Tumblr Architecture - 15 Billion Page Views a Month and Harder to Scale than Twitter

Monday, February 13, 2012 at 9:15AM

Todd Hoff in Example

With over 15 billion page views a month Tumblr has become an insanely popular blogging platform. Users may like Tumblr for its simplicity, its beauty, its strong focus on user experience, or its friendly and engaged community, but like it they do.

Growing at over 30% a month has not been without challenges. Some reliability problems among them. It helps to realize that Tumblr operates at surprisingly huge scales: 500 million page views a day, a peak rate of ~40k requests per second, ~3TB of new data to store a day, all running on 1000+ servers.

One of the common patterns across successful startups is the perilous chasm crossing from startup to wildly successful startup. Finding people, evolving infrastructures, servicing old infrastructures, while handling huge month over month increases in traffic, all with only four engineers, means you have to make difficult choices about what to work on. This was Tumblr's situation. Now with twenty engineers there's enough energy to work on issues and develop some very interesting solutions.

Tumblr started as a fairly typical large LAMP application. The direction they are moving in now is towards a distributed services model built around Scala, HBase, Redis, Kafka, Finagle, and an intriguing cell based architecture for powering their Dashboard. Effort is now going into fixing

short term problems in their PHP application, pulling things out, and doing it right using services.

The theme at Tumblr is transition at massive scale. Transition from a LAMP stack to a somewhat bleeding edge stack. Transition from a small startup team to a fully armed and ready development team churning out new features and infrastructure. To help us understand how Tumblr is living this theme is startup veteran Blake Matheny, Distributed Systems Engineer at Tumblr. Here's what Blake has to say about the House of Tumblr:

Site: http://www.tumblr.com/

Stats

- 500 million page views a day
- 15B+ page views month
- ~20 engineers
- Peak rate of ~40k requests per second
- 1+ TB/day into Hadoop cluster
- Many TB/day into MySQL/HBase/Redis/Memcache
- Growing at 30% a month
- ~1000 hardware nodes in production
- Billions of page visits per month per engineer
- Posts are about 50GB a day. Follower list updates are about 2.7TB a day.
- Dashboard runs at a million writes a second, 50K reads a second, and it is growing.

Software

- OS X for development, Linux (CentOS, Scientific) in production
- Apache

- PHP, Scala, Ruby
- Redis, HBase, MySQL
- Varnish, HA-Proxy, nginx,
- Memcache, Gearman, Kafka, Kestrel, Finagle
- Thrift, HTTP
- Func - a secure, scriptable remote control framework and API
- Git, Capistrano, Puppet, Jenkins

Hardware

- 500 web servers
- 200 database servers (many of these are part of a spare pool we pulled from for failures)
  - 47 pools
  - 30 shards
- 30 memcache servers
- 22 redis servers
- 15 varnish servers
- 25 haproxy nodes
- 8 nginx
- 14 job queue servers (kestrel + gearman)

Architecture

- Tumblr has a different usage pattern than other social networks.
  - With 50+ million posts a day, an average post goes to many hundreds of people. It’s not just one or two users that have millions of followers. The graph for Tumblr users has hundreds of followers. This is different than any other social network and is what makes Tumblr so challenging to scale.
  - #2 social network in terms of time spent by users. The content is engaging. It’s images and videos. The posts aren’t byte

sized. They aren't all long form, but they have the ability. People write in-depth content that's worth reading so people stay for hours.
    - Users form a connection with other users so they will go hundreds of pages back into the dashboard to read content. Other social networks are just a stream that you sample.
    - Implication is that given the number of users, the average reach of the users, and the high posting activity of the users, there is a huge amount of updates to handle.
- Tumblr runs in one colocation site. Designs are keeping geographical distribution in mind for the future.
- Two components to Tumblr as a platform: public Tumblelogs and Dashboard
    - Public Tumblelog is what the public deals with in terms of a blog. Easy to cache as its not that dynamic.
    - Dashboard is similar to the Twitter timeline. Users follow real-time updates from all the users they follow.
        - Very different scaling characteristics than the blogs. Caching isn't as useful because every request is different, especially with active followers.
        - Needs to be real-time and consistent. Should not show stale data. And it's a lot of data to deal with. Posts are only about 50GB a day. Follower list updates are 2.7TB a day. Media is all stored on S3.
    - Most users leverage Tumblr as tool for consuming of content. Of the 500+ million page views a day, 70% of that is for the Dashboard.
    - Dashboard availability has been quite good. Tumblelog hasn't been as good because they have a legacy infrastructure that has been hard to migrate away from. With a small team they had to pick and choose what they addressed for scaling issues.

