

- Declares *message* definitions
  - Simple Java/C++-like format
- Messages have one or more *fields*
- Fields are of a specific *type*
  - uint32, string, bytes, etc.
- Fields have a *specifier*
  - required, optional, repeated
- Submessages and enumerations too!

example .proto file

```
package drizzled.message;
option optimize_for = SPEED;

/*
  Context for a transaction.
*/
message TransactionContext
{
  required uint32 server_id = 1; /* Unique identifier of a server */
  required uint64 transaction_id = 2; /* Globally-unique transaction ID */
  required uint64 start_timestamp = 3; /* Timestamp of when the transaction started */
  required uint64 end_timestamp = 4; /* Timestamp of when the transaction ended */
}
```

- `package` sets the namespace for the generated code
  - In C++, the TransactionContext class would be created in the `drizzled::message::` namespace
- To compile the .proto, we use the protoc compiler:

```
$> protoc --cpp_out=. transaction.proto
```

generated code files

- For C++, protoc produces two files, one header and one source file
  - transaction.pb.h, transaction.pb.cc
- To use these classes, simply #include the header file and start using your new message classes:

```
#include "transaction.pb.h";

using namespace drizzled;

message::TransactionContext tc;
tc.set_transaction_id(100000);
...
```

The C++ POD GPB API in one slide

- To access the data, method is same as the field
- To set the data, append `set_` to the field name
- To check existence, append `has_` to the field name
- To add a new repeated field, append `add_` to the field name
- To get a pointer to a field that is a submessage, append `mutable_` to the field name
  - All memory for fields is managed by GPB; when you delete the main object, all memory is freed

serializing GPB messages

- Serialize to a C++ stream:

```
message::Transaction transaction;
// fill the transaction's fields...
fstream output("myfile", ios::out | ios::binary);
transaction.SerializeToOstream(&output);
```

- or a file descriptor:

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
#include <stdio.h>

using namespace google;

int myfile= open("myfile", O_WRONLY);
protobuf::io::ZeroCopyOutputStream *output= new protobuf::io::FileOutputStream(myfile);
transaction.SerializeToZeroCopyStream(output);
```

- or a std::string:

```
string buffer("");
transaction.SerializeToString(&buffer);
```

serialize to raw bytes

- Full control...serializing to raw bytes:

```
#include <google/protobuf/io/coded_stream.h>
#include <vector>

using namespace google;

size_t message_byte_length= transaction.ByteSize();
vector<uint8_t> buffer;
uint8_t *ptr= &buffer[0];

buffer.reserve(message_byte_length + sizeof(uint32_t));

/*
 * Write the length of the message then the serialized
 * message to the raw byte buffer
 */
ptr= protobuf::io::CodedOutputStream::WriteLittleEndian32ToArray(
    static_cast<uint32_t>(message_byte_length), ptr);

ptr= transaction.SerializeWithCachedSizesToArray(ptr);
```

parsing serialized GPB messages

- Parsing from a C++ stream:

```
message::Transaction transaction;
fstream output("myfile", ios::in | ios::binary);
transaction.ParseFromIstream(&output);
```

- or a file descriptor:

```
#include <google/protobuf/io/zero_copy_stream_impl.h>
#include <stdio.h>

using namespace google;

