Reddit: Lessons Learned from Mistakes Made Scaling to 1 Billion Pageviews a Month

Monday, August 26, 2013 at 8:44AM

Todd Hoff in architecture

Jeremy Edberg, the first paid employee at reddit, teaches us a lot about how to create a successful social site in a really good talk he gave at the RAMP conference. Watch it here at Scaling Reddit from 1 Million to 1 Billion–Pitfalls and Lessons.

Jeremy uses a virtue and sin approach. Examples of the mistakes made in scaling reddit are shared and it turns out they did a lot of good stuff too. Somewhat of a shocker is that Jeremy is now a Reliability Architect at Netflix, so we get a little Netflix perspective thrown in for free.

Some of the lessons that stood out most for me:

- **Think of SSDs as cheap RAM, not expensive disk**. When reddit moved from spinning disks to SSDs for the database the number of servers was reduced from 12 to 1 with a ton of headroom. SSDs are 4x more expensive but you get 16x the performance. Worth the cost.
- **Give users a little bit of power, see what they do with it, and turn the good stuff into features**. One of the biggest revelations for me was how much reddit learns from its users and how much it relies on users to make the site run smoothly. Users are going to tell you a lot of things you don’t know. For example, reddit gold started as a joke in the community. They made it a product and users love it.
- **It’s not necessary to build a scalable architecture from the start.** You don’t know what your feature set will be when you start out so you want know what your scaling problems will be. Wait until your site grows so you can learn where your scaling problems are going to be.
- **Treat nonlogged in users as second class citizens.** By always giving logged out always cached content Akamai bears the brunt for reddit’s traffic. Huge performance improvement.

There's lots more. Here's my gloss of the talk where we learn many lessons from the mistakes made in the early days of scaling reddit:

Stats

- Traffic doubles about every 15 months.
- Last month reddit had: 67,328,706 unique visitors from 177 different countries view 4,692,494,641 pages. This talk ends at about the 1 billion pageview point. How different the architecture is now is unclear.
- 28 employees
- Approximately 2.4 million unique visitors per employee. (link)
- Thousands of volunteer moderators
- As of 2012 they had 240 servers supporting 2 billion pageviews a month and 2TB of data in Postgres. All high-traffic data was moved off of EBS and onto local ephemeral disks.

Origin Story

- **Reddit started in 2005**. Went to Y Combinator with the idea of ordering food via text and were rejected. They came back and talked with Paul Graham and came up with the idea of building the frontpage of the internet. And that's reddit. At the time they did not know about Digg.

- Started in a datacenter and **moved functionality to EC2 over time**.

    - Originally started with EC2 in 2006 to store and serve logos using S3.

    - In 2007 used S3 for thumbnails.

    - In 2008 EC2 used for batch processing using a VPN tunnel to the datacenter.

    - In 2009 EC2 used for the entire site. Took site down for an entire day to move the data over to EC2. Great example of *Data Gravity* that is talked about later.

EC2

- **Motivators for moving to EC2**

    - Racking and stacking is not fun. Didn't want to rent more cabinets and buy more servers.

    - Outgrew datacenter and in the early days growth was unpredictable.

    - Cost was beneficial for their use case of a team of 4 people. EC2 was 29% cheaper than their San Francisco datacenter.

- **EC2 is not a magic bullet**. You suffer from higher network latency and noisy neighbors, so plan to work around it. Benefit is that you can grow as you need to.

- **Keep track of resource limits on EC2**

    - All resources have per account limits.
    - Amazon doesn't even know what some of their limits are.
    - Track limits and get them raised ahead of when you need them.
    - Catch exceptions to notice when limits have been reached.

Architecture

- **Reddit architecture is straightforward**. Users connect to a web tier which talks to an application tier. The application tier talks to memcache, Cassandra, and Postgres. Postgres uses a master-slave configuration. A batch system makes use of Cassandra and Postgres.

