Migrating Uber from MySQL to PostgreSQL

Evan Klitzke

Uber, Inc.

March 13, 2013

Outline

Background

Uber is a mobile application that lets you request a towncar or taxi on demand from your phone.

All of the things that you would expect (client information, driver information, vehicle information, trip data, etc.) are stored in a relational database; when I started (September, 2012) that database was MySQL.

Background (cont.)

All of the parts of Uber that touch the database are written in Python.

Most of the database interaction is through SQLAlchemy, with a handful of hand-written queries (i.e. `cursor.execute()`).

Also, some business intelligence/MBA types issuing ad hoc queries directly to read slaves using an OS X MySQL GUI tool (Sequel Pro).

The Objective

The objective was to move our ~50GB MySQL database to PostgreSQL.

Why? A bunch of reasons, but one of the most important was the availability of PostGIS.

Obstacles

Minimizing downtime was essential; the real cost of downtime is easily calculable and significant.

Migration (work began in October 2012) had to be coordinated around the schedule of our two biggest days: weekend before Halloween, and New Year's Eve.

Obstacles cont.

Additionally, at the time the project was conceived, growth was sustained at $> 10x$ per year for the previous two years.

Getting the project done expediently was a big concern, since the database was growing at about that rate. Migrating a 500GB database would have been hugely problematic.

Work Distribution

The work was to be mostly done by a single person (me), with help for whatever I couldn't do myself.

Plan of Attack

To do a migration, there's basically two things you need to do:

1. Figure out how to move the data out of MySQL and into PostgreSQL
2. Find all of the MySQLisms in the code, and make that code work with PostgreSQL

“Replication” Plan

The initial plan for a migration was something like:

“Let’s hack up SQLAlchemy to issue INSERT/UPDATE/DELETE statements to both MySQL and PostgreSQL, get the code working with PostgreSQL, and then switch over (with approximately zero downtime).”

“Replication” Plan, Problems

There are some problems with this plan of attack:

Primary problem: statements are well defined within a transaction, but the implicit ordering (i.e. serialization) of concurrent transactions is not clear (i.e. there are timing problems).

Serialization Problem

T_1: `UPDATE foo SET val = val + 1 WHERE ...;`

T_2: `INSERT INTO foo ...;`

If transactions T_1 and T_2 are issued from requests that are being processed concurrently, it's not possible to tell which transaction committed first, i.e. whether T_1 updated the value inserted by T_2.

Serialization Problem

The problem of serializing transactions is normally resolved by the database when it writes transactional data to the binlog (MySQL) or the WAL (Postgres).

In theory, it would have been possible to still to the strategy of “replicating” MySQL statements to Postgres using `mysqlbinlog` to read MySQL binglogs and replaying the statements, but this is complicated.

Downtime Plan

The alternate, much simpler plan is:

Take some partial downtime (i.e. now database writes allowed), dump the data from MySQL to Postgres, if everything looks OK turn on writes to Postgres.

Conversion

OK, how much downtime will we need? And how can we convert the data?

MySQL has a utility called `mysqldump`, similar to Postgres' `pg_dump`, that can dump MySQL tables in a variety of formats.

Conversion

Initial attempt:

Wow, `mysqldump` has a `--compatible=postgresql` option! How convenient!

Conversion

It turns out that `--compatible=postgresql` just turns off a few MySQL pragmas at the start of the output. Data still uses backticks for escaping, MySQL quoting syntax, integers for booleans, etc.

MySQLisms (escaping)

Table/column name escaping conventions:

MySQL:	`INSERT INTO ‘foo‘ ...`
Postgres:	`INSERT INTO "foo" ...`

MySQLisms (quoting)

Quoting conventions:

MySQL: `SELECT 'hello \'world\'';`
Postgres: `SELECT 'hello ''world''';`

MySQLisms (booleans)

In MySQL, `BOOL` and `BOOLEAN` are synonyms for, and indistinguishable from, `TINYINT(1)`.

In other words, there is not a boolean type that is distinct from the integer type.

Attempt 1

My first attempt was to take the SQL emitted by `mysqldump`, and to try to massage it into Postgres-compatible SQL with regexes.

There are various things on GitHub/Stack Overflow that can already do this, with varying levels of correctness.

Attempt 1

Lesson learned: using regular expressions to parse and munge strings is really, really slow.

Initial experiments showed that to even get a basic level of correctness, it would take tens of hours to munge all of the output of `mysqldump`.

XML Attempt

Next I discovered `mysqldump --xml` which emits XML, and Postgres' `COPY` format.

Postgres `COPY` format is a text (or binary) format optimized for doing bulk data loading, which is exactly what we were trying to do.