int myfile= open("myfile", O_RDONLY);
protobuf::io::ZeroCopyOutputStream *intput= new protobuf::io::FileInputStream(myfile);
transaction.ParseFromZeroCopyStream(input);
```

- or a std::string:

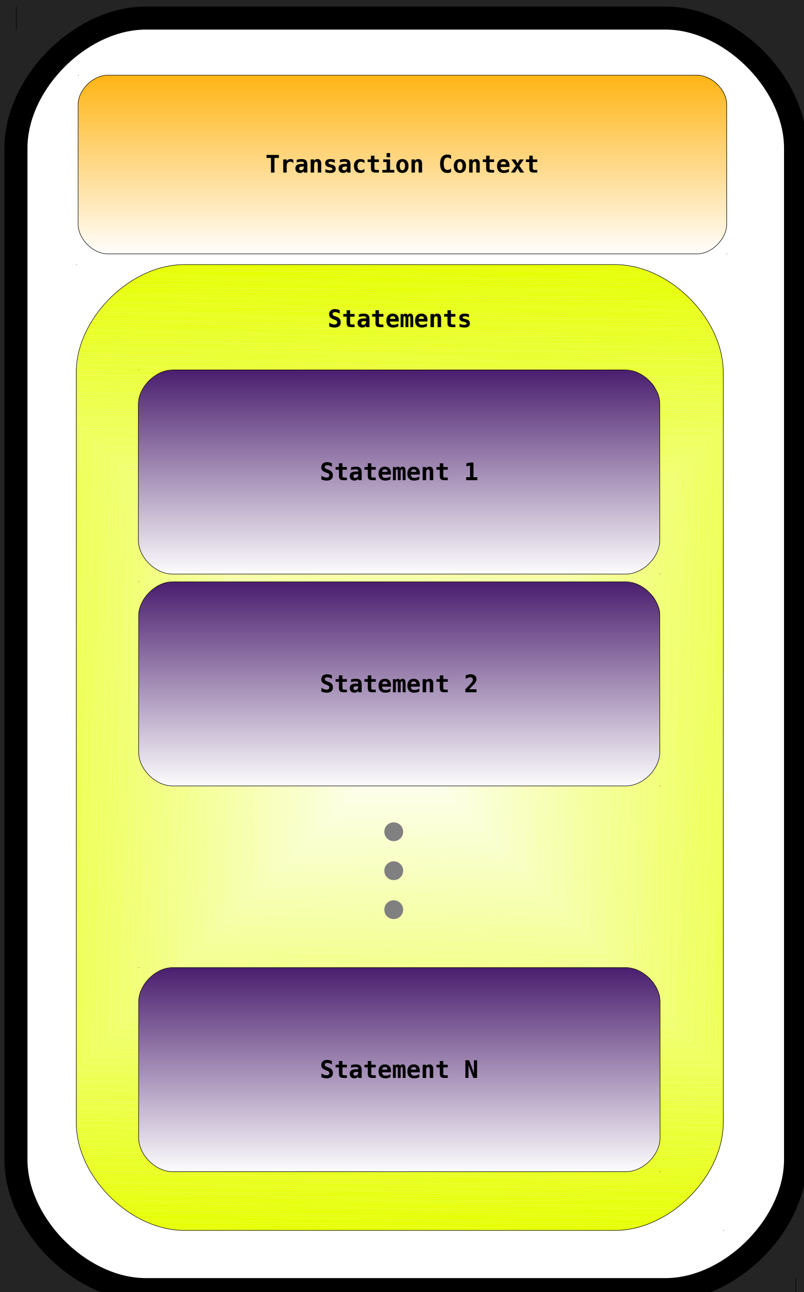
```
string buffer("");
transaction.SerializeToString(&buffer);

message::Transaction copy_transaction;
copy_transaction.ParseFromString(buffer);
```

The Transaction message

the Transaction message

- The Transaction message is the basic unit of work in the replication system
- Compressed binary format
- Represents a set of changes that were made to a server
- Most of the time, the Transaction message represents the work done in a single SQL transaction
  - Large SQL transactions may be broken into multiple Transaction messages

the Transaction message format



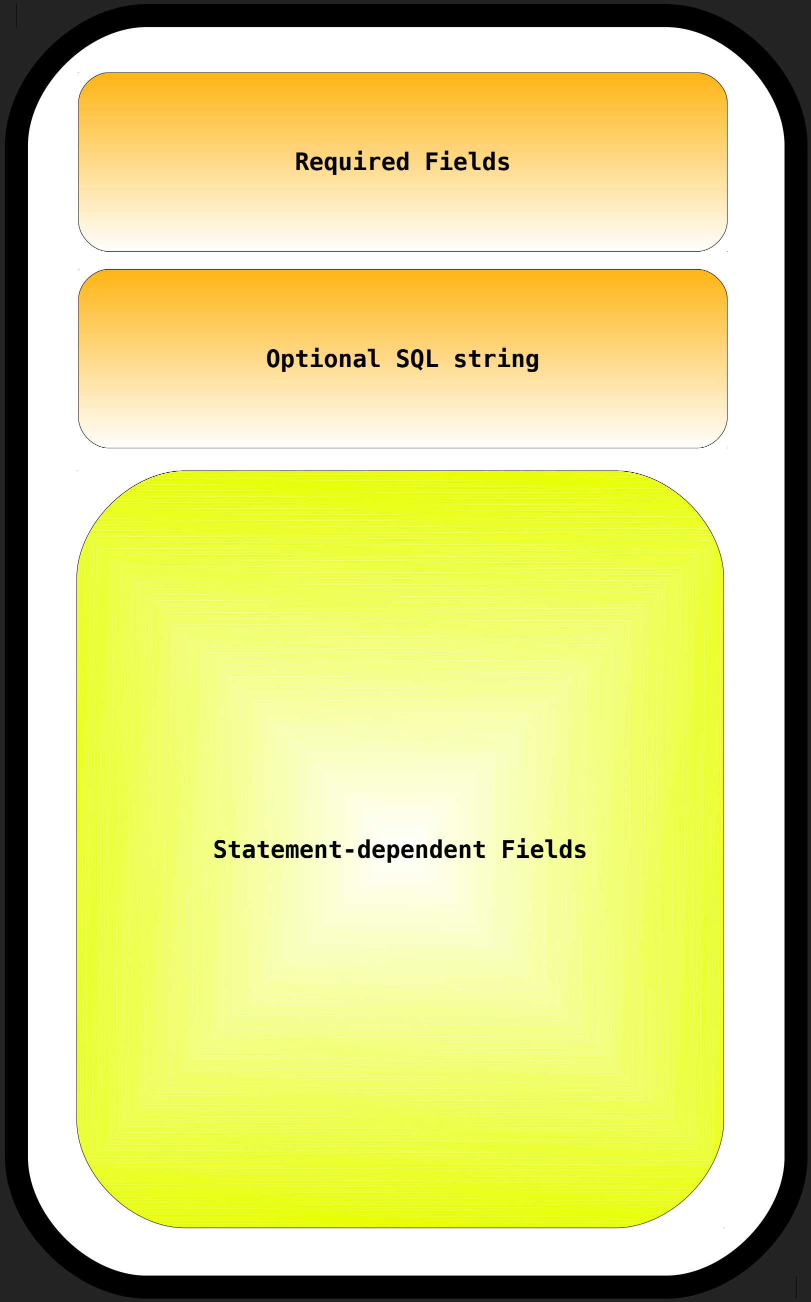
- TransactionContext
  - Transaction ID
  - Start and end timestamps
  - Server ID
  - Channel ID (optional)
- Statements
  - One or more Statement submessages
  - Describes the rows modified in a SQL statement

TransactionContext message