- In contrast Netflix use a service-oriented architecture where components talk to each other using REST APIs.

    - **Advantages**: Easier auto-scaling because just the service that is having the problem needs to scale; Easier capacity planning; Problems can be identified more easily because they are isolated behind REST calls; Effects of change are narrowed; More efficient local caching.

    - **Disadvantages**: Need multiple dev teams or devs to work on multiple services so you need more people; Need a common platform to prevent work duplication; Too much overhead for a small team just starting out.

- **Postgres is a great database**. It makes a wonderful, really fast key-value store.

- **Email is a hard problem**. It's hard to get deliverability right. Went with their own email server to begin with but today would probably go with an email service provider.

- **Queues were a saviour**. When passing work between components put it into a queue. You get a nice little buffer. (Reddit uses RabbitMQ for message queuing)

- **Mix of both HAProxy and Nginx**. Some traffic is directed to each. For load balancing went to HAProxy after trying Nginx (had load balancing breakdowns). It does L7 load balancing. Nginx was still used to terminate SSL and serve static content.

Code

- **Frameworks**. Used the Pylons (Django was too slow), a Python based framework, from the start. Made it easy to get started. Eventually they break down because they won't match your use case. Ended up making a lot of changes to Pylon that made it difficult to upgrade to new versions (now fixed). Would use Pyramid (new name for Pylons) again.

- **Event of thread based?** Thread based can size ahead of time, but the size can be wrong. Event based can handle more connections but when you hit a wall you hit a wall. Do you want to spend more time planning your thread pool size or suddenly hitting a wall?

- **Open Source is good**. Reddit is built on Open Source. Not having to pay for software is good, especially when getting started.

Data

- **Data is the most important asset your business will have**. Companies like Facebook, Google and Flickr have built their companies on data.

- **Data gravity**. Where you put your data you're going to need to put your application near it. The idea that all your applications are going to want to rotate around your data. Data creates a gravity well where everything need to be near it because data is the hardest thing to move. The bigger the dataset the harder it is to move. Currently it would be very expensive to move out of EC2. This is why EC2 allows you to ingress data for free and charge you to take it out. They want you to put all your data in the cloud.

- **Relational vs. Non-relational**. Most data at reddit is key-value and it is stored in Postgres. Everything that deals with money is kept in a relational database because of transactions and easier analytics.

- **Postgres is bullet proof**. It's rock solid. They never had a problem with Postgres itself. If they had a problem it was with the things around it, like the replication system written in Python. Hard to find experts in Postgres.

    - Postgres was picked over Cassandra for the key-value store because Cassandra didn't exist at the time. Plus Postgres is very fast and now natively supports KV.

- **Sharding**. Writes are split across four master databases: links, accounts, subreddits, comments, votes, and misc.

    - Each has slaves. The vote database has one master one slave. The comment database has one master and 12 slaves.

    - Avoid reading from the master if possible and direct reads to the slaves to keep the master dedicated to writes.

    - Client libraries would load balance across slaves and try a new slave if one was busy.

    - Wrote a database access layer called 'thing'

    - This approach worked for a long time. A combination of sharded databases, read slaves, and tracking read slave performance for load balancing.

- **Cassandra**.
    - Fast writes, fast negative lookups, easy incremental scalability, no single point of failure
    - At Netflix data is distributed around to three different zones. A copy of all data is in all three zones. If a zone is lost then they can still run.
    - Switching vote data into Cassandra was a huge win at reddit. Cassandra's bloom filters enabled really fast negative lookups. For comments it's very fast to tell which comments you didn't vote on, so the negative answers come back quickly. (more on this topic)

Social

- **In 2008 reddit was open sourced**
    - Users could read the code and know their was no vote tampering.
    - Users could add the features they always wanted to add and reddit would host it. This didn't work as people didn't really want to write code.
    - Hiring. Other people could know the code so it would be easier to hire people. This was the reasoning used to sell the idea to corporate.
- **The worm incident**. Somebody figured out how to write a worm by injecting extra javascript into a page. There was no intention of it getting out but get out it did. On the day one of the founders was getting married. The entire team was on a plane returning from the wedding. But a user had already responded with a patch that would stop the worm from spreading. Having the code open sourced allowed the community to help in a time of crisis.

