Articles / Blurbs on Very Large Scale Data Centers.

Sekar Last Update Feb 4 2011

Google Pro Tip: Use Back-of-the-envelope-calculations to Choose the Best Design

Wednesday, January 26, 2011 at 8:44AM

How do you know which is the "best" design for a given problem? If, for example, you were given the problem of generating an image search results page of 30 thumbnails, would you load images sequentially? In parallel? Would you cache? How would you decide?

If you could harness the power of the multiverse you could try every possible option in the design space

and see which worked best. But that's crazy impractical, isn't it?

Another option is to consider the order of various algorithm alternatives. As a prophet for the Golden Age of Computational Thinking, Google would definitely do this, but what else might Google do?

****

Use Back-of-the-envelope Calculations to Evaluate Different Designs

Jeff Dean, Head of Google's School of Infrastructure Wizardry—instrumental in many of Google's key systems: ad serving, BigTable; search, MapReduce, ProtocolBuffers—advocates evaluating different designs using **back-of-the-envelope calculations**. He gives the full story in this Stanford video presentation.

Back-of-the-envelope calculations are estimates you create using a combination of thought experiments and common performance numbers to a get a good feel for which designs will meet your requirements. Dr. Dean thinks an important skill for every software engineer is the ability to estimate the performance of alternative systems, using back of the envelope calculations, without having to build them.

He gives a great example of the process in the video, but first...

Numbers Everyone Should Know

To evaluate design alternatives you first need a good sense of how long typical operations will take. Dr. Dean gives this list:

- L1 cache reference 0.5 ns
- Branch mispredict 5 ns
- L2 cache reference 7 ns
- Mutex lock/unlock 100 ns
- Main memory reference 100 ns
- Compress 1K bytes with Zippy 10,000 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from network 10,000,000 ns
- Read 1 MB sequentially from disk 30,000,000 ns
- Send packet CA->Netherlands->CA 150,000,000 ns

Some things to notice:

- Notice the magnitude differences in the performance of different options.
- Datacenters are far away so it takes a long time to send anything between them.
- Memory is fast and disks are slow.
- By using a cheap compression algorithm a lot (by a factor of 2) of network bandwidth can be saved.
- Writes are 40 times more expensive than reads.
- Global shared data is expensive. This is a fundamental limitation of distributed systems. The lock contention in shared heavily written objects kills performance as transactions become serialized and slow.

- Architect for scaling writes.
- Optimize for low write contention.
- Optimize wide. Make writes as parallel as you can.

Example: Generate Image Results Page of 30 Thumbnails

The is the example given in the video. Two design alternatives are used as design thought experiments.

Design 1 - Serial

- Read images serially. Do a disk seek. Read a 256K image and then go on to the next image.
- Performance: 30 seeks * 10 ms/seek + 30 * 256K / 30 MB /s = 560ms

Design 2 - Parallel

- Issue reads in parallel.
- Performance: 10 ms/seek + 256K read / 30 MB/s = 18ms
- There will be variance from the disk reads, so the more likely time is 30-60ms

Which design is best? It depends on you requirements, but given the back-of-the-envelope calculations you have a quick way to compare them without building them.

Now you have a framework for asking yourself other design questions and comparing different design variations:

- Does it make sense to cache single thumbnail images?
- Should you cache a whole set of images in one entry?
- Does it make sense to precompute the thumbnails?

To make these estimates realistic you'll have to know the performance of your services. If there is an unknown variable then perhaps you could rapidly prototype just that part to settle the question. To know if caching is a good design alternative, for example, you'll have to know how long it takes to write into your cache.

Lessons Learned

- Back-of-the-envelope calculations allow you to take a look at different variations.
- When designing your system, these are the kind of calculations you should be doing over and over in your head.
- Know the back of the envelope numbers for the building blocks of your system. It's not good enough to just know the generic performance numbers, you have to know how your subsystems perform. You can't make decent back-of-the-envelope calculations if you don't know what's going on.
- Monitor and measure every part of your system so you can make these sorts of projections from real data.

I personally quite like this approach. It seems much more grounded in the end-to-end nature of a system than is common. The practice today is to focus on the trickeration of various algorithms, which are really a researchable and pluggable part of this larger, more holistic analysis.

Related Articles

- Numbers Everyone Should Know
- The Back of the Napkin by Dan Roam
- A Physicist Explains Why Parallel Universes May Exist by Brian Green

****

Numbers Everyone Should Know

Wednesday, February 18, 2009 at 8:14AM

Google AppEngine Numbers

This group of numbers is from Brett Slatkin in Building Scalable Web Apps with Google App Engine.