```
message Transaction
{
  required TransactionContext transaction_context = 1;
  repeated Statement statement = 2;
}

message TransactionContext
{
  required uint32 server_id = 1; /* Unique identifier of a server */
  required uint64 transaction_id = 2; /* Channel-unique transaction ID */
  required uint64 start_timestamp = 3; /* Timestamp of when the transaction started */
  required uint64 end_timestamp = 4; /* Timestamp of when the transaction ended */
  optional uint32 channel_id = 5; /* Scope of uniqueness of transaction ID */
}
```

- Would you add additional fields?
  - user_id? session_id? something else?
- Add fields as optional, recompile, able to use those custom fields right away in your plugins
  - Now *that's* extensible!

the Statement message format



- Required fields
  - Type
  - Start and end timestamps
- Optional SQL string
- Statement-dependent fields
  - For DML: header and data message
  - For DDL: submessage representing a DDL statement

the Statement message

```
message Statement
{
  enum Type
  {
    ROLLBACK = 0; /* A ROLLBACK indicator */
    INSERT = 1; /* An INSERT statement */
    DELETE = 2; /* A DELETE statement */
    UPDATE = 3; /* An UPDATE statement */
    TRUNCATE_TABLE = 4; /* A TRUNCATE TABLE statement */
    CREATE_SCHEMA = 5; /* A CREATE SCHEMA statement */
    ALTER_SCHEMA = 6; /* An ALTER SCHEMA statement */
    DROP_SCHEMA = 7; /* A DROP SCHEMA statement */
    CREATE_TABLE = 8; /* A CREATE TABLE statement */
    ALTER_TABLE = 9; /* An ALTER TABLE statement */
    DROP_TABLE = 10; /* A DROP TABLE statement */
    SET_VARIABLE = 98; /* A SET statement */
    RAW_SQL = 99; /* A raw SQL statement */
  }
  required Type type = 1; /* The type of the Statement */
  required uint64 start_timestamp = 2; /* Nanosecond precision timestamp of when the
                                          Statement was started on the server */
  required uint64 end_timestamp = 3; /* Nanosecond precision timestamp of when the
                                        Statement finished executing on the server */
  optional string sql = 4; /* May contain the original SQL string */

  /* ... (cont'd on later slide) */
}
```

getting data from the message

- For data fields in a message, to get the value of the field, simply call a method the same as the name of the field:

```
message::Transaction &transaction= getSomeTransaction();
const message::TransactionContext &trx_ctx= transaction.transaction_context();

cout << "Transaction ID: " << trx_ctx.transaction_id << endl;
```

- Enumerations are also easily used:

```
message::Statement::Type type= statement.type();

switch (type)
{
  case message::Statement::INSERT:
    // do something for an insert...
  case message::Statement::UPDATE:
    // do something for an update...
}
```

accessing a repeated element

- Elements in a repeated field are accessed via an index, and a $fieldname_size() method returns the number of elements:

```
using namespace drizzled;

const message::Transaction &transaction= getSomeTransaction();

/* Get the number of elements in the repeated field */
size_t num_statements= transaction.statement_size();

for (size_t x= 0; x < num_statements; ++x)
{
  /* Access the element via the 0-based index */
  const message::Statement &statement= transaction.statement(x);

  /* For optional fields, a has_$fieldname() method is available
     to check for existence */
  if (statement.has_sql())
  {
    cout << statement.sql() << endl;
  }
}
```

the specific Statement message

```
message Statement
{
  /* ... cont'd from a previous slide */

  /*
   * Each Statement message may contain one or more of
   * the below sub-messages, depending on the Statement's type.
   */
  optional InsertHeader insert_header = 5;
  optional InsertData insert_data = 6;
  optional UpdateHeader update_header = 7;
  optional UpdateData update_data = 8;
  optional DeleteHeader delete_header = 9;
  optional DeleteData delete_data = 10;
  optional TruncateTableStatement truncate_table_statement = 11;
  optional CreateSchemaStatement create_schema_statement = 12;
  optional DropSchemaStatement drop_schema_statement = 13;
  optional AlterSchemaStatement alter_schema_statement = 14;
  optional CreateTableStatement create_table_statement = 15;
  optional AlterTableStatement alter_table_statement = 16;
  optional DropTableStatement drop_table_statement = 17;
  optional SetVariableStatement set_variable_statement = 18;
}
```

- Example: for an INSERT SQL statement, the Statement message will contain an **`insert_header`** and **`insert_data`** field

insert header and data messages

