A Drizzle Code Excursion

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

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 *staging*. We push code to *staging* 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

Proposing your branch for merging

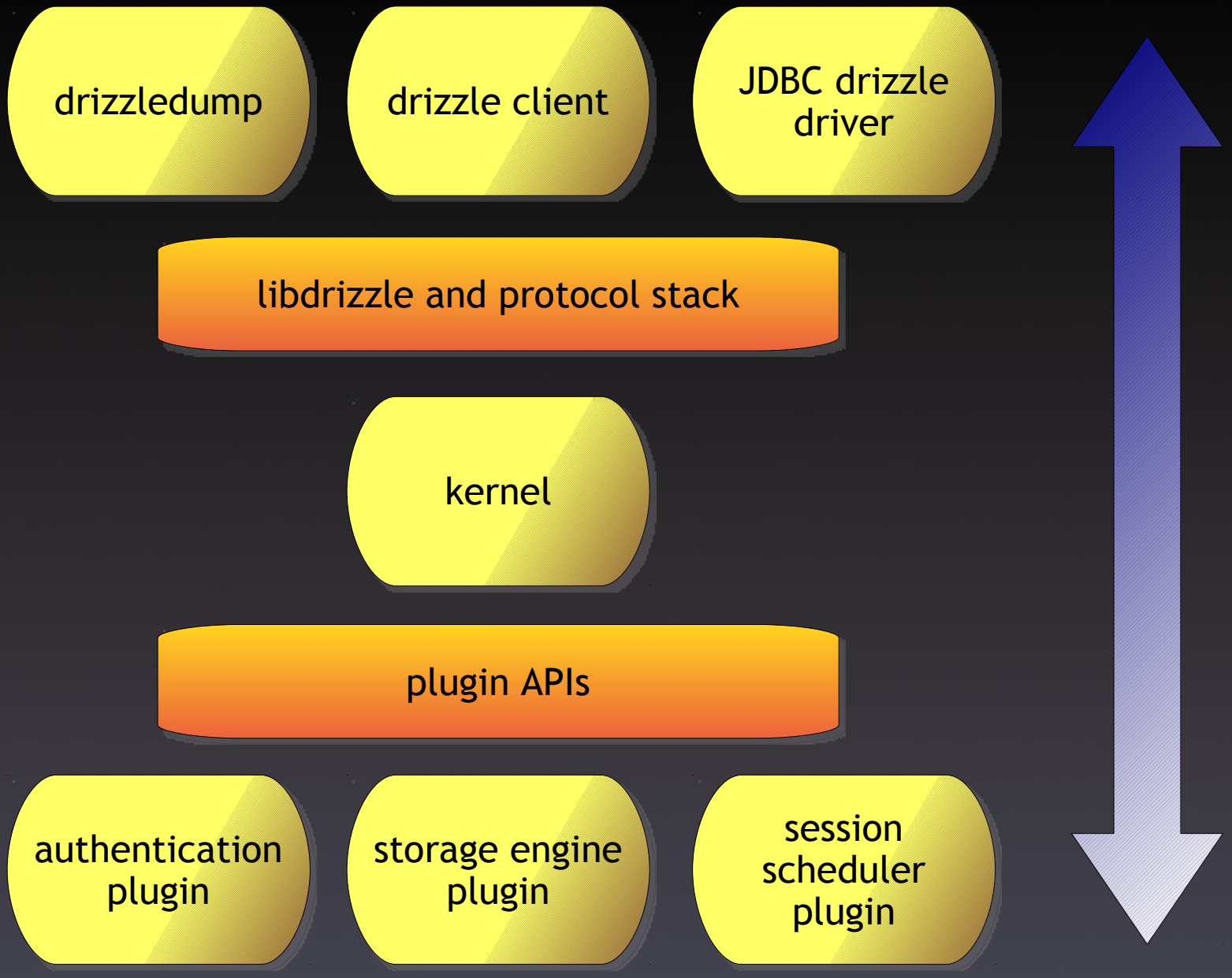
- Click “Propose for merging into another branch”
- Select lp:drizzle
- Write a comment about the code in your branch
- Click Propose Merge button
- Email sent to drizzle-developers to review your code
- Code review done online
  - Don't worry, we don't bite :)

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
  - Request mentoring on your task
  - Offer mentoring to someone else!

