Flickr Architecture

Tuesday, November 13, 2007 at 6:04PM

Todd Hoff in Apache, Example, Java, Linux, MySQL, PHP, Perl, Shard

Update: Flickr hits 2 Billion photos served. That's a lot of hamburgers.

Flickr is both my favorite bird and the web's leading photo sharing site. Flickr has an amazing challenge, they must handle a vast sea of ever expanding new content, ever increasing legions of users, and a constant stream of new features, all while providing excellent performance. How do they do it?

Site: http://www.flickr.com

Information Sources

- Flickr and PHP (an early document)
- Capacity Planning for LAMP
- Federation at Flickr: Doing Billions of Queries a Day by Dathan Pattishall.
- Building Scalable Web Sites by Cal Henderson from Flickr.
- Database War Stories #3: Flickr by Tim O'Reilly
- Cal Henderson's Talks. A lot of useful PowerPoint presentations.

Platform

- PHP
- MySQL
- Shards
- Memcached for a caching layer.
- Squid in reverse-proxy for html and images.

- Linux (RedHat)
- Smarty for templating
- Perl
- PEAR for XML and Email parsing
- ImageMagick, for image processing
- Java, for the node service
- Apache
- SystemImager for deployment
- Ganglia for distributed system monitoring
- Subcon stores essential system configuration files in a subversion repository for easy deployment to machines in a cluster.
- Cvsup for distributing and updating collections of files across a network.

The Stats

- More than 4 billion queries per day.
- ~35M photos in squid cache (total)
- ~2M photos in squid's RAM
- ~470M photos, 4 or 5 sizes of each
- 38k req/sec to memcached (12M objects)
- 2 PB raw storage (consumed about ~1.5TB on Sunday
- Over 400,000 photos being added every day

The Architecture

- A pretty picture of Flickr's architecture can be found on this slide . A simple depiction is:

-- Pair of ServerIron's
---- Squid Caches
------ Net App's
---- PHP App Servers
------ Storage Manager

------ Master-master shards
------ Dual Tree Central Database
------ Memcached Cluster
------ Big Search Engine

- The Dual Tree structure is a custom set of changes to MySQL that allows scaling by incrementally adding masters without a ring architecture. This allows cheaper scaling because you need less hardware as compared to master-master setups which always requires double the hardware.
- The central database includes data like the 'users' table, which includes primary user
keys (a few different IDs) and a pointer to which shard a users' data can be found on.

- Use dedicated servers for static content.
- Talks about how to support Unicode.
- Use a share nothing architecture.
- Everything (except photos) are stored in the database.
- Statelessness means they can bounce people around servers and it's easier to make their APIs.
- Scaled at first by replication, but that only helps with reads.
- Create a search farm by replicating the portion of the database they want to search.
- Use horizontal scaling so they just need to add more machines.
- Handle pictures emailed from users by parsing each email is it's delivered in PHP. Email is parsed for any photos.
- Earlier they suffered from Master-Slave lag. Too much load and they had a single point of failure.
- They needed the ability to make live maintenance, repair data, and so forth, without taking the site down.
- Lots of excellent material on capacity planning. Take a look in the

Information Sources for more details.

- Went to a federated approach so they can scale far into the future:

- Shards: My data gets stored on my shard, but the record of performing action on your comment, is on your shard. When making a comment on someone else's' blog
- Global Ring: Its like DNS, you need to know where to go and who controls where you go. Every page view, calculate where your data is, at that moment of time.
- PHP logic to connect to the shards and keep the data consistent (10 lines of code with comments!)

- Shards:

- Slice of the main database
- Active Master-Master Ring Replication: a few drawbacks in MySQL 4.1, as honoring commits in Master-Master. AutoIncrement IDs are automated to keep it Active Active.
- Shard assignments are from a random number for new accounts
- Migration is done from time to time, so you can remove certain power users. Needs to be balanced if you have a lot of photos… 192,000 photos, 700,000 tags, will take about 3-4 minutes. Migration is done manually.

- Clicking a Favorite:

- Pulls the Photo owners Account from Cache, to get the shard location (say on shard-5)
- Pulls my Information from cache, to get my shard location (say on shard-13)
- Starts a “distributed transaction” - to answer the question: Who favorited the photo? What are my favorites?

- Can ask question from any shard, and recover data. Its absolutely redundant.
- To get rid of replication lag…

- every page load, the user is assigned to a bucket
- if host is down, go to next host in the list; if all hosts are down, display an error page. They don’t use persistent connections, they build

connections and tear it down. Every page load thus, tests the connection.
- Every users reads and writes are kept in one shard. Notion of replication lag is gone.
- Each server in shard is 50% loaded. Shut down 1/2 the servers in each shard. So 1 server in the shard can take the full load if a server of that shard is down or in maintenance mode. To upgrade you just have to shut down half the shard, upgrade that half, and then repeat the process.
- Periods of time when traffic spikes, they break the 50% rule though. They do something like 6,000-7,000 queries per second. Now, its designed for at most 4,000 queries per second to keep it at 50% load.
- Average queries per page, are 27-35 SQL statements. Favorites counts are real time. API access to the database is all real time. Achieved the real time requirements without any disadvantages.
- Over 36,000 queries per second - running within capacity threshold. Burst of traffic, double 36K/qps.
- Each Shard holds 400K+ users data.