```
/*
 * Represents statements which insert data into the database:
 *
 * INSERT
 * INSERT SELECT
 * LOAD DATA INFILE
 * REPLACE (is a delete and an insert)
 *
 * @note
 *
 * Bulk insert operations will have >1 data segment, with the last data
 * segment having its end_segment member set to true.
 */
message InsertHeader
{
  required TableMetadata table_metadata = 1; /* Metadata about the table affected */
  repeated FieldMetadata field_metadata = 2; /* Metadata about fields affected */
}

message InsertData
{
  required uint32 segment_id = 1; /* The segment number */
  required bool end_segment = 2; /* Is this the final segment? */
  repeated InsertRecord record = 3; /* The records inserted */
}

/*
 * Represents a single record being inserted into a single table.
 */
message InsertRecord
{
  repeated bytes insert_value = 1;
}
```

tip: statement_transform

- Looking for examples of how to use the Transaction and Statement messages?
- The `/drizzled/message/transaction.proto` file has extensive documentation
- Also check out the statement_transform library in `/drizzled/message/statement_transform.cc`
- Shows how to contruct SQL statements from the information in a Transaction message
- The statement_transform library is used in utility programs such as `/drizzled/message/table_raw_reader.cc`

Code walkthrough of the Filtered Replicator module

replicators can filter/transform

- `plugin::TransactionReplicator`'s function is to *replicate* the Transaction message to the `plugin::TransactionApplier` in a replication stream
- You can *filter* or *transform* a Transaction message before passing it off to the applier
- Only one method in the API:

```
/**
 * Replicate a Transaction message to a TransactionApplier.
 *
 * @param Pointer to the applier of the command message
 * @param Reference to the current session
 * @param Transaction message to be replicated
 */
virtual ReplicationReturnCode replicate(TransactionApplier *in_applier,
                                        Session &session,
                                        message::Transaction &to_replicate)= 0;
```

module overview

- Allows filtering of transaction messages by schema name or table name
  - We construct a new transaction message containing only Statement messages that have not been filtered
- Includes support for the use of regular expressions
- Schemas and tables to filter are specified in system variables
  - filtered_replicator_filteredschemas
  - filtered_replicator_filteredtables

module initialization

- Very similar to what we saw with the default replicator:

```
static FilteredReplicator *filtered_replicator= NULL;

static int init(plugin::Context &context)
{
  filtered_replicator= new(std::nothrow)
    FilteredReplicator("filtered_replicator",
                       sysvar_filtered_replicator_sch_filters,
                       sysvar_filtered_replicator_tab_filters);
  if (filtered_replicator == NULL)
  {
    return 1;
  }
  context.add(filtered_replicator);
  return 0;
}
```

obtaining schema/table name

- For each statement in the transaction message, we obtain the schema name and table name in the `parseStatementTableMetadata` method:

```
void parseStatementTableMetadata(const message::Statement &in_statement,
                                 string &in_schema_name,
                                 string &in_table_name) const
{
  switch (in_statement.type())
  {
    case message::Statement::INSERT:
    {
      const message::TableMetadata &metadata= in_statement.insert_header().table_metadata();
      in_schema_name.assign(metadata.schema_name());
      in_table_name.assign(metadata.table_name());
      break;
    }
    case message::Statement::UPDATE:
    …
  }
}
```

filtering by schema name

- We search through the list of schemas to filter to see if there is a match

```
pthread_mutex_lock(&sch_vector_lock);
vector<string>::iterator it= find(schemas_to_filter.begin(),
                                  schemas_to_filter.end(),
                                  schema_name);
if (it != schemas_to_filter.end())
{
  pthread_mutex_unlock(&sch_vector_lock);
  return true;
}
pthread_mutex_unlock(&sch_vector_lock);
```

regular expression filtering

- We use pcre to perform regular expression filtering if enabled:

```
/*
 * If regular expression matching is enabled for schemas to filter, then
 * we check to see if this schema name matches the regular expression that
 * has been specified.
 */
if (sch_regex_enabled)
{
  int32_t result= pcre_exec(sch_re,
                            NULL,
                            schema_name.c_str(),
                            schema_name.length(),
                            0,
                            0,
                            NULL,
                            0);
  if (result >= 0)
  {
    return true;
  }
}
```

filtering Statements

- Schema and table name are converted to lower case since we store the list of schemas and tables to filter in lower case
- If neither matches a filtering condition, we add the statement to our new filtered transaction:

```
/* convert schema name and table name to lower case */
std::transform(schema_name.begin(), schema_name.end(),
               schema_name.begin(), ::tolower);
std::transform(table_name.begin(), table_name.end(),
               table_name.begin(), ::tolower);


if (! isSchemaFiltered(schema_name) &&
    ! isTableFiltered(table_name))
{
  message::Statement *s= filtered_transaction.add_statement();
  *s= statement; /* copy construct */
}
```

pass Transaction on to applier

- Finally, we pass on our filtered transaction to an applier:

```
if (filtered_transaction.statement_size() > 0)
{
  /*
   * We can now simply call the applier's apply() method, passing
   * along the supplied command.
   */
  message::TransactionContext *tc= filtered_transaction.mutable_transaction_context();
  *tc= to_replicate.transaction_context(); /* copy construct */
  return in_applier->apply(in_session, filtered_transaction);
}
```

system variables

- Control module's configuration
- Each system variable has two associated functions
  - A check function which can verify the input is correct
  - An update function which actually updates the value of the variable
- System variable handling will be over-hauled in Drizzle so not essential to understand how these currently work

Code walkthrough of the Transaction Log module

appliers can log/analyze/apply

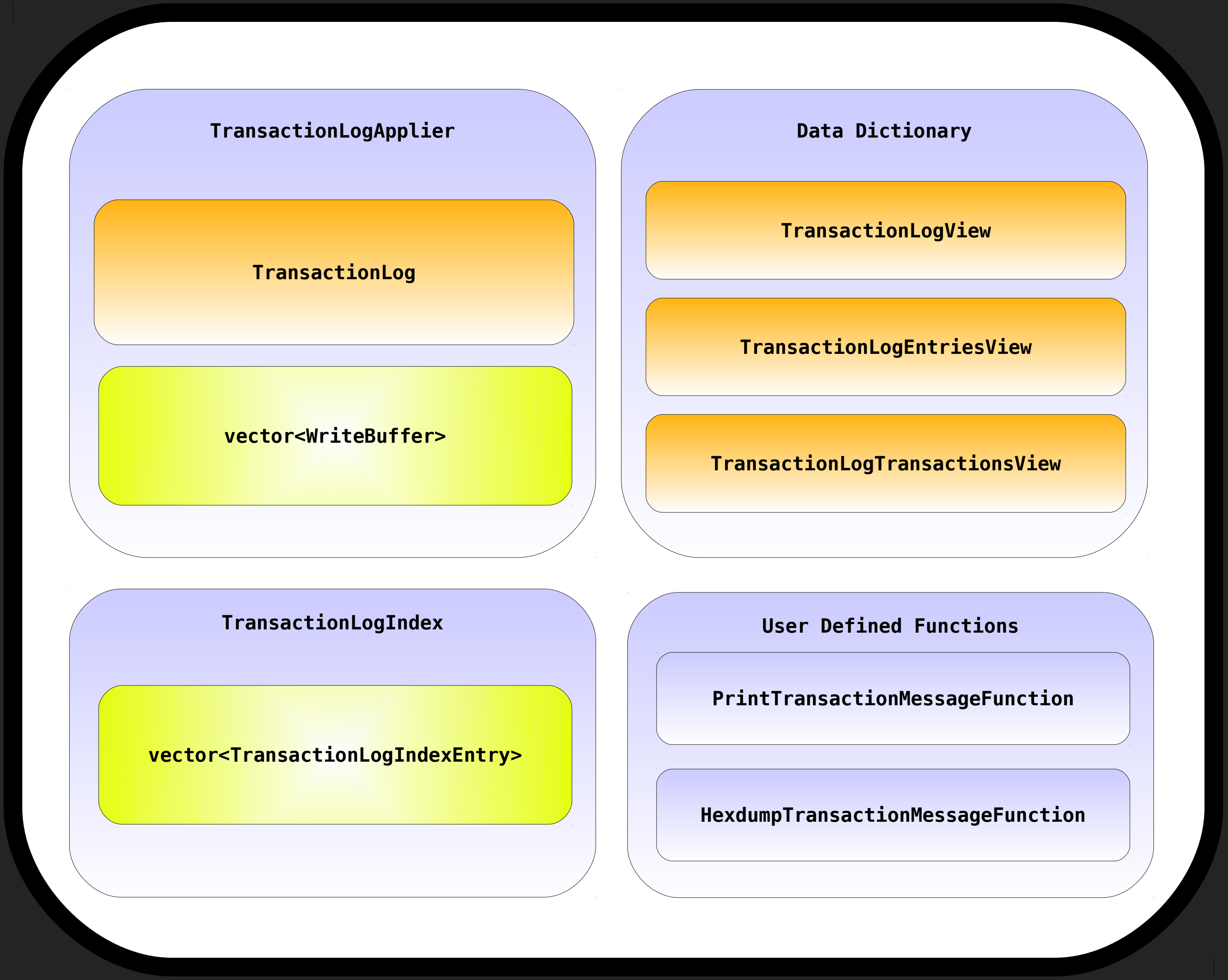
- **`plugin::TransactionApplier`**'s function is to *apply* the Transaction message to some target or analyze the transaction in some way
- You cannot modify the Transaction message
  - If you need to modify the message, you likely should be using **`TransactionReplicator::replicate()`**
- Only one method in the API:

```
/**
 * Applies a Transaction message to some target
 *
 * @param Reference to the current session
 * @param Transaction message to be applied
 */
virtual ReplicationReturnCode apply(Session &session,
                                    const message::Transaction &to_apply)= 0;
```

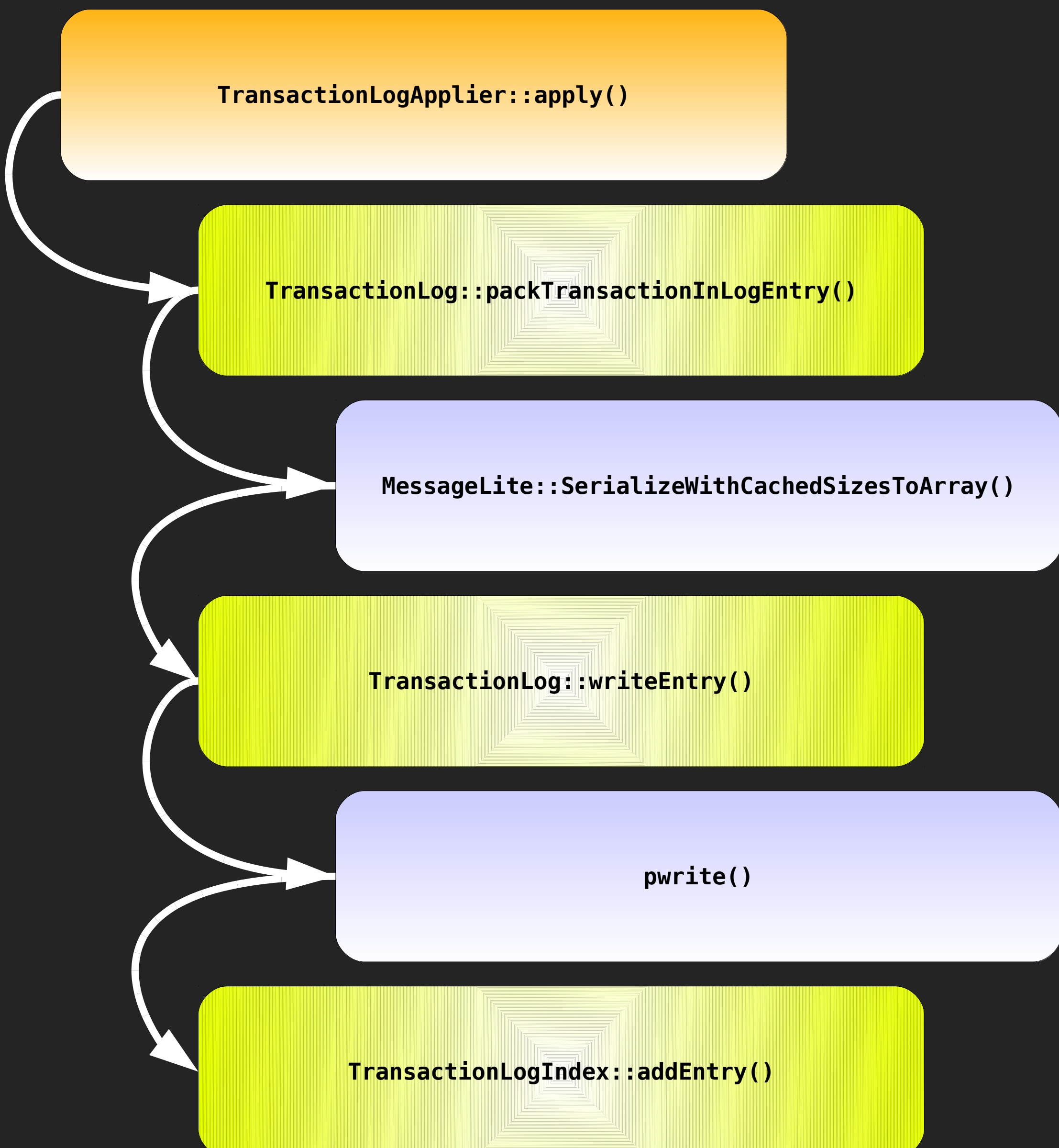
module overview

- Provides a log of compressed, serialized Transaction messages
- Supports checksumming of written messages
- Flexible file sync behaviour
  - Similar to **`innodb_flush_log_at_trx_commit`**
- Uses a scoreboard of write buffers to minimize memory usage
- Components are all plugin examples
  - TransactionApplier, Data Dictionary, user-defined Functions

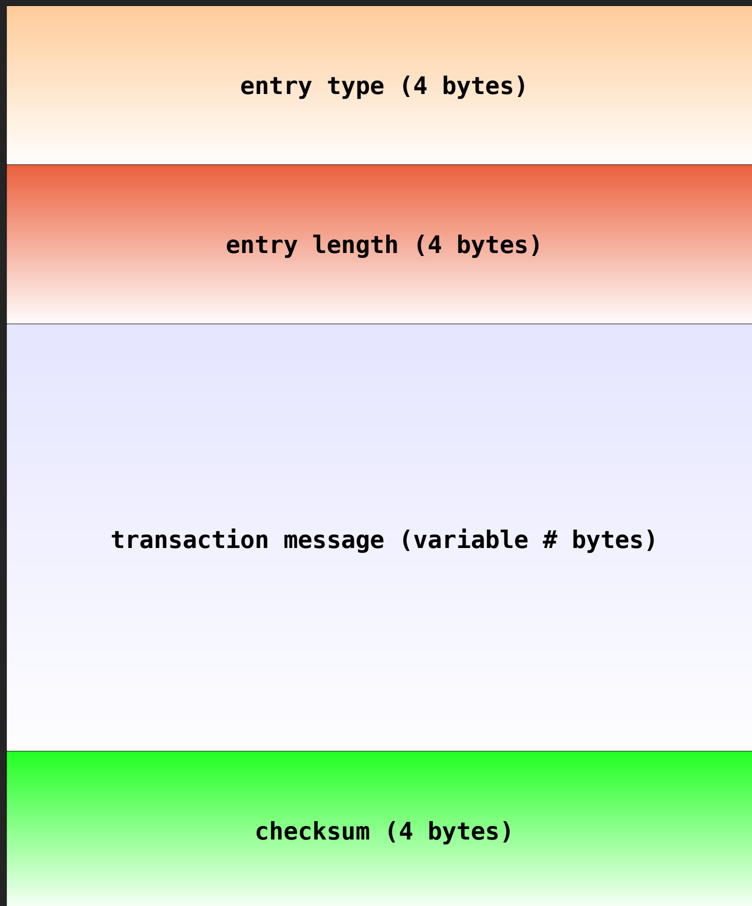
transaction log components



code flow through module



transaction log entry format



TransactionLogApplier header