Inside the Code

Overview of the Drizzle Code Base


drizzledump
drizzle client
JDBC drizzle driver
libdrizzle and protocol stack
kernel
plugin APIs
authentication plugin
storage engine plugin
session scheduler plugin

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
- /gnulib
  - Portability headers

Directory organization (cont'd)

- /mystrings
  - Character set handling library
  - Comes from MySQL's strings directory
  - May go away with move to full C++ UTF8
- /mysys
  - 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!

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 (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

/plugin (module code)

- Lots of plugin examples and default implementations
  - Authentication
  - Replication
  - Serial event log writing
  - 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
```

A Word About Style

Consistent Rules for Coding

Code Style Rules

- Yes, these are enforced in code review... :)
- Consistency is the key
- Nobody agrees with all of the style, but everyone should follow it
- Otherwise the code is very difficult to navigate
- No TABs
- TABs should be expanded as spaces
- 2 space indentation

Class Names

- Pascal casing, no underscores
- Inconsistent in code...cleanup underway
- CORRECT:

```
class MyClassName;
```

- INCORRECT:

```
class My_Class_Name;
```

- INCORRECT:

```
class MY_CLASS_NAME;
```

Class *Method* Names

- Camel casing, no underscores
- Inconsistent in code...cleanup underway
- CORRECT:

```
int getSomeValue();
```

- INCORRECT:

```
int get_some_value();
```

- INCORRECT:

```
int GetSomeValue();
```

Classes

- Keep class member variable *protected* or *private* unless there is a good reason not to
- Write *public* accessors and setters for these member variables
- General rules of class design:
  - Only expose the classes' API
  - Only expose what is necessary to expose
  - Keep private as much as possible

Assignment

- Zero spaces before assignment operator
- One and only one space afterwards
- CORRECT:

```
uint32_t my_counter= 0;
```

- INCORRECT:

```
uint32_t my_counter = 0;
```

- INCORRECT:

```
uint32_t my_counter=      0;
```

Comparison

- One and only one space before and after comparison operator
- CORRECT:

```
if (my_counter == 1)
```

- INCORRECT:

```
if (my_counter==1)
```

- INCORRECT:

```
if ( my_counter== 1 )
```

Braces

- Braces should be on their own line
- *else* should be on its own line
- CORRECT:

```
if (my_counter == 1)
{
  // do something
}
```

- INCORRECT:

```
if (my_counter == 1) {
  // do something
}
```

Braces (cont'd)

- Classes and namespaces follow same standard
- Same with switch!
- CORRECT:

```
class MyClass :public SomeOtherClass
{
private:
  int my_counter;
};
```

- INCORRECT:

```
class MyClass :public SomeOtherClass {
  private:
  int my_counter;
};
```

If in doubt...

Check the Wiki:

http://drizzle.org/wiki/Coding_Standards

Under the Hood

Kernel Code Walk-through

Drizzle kernel

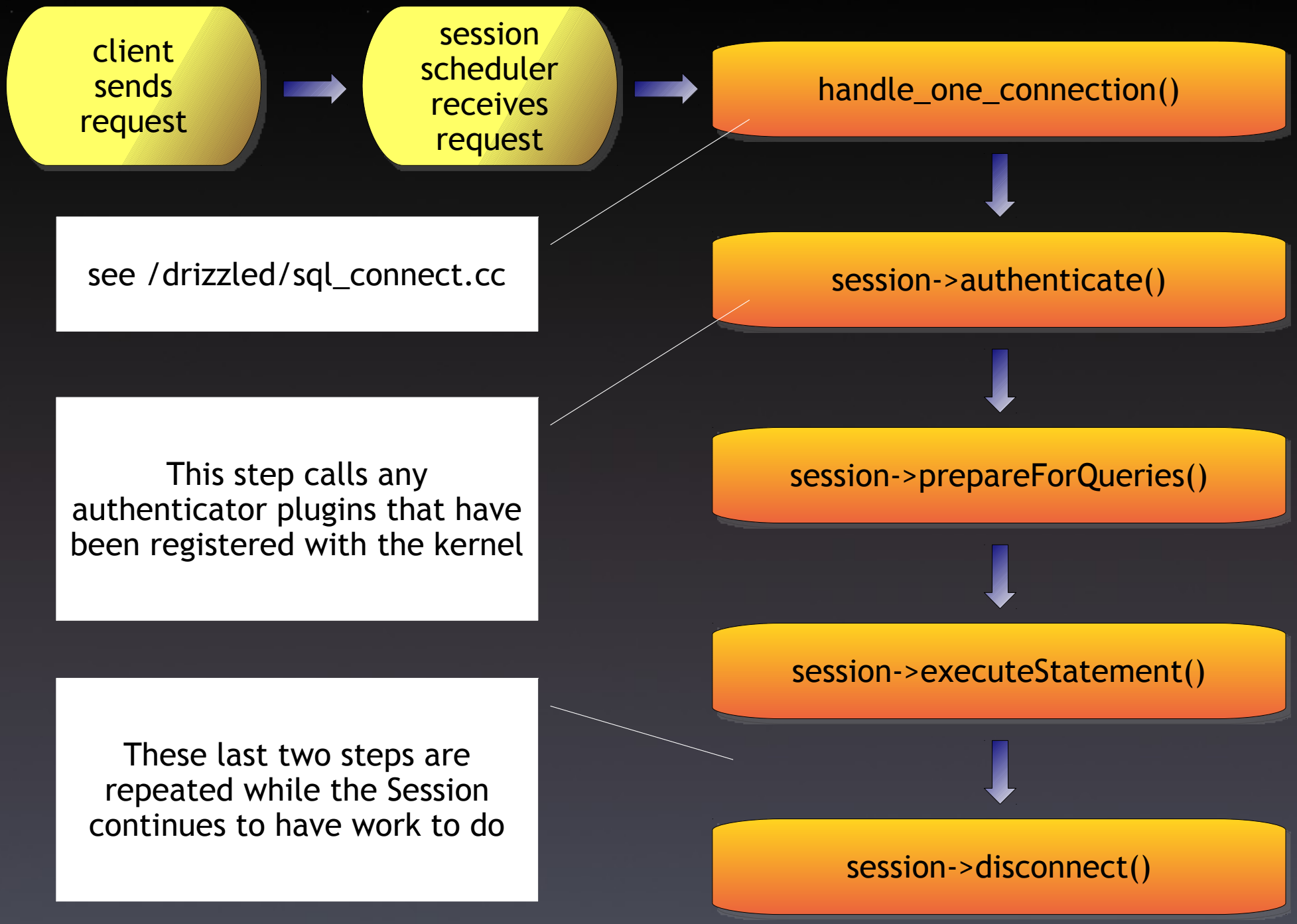
- Written in C++
  - Not C, Not C+
- Responsible for the “runtime” and coordinating communication between various plugins, clients, and itself
- Big parts:
  - Session handling
  - SQL statement parsing and optimization
  - Execution of parsed statements
  - Registering and communicating with plugins

The Session

- Session != OS Thread
- Represents the series of SQL commands received from a client
- Currently under heavy refactoring
  - So don't assume anything about it!
- Defined in /drizzled/session.h
- Contains its own separate memory area, called a mem_root, for memory allocated that lives for the lifetime of the Session object

Session handling

- Sessions are allocated in handle_connections_sockets()
  - see /drizzled/drizzled.cc
- Session pointer is passed to create_new_thread(Session *)
  - see /drizzled/drizzled.cc
- Session pointer is passed to the registered session scheduler via scheduler.add_connection(Session *)
  - Session scheduler then is responsible for it...


client sends request
session scheduler receives request
handle_one_connection()
see /drizzled/sql_connect.cc
session->authenticate()
This step calls any authenticator plugins that have been registered with the kernel
session->prepareForQueries()
session->executeStatement()
These last two steps are repeated while the Session continues to have work to do
session->disconnect()

session->executeStatement()

- Lots 'o stuff happening
- Depends on the command received from the client
- Eventually, the mysql_execute_command() function is reached, which dispatches the execution to the drizzled::Statement subclass created in the parser
  - Command is an integer SQLCOM_XXX
  - See /drizzled/sql_parse.cc
- The actual drizzled::Statement subclass has its execute() method called

Parsing of a statement

- Most SQLCOM_XXX commands have a corresponding string of SQL text passed to the execute_sqlcom_xxx() method
- This string must be parsed
- Grammar stored in a Yacc file
  - see /drizzled/sql_yacc.yy
- DRIZZLEparse() and DRIZZLElex() are the two functions which handle parsing
  - see /drizzled/sql_parse.cc
  - see /drizzled/sql_lex.cc

Parsing (cont'd)

- The parsing process actually does a lot more than just lex and parse the statement's SQL string
  - This is unfortunate, because it makes modifying and modularizing the parser difficult
  - Work is underway to address this
- The parsing process allocates a series of Item class objects, and constructs a LEX object which represents the parsed statement
- The LEX *is not* an abstract syntax tree, nor is it a compiled execution plan

Parsing (cont'd)

- After the LEX is constructed, it may go through some post-processing (particularly in the case of a SELECT statement)
- The LEX is eventually tacked onto the Session so that routines processing the statement can refer to its parsed structure
  - see /drizzled/sql_lex.h
  - see /drizzled/sql_lex.cc
- After this point, the type of command being executed determines what happens next...

Example: SQLCOM_SELECT

- Here is the some code from mysql_execute_command()