Writes are expensive!

 Datastore is transactional: writes require disk access

 Disk access means disk seeks

 Rule of thumb: 10ms for a disk seek

 Simple math: 1s / 10ms = 100 seeks/sec maximum

 Depends on:
* The size and shape of your data
* Doing work in batches (batch puts and gets)

Reads are cheap!

- Reads do not need to be transactional, just consistent
- Data is read from disk once, then it's easily cached
- All subsequent reads come straight from memory
- Rule of thumb: 250usec for 1MB of data from memory
- Simple math: 1s / 250usec = 4GB/sec maximum

* For a 1MB entity, that's 4000 fetches/sec

Numbers Miscellaneous

This group of numbers is from a presentation Jeff Dean gave at a Engineering All-Hands Meeting at Google.

- L1 cache reference 0.5 ns
- Branch mispredict 5 ns
- L2 cache reference 7 ns
- Mutex lock/unlock 100 ns
- Main memory reference 100 ns
- Compress 1K bytes with Zippy 10,000 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from network 10,000,000 ns

- Read 1 MB sequentially from disk 30,000,000 ns
- Send packet CA->Netherlands->CA 150,000,000 ns

The Lessons

- Writes are 40 times more expensive than reads.
- Global shared data is expensive. This is a fundamental limitation of distributed systems. The lock contention in shared heavily written objects kills performance as transactions become serialized and slow.
- Architect for scaling writes.
- Optimize for low write contention.
- Optimize wide. Make writes as parallel as you can.

The Techniques

Keep in mind these are from a Google AppEngine perspective, but the ideas are generally applicable.

Sharded Counters

We always seem to want to keep count of things. But BigTable doesn't keep a count of entities because it's a key-value store. It's very good at getting data by keys, it's not interested in how many you have. So the job of keeping counts is shifted to you.

The naive counter implementation is to lock-read-increment-write. This is fine if there a low number of writes. But if there are frequent updates there's high contention. Given the the number of writes that can be made per second is so limited, a high write load serializes and slows down the whole process.

The solution is to shard counters. This means:

- Create N counters in parallel.
- Pick a shard to increment transactionally at random for each item counted.
- To get the real current count sum up all the sharded counters.
- Contention is reduced by 1/N. Writes have been optimized because they have been spread over the different shards. A bottleneck around shared state has been removed.

This approach seems counter-intuitive because we are used to a counter being a single incrementable variable. Reads are cheap so we replace having a single easily read counter with having to make multiple reads to recover the actual count. Frequently updated shared variables are expensive so we shard and parallelize those writes.

With a centralized database letting the database be the source of sequence numbers is doable. But to scale writes you need to partition and once you partition it becomes difficult to keep any shared state like counters. You might argue that so common a feature should be provided by GAE and I would agree 100 percent, but it's the ideas that count (pun intended).

Paging Through Comments

How can comments be stored such that they can be paged through in roughly the order they were entered?

Under a high write load situation this is a surprisingly hard question to answer. Obviously what you want is just a counter. As a comment is made you get a sequence number and that's the order comments are displayed. But as we saw in the last section shared state like a single counter won't scale in high write environments.

A sharded counter won't work in this situation either because summing the shared counters isn't transactional. There's no way to guarantee each comment will get back the sequence number it allocated so we could have duplicates.

Searches in BigTable return data in alphabetical order. So what is needed for a key is something unique and alphabetical so when searching through comments you can go forward and backward using only keys.

A lot of paging algorithms use counts. Give me records 1-20, 21-30, etc. SQL makes this easy, but it doesn't work for BigTable. BigTable knows how to get things by keys so you must make keys that return data in the proper order.

In the grand old tradition of making unique keys we just keep appending stuff until it becomes unique. The suggested key for GAE is: **time stamp** + user ID + user comment ID.

Ordering by date is obvious. The good thing is getting a time stamp is a local decision, it doesn't rely on writes and is scalable. The problem is timestamps are not unique, especially with a lot of users.

So we can add the user name to the key to distinguish it from all other comments made at the same time. We already have the user name so this too is a cheap call.

Theoretically even time stamps for a single user aren't sufficient. What we need then is a sequence number for each user's comments.

And this is where the GAE solution turns into something totally unexpected. Our goal is to remove write contention so we want to parallelize writes. And we have a lot available storage so we don't have to worry about that.