Old Tumblr

- When the company started on Rackspace it gave each custom domain blog an A record. When they outgrew Rackspace there were too many users to migrate. This is 2007. They still have custom domains on Rackspace. They route through Rackspace back to their colo space using HAProxy and Varnish. Lots of legacy issues like this.
- A traditional LAMP progression.
  - Historically developed with PHP. Nearly every engineer programs in PHP.
  - Started with a web server, database server and a PHP application and started growing from there.
  - To scale they started using memcache, then put in front-end caching, then HAProxy in front of the caches, then MySQL sharding. MySQL sharding has been hugely helpful.
  - Use a squeeze everything out of a single server approach. In the past year they've developed a couple of backend services in C: an ID generator and Staircar, using Redis to power Dashboard notifications
- The Dashboard uses a scatter-gather approach. Events are displayed when a user access their Dashboard. Events for the users you follow are pulled and displayed. This will scale for another 6 months. Since the data is time ordered sharding schemes don't work particularly well.

New Tumblr

- Changed to a JVM centric approach for hiring and speed of development reasons.
- Goal is to move everything out of the PHP app into services and make the app a thin layer over services that does request

authentication, presentation, etc.

- Scala and Finagle Selection
  - Internally they had a lot of people with Ruby and PHP experience, so Scala was appealing.
  - Finagle was a compelling factor in choosing Scala. It is a library from Twitter. It handles most of the distributed issues like distributed tracing, service discovery, and service registration. You don’t have to implement all this stuff. It just comes for free.
  - Once on the JVM Finagle provided all the primitives they needed (Thrift, ZooKeeper, etc).
  - Finagle is being used by Foursquare and Twitter. Scala is also being used by Meetup.
  - Like the Thrift application interface. It has really good performance.
  - Liked Netty, but wanted out of Java, so Scala was a good choice.
  - Picked Finagle because it was cool, knew some of the guys, it worked without a lot of networking code and did all the work needed in a distributed system.
  - Node.js wasn’t selected because it is easier to scale the team with a JVM base. Node.js isn’t developed enough to have standards and best practices, a large volume of well tested code. With Scala you can use all the Java code. There’s not a lot of knowledge of how to use it in a scalable way and they target 5ms response times, 4 9s HA, 40K requests per second and some at 400K requests per second. There’s a lot in the Java ecosystem they can leverage.
- Internal services are being shifted from being C/libevent based to being Scala/Finagle based.
- Newer, non-relational data stores like HBase and Redis are being used, but the bulk of their data is currently stored in a heavily

partitioned MySQL architecture. Not replacing MySQL with HBase.

- HBase backs their URL shortner with billions of URLs and all the historical data and analytics. It has been rock solid. HBase is used in situations with high write requirements, like a million writes a second for the Dashboard replacement. HBase wasn't deployed instead of MySQL because they couldn't bet the business on HBase with the people that they had, so they started using it with smaller less critical path projects to gain experience.
- Problem with MySQL and sharding for time series data is one shard is always really hot. Also ran into read replication lag due to insert concurrency on the slaves.
- Created a common services framework.
  - Spent a lot of time upfront solving operations problem of how to manage a distributed system.
  - Built a kind of Rails scaffolding, but for services. A template is used to bootstrap services internally.
  - All services look identical from an operations perspective. Checking statistics, monitoring, starting and stopping all work the same way for all services.
  - Tooling is put around the build process in SBT (a Scala build tool) using plugins and helpers to take care of common activities like tagging things in git, publishing to the repository, etc. Most developers don't have to get in the guts of the build system.
- Front-end layer uses HAProxy. Varnish might be hit for public blogs. 40 machines.
- 500 web servers running Apache and their PHP application.
- 200 database servers. Many database servers are used for high availability reasons. Commodity hardware is used an the MTBF is surprisingly low. Much more hardware than expected is lost so there are many spares in case of failure.