XML Attempt

Web developers like to poke fun of XML, but:

1. XML actually can be parsed very quickly
2. Unlike string munging, we don't need to `memcpy` lots of data
3. XML format is well defined; in particular, it's easy to distinguish `0`, `NULL`, `'NULL'`, etc. in the `mysqldump --xml` output
4. SAX style parsers can easily parse huge documents with minimal overhead

XML Attempt

New plan of attack:

Write a C or C++ program using `libxml2` in event oriented mode (i.e. SAX mode) to parse the `mysqldump` output, and generate Postgres `COPY` format.

XML Attempt

Another nice thing about `mysqldump --xml` is that it can include table schemas in an easily parseable and well-defined format.

This includes column types, primary key definitions, all indexes defined on each table, etc.

XML Attempt

Initial testing showed that I could dump MySQL, parse the XML, and emit the `COPY` data in less than an hour for any table.

XML Attempt

Except a few tables. The XML standard defines a character as:

```
Char ::= #x9 | #xA | #xD | [#x20-#xD7FF]
|[#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

...which allows `\t`, `\r`, and `\n` but not other low value control characters (even though those characters are valid Unicode codepoints, and valid in UTF-8).

XML Attempt

It turns out `libxml2` enforces the constraint on these character values, and `mysqldump` does not respect it.

It also turns out that our `sms_log` table, which contains logs of all SMS messages we've ever sent or received, had random low order bytes (i.e. garbage) prefixed or postfixed to a very small percentage of messages.

XML Attempt

Solution: I added a flag to my program `xml2pgcopy` to strip binary data from the input stream, which adds a small amount of additional CPU overhead.

This is fine, we don't care about the garbage bytes that random, old SMS messages had on them.

xml2pgcopy

The final implementation ended up being written in C++, using `libxml2`, and is very fast.

With the 50GB database, I was able to fully convert from MySQL to Postgres, including adding indexes, in about 45 minutes by loading multiple tables in parallel.

Outline

Code Changes

The other part of a project like this, is making the code work.

Major Errors

Most problems fell into these categories:

- Confusing boolean/integer types
- Truncation problems
- Timezone problems
- Missing required values
- Enums
- Case sensitivitry
- Implicit Ordering
- `GROUP BY` behavior

Boolean Errors

As noted before, MySQL does not truly have a boolean type; `BOOL` and `BOOLEAN` are synonyms for `TINYINT(1)`.

These are the easiest errors to fix, usually it involves changing Python code to use `True` and `False` instead of `1` and `0`.

Truncation

In MySQL, overflow of `VARCHAR` columns is not an error (it does, however, generate a "warning" which no one looks at).

Tons and tons of test cases and fixtures were generating fake data that overflowed, and I had to manually fix all of these tests. Only in one or two places were we actually truncating in production.

Timezone Issues

Production servers were already all UTC, and we were already using UTC for values in `datetime` columns.

MySQL provides a convenient function `UTC_TIMESTAMP()` which is not present in PostreSQL; useful for developers doing local dev on machines not using UTC. Fix for this was to create a Postgres stored procedure called `UTC_TIMESTAMP`.

Missing Required Values

In MySQL, you can omit values any column in an `INSERT` statement, even if the column is `NOT NULL`, and even if it does not have a default value.

A *lot* of Uber code relied on this, especially test cases; fixing this is tedious but straightforward, just make sure you supply all values!

ENUMs

In MySQL, `ENUMs` are effectively the same type as strings, and you can insert invalid `ENUM` values without error! Also, they are not case sensitive.

We had to fix a *lot* of code for this, including places where we were unknowingly losing data due to misspelled `ENUMs`. I also saw some cute SQLAlchemy declarations like this:

```
Column('somecolumn', Enum('foo, bar'), ...)
```

Implicit Ordering

In MySQL (specifically InnoDB), a “secondary” index on `(foo)` is implemented as a B-tree whose leaves contain values of `(foo, row_id)`, an index on `(foo, bar)` has leaves containing values of `(foo, bar, row_id)`, etc.

This means that all rows fetched using an index, including table scans, are implicitly clustered on/sorted by the primary key when an auto-increment key is used.

Implicit Ordering (cont.)

```
SELECT * FROM users LIMIT 10;
```

In MySQL, this scans the rows in PK order, and is implicitly equivalent to the same query with `ORDER BY id ASC`.

Implicit Ordering (cont.)