With these forces in mind, the idea is to create a counter per user. When a user adds a comment it's added to a user's comment list and a sequence number is allocated. Comments are added in a transactional context on a per user basis using Entity Groups. So each comment add is guaranteed to be unique because updates in an Entity Group are serialized.

The resulting key is guaranteed unique and sorts properly in alphabetical order. When paging a query is made across entity groups using the ID index. The results will be in the correct order. Paging is a matter of getting the previous and next keys in the query for the current page. These keys can then be used to move through index.

I certainly would have never thought of this approach. The idea of keeping per user comment indexes is out there. But it cleverly follows

the rules of scaling in a distributed system. Writes and reads are done in parallel and that's the goal. Write contention is removed.

Jan112011

Google Megastore - 3 Billion Writes and 20 Billion Read Transactions Daily

Tuesday, January 11, 2011 at 11:39PM

A giant step into the fully distributed future has been taken by the Google App Engine team with the release of their High Replication Datastore. The HRD is targeted at mission critical applications that require data replicated to at least three datacenters, full ACID semantics for entity groups, and lower consistency guarantees across entity groups.

This is a major accomplishment. Few organizations can implement a true multi-datacenter datastore. Other than SimpleDB, how many other publicly accessible database services can operate out of multiple datacenters? Now that capability can be had by anyone. But there is a price, literally and otherwise. Because the HRD uses three times the resources as Google App Engine's Master/Slave datastatore, it will cost three times as much. And because it is a distributed database, with all that implies in the CAP sense, developers will have to be very careful in how they architect their applications because as costs increased, reliability increased, complexity has increased, and performance has decreased. This is why HRD is targeted ay mission

critical applications, you gotta want it, otherwise the Master/Slave datastore makes a lot more sense.

The technical details behind the HRD are described in this paper, Megastore: Providing Scalable, Highly Available Storage for Interactive Services. This is a wonderfully written and accessible paper, chocked full of useful and interesting details. James Hamilton wrote an excellent summary of the paper in Google Megastore: The Data Engine Behind GAE. There are also a few useful threads in Google Groups that go into some more details about how it works, costs, and performance (the original announcement, performance comparison).

Some Megastore highlights:

- Megastore *blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS*. It has been used internally by Google for several years, on more than 100 production applications, to handle more than three billion write and 20 billion read transactions daily, and store a petabyte of data across many global datacenters.
- *Megastore is a storage system developed to meet the storage requirements of today's interactive online services. It is novel in that it blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS. It uses synchronous replication to achieve high availability and a consistent view of the data. In brief, it provides fully serializable ACID semantics over distant replicas with low enough latencies to support interactive applications. We accomplish this by taking a middle ground in the RDBMS vs. NoSQL design space: we partition the datastore and replicate each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. We provide*

traditional database features, such as secondary indexes, but only those features that can scale within user-tolerable latency limits, and only with the semantics that our partitioning scheme can support. We contend that the data for most Internet services can be suitably partitioned (e.g., by user) to make this approach viable, and that a small, but not spartan, set of features can substantially ease the burden of developing cloud applications.

- Paxos is used to manage synchronous replication between datacenters. This provides the highest level of availability for reads and writes at the cost of higher-latency writes. Typically Paxos is used only for coordination, Megastore also uses it to perform write operations.
- Supports 3 levels of read consistency: current, snapshot, and inconsistent reads.
- Entity groups are now a unit of consistency as well as a unit of transactionality. Entity groups seem to be like little separate databases. Each is *independently and synchronously replicated over a wide area. The underlying data is stored in a scalable NoSQL datastore in each datacenter.*
- The App Engine Datastore doesn't support transactions across multiple entity groups because it will greatly limit the write throughput when not operating on an entity group, though Megastore does support these operations.
- Entity groups are an apriori grouping of data for fast operations. Their size and composition must be balanced. Examples of entity groups are: an email account for a user; a blog would have a profile entity group and more groups to hold posts and meta data for each blog. Each application will have to find natural ways to draw entity group boundaries. Fi ne-grained entity groups will force

expensive cross-group operations. Groups with too much unrelated data will cause unrelated writes to be serialized which degrades throughput. This a process that ironically seems a little like normalizing and will probably prove just as frustrating.