- 6 backend services to support the PHP application. A team is dedicated to develop the backend services. A new service is rolled out every 2-3 weeks. Includes dashboard notifications, dashboard secondary index, URL shortener, and a memcache proxy to handle transparent sharding.
- Put a lot of time and effort and tooling into MySQL sharding. MongoDB is not used even though it is popular in NY (their location). MySQL can scale just fine..
- Gearman, a job queue system, is used for long running fire and forget type work.
- Availability is measured in terms of reach. Can a user reach custom domains or the dashboard? Also in terms of error rate.
- Historically the highest priority item is fixed. Now failure modes are analyzed and addressed systematically. Intention is to measure success from a user perspective and an application perspective. If part of a request can't be fulfilled that is account for
- Initially an Actor model was used with Finagle, but that was dropped. For fire and forget work a job queue is used. In addition, Twitter's utility library contains a Futures implementation and services are implemented in terms of futures. In the situations when a thread pool is needed futures are passed into a future pool. Everything is submitted to the future pool for asynchronous execution.
- Scala encourages no shared state. Finagle is assumed correct because it's tested by Twitter in production. Mutable state is avoided using constructs in Scala or Finagle. No long running state machines are used. State is pulled from the database, used, and writte n back to the database. Advantage is developers don't need to worry about threads or locks.
- 22 Redis servers. Each server has 8 - 32 instances so 100s of Redis instances are used in production.
    - Used for backend storage for dashboard notifications.

- A notification is something like a user liked your post. Notifications show up in a user's dashboard to indicate actions other users have taken on their content.
- High write ratio made MySQL a poor fit.
- Notifications are ephemeral so it wouldn't be horrible if they were dropped, so Redis was an acceptable choice for this function.
- Gave them a chance to learn about Redis and get familiar with how it works.
- Redis has been completely problem free and the community is great.
- A Scala futures based interface for Redis was created. This functionality is now moving into their Cell Architecture.
- URL shortener uses Redis as the first level cache and HBase as permanent storage.
- Dashboard's secondary index is built around Redis.
- Redis is used as Gearman's persistence layer using a memcache proxy built using Finagle.
- Slowly moving from memcache to Redis. Would like to eventually settle on just one caching service. Performance is on par with memcache.

Internal Firehose

- Internally applications need access to the activity stream. An activity steam is information about users creating/deleting posts, liking/unliking posts, etc. A challenge is to distribute so much data in real-time. Wanted something that would scale internally and that an application ecosystem could reliably grow around. A central point of distribution was needed.
- Previously this information was distributed using Scribe/Hadoop. Services would log into Scribe and begin tailing and then pipe that

data into an app. This model stopped scaling almost immediately, especially at peak where people are creating 1000s of posts a second. Didn't want people tailing files and piping to grep.
- An internal firehose was created as a message bus. Services and applications talk to the firehose via Thrift.
- LinkedIn's Kafka is used to store messages. Internally consumers use an HTTP stream to read from the firehose. MySQL wasn't used because the sharding implementation is changing frequently so hitting it with a huge data stream is not a good idea.
- The firehose model is very flexible, not like Twitter's firehose in which data is assumed to be lost.
    - The firehose stream can be rewound in time. It retains a week of data. On connection it's possible to specify the point in time to start reading.
    - Multiple clients can connect and each client won't see duplicate data. Each client has a client ID. Kafka supports a consumer group idea. Each consumer in a consumer group gets its own messages and won't see duplicates. Multiple clients can be created using the same consumer ID and clients won't see duplicate data. This allows data to be processed independently and in parallel. Kafka uses ZooKeeper to periodically checkpoint how far a consumer has read.

Cell Design for Dashboard Inbox

- The current scatter-gather model for providing Dashboard functionality has very limited runway. It won't last much longer.
    - The solution is to move to an inbox model implemented using a Cell Based Architecture that is similar to Facebook Messages.
    - An inbox is the opposite of scatter-gather. A user's dashboard, which is made up posts from followed users and actions taken

by other users, is logically stored together in time order.
    - Solves the scatter gather problem because it's an inbox. You just ask what is in the inbox so it's less expensive then going to each user a user follows. This will scale for a very long time.
- Rewriting the Dashboard is difficult. The data has a distributed nature, but it has a transactional quality, it's not OK for users to get partial updates.
    - The amount of data is incredible. Messages must be delivered to hundreds of different users on average which is a very different problem than Facebook faces. Large date + high distribution rate + multiple datacenters.
    - Spec'ed at a million writes a second and 50K reads a second. The data set size is 2.7TB of data growth with no replication or compression turned on. The million writes a second is from the 24 byte row key that indicates what content is in the inbox.
    - Doing this on an already popular application that has to be kept running.
- Cells
    - A cell is a self-contained installation that has all the data for a range of users. All the data necessary to render a user's Dashboard is in the cell.
    - Users are mapped into cells. Many cells exist per data center.
    - Each cell has an HBase cluster, service cluster, and Redis caching cluster.
    - Users are homed to a cell and all cells consume all posts via firehose updates.
    - Each cell is Finagle based and populates HBase via the firehose and service requests over Thrift.
    - A user comes into the Dashboard, users home to a particular cell, a service node reads their dashboard via HBase, and passes the data back.

