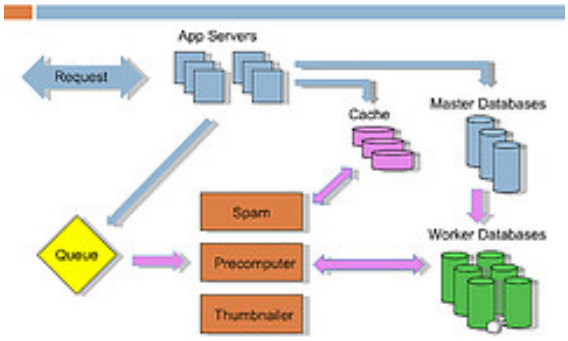
7 Lessons Learned While Building Reddit to 270 Million Page Views a Month

Monday, May 17, 2010 at 8:05AM

Todd Hoff in Example, queue



Steve Huffman, co-founder of social news site Reddit, gave an excellent presentation (slides, transcript) on the lessons he learned while building and growing Reddit to 7.5 million users per month, 270 million page views per month, and 20+ database servers.

Steve says a lot of the lessons were really obvious, so you may not find a lot of completely new ideas in the presentation. But Steve has an earnestness and genuineness about him that is so obviously grounded in experience that you can't help but think deeply about what you could be doing different. And if Steve didn't know about these lessons, I'm betting others don't either.

There are seven lessons, each has their own summary section: Lesson one: Crash Often; Lesson 2: Separation of Services; Lesson 3: Open Schema; Lesson 4: Keep it Stateless; Lesson 5: Memcache; Lesson 6: Store Redundant Data; Lesson 7: Work Offline.

By far the most surprising feature of their architecture is in Lesson Six, whose essential idea is: The key to speed is to precompute everything and cache It. They turn the precompute knob up to 11. It sounds like nearly everything you see on Reddit has been precomputed and cached, regardless of the number of versions they need to create. For example, they precompute all 15 different sort orders (hot, new, top, old, this week. etc) for listings when someone submits a link. Normally developers would be afraid of going this extreme, being this wasteful. But they thought it's better to wasteful upfront than slow. **Wasting disk and memory is better than keeping users waiting**. So if you've been holding back, go to 11, you have a good precedent.

Lesson one: Crash Often

The essence of this lesson is: **automatically restart failed and cancerous services**.

The downside of running your own system in a colo is that you are on the hook for maintenance. When your service dies you have to fix it now, even at 2AM. This is a constant tension in your life. You have to take a computer with you everywhere and you know that anytime anyone calls it could be another disaster you have to fix. It ruins your life.

One way to mitigate this problem is restart process that have died or become cancerous. Reddit uses Supervise to automatically restart applications. Special monitoring programs kill processes that use too much memory, use too much CPU, or aren't responsive. Instead of worrying just restart and the system is up. Of course you have to read the logs and find a root cause, but until then it keeps you sane.

Lesson 2: Separation of Services

The essence of this lesson is: **group like processes and data on different boxes**.

Doing too much work on one box causes **a lot of context switching between jobs**. Try to make each database server serve the same kind of database in the same way. This means all your indices will be cached and they won't be paged in and out. Keep everything as similar as possible together. Don't use Python threads. They are slow. They put everything in separate multiple processes. Services like spam, and thumbnails, query caching. It allows you to put them on different machines easily. You already solved problems of communicating between process. Once solved it keeps the architecture clean and it's easier to grow.

Lesson 3: Open Schema

The essence of this lesson is: **don't worry about the schema**.

They used to spend a lot of time worrying about the database, keeping everthing nice and normalized. **You shouldn't have to worry about the database**. Schema updates are very slow when you get bigger. Adding a column to 10 million rows takes locks and doesn't work. They used replication for backup and for scaling. Schema updates and maintaining replication is a pain. They would have to restart replication and could go a day without backups. Deployments are a pain because you have to orchestrate how new software and new database upgrades happen together.

Instead, they keep a Thing Table and a Data Table. Everything in Reddit is a Thing: users, links, comments, subreddits, awards, etc. Things keep common attribute like up/down votes, a type, and creation date. The Data table has three columns: thing id, key, value. There's a row for every attribute. There's a row for title, url, author, spam votes, etc. When they add new features they didn't have to worry about the database anymore. They didn't have to add new tables for new things or worry about upgrades. Easier for development, deployment, maintenance. The price is you can't use cool relational features. There are no joins in the database and you must manually enforce consistency. No joins means it's really easy to distribute data to different machines. You don't have to worry about foreign keys are doing joins or how to split the data up. Worked out really well. Worries of using a relational database are a thing of the past.

Lesson 4: Keep it Stateless

Goal is for every app server to handle every type of request. As they grew they had more machines so they couldn't rely on an in app server caches. They originally replicated state to each app server which was a waste of memory. They couldn't use memcached because they kept such a large amount of fine grained it was too slow. They rewrote to use memcache and don't store any state in app servers. Makes it easy if app servers fail. And to scale you can just add more app servers.

Lesson 5: Memcache

The essence of this lesson is: **memcache everything**.

They store everything in memcache: 1. Database data 2. Session data 3. Rendered pages 4. Memoizing (remember previously calculated results) internal functions 5. Rate-limiting user actions, crawlers 6. Storing pre-computing listings/pages 7. Global locking.

They store more data now in Memcachedb than Postgres. It's like memcache but stores to disk. Very fast. All queries are generated by same piece of control and is cached in memcached. Change password Links and associated state are cached for 20 minutes or so. Same for Captchas. Used for links they don't want to store forever.

