Scaling Twitter: Making Twitter 10000 Percent Faster

Saturday, June 27, 2009 at 11:46PM

Todd Hoff in Example, Memcached, RoR, Twitter

Update 6: Some interesting changes from Twitter's Evan Weaver: everything in RAM now, database is a backup; peaks at 300 tweets/second; every tweet followed by average 126 people; vector cache of tweet IDs; row cache; fragment cache; page cache; keep separate caches; GC makes Ruby optimization resistant so went with Scala; Thrift and HTTP are used internally; 100s internal requests for every external request; rewrote MQ but kept interface the same; 3 queues are used to load balance requests; extensive A/B testing for backwards capability; switched to C memcached client for speed; optimize critical path; faster to get the cached results from the network memory than recompute them locally.
Update 5: Twitter on Scala. A Conversation with Steve Jenson, Alex Payne, and Robey Pointer by Bill Venners. A fascinating discussion of why Twitter moved to the Java JVM for their server infrastructure (long lived processes) and why they moved to Scala to program against it (high level language, static typing, functional). Ruby is used on the front-end but wasn't performant or reliable enough for the back-end.
Update 4: Improving Running Components at Twitter by Evan Weaver. Tells how Twitter changed their infrastructure to go from handling 3 requests to 139 requests a second. They moved to a messaging model, asynchronous process, 3 levels of cache, and moved their middleware to a mixture C and Scala/JVM.
Update 3: Upgrading Twitter without service disruptions by Gojko Adzic. Lots of good updates on the new Twitter architecture.

Update 2: a commenter in Twitter Fails Macworld Keynote Test said this entry needs to be updated. LOL. My uneducated guess is it's not a language or architecture problem, but more a problem of not being able to add hardware fast enough into their data center. The predictability of this problem is debatable, but once you have it, it's hard to fix.
Update: Twitter releases Starling - light-weight persistent queue server that speaks the MemCache protocol. It was built to drive Twitter's backend, and is in production across Twitter's cluster.

Twitter started as a side project and blew up fast, going from 0 to millions of page views within a few terrifying months. Early design decisions that worked well in the small melted under the crush of new users chirping tweets to all their friends. Web darling Ruby on Rails was fingered early for the scaling problems, but Blaine Cook, Twitter's lead architect, held Ruby blameless:

> *For us, it's really about scaling horizontally - to that end, Rails and Ruby haven't been stumbling blocks, compared to any other language or framework. The performance boosts associated with a "faster" language would give us a 10-20% improvement, but thanks to architectural changes that Ruby and Rails happily accommodated, Twitter is 10000% faster than it was in January.*

If Ruby on Rails wasn't to blame, how did Twitter learn to scale ever higher and higher?

Update: added slides Small Talk on Getting Big. Scaling a Rails App & all that Jazz

Site: http://twitter.com

Information Sources

- Scaling Twitter Video by Blaine Cook.
- Scaling Twitter Slides
- Good News blog post by Rick Denatale
- Scaling Twitter blog post Patrick Joyce.
- Twitter API Traffic is 10x Twitter's Site.
- A Small Talk on Getting Big. Scaling a Rails App & all that Jazz - really cute dog picks

The Platform

- Ruby on Rails
- Erlang
- MySQL
- Mongrel - hybrid Ruby/C HTTP server designed to be small, fast, and secure
- Munin
- Nagios
- Google Analytics
- AWStats - real-time logfile analyzer to get advanced statistics
- Memcached

The Stats

- Over 350,000 users. The actual numbers are as always, very super super top secret.
- 600 requests per second.
- Average 200-300 connections per second. Spiking to 800 connections per second.
- MySQL handled 2,400 requests per second.
- 180 Rails instances. Uses Mongrel as the "web" server.
- 1 MySQL Server (one big 8 core box) and 1 slave. Slave is read only

for statistics and reporting.

- 30+ processes for handling odd jobs.
- 8 Sun X4100s.
- Process a request in 200 milliseconds in Rails.
- Average time spent in the database is 50-100 milliseconds.
- Over 16 GB of memcached.

The Architecture

● Ran into very public scaling problems. The little bird of failure popped up a lot for a while.

● Originally they had no monitoring, no graphs, no statistics, which makes it hard to pinpoint and solve problems. Added Munin and Nagios. There were difficulties using tools on Solaris. Had Google analytics but the pages weren't loading so it wasn't that helpful :-)

● Use caching with memcached a lot.

- For example, if getting a count is slow, you can memoize the count into memcache in a millisecond.

- Getting your friends status is complicated. There are security and other issues. So rather than doing a query, a friend's status is updated in cache instead. It never touches the database. This gives a predictable response time frame (upper bound 20 msecs).