    - Background tasks consume from the firehose to populate tables and process requests.
    - A Redis caching layer is used for posts inside a cell.
- Request flow: a user publishes a post, the post is written to the firehose, all of the cells consume the posts and write that post content to post database, the cells lookup to see if any of the followers of the post creator are in the cell, if so the follower inboxes are updated with the post ID.
- Advantages of cell design:
    - Massive scale requires parallelization and parallelization requires components be isolated from each other so there is no interaction. Cells provide a unit of parallelization that can be adjusted to any size as the user base grows.
    - Cells isolate failures. One cell failure does not impact other cells.
    - Cells enable nice things like the ability to test upgrades, implement rolling upgrades, and test different versions of software.
- The key idea that is easy to miss is: all posts are replicated to all cells.
    - Each cell stores a single copy of all posts. Each cell can completely satisfy a Dashboard rendering request. Applications don’t ask for all the post IDs and then ask for the posts for those IDs. It can return the dashboard content for the user. Every cell has all the data needed to fulfill a Dashboard request without doing any cross cell communication.
    - Two HBase tables are used: one that stores a copy of each post. That data is small compared to the other table which stores every post ID for every user within that cell. The second table tells what the user’s dashboard looks like which means they don’t have to go fetch all the users a user is following. It also means across clients they’ll know if you read a post and

viewing a post on a different device won't mean you read the same content twice. With the inbox model state can be kept on what you've read.
    - Posts are not put directly in the inbox because the size is too great. So the ID is put in the inbox and the post content is put in the cell just once. This model greatly reduces the storage needed while making it simple to return a time ordered view of an users inbox. The downside is each cell contains a complete copy of call posts. Surprisingly posts are smaller than the inbox mappings. Post growth per day is 50GB per cell, inbox grows at 2.7TB a day. Users consume more than they produce.
    - A user's dashboard doesn't contain the text of a post, just post IDs, and the majority of the growth is in the IDs.
    - As followers change the design is safe because all posts are already in the cell. If only follower posts were stored in a cell then cell would be out of date as the followers changed and some sort of back fill process would be needed.
    - An alternative design is to use a separate post cluster to store post text. The downside of this design is that if the cluster goes down it impacts the entire site. Using the cell design and post replication to all cells creates a very robust architecture.
- A user having millions of followers who are really active is handled by selectively materializing user feeds by their access model (see Feeding Frenzy).
    - Different users have different access models and distribution models that are appropriate. Two different distribution modes: one for popular users and one for everyone else.
    - Data is handled differently depending on the user type. Posts from active users wouldn't actually be published, posts would selectively materialized.
    - Users who follow millions of users are treated similarly to

users who have millions of followers.

- Cell size is hard to determine. The size of cell is the impact site of a failure. The number of users homed to a cell is the impact. There's a tradeoff to make in what they are willing to accept for the user experience and how much it will cost.
- Reading from the firehose is the biggest network issue. Within a cell the network traffic is manageable.
- As more cells are added cells can be placed into a cell group that reads from the firehose and then replicates to all cells within the group. A hierarchical replication scheme. This will also aid in moving to multiple datacenters.

On Being a Startup in New York

- NY is a different environment. Lots of finance and advertising. Hiring is challenging because there's not as much startup experience.
- In the last few years NY has focused on helping startups. NYU and Columbia have programs for getting students interesting internships at startups instead of just going to Wall Street. Mayor Bloomberg is establishing a local campus focused on technology.

Team Structure

- Teams: infrastructure, platform, SRE, product, web ops, services.
- Infrastructure: Layer 5 and below. IP address and below, DNS, hardware provisioning.
- Platform: core app development, SQL sharding, services, web operations.
- SRE: sits between service team and web ops team. Focused on more immediate needs in terms of reliability and scalability.
- Service team: focuses on things that are slightly more strategic, that

are a month or two months out.
- Web ops: responsible for problem detection and response, and tuning.