```
class TransactionLogApplier: public drizzled::plugin::TransactionApplier
{
public:
  TransactionLogApplier(const std::string name_arg,
                        TransactionLog *in_transaction_log,
                        uint32_t in_num_write_buffers);

  /** Destructor */
  ~TransactionLogApplier();

  /**
   * Applies a Transaction to the transaction log
   *
   * @param Session descriptor
   * @param Transaction message to be replicated
   */
  drizzled::plugin::ReplicationReturnCode
  apply(drizzled::Session &in_session,
        const drizzled::message::Transaction &to_apply);
private:
  TransactionLog &transaction_log;
  /* This Applier owns the memory of the associated TransactionLog - so we
    have to track it. */
  TransactionLog *transaction_log_ptr;
  uint32_t num_write_buffers; ///< Number of write buffers used
  std::vector<WriteBuffer *> write_buffers; ///< array of write buffers

  /**
   * Returns the write buffer for the supplied session
   *
   * @param Session descriptor
   */
  WriteBuffer *getWriteBuffer(const drizzled::Session &session);
};
```

TransactionLog header

```
class TransactionLog
{
public:
  static size_t getLogEntrySize(const drizzled::message::Transaction &trx);

  uint8_t *packTransactionIntoLogEntry(const drizzled::message::Transaction &trx,
                                       uint8_t *buffer,
                                       uint32_t *checksum_out);

  off_t writeEntry(const uint8_t *data, size_t data_length);
private:
  static const uint32_t HEADER_TRAILER_BYTES= sizeof(uint32_t) + /* 4-byte msg type header */
                                              sizeof(uint32_t) + /* 4-byte length header */
                                              sizeof(uint32_t); /* 4 byte checksum trailer */

  int syncLogFile();

  int log_file; ///< Handle for our log file
  drizzled::atomic<off_t> log_offset; ///< Offset in log file where we write next entry
  uint32_t sync_method; ///< Determines behaviour of syncing log file
  time_t last_sync_time; ///< Last time the log file was synced
  bool do_checksum; ///< Do a CRC32 checksum when writing Transaction message to log?
};
```

TransactionLogApplier::apply()

```
plugin::ReplicationReturnCode
TransactionLogApplier::apply(Session &in_session,
                             const message::Transaction &to_apply)
{
  size_t entry_size= TransactionLog::getLogEntrySize(to_apply);
  WriteBuffer *write_buffer= getWriteBuffer(in_session);

  uint32_t checksum;

  write_buffer->lock();
  write_buffer->resize(entry_size);
  uint8_t *bytes= write_buffer->getRawBytes();
  bytes= transaction_log.packTransactionIntoLogEntry(to_apply,
                                                     bytes,
                                                     &checksum);

  off_t written_to= transaction_log.writeEntry(bytes, entry_size);
  write_buffer->unlock();

  /* Add an entry to the index describing what was just applied */
  transaction_log_index->addEntry(TransactionLogEntry(ReplicationServices::TRANSACTION,
                                                      written_to,
                                                      entry_size),
                                  to_apply,
                                  checksum);
  return plugin::SUCCESS;
}
```

TransactionLog::packTransactionIntoLogEntry()