- ActiveRecord objects are huge so that's why they aren't cached. So they want to store critical attributes in a hash and lazy load the other attributes on access.

- 90% of requests are API requests. So don't do any page/fragment caching on the front-end. The pages are so time sensitive it doesn't do any good. But they cache API requests.

● Messaging

- Use message a lot. Producers produce messages, which are queued, and then are distributed to consumers. Twitter's main functionality is to act as a messaging bridge between different formats (SMS, web, IM, etc).

- Send message to invalidate friend's cache in the background instead of doing all individually, synchronously.
- Started with DRb, which stands for distributed Ruby. A library that allows you to send and receive messages from remote Ruby objects via TCP/IP. But it was a little flaky and single point of failure.
- Moved to Rinda, which a shared queue that uses a tuplespace model, along the lines of Linda. But the queues are persistent and the messages are lost on failure.
- Tried Erlang. Problem: How do you get a broken server running at Sunday Monday with 20,000 users waiting? The developer didn't know. Not a lot of documentation. So it violates the use what you know rule.
- Moved to Starling, a distributed queue written in Ruby.
- Distributed queues were made to survive system crashes by writing them to disk. Other big websites take this simple approach as well.

● SMS is handled using an API supplied by third party gateway's. It's very expensive.

● Deployment

- They do a review and push out new mongrel servers. No graceful way yet.
- An internal server error is given to the user if their mongrel server is replaced.
- All servers are killed at once. A rolling blackout isn't used because the message queue state is in the mongrels and a rolling approach would cause all the queues in the remaining mongrels to fill up.

● Abuse

- A lot of down time because people crawl the site and add everyone as friends. 9000 friends in 24 hours. It would take down the site.
- Build tools to detect these problems so you can pinpoint when and where they are happening.
- Be ruthless. Delete them as users.

● Partitioning

- Plan to partition in the future. Currently they don't. These changes have

been enough so far.
- The partition scheme will be based on time, not users, because most requests are very temporally local.
- Partitioning will be difficult because of automatic memoization. They can't guarantee read-only operations will really be read-only. May write to a read-only slave, which is really bad.

● Twitter's API Traffic is 10x Twitter's Site
- Their API is the most important thing Twitter has done.
- Keeping the service simple allowed developers to build on top of their infrastructure and come up with ideas that are way better than Twitter could come up with. For example, Twitterrific, which is a beautiful way to use Twitter that a small team with different priorities could create.

● Monit is used to kill process if they get too big.

Lessons Learned

● Talk to the community. Don't hide and try to solve all problems yourself. Many brilliant people are willing to help if you ask.

● Treat your scaling plan like a business plan. Assemble a board of advisers to help you.

● Build it yourself. Twitter spent a lot of time trying other people's solutions that just almost seemed to work, but not quite. It's better to build some things yourself so you at least have some control and you can build in the features you need.

● Build in user limits. People will try to bust your system. Put in reasonable limits and detection mechanisms to protect your system from being killed.

● Don't make the database the central bottleneck of doom. Not everything needs to require a gigantic join. Cache data. Think of other creative ways to get the same result. A good example is talked about in Twitter, Rails, Hammers, and 11,000 Nails per Second.

● Make your application easily partitionable from the start. Then you always have a way to scale your system.

- Realize your site is slow. Immediately add reporting to track problems.
- Optimize the database.

- Index everything. Rails won't do this for you.
- Use explain to how your queries are running. Indexes may not be being as you expect.
- Denormalize a lot. Single handedly saved them. For example, they store all a user IDs friend IDs together, which prevented a lot of costly joins.
- Avoid complex joins.
- Avoid scanning large sets of data.

- Cache the hell out of everything. Individual active records are not cached, yet. The queries are fast enough for now.
- Test everything.

- You want to know when you deploy an application that it will render correctly.
- They have a full test suite now. So when the caching broke they were able to find the problem before going live.

- Long running processes should be abstracted to daemons.
- Use exception notifier and exception logger to get immediate notification of problems so you can address the right away.
- Don't do stupid things.

- Scale changes what can be stupid.
- Trying to load 3000 friends at once into memory can bring a server down, but when there were only 4 friends it works great.

- Most performance comes not from the language, but from application design.
- Turn your website into an open service by creating an API. Their API is a huge reason for Twitter's success. It allows user's to create an ever expanding and ecosystem around Twitter that is difficult to compete with. You can never do all the work your user's can do and you probably won't be as creative. So open you application up and make it easy for others to integrate your application with theirs.

Related Articles

- For a discussion of partitioning take a look at Amazon Architecture, An Unorthodox Approach to Database Design : The Coming of the Shard, Flickr Architecture
- The Mailinator Architecture has good strategies for abuse protection.
- GoogleTalk Architecture addresses some interesting issues when scaling social networking sites.