Software Deployment

- Started with a set of rsync scripts that distributed the PHP application everywhere. Once the number of machines reached 200 the system started having problems, deploys took a long time to finish and machines would be in various states of the deploy process.
- The next phase built the deploy process (development, staging, production) into their service stack using Capistrano. Worked for services on dozens of machines, but by connecting via SSH it started failing again when deploying to hundreds of machines.
- Now a piece of coordination software runs on all machines. Based around Func from RedHat, a lightweight API for issuing commands to hosts. Scaling is built into Func.
- Build deployment is over Func by saying do X on a set of hosts, which avoids SSH. Say you want to deploy software on group A. The master reaches out to a set of nodes and runs the deploy command.
- The deploy command is implemented via Capistrano. It can do a git checkout or pull from the repository. Easy to scale because they are talking HTTP. They like Capistrano because it supports simple directory based versioning that works well with their PHP app. Moving towards versioned updates, where each directory contains a SHA so it’s easy to check if a version is correct.
- The Func API is used to report back status, to say these machines have these software versions.
- Safe to restart any of their services because they’ll drain off connections and then restart.

- All features run in dark mode before activation.

Development

- Started with the philosophy that anyone could use any tool that they wanted, but as the team grew that didn't work. Onboarding new employees was very difficult, so they've standardized on a stack so they can get good with those, grow the team quickly, address production issues more quickly, and build up operations around them.
- Process is roughly Scrum like. Lightweight.
- Every developer has a preconfigured development machine. It gets updates via Puppet.
- Dev machines can roll changes, test, then roll out to staging, and then roll out to production.
- Developers use vim and Textmate.
- Testing is via code reviews for the PHP application.
- On the service side they've implemented a testing infrastructure with commit hooks, Jenkins, and continuous integration and build notifications.

Hiring Process

- Interviews usually avoid math, puzzles, and brain teasers. Try to ask questions focused on work the candidate will actually do. Are they smart? Will they get stuff done? But measuring "gets things done" is difficult to assess. Goal is to find great people rather than keep people out.
- Focused on coding. They'll ask for sample code. During phone interviews they will use Collabedit to write shared code.
- Interviews are not confrontational, they just want to find the best people. Candidates get to use all their tools, like Google, during the

interview. The idea is developers are at their best when they have tools so that's how they run the interviews.

- Challenge is finding people that have the scaling experience they require given Tumblr's traffic levels. Few companies in the world are working on the problems they are.
    - Example, for a new ID generator they needed A JVM process to generate service responses in less the 1ms at a rate at 10K requests per second with a 500 MB RAM limit with High Availability. They found the serial collector gave the lowest latency for this particular work load. Spent a lot of time on JVM tuning.
- On the Tumblr Engineering Blog they've posted memorials giving their respects for the passing of Dennis Ritchie & John McCarthy. It's a geeky culture.

Lessons learned

- Automation everywhere.
- MySQL (plus sharding) scales, apps don't.
- Redis is amazing.
- Scala apps perform fantastically.
- Scrap projects when you aren't sure if they will work.
- Don't hire people based on their survival through a useless technological gauntlet. Hire them because they fit your team and can do the job.
- Select a stack that will help you hire the people you need.
- Build around the skills of your team.
- Read papers and blog posts. Key design ideas like the cell architecture and selective materialization were taken from elsewhere.
- Ask your peers. They talked to engineers from Facebook, Twitter, LinkedIn about their experiences and learned from them. You may

not have access to this level, but reach out to somebody somewhere.
- Wade, don’t jump into technologies. They took pains to learn HBase and Redis before putting them into production by using them in pilot projects or in roles where the damage would be limited.

I’d like to thank Blake very much for the interview. He was very generous with his time and patient with his explanations. Please contact me if you would like to talk about having your architecture profiled.

Related Articles

- Tumblr Engineering Blog - contains a lot of good articles
- Building Network Services with Finagle and Ostrich by Nathan Hamblen - Finagle is awesome
- Ostrich - A stats collector & reporter for Scala servers
- ID Generation at Scale by Blake Matheny
- Finagle - a network stack for the JVM that you can use to build asynchronous Remote Procedure Call (RPC) clients and servers in Java, Scala, or any JVM-hosted language. Finagle provides a rich set of protocol-independent tools.
- Finagle Redis client from Tumblr
- Tumblr. Massively Sharded MySQL by Evan Elias - one of the better presentations on MySQL sharding available
- Staircar: Redis-powered notifications by Blake Matheny
- Flickr Architecture - talks about Flickr's cell architecture

---