How Does Reddit Make Money?

- Sidebox ads, self-serve ads, merchandise, reddit gold, marketplace.
- Note that reddit is not yet profitable (link). Which brings up the question is if a site like reddit can ever be profitable on the cloud?
- Also note that reddit isn't owned by Condé Nast anymore, so it is independent. (link)

Mistakes

- **Did not account for increased latency after moving to EC2**. In the datacenter they had submillisecond access between machines so it was possible to make a 1000 calls to memache for one page load. Not so on EC2. Memcache access times increased 10x to a millisecond which made their old approach unusable. Fix was to batch calls to memcache so a large number of gets are in one request.
- **Promises promises**. Amazon doesn't always deliver on promises and work around that. Design around failures instead of attempting to fix them. (There's no reference here, maybe EBS?)
- **Used bleeding edge products in production**. Cassandra was use when it was still early in its development cycle. Really great now, but it was problematic.
- **Should have offloaded a lot more of the work to the client earlier**. The server did a lot of the page rendering when it could have been pushed to the client. Facebook is the master of this. You get a rectangle with a lot of divs and API calls are made to fill out all the divs. That's how they

wish reddit was done in the first place. It would have scaled much better. It also helps debugging as it's easy to determine which API calls become problems.

- **Not having enough monitoring and using a monitoring system that isn’t virtualization friendly**. Started with Ganglia which produced great graphs but was hard to use and rapidly change, especially in a virtual environment with instances coming and going all the time.
- **Did not expire data**. At reddit you can see comments all the way back to the beginning of time. They’ve started to put limits so you can’t vote on old comments or add comments to old threads. This causes data to grow and grow over time which makes it more and more difficult to keep the hot data in the database.
- **Not using consistent hashing**. When hashing to a cache the problem is if you need to add more cache you are stuck because all your data is in one or however many caches you are hashing too. You can’t rebalance when growing the cache. Consistent hashing is a way around this problem. Solved by moving to Cassandra.

Lessons Learned

- **The key to scaling is finding the bottlenecks before your users do**.
- **Using a proxy was a huge boon to scaling**. User could be routed based on the URLs they were hitting. Reddit had a system that would monitor how long every URL took to service. People were put into different lanes. Slow traffic went one place and fast traffic went another. Splitting traffic based on the median speed of response was a huge boost.
- **Automate all the things**. Your life will be so much easier if you treat your infrastructure like you treat your code. Everything should come up and configure itself automatically.
- **It’s not necessary to build a scalable architecture from the start**. You don’t know what your feature set will be when you start out so you want know what your scaling problems will be. Wait until your site grows so you can learn where your scaling problems are going to be.
- **Don’t use a Service Oriented Architecture when just starting out**. Keep it in mind, it’s a good place to go when you’re medium sized, otherwise there’s just too much overhead.
- **Don’t follow fads**. Sometimes fads do work, node.js for example.
- **Put a limit on everything**. Everything that can happen repeatedly put a high limit on it and raise or lower the limit as needed. Block users if the limit is passed. This protects the service. Example is uploading files of logos for subreddits. Users figured out they could upload really big files and harm the system. Don’t accept huge text blobs either. Someone will figure out how to send you 5GB of text.
- **Plan for 3**. When designing always assume there’s going to be a whole bunch of what you are doing. Application servers, databases, caches. Always assume you are going to have more than one from the start. It will be much easier to horizontally scale in the future.
- **Recode Python functions in C**. As reddit was scaling to get speed they pulled out the most repetitive functions from Python and recoded in C. Particularly filters, markdown rendering, and memcache calls. What’s nice about Python is calling out to C is easy and efficient.
- **Stay as schemaless as possible**. It makes it easy to add features. All you need to do is add new properties without having to alter tables.
- **Expire data**. Lock out old threads and create a fully rendered page and cache it. This is how to deal with all the old data that threatens to overwhelm your database. At the same time, don’t allow voting on old comments or adding comments to old threads. Users will rarely notice.