They built memoization into their framework. Results that are calculated are also cached: normalized pages, listings, everything.

Rate-limit everything using memcache + expiration dates. A good way to protect your system from attacks. Without a rate limiting subsystem a single malicious user could take down the system. Not good. So for users and crawlers they keep a lot of it in memcache. If the user comes again within a second they get bounced. Regular users don't click that fast so they want notice. The Google crawler will hit you has fast as you let it, so when gets slow just crank up the rate limiter and it quiets the system down without hurting users.

Everything on Reddit is a listing: the front page, in box, comment pages. All are precomputed and dumped into the cache. When you get a listing it's taken from the cache. Every link and every comment is probably stored in a 100 different versions. For example, a link with 2 votes that's 30 seconds old is rendered and cached separately. When it hits 30 seconds it's rendered again. And so on. Every little piece of HTML comes from cache so the CPU isn't wasted on rendering. When things get slow just add more cache.

When messing with their fragile inconsistent database they use memcache as a global lock. Works for them even thought it's not the best way.

Lesson 6: Store Redundant Data

The essence of this lesson is: **the key to speed is to precompute everything and cache it**.

The way to make a slow website is have a perfectly normalized database, collect it all on demand, and then render it. It takes forever on every single request. So if you have data that might be displayed in several different formats, like links on front page, in-box, or profile, store all those representations separately. So when somebody comes and gets the data it's already there.

Every listing has 15 different sort orders (hot, new, top, old, this week). When someone submits a link they recalculate all the possible listing that link could effect. It may be a little wasteful upfront, but it's it better to wasteful upfront than slow. Wasting disk and memory is better than keeping users waiting.

Lesson 7: Work Offline

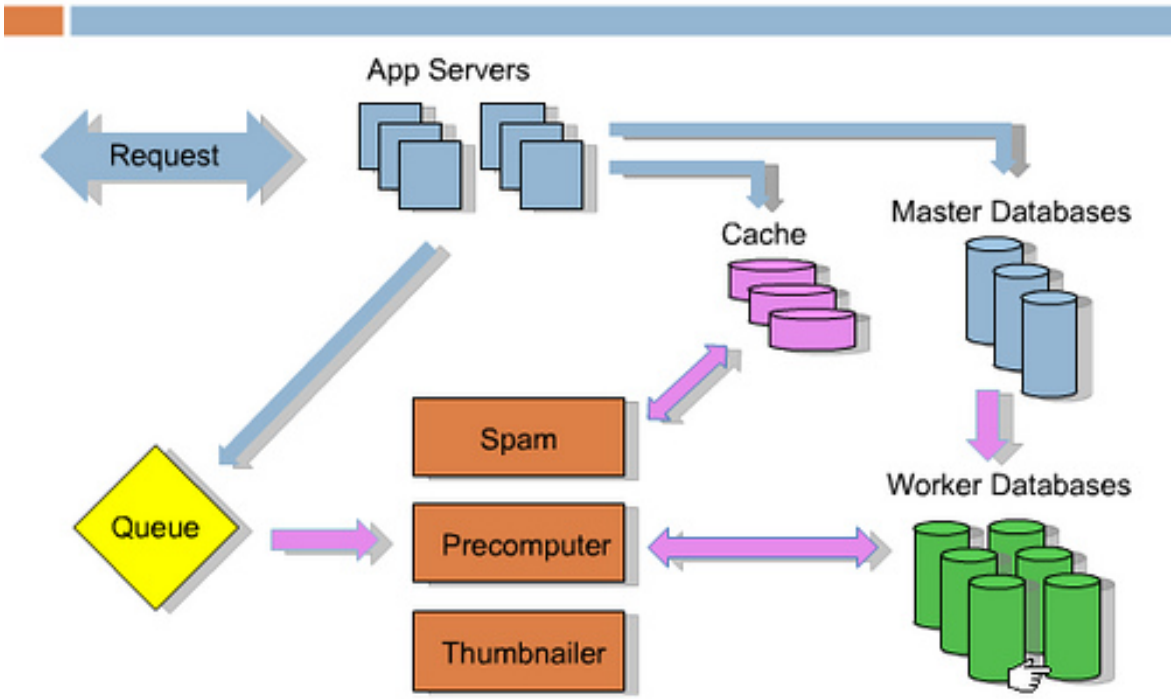
The essence of this lesson is: **do the minimal amount of work on the backend and**

tell the user you are done.

If you need to do something do it while the user isn't waiting for you. Put it in a queue. When a user votes on Reddit that updates listings, a user's Karma, and lots of other stuff. So on a vote the database is updated to know that the vote happened, then a job is put in the queue, the job knows the 20 things that need to be updated. When the user comes back everything has been precached for them.

Work they do offline: 1. Precompute listings 2. Fetch thumbnails 3. Detect cheating. 4. Remove spam 5. Compute awards 6. Update search index .

There's **no need to do these things while the user is waiting on you**. For example, the incentive to cheat is higher now as Reddit has grown larger, so they spend a lot of time in the backend while people are voting to detect cheating. But they do it live in the background so it doesn't slow down the user experience. The diagram of the architecture from the presentation is:



The blue arrows are what happens when a request comes in. Say someone submits a link or vote, it goes to the cache, master database, and job queue. Then they return to the user. Then the rest happens offline, those are represented by the pink arrows. Services like Spam, Precomputer, and Thumnailer read from the queue, do the work, and update database as required. Key piece of technology is RabbitMQ.

Related Articles

- Queue Related Articles on HighScalability
- reddit's May 2010 "State of the Servers" report