- Queries that require strongly consistent results must be restricted to a single entity group. Queries across entity groups may return stale results This is a major change for programmers. The Master/Slave datastore defaulted to strongly consistent results for all queries, because reads and writes were from the master replica by default. With multiple datacenters the world is a lot ore complicated. This is clear from some the Google group comments too. Performance will vary quite a bit where entities are located and how they are grouped.
- Applications will remain fully available during planned maintenance periods, as well as during most unplanned infrastructure issues. The Master/Slave datastore was subject to periodic maintenance windows. If availability is job one for your application the HRD is a big win.
- Backups and redundancy are achieved via synchronous replication, snapshots, and incremental log backups.
- The datastore API does not change at all.
- Writes to a single entity group are strongly consistent.
- Writes are limited to an estimated 1 per second per entity group, so HRD is not a good match when high usage is expected. This number is not a strict limit, but a good rule of thumb for write performance.
- With eventual consistency, more than 99.9% of your writes are available for queries within a few seconds.
- Only new applications can choose the HRD option. An existing application must be moved to a new application.

- Performance can be improved at the expense of consistency by setting the read_policy to eventually consistent. This will be bring performance similar to that of Master/Slave datastore. Writes are not affected by this flag, it only works for read performance, which are already fast in the 99% case, so it doesn't have a very big impact. Applications doing batch gets will notice impressive speed ups from using this flag.
- One application can't mix Master/Slave with HRD. The reasoning is HRD can serve out of multiple datacenters and Master/Slave can not, so there's no way to ensure in failure cases that apps are running in the right place. So if you planned to use an expensive HRD for critical data and the less expensive Master/Slave for less critical data, you can't do that. You might be thinking to delegate Master/Slave operations to another application, but splitting up applications that way is against the TOS.
- Once HRD is selected your choice can't be changed. So if you would like to start with the cheeper Master/Slave for customers who want to pay less and use HRD who would like to pay for a premium service, you can't do that.
- There's no automated migration of Master/Slave data, the HRD data. The application must write that code. The reasoning is the migration will require a read-only period and the application is in the best position to know how to minimize that downtime. There are some tools and code provided to make migration easier.
- Moving to a caching based architecture will be even more important to hide some of the performance limitations of HRD. Cache can include memcache, cookies, or state put in a URL.

- Though HRD is based on the Megastore, they do not seem to be the same thing, not every feature of the Megastore may be made available in HRD. To what extent this is true I'm not sure.

Related Articles

- May 25th Datastore Outage Post-mortem
- Paxos Made Live – An Engineering Perspective
- Choosing a Datastore
- Using the High Replication Datastore
- Bigtable: A Distributed Storage System for Structured Data
- How Google Serves Data From Multiple Datacenters
- ZooKeeper - A Reliable, Scalable Distributed Coordination System
- Consensus Protocols: Paxos
- Yahoo!'S PNUTS Database: Too Hot, Too Cold Or Just Right?

YouTube Architecture

Wednesday, March 12, 2008 at 3:54PM

Update 2: YouTube Reaches One Billion Views Per Day. *That's at least 11,574 views per second, 694,444 views per minute, and 41,666,667 views per hour.*
Update: YouTube: The Platform. YouTube adds a new rich set of APIs in order to become your video platform leader--all for free. Upload, edit, watch, search, and comment on video from your own site without visiting YouTube. Compose your site internally from APIs because you'll need to expose them later anyway.

YouTube grew incredibly fast, to over 100 million video views per day, with only a handful of people responsible for scaling the site. How did they manage to deliver all that video to all those users? And how have they evolved since being acquired by Google?

Information Sources

1. Google Video

Platform

1. Apache
2. Python
3. Linux (SuSe)
4. MySQL
5. psyco, a dynamic python->C compiler
6. lighttpd for video instead of Apache

What's Inside?

The Stats

1. Supports the delivery of over 100 million videos per day.
2. Founded 2/2005
3. 3/2006 30 million video views/day
4. 7/2006 100 million video views/day
5. 2 sysadmins, 2 scalability software architects
6. 2 feature developers, 2 network engineers, 1 DBA

Recipe for handling rapid growth

```
while (true)
{
identify_and_fix_bottlenecks();
drink();
sleep();
notice_new_bottleneck();
}
```

This loop runs many times a day.