- **Think of SSDs as cheap RAM, not expensive disk**. When reddit moved from spinning disks to SSDs for the database the number of servers was reduced from 12 to 1 with a ton of headroom. SSDs are 4x more expensive but you get 16x the performance. Worth the cost. Some of the biggest Cassandra nodes at Netflix and reddit are on SSDs and it made a massive improvement.

- **Each tool has a different use case**. Memcache has no guarantees about durability, but is very fast, so the vote data is stored there to make rendering of pages as quick as possible. Cassandra is durable and fast, and gives fast negative lookups because of its bloom filter, so it was good for storing a durable copy of the votes for when the data isn't in memcache. Postgres is rock solid and relational, so it’s a good place to store votes as a backup for Cassandra (all data in Cassandra could be generated from Postgres if necessary) and also for doing batch processing, which sometimes needed the relational capabilities.

- **Treat nonlogged in users as second class citizens**. Logged out users were about 80% of the traffic, now it’s closer to 50%. By always giving logged out always cached content Akamai bears the brunt for reddit’s traffic. Huge performance improvement. Side benefit if reddit goes down and you aren’t logged in you might never know.

- **Put everything into a queue**. Votes, comments, thumbnail creation, precomputed queries, spam processing and corrections. Queues allow you to know when there’s a problem by monitoring queue lengths. Side benefit is queues hide problems from users because things like vote requests are in the queue and if they aren’t applied immediately nobody notices.

- **Keep data in multiple Availability Zones**.

- **Avoid keeping state on a single instance**.

- **Take frequent snapshots of EBS disks**.

- **Do not keep secret keys on the instance**. Amazon now provides a service to give you on instance keys.

- **Divide functions by Security Group**.

- **Provide an API**. Programmers will build on your platform. The reddit iPhone applications, for example, are built by other people using the API.

- **Be an active in your own community**. Reddit users love that reddit admins are actually on the site and interacting with them.

- **Let users do the work for you**. On a site with user input one problem is always cheating, spam, and fraud. Most of the work of moderation is done by thousands of volunteers who take care of most of the spam problem. It works amazingly well and is one of the reasons the reddit team can remain small.

- **Give users a little bit of power, see what they do with it, and turn the good stuff into features**. For example, when the ability to add CSS to subreddits was added they saw what people were doing and added many of the common things as feature for everyone. This also makes users excited about doing stuff on reddit because they like that sense of control. Lots of other examples.

- **Listen to your users**. Users are going to tell you a lot of things you don’t know that you probably want to know. For example, reddit gold started as a joke in the community. They made it a product and users love it.

Related Articles

- On Hacker News / On Reddit

- 7 Lessons Learned While Building Reddit To 270 Million Page Views A Month
- reddit myth busters
- I run reddit's servers (and do a bunch of other stuff too). AMA. (3 years old)
- January 2012 - State of the Servers (2012)
- Reddit, Netflix, and beyond: Building Scalable and Reliable Architectures in the Cloud
- Reddit’s database has two tables

- Hacker News discussion of the talk

- A lot has changed. It wasn’t looking all that good as little as 2 years ago:
  - Dear admin. Let's be frank and honest about it. Reddit is not healthy. No other top internet site runs as slowly, or is down as often. It's becoming a daily joke. Why don't we have a proper discussion about what needs to be done?
  - Are any of the reddit site problems Python/Pylons related?
  - Dear Reddit: if this is really your architecture, it totally fails. Every single request goes through your Python app servers? Why don't you use an HTTP cache (e.g. Varnish), ESI, and a CDN to speed up the site?
