How FriendFeed uses MySQL to store schema-less data

February 27, 2009

Background

We use MySQL for storing all of the data in FriendFeed. Our database has grown a lot as our user base has grown. We now store over 250 million entries and a bunch of other data, from comments and "likes" to friend lists.

As our database has grown, we have tried to iteratively deal with the scaling issues that come with rapid growth. We did the typical things, like using read slaves and memcache to increase read throughput and sharding our database to improve write throughput. However, as we grew, scaling our existing features to accomodate more traffic turned out to be much less of an issue than adding *new* features.

In particular, making schema changes or adding indexes to a database with more than 10 - 20 million rows completely locks the database for hours at a time. Removing old indexes takes just as much time, and not removing them hurts performance because the database will continue to read and write to those unused blocks on every `INSERT`, pushing important blocks out of memory. There are complex operational procedures you can do to circumvent these problems (like setting up the new index on a slave, and then swapping the slave and the master), but those procedures are so error prone and heavyweight, they implicitly discouraged our adding features that would require schema/index changes. Since our databases are all heavily sharded, the relational features of MySQL like `JOIN` have never been useful to us, so we decided to look outside of the realm of RDBMS.

Lots of projects exist designed to tackle the problem storing data with flexible schemas and building new indexes on the fly (e.g., CouchDB). However, none of them seemed widely-used enough by large sites to inspire confidence. In the tests we read about and ran ourselves, none of the projects were stable or battle-tested enough for our needs (see this somewhat outdated article on CouchDB, for example). MySQL works. It doesn't corrupt data. Replication works. We understand its limitations already. We like MySQL for storage, just not RDBMS usage patterns.

After some deliberation, we decided to implement a "schema-less" storage system on top of MySQL rather than use a completely new storage system. This post attempts to describe the high-level details of the system. We are curious how other large sites have tackled these problems, and we thought some of the design work we have done might be useful to other developers.

Overview

Our datastore stores schema-less bags of properties (e.g., JSON objects or Python dictionaries). The only required property of stored entities is `id`, a 16-byte UUID. The rest of the entity is opaque as far as the datastore is concerned. We can change the "schema" simply by storing new properties.

We index data in these entities by storing indexes in separate MySQL tables. If we want to index three properties in each entity, we will have three MySQL tables - one for each index. If we want to stop using an index, we stop writing to that table from our code and, optionally, drop the table from MySQL. If we want a new index, we make a new MySQL table for that index and run a process to asynchronously populate the index without disrupting our live service.

As a result, we end up having more tables than we had before, but adding and removing indexes is easy. We have heavily optimized the process that populates new indexes (which we call "The Cleaner") so that it fills new indexes rapidly without disrupting the site. We can store new properties and index them in a day's time rather than a week's time, and we don't need to swap MySQL masters and slaves or do any other scary operational work to make it happen.

Details

In MySQL, our entities are stored in a table that looks like this:

```
CREATE TABLE entities (
    added_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    id BINARY(16) NOT NULL,
    updated TIMESTAMP NOT NULL,
    body MEDIUMBLOB,
    UNIQUE KEY (id),
    KEY (updated)
) ENGINE=InnoDB;
```

The `added_id` column is present because InnoDB stores data rows physically in primary key order. The `AUTO_INCREMENT` primary key ensures new entities are written sequentially on disk after old entities, which helps for both read and write locality (new entities tend to be read more frequently than old entities since FriendFeed pages are ordered reverse-chronologically). Entity bodies are stored as zlib-compressed, pickled Python dictionaries.

Indexes are stored in separate tables. To create a new index, we create a new table storing the attributes we want to index on all of our database shards. For example, a typical entity in FriendFeed might look like this:

```
{
    "id": "71f0c4d2291844cca2df6f486e96e37c",
    "user_id": "f48b0440ca0c4f66991c4d5f6a078eaf",
    "feed_id": "f48b0440ca0c4f66991c4d5f6a078eaf",
    "title": "We just launched a new backend system for FriendFeed!",
    "link": "http://friendfeed.com/e/71f0c4d2-2918-44cc-a2df-6f486e96e
    "published": 1235697046,
    "updated": 1235697046,
}
```

We want to index the `user_id` attribute of these entities so we can render a page of all the entities a given user has posted. Our index table looks like this:

```
CREATE TABLE index_user_id (
    user_id BINARY(16) NOT NULL,
    entity_id BINARY(16) NOT NULL UNIQUE,
    PRIMARY KEY (user_id, entity_id)
) ENGINE=InnoDB;
```

Our datastore automatically maintains indexes on your behalf, so to start an instance of our datastore that stores entities like the structure above with the given indexes, you would write (in Python):

```
user_id_index = friendfeed.datastore.Index(
    table="index_user_id", properties=["user_id"], shard_on="user_id")
datastore = friendfeed.datastore.DataStore(
    mysql_shards=["127.0.0.1:3306", "127.0.0.1:3307"],
    indexes=[user_id_index])

new_entity = {
    "id": binascii.a2b_hex("71f0c4d2291844cca2df6f486e96e37c"),
    "user_id": binascii.a2b_hex("f48b0440ca0c4f66991c4d5f6a078eaf"),
    "feed_id": binascii.a2b_hex("f48b0440ca0c4f66991c4d5f6a078eaf"),
    "title": u"We just launched a new backend system for FriendFeed!",
    "link": u"http://friendfeed.com/e/71f0c4d2-2918-44cc-a2df-6f486e96
    "published": 1235697046,
    "updated": 1235697046,
}
datastore.put(new_entity)
entity = datastore.get(binascii.a2b_hex("71f0c4d2291844cca2df6f486e96e
entity = user_id_index.get_all(datastore, user_id=binascii.a2b_hex("f4
```