```
SELECT * FROM trips WHERE status =
'completed';
```

Without an index on `status`, the query scans the rows in PK order, so the same implicit ordering on `id` will occur.

With an index on `status`, the query scans the row in order of `(status, id)`, and thus there is *still* an implicit ordering on `id`! (This would *not* be true for a "range" query, however.)

Implicit Ordering (cont.)

Disclaimer: I am not a Postgres developer, this is based on what I've read online.

Postgres does *not* implicitly cluster indexes on the primary key.

Postgres indexes point to the on-disk location/offset of the row tuple, not the row id.

Implicit Ordering (cont.)

Index details aside, the problem related to Python code was pages that implicitly relied on clustering on the PK in MySQL.

Usually the code is trying to get thing in order from oldest to newest (e.g. code paginating some results), and the auto-increment PK is an indirect proxy for that behavior.

Implicit Ordering (cont.)

There’s no easy fix for this, and errors can be hard to reproduce.

Solution is to add `ORDER BY id` to queries as you find code that implicitly relies on ordering.

GROUP BY

In MySQL you can do this:

```
SELECT foo, bar FROM table GROUP BY foo;
```

According to the Postgres/SQL specification, this query is invalid since bar is not an aggregated column.

GROUP BY (cont.)

MySQL chooses the value of `bar` by using the *first* value encountered, which can be predictable in certain cases depending on the index the query uses.

There are some advanced use cases for this, but thankfully all of the queries I found were doing this unintentionally, without actually relying on this behavior.

Case Sensitivity

By default, MySQL is case insensitive (although you can change the collation to make it case sensitive). By default, Postgres is case sensitive.

Of course, we were using the default case insensitive collation for MySQL and depended on that behavior in numerous places that were mostly well tested, but didn't test case sensitivity issues.

Case Sensitivity

Case sensitivity issues were ultimately where most of the bugs from the migration came from.

For instance, for a few days we were accepting signups from emails that differed only in case, which required auditing/merging numerous user accounts after fixing the bug. (Also promotion codes, etc.)

Case Sensitivity

My advice: if you're using MySQL, turn on case sensitive collation *right now*.

There's no good reason to use a case insensitive collation on any modern application, and there are workarounds for both MySQL and Postgres that allow you to do case insensitive checking on email columns and so forth.

SQLAlchemy autoflush

An aside on SQLAlchemy:

SQLAlchemy autoflushing can decouple where Python statements that interact with the database models happen from where SQL statements are actually issued to the database (e.g. statements aren't issued until a `Session.commit()` happens).

This can make debugging very difficult! Use autoflush carefully.

Outline

Production Setup

Our production setup is like:

Python $\rightarrow$ PgBouncer $\rightarrow$ HAProxy $\rightarrow$ Postgres

We have PgBouncer and HAProxy running locally on every machine running Python code (to prevent SPOFs), only the connection from HAProxy to Postgres goes "over the network".

SQLAlchemy was set to use a `NullPool`, i.e. each request reconnects to PgBouncer, which maintains semi-persistent connections to the backend.

Downtime Plan

The actual real-time part of tracking cars and users is a Node.js thing that keeps state in memory and persists to Redis.

We hacked up the Node.js thing to allow queueing trip data, to allow users/drivers to make trips while the Python API/database is unavailable. During "downtime", most things work except signing up and actually charging credit cards.

First Attempt

First conversion attempt was early February. Conversion takes about two hours. We turn on read-only queries against Postgres, and get a constant stream of errors like:

```
OperationalError: server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
```

The errors were happening *only* on the master, not on slaves. After several hours of debugging, we ultimately and tragically had to roll back to MySQL before enabling writes. Much cursing was done.

The Problem

Due to a last minute change, we had mistakenly disabled SQLAlchemy connection pooling only on the slaves, and not on the master.

Since the migration occurred in the middle of the night, traffic was really low, and PgBouncer was timing out persistent connections being pooled by SQLAlchemy.

The *next* issued query after a timeout would generate the error described, when SQLAlchemy tried to actually use the connection.

Interim

In retrospect, a failed first attempt was actually kind of good.

We were able to find and fix various small bugs, figure out what additional monitoring metrics we wanted, get more testing in, etc.

The Second Attempt

We made the second conversion attempt about two weeks after the initial attempt... to great success!

In the end, the conversion took something like 90 minutes plus additional time for read-only mode testing, about twice as long as when I started the project three months earlier (due to the growth of some large tables).

xml2pgcopy, redux

We plan to release code for `xml2pgcopy`, the C++ program I wrote to convert `mysqldump --format=xml` output to Postgres `COPY` format today.

This tool worked for Uber's dataset, but likely needs modification if you're using extensively different data types, or MySQL features that we didn't use. It should be a good jumping off point, though.