- A lot of data is stored twice. For example, a comment is part of the relation between the commentor and the commentee. Where is the comment stored? How about both places? Transactions are used to prevent out of sync data: open transaction 1, write commands, open transaction 2, write commands, commit 1st transaction if all is well, commit 2nd transaction if 1st committed. but there still a chance for failure when a box goes down during the 1st commit.
- Search:

- Two search back-ends: shards 35k qps on a few shards and Yahoo!’s (proprietary) web search
- Owner’s single tag search or a batch tag change (say, via Organizr) goes to the Shards due to real-time requirements, everything else goes to Yahoo!’s engine (probably about 90% behind the real-time goodness)
- Think of it such that you’ve got Lucene-like search
- Hardware:

- EMT64 w/RHEL4, 16GB RAM

- 6-disk 15K RPM RAID-10.
- Data size is at 12 TB of user metadata (these are not photos, this is just innodb ibdata files - the photos are a lot larger).
- 2U boxes. Each shard has~120GB of data.

- Backup procedure:

- ibbackup on a cron job, that runs across various shards at different times. Hotbackup to a spare.
- Snapshots are taken every night across the entire cluster of databases.
- Writing or deleting several huge backup files at once to a replication filestore can wreck performance on that filestore for the next few hours as it replicates the backup files. Doing this to an in-production photo storage filer is a bad idea.
- However much it costs to keep multiple days of backups of all of your data, it's worth it. Keeping staggered backups is good for when you discover something gone wrong a few days later. something like 1, 2, 10 and 30 day backups.

- Photos are stored on the filer. Upon upload, it processes the photos, gives you different sizes, then its complete. Metadata and points to the filers, are stored in the database.
- Aggregating the data: Very fast, because its a process per shard. Stick it into a table, or recover data from another copy from other users shards.
- max_connections = 400 connections per shard, or 800 connections per server & shard. Plenty of capacity and connections. Thread cache is set to 45, because you don’t have more than 45 users having simultaneous activity.
- Tags:

- Tags do not fit well with traditional normalized RDBMs schema design. Denormalization or heavy caching is the only way to generate a tag cloud in milliseconds for hundreds of millions of tags.
- Some of their data views are calculated offline by dedicated processing clusters which save the results into MySQL because some relationships are so complicated to calculate it would absorb all the database CPU

cycles.

- Future Direction:

- Make it faster with real-time BCP, so all data centers can receive writes to the data layer (db, memcache, etc) all at the same time. Everything is active nothing will ever be idle.

Lessons Learned

- **Think of your application as more than just a web application**. You'll have REST APIs, SOAP APIs, RSS feeds, Atom feeds, etc.

- **Go stateless**. Statelessness makes for a simpler more robust system that can handle upgrades without flinching.

- Re-architecting your database sucks.

- **Capacity plan**. Bring capacity planning into the product discussion EARLY. Get buy-in from the $$$ people (and engineering management) that it’s something to watch.

- **Start slow**. Don’t buy too much equipment just because you’re scared/ happy that your site will explode.

- **Measure reality**. Capacity planning math should be based on real things, not abstract ones.

- **Build in logging and metrics**. Usage stats are just as important as server stats. Build in custom metrics to measure real-world usage to server-based stats.

- **Cache**. Caching and RAM is the answer to everything.

- **Abstract**. Create clear levels of abstraction between database work, business logic, page logic, page mark-up and the presentation layer. This

supports quick turn around iterative development.

- **Layer**. Layering allows developers to create page level logic which designers can use to build the user experience. Designers can ask for page logic as needed. It's a negotiation between the two parties.

- **Release frequently**. Even every 30 minutes.

- **Forget about small efficiencies**, about 97% of the time. Premature optimization is the root of all evil.

- **Test in production**. Build into the architecture mechanisms (config flags, load balancing, etc.) with which you can deploy new hardware easily into (and out of) production.

- **Forget benchmarks**. Benchmarks are fine for getting a general idea of capabilities, but not for planning. Artificial tests give artificial results, and the time is better used with testing for real.

- **Find ceilings**.

- What is the maximum something that every server can do ?
- How close are you to that maximum, and how is it trending ?
- MySQL (disk IO ?)
- SQUID (disk IO ? or CPU ?)
- memcached (CPU ? or network ?)

- **Be sensitive to the usage patterns for your type of application**.

- Do you have event related growth? For example: disaster, news event.
- Flickr gets 20-40% more uploads on first work day of the year than any previous peak the previous year.
- 40-50% more uploads on Sundays than the rest of the week, on average

- **Be sensitive to the demands of exponential growth**. More users means more content, more content means more connections, more

connections mean more usage.

- **Plan for peaks**. Be able to handle peak loads up and down the stack.