The `Index` class above looks for the `user_id` property in all entities and automatically maintains the index in the `index_user_id` table. Since our database is sharded, the `shard_on` argument is used to determine which shard the index gets stored on (in this case, `entity["user_id"] % num_shards`).

You can query an index using the index instance (see `user_id_index.get_all` above). The datastore code does the "join" between the `index_user_id` table and the `entities` table in Python, by first querying the `index_user_id` tables on all database shards to get a list of entity IDs and then fetching those entity IDs from the `entities` table.

To add a new index, e.g., on the `link` property, we would create a new table:

```
CREATE TABLE index_link (
    link VARCHAR(735) NOT NULL,
    entity_id BINARY(16) NOT NULL UNIQUE,
    PRIMARY KEY (link, entity_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

We would change our datastore initialization code to include this new index:

```
user_id_index = friendfeed.datastore.Index(
    table="index_user_id", properties=["user_id"], shard_on="user_id")
link_index = friendfeed.datastore.Index(
    table="index_link", properties=["link"], shard_on="link")
datastore = friendfeed.datastore.DataStore(
    mysql_shards=["127.0.0.1:3306", "127.0.0.1:3307"],
    indexes=[user_id_index, link_index])
```

And we could populate the index asynchronously (even while serving live traffic) with:

```
./rundatastorecleaner.py --index=index_link
```

Consistency and Atomicity

Since our database is sharded, and indexes for an entity can be stored on different shards than the entities themselves, consistency is an issue. What if the process crashes before it has written to all the index tables?

Building a transaction protocol was appealing to the most ambitious of FriendFeed engineers, but we wanted to keep the system as simple as possible. We decided to loosen constraints such that:

- The property bag stored in the main `entities` table is canonical
- Indexes may not reflect the actual entity values

Consequently, we write a new entity to the database with the following steps:

1. Write the entity to the `entities` table, using the ACID properties of InnoDB
2. Write the indexes to all of the index tables on all of the shards

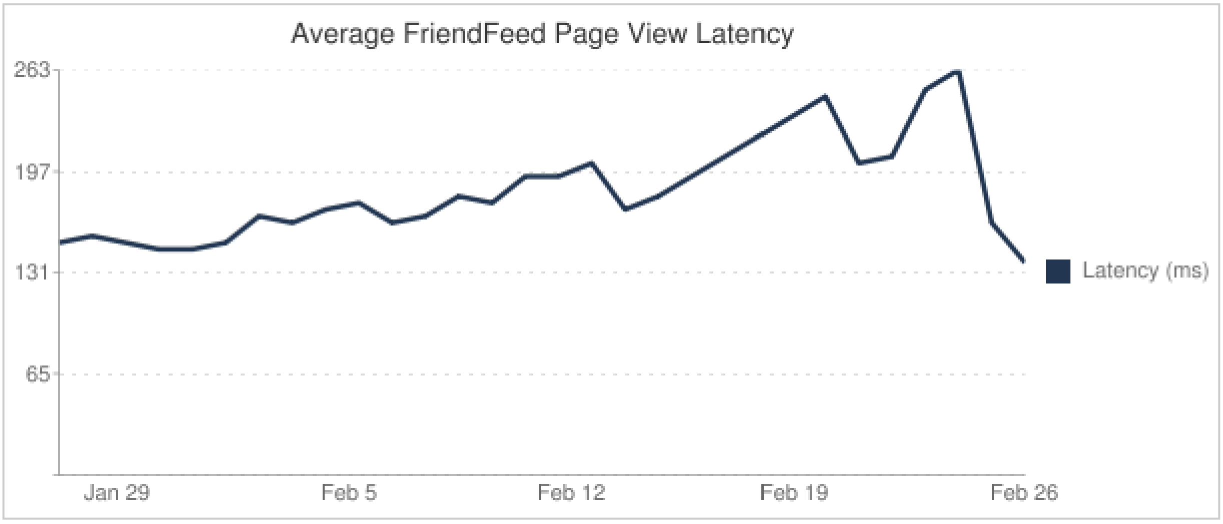
When we read from the index tables, we know they may not be accurate (i.e., they may reflect old property values if writing has not finished step 2). To ensure we don't return invalid entities based on the constraints above, we use the index tables to determine which entities to read, but we re-apply the query filters on the entities themselves rather than trusting the integrity of the indexes:

1. Read the `entity_id` from all of the index tables based on the query
2. Read the entities from the `entities` table from the given entity IDs
3. Filter (in Python) all of the entities that do not match the query conditions based on the actual property values

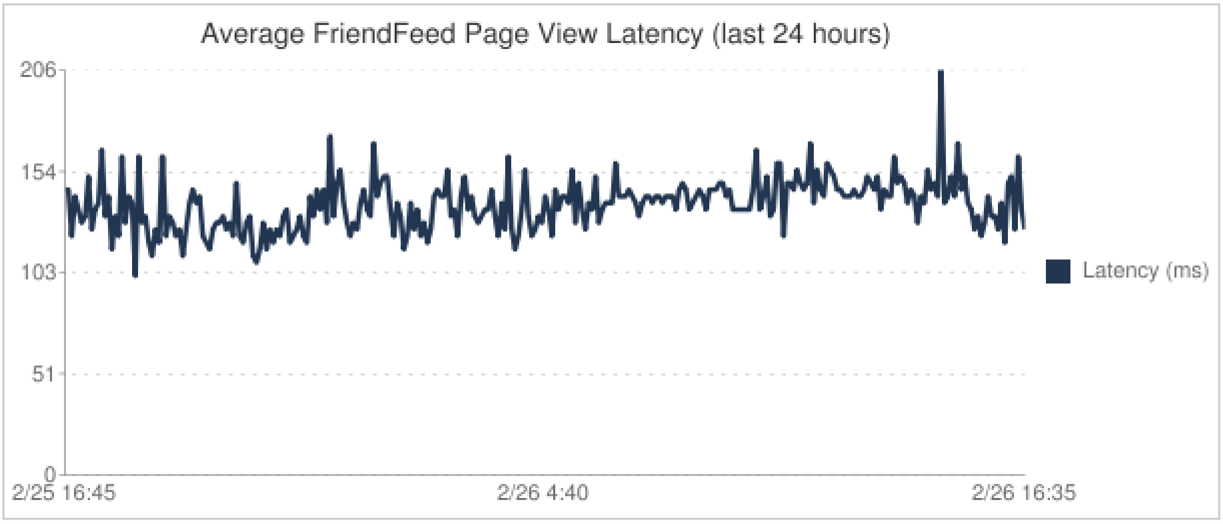
To ensure that indexes are not missing perpetually and inconsistencies are eventually fixed, the "Cleaner" process I mentioned above runs continously over the entities table, writing missing indexes and cleaning up old and invalid indexes. It cleans recently updated entities first, so inconsistencies in the indexes get fixed fairly quickly (within a couple of seconds) in practice.

Performance

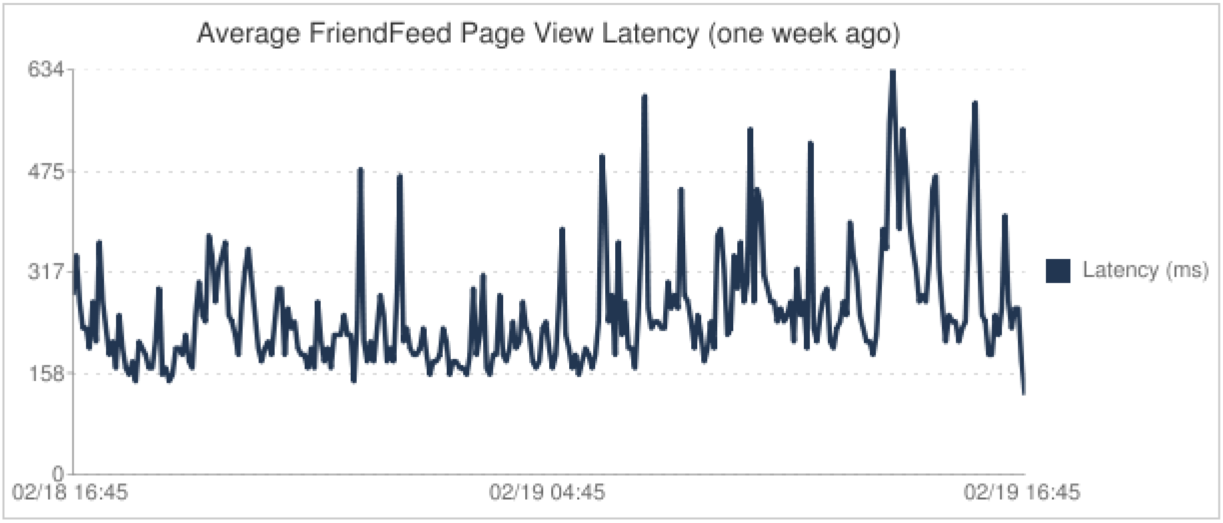
We have optimized our primary indexes quite a bit in this new system, and we are quite pleased with the results. Here is a graph of FriendFeed page view latency for the past month (we launched the new backend a couple of days ago, as you can tell by the dramatic drop):



In particular, the latency of our system is now remarkably stable, even during peak mid-day hours. Here is a graph of FriendFeed page view latency for the past 24 hours:



Compare this to one week ago:



The system has been really easy to work with so far. We have already changed the indexes a couple of times since we deployed the system, and we have started converting some of our biggest MySQL tables to use this new scheme so we can change their structure more liberally going forward.

Y Discussion