Instagram Engineering Follow

Dec 30, 2012 · 5 min read

Sharding & IDs at Instagram

With more than 25 photos and 90 likes every second, we store a lot of data here at Instagram. To make sure all of our important data fits into memory and is available quickly for our users, we've begun to shard our data□—□in other words, place the data in many smaller buckets, each holding a part of the data.

Our application servers run Django with PostgreSQL as our back-end database. Our first question after deciding to shard out our data was whether PostgreSQL should remain our primary data-store, or whether we should switch to

something else. We evaluated a few different NoSQL solutions, but ultimately decided that the solution that best suited our needs would be to shard our data across a set of PostgreSQL servers.

Before writing data into this set of servers, however, we had to solve the issue of how to assign unique identifiers to each piece of data in the database (for example, each photo posted in our system). The typical solution that works for a single database□—□just using a database's natural auto-incrementing primary key feature□—□no longer works when data is being inserted into many databases at the same time. The rest of this blog post addresses how we tackled this issue.

Before starting out, we listed out what features were essential in our system:

1. Generated IDs should be sortable by time (so a list of photo IDs, for example, could be sorted without fetching more information about the photos)

2. IDs should ideally be 64 bits (for smaller indexes, and better storage in systems like Redis)

3. The system should introduce as few new ‘moving parts’ as possible□—□a large part of how we’ve been able to scale Instagram with very few engineers is by choosing simple, easy-to-understand solutions that we trust.

Existing solutions

Many existing solutions to the ID generation problem exist; here are a few we considered:

Generate IDs in web application

This approach leaves ID generation entirely up to your application, and not up to the database at all. For example, MongoDB’s ObjectId, which is 12 bytes long and encodes the timestamp as the first component. Another popular approach is to use UUIDs.

Pros:

1. Each application thread generates IDs independently, minimizing points of failure and contention for ID generation
2. If you use a timestamp as the first component of the ID, the IDs remain time-sortable

Cons:

1. Generally requires more storage space (96 bits or higher) to make reasonable uniqueness guarantees
2. Some UUID types are completely random and have no natural sort

Generate IDs through dedicated service

Ex: Twitter's Snowflake, a Thrift service that uses Apache ZooKeeper to coordinate nodes and then generates 64-bit unique IDs

Pros:

1. Snowflake IDs are 64-bits, half the size of a UUID
2. Can use time as first component and remain sortable
3. Distributed system that can survive nodes dying

Cons:

1. Would introduce additional complexity and more ‘moving parts’ (ZooKeeper, Snowflake servers) into our architecture

DB Ticket Servers

Uses the database’s auto-incrementing abilities to enforce uniqueness. Flickr uses this approach, but with two ticket DBs (one on odd numbers, the other on even) to avoid a single point of failure.

Pros:

1. DBs are well understood and have pretty predictable scaling factors

Cons:

1. Can eventually become a write bottleneck (though Flickr reports that, even at huge scale, it's not an issue).
2. An additional couple of machines (or EC2 instances) to admin
3. If using a single DB, becomes single point of failure. If using multiple DBs, can no longer guarantee that they are sortable over time.

Of all the approaches above, Twitter's Snowflake came the closest, but the additional complexity required to run an ID service was a point against it. Instead, we took a conceptually similar approach, but brought it inside PostgreSQL.

Our solution

Our sharded system consists of several thousand ‘logical’ shards that are mapped in code to far fewer physical shards. Using this approach, we can start with just a few database servers, and eventually move to many more, simply by moving a set of logical shards from one database to another, without having to re-bucket any of our data. We used Postgres’ schemas feature to make this easy to script and administrate.

Schemas (not to be confused with the SQL schema of an individual table) are a logical grouping feature in Postgres. Each Postgres DB can h2have several schemas, each of which can contain one or more tables. Table names must only be unique per-schema, not per-DB, and by default Postgres places everything in a schema named ‘public’.

Each ‘logical’ shard is a Postgres schema in our system, and each sharded table (for example, likes on our photos) exists inside each schema.

We've delegated ID creation to each table inside each shard, by using PL/PGSQL, Postgres' internal programming language, and Postgres' existing auto-increment functionality.

Each of our IDs consists of:

- 41 bits for time in milliseconds (gives us 41 years of IDs with a custom epoch)
- 13 bits that represent the logical shard ID
- 10 bits that represent an auto-incrementing sequence, modulus 1024. This means we can generate 1024 IDs, per shard, per millisecond

Let's walk through an example: let's say it's September 9th, 2011, at 5:00pm and our 'epoch' begins on January 1st, 2011. There have been 1387263000 milliseconds since the beginning of our epoch, so to start our ID, we fill the left-most 41 bits with this value with a left-shift:

```
id = 1387263000 <<(64-41)
```

Next, we take the shard ID for this particular piece of data we're trying to insert. Let's say we're sharding by user ID, and there are 2000 logical shards; if our user ID is 31341, then the shard ID is 31341 % 2000 -> 1341. We fill the next 13 bits with this value:

```
id |= 1341 <<(64-41-13)
```

Finally, we take whatever the next value of our auto-increment sequence (this sequence is unique to each table in each schema) and fill out the remaining bits. Let's say we'd generated 5,000 IDs for this table already; our next value is 5,001, which we take and mod by 1024 (so it fits in 10 bits) and include it too:

```
id |= (5001 % 1024)
```

We now have our ID, which we can return to the application server using the RETURNING keyword as part of the INSERT.

Here's the PL/PGSQL that accomplishes all this (for an example schema insta5):

```
CREATE OR REPLACE FUNCTION insta5.next_id(OUT result bigint) AS $$
DECLARE
    our_epoch bigint := 1314220021721;
    seq_id bigint;
    now_millis bigint;
    shard_id int := 5;
BEGIN
    SELECT nextval('insta5.table_id_seq') %% 1024 INTO seq_id;

    SELECT FLOOR(EXTRACT(EPOCH FROM clock_timestamp()) * 1000) INTO
now_millis;
```

```
    result := (now_millis - our_epoch) << 23;
    result := result | (shard_id <<10);
    result := result | (seq_id);
END;
    $$ LANGUAGE PLPGSQL;
```

And when creating the table, we do:

```
CREATE TABLE insta5.our_table (
    "id" bigint NOT NULL DEFAULT insta5.next_id(),
    ...rest of table schema...
  )
```

And that's it! Primary keys that are unique across our application (and as a bonus, contain the shard ID in them for easier mapping). We've been rolling this approach into production and are happy with the results so far. Interested in helping us figure out these problems at scale? We're hiring!

Discuss this post on Hacker News.

Infra Open Source

80 6

Instagram Engineering

Follow

Never miss a story from **Instagram Engineering**, when you sign up for Medium. Learn more

GET UPDATES