```
uint8_t *TransactionLog::packTransactionIntoLogEntry(const message::Transaction &trx,
                                                     uint8_t *buffer,
                                                     uint32_t *checksum_out)
{
  uint8_t *orig_buffer= buffer;
  size_t message_byte_length= trx.ByteSize();

  /*
   * Write the header information, which is the message type and
   * the length of the transaction message into the buffer
   */
  buffer= protobuf::io::CodedOutputStream::WriteLittleEndian32ToArray(
      static_cast<uint32_t>(ReplicationServices::TRANSACTION), buffer);
  buffer= protobuf::io::CodedOutputStream::WriteLittleEndian32ToArray(
      static_cast<uint32_t>(message_byte_length), buffer);

  /*
   * Now write the serialized transaction message, followed
   * by the optional checksum into the buffer.
   */
  buffer= trx.SerializeWithCachedSizesToArray(buffer);

  if (do_checksum)
  {
    *checksum_out= drizzled::algorithm::crc32(
        reinterpret_cast<char *>(buffer) - message_byte_length, message_byte_length);
  }
  else
    *checksum_out= 0;

  /* We always write in network byte order */
  buffer= protobuf::io::CodedOutputStream::WriteLittleEndian32ToArray(*checksum_out, buffer);
  /* Reset the pointer back to its original location... */
  buffer= orig_buffer;
  return orig_buffer;
}
```

TransactionLog::writeEntry()

```
off_t TransactionLog::writeEntry(const uint8_t *data, size_t data_length)
{
  ssize_t written= 0;

  /* Do an atomic increment on the offset of the log file position */
  off_t cur_offset= log_offset.fetch_and_add(static_cast<off_t>(data_length));

  /* Write the full buffer in one swoop */
  do
  {
    written= pwrite(log_file, data, data_length, cur_offset);
  }
  while (written == -1 && errno == EINTR); /* Just retry the write when interrupted */

  if (unlikely(written != static_cast<ssize_t>(data_length)))
  {
    errmsg_printf(ERRMSG_LVL_ERROR,
                  _("Failed to write full size of log entry.  Tried to write %" PRId64
                    " bytes at offset %" PRId64 ", but only wrote %" PRId32
                    " bytes.  Error: %s\n"),
                  static_cast<int64_t>(data_length),
                  static_cast<int64_t>(cur_offset),
                  static_cast<int64_t>(written),
                  strerror(errno));
  }

  int error_code= syncLogFile();

  if (unlikely(error_code != 0))
  {
    errmsg_printf(ERRMSG_LVL_ERROR,
                  _("Failed to sync log file. Got error: %s\n"),
                  strerror(errno));
  }
  return cur_offset;
}
```

What's up with the Publisher and Subscriber plugins?

we need your input

- These plugin's APIs are still being developed
- The idea is for responsibility to be divided like so:
  - plugin::Publisher will be responsible for describing the state of each replication channel and communicating with subscribers on separate ports
    - Think: a Publisher is a specialized server for each subscriber
  - plugin::Subscriber will be responsible for pulling data from a plugin::Publisher and applying that data to a replica node
    - Think: relay-log.info and master.info files as a C++ class interface

Possible SQL API

- SQL API for communications yet to be finalized
- Possible SQL to run on a replica node:

```
SUBSCRIBE TO <host> [CHANNEL n]
  [UNTIL [<timestamp> | <transaction_id>]]
```

- Possible SQL to create a snapshot archive for shipping to a new node for starting up a new replica:

```
BACKUP <schema_list> TO <archive_filename>
  [UNTIL [<timestamp> | <transaction_id>]]
```

rabbitmq and replication

- Developed by Marcus Eriksson
  - http://developian.com
- Can replicate externally or internally
  - External by reading the Drizzle transaction log and sending logs to RabbitMQ
    - Multi-threaded applier constructs SQL statements from transaction messages in log files on replica
  - Internal via a C++ plugin
    - /plugin/rabbitmq/
    - Implements plugin::TransactionApplier
    - Sends transaction message to RabbitMQ

A Memcached Query Cache

- Google Summer of Code project
- Two students
  - Djellel Difallah
  - Siddharth Singh
- Uses plugin::TransactionApplier and plugin::QueryCache to implement a query cache with fine-grained invalidation
  - MySQL Query Cache has very *coarse* invalidation
- plugin::TransactionApplier API uses the row-based Transaction message to determine tuple ranges that must be invalidated

Drizzle Developer Day this Friday

- Mezzanine level, this Friday, see drizzle wiki
- Hackfest
  - Come with ideas, leave with working programs
- We'll teach you how to create **`INFORMATION_SCHEMA`** and **`DATA_DICTIONARY`** views for your modules
  - In 15 minutes. Yeah, it's that easy.
- We'll demonstrate creating user-defined functions
- Like Python?
  - We'll show you how to read the trx log in 15 lines of Python code