```
lex->statement->execute();
```

- The lex->statement is the object that is a subclass of drizzled::Statement that is built in the parser
- Each execute() method of the Statement classes executes a different code path – for SELECT, the exec_sqlcom_select() method is invoked

Optimization of SELECT statements

- During execution of SELECT statements, the optimizer “module” is called
  - It's not really a module, more of a loose collection of classes and functions in /drizzled/optimizer/
  - See /drizzled/sql_select.cc
  - See /drizzled/join.cc
  - See /drizzled/optimizer/range.cc
- The Join class is the dominant class used in the optimizer's routines
- There is also a JoinTab class which contains information about the tables in a SQL join

Optimization (cont'd)

- It may not be obvious by looking at the code, but the Join class' responsibility is to query the storage engine (plugin::StorageEngine and plugin::Cursor) and determine how best to perform the nested loops join algorithm
- In other words, determine the best access plan to the data in the storage engine
  - choose_plan():/drizzled/join.cc
  - best_access_path():/drizzled/join.cc
  - Join::prepare(), Join::optimize()

Execution

- Nested loops join algorithm
- Implemented using the READ_RECORD struct and a set of routines in /drizzled/sql_select.cc
  - join_read_system()
  - join_read_const()
  - join_read_key(), etc...
- Think of READ_RECORD as a rudimentary cursor over the storage engine's raw records
- READ_RECORD has a variable read_record of type pointer to function, which controls reading
  - See /drizzled/records.cc

The Plugin System

- plugin::Registry singleton
  - see /drizzled/plugin/registry.cc
- Allows plugins to register with the kernel as responders to some type of event
- Each plugin defines an init function which is passed to the plugin::Registry during registration
- This function is called when the kernel “spools up” the plugins on startup

plugins (cont'd)

- Depending on the plugin, the interface (API) between the plugin and the kernel may be messy
- We're working on cleaning up all of these APIs
- We're moving towards having plugins communicate with the kernel via GPB messages and not passing internal structure pointers back and forth
  - Example: The transaction log
  - see /plugin/transaction_log/*
  - see /drizzled/transaction_services.cc

Easy First Steps

Where to start?

don't dig too deep!

- It's best to start with small, attainable goals
- Very easy to go down “ratholes” in the code
- Have clear, well-defined tasks
- Stay out of the optimizer until you've coded on Drizzle for >3 months ;)
- Lots of little tasks that make it easy to get your feet wet and feel like you've gotten stuff accomplished...

get your feet wet

- Refactoring and code cleanup
  - Replacing custom code with STL or libc
  - Cleaning up style and indentation problems
- Documenting the large parts of the source code which are undocumented
  - Great way to learn the source code without altering
- Creating test cases
  - Look at where the source code is weak on test coverage: http://drizzle.org/lcov/
  - Work on creating tests to cover missing spots or remove dead code