Web Servers

1. NetScalar is used for load balancing and caching static content.
2. Run Apache with mod_fast_cgi.
3. Requests are routed for handling by a Python application server.
4. Application server talks to various databases and other informations sources to get all the data and formats the html page.
5. Can usually scale web tier by adding more machines.
6. The Python web code is usually NOT the bottleneck, it spends most of its time blocked on RPCs.
7. Python allows rapid flexible development and deployment. This is critical given the competition they face.
8. Usually less than 100 ms page service times.
9. Use psyco, a dynamic python->C compiler that uses a JIT compiler approach to optimize inner loops.
10. For high CPU intensive activities like encryption, they use C extensions.
11. Some pre-generated cached HTML for expensive to render blocks.
12. Row level caching in the database.
13. Fully formed Python objects are cached.

14. Some data are calculated and sent to each application so the values are cached in local memory. This is an underused strategy. The fastest cache is in your application server and it doesn't take much time to send precalculated data to all your servers. Just have an agent that watches for changes, precalculates, and sends.

Video Serving

 Costs include bandwidth, hardware, and power consumption.

 Each video hosted by a mini-cluster. Each video is served by more than one machine.

 Using a a cluster means:
- More disks serving content which means more speed.
- Headroom. If a machine goes down others can take over.
- There are online backups.

 Servers use the lighttpd web server for video:
- Apache had too much overhead.
- Uses epoll to wait on multiple fds.
- Switched from single process to multiple process configuration to handle more connections.

 Most popular content is moved to a CDN (content delivery network):
- CDNs replicate content in multiple places. There's a better chance of content being closer to the user, with fewer hops, and content will run over a more friendly network.
- CDN machines mostly serve out of memory because the content is so popular there's little thrashing of content into and out of memory.

 Less popular content (1-20 views per day) uses YouTube servers in various colo sites.

- There's a long tail effect. A video may have a few plays, but lots of videos are being played. Random disks blocks are being accessed.
- Caching doesn't do a lot of good in this scenario, so spending money on more cache may not make sense. This is a very interesting point. If you have a long tail product caching won't always be your performance savior.
- Tune RAID controller and pay attention to other lower level issues to help.
- Tune memory on each machine so there's not too much and not too little.

Serving Video Key Points

1. Keep it simple and cheap.
2. Keep a simple network path. Not too many devices between content and users. Routers, switches, and other appliances may not be able to keep up with so much load.
3. Use commodity hardware. More expensive hardware gets the more expensive everything else gets too (support contracts). You are also less likely find help on the net.
4. Use simple common tools. They use most tools build into Linux and layer on top of those.
5. Handle random seeks well (SATA, tweaks).

Serving Thumbnails

 Surprisingly difficult to do efficiently.

 There are a like 4 thumbnails for each video so there are a lot more thumbnails than videos.

 Thumbnails are hosted on just a few machines.

 Saw problems associated with serving a lot of small objects:
- Lots of disk seeks and problems with inode caches and page caches at OS level.

- Ran into per directory file limit. Ext3 in particular. Moved to a more hierarchical structure. Recent improvements in the 2.6 kernel may improve Ext3 large directory handling up to 100 times, yet storing lots of files in a file system is still not a good idea.
- A high number of requests/sec as web pages can display 60 thumbnails on page.
- Under such high loads Apache performed badly.
- Used squid (reverse proxy) in front of Apache. This worked for a while, but as load increased performance eventually decreased. Went from 300 requests/second to 20.
- Tried using lighttpd but with a single threaded it stalled. Run into problems with multiprocesses mode because they would each keep a separate cache.
- With so many images setting up a new machine took over 24 hours.
- Rebooting machine took 6-10 hours for cache to warm up to not go to disk.

 To solve all their problems they started using Google's BigTable, a distributed data store:
- Avoids small file problem because it clumps files together.
- Fast, fault tolerant. Assumes its working on a unreliable network.
- Lower latency because it uses a distributed multilevel cache. This cache works across different collocation sites.
- For more information on BigTable take a look at Google Architecture, GoogleTalk Architecture, and BigTable.

Databases

1. The Early Years
   - Use MySQL to store meta data like users, tags, and descriptions.
   - Served data off a monolithic RAID 10 Volume with 10 disks.
   - Living off credit cards so they leased hardware. When they needed more hardware to handle load it took a few days to order and get delivered.

- They went through a common evolution: single server, went to a single master with multiple read slaves, then partitioned the database, and then settled on a sharding approach.
- Suffered from replica lag. The master is multi-threaded and runs on a large machine so it can handle a lot of work. Slaves are single threaded and usually run on lesser machines and replication is asynchronous, so the slaves can lag significantly behind the master.
- Updates cause cache misses which goes to disk where slow I/O causes slow replication.
- Using a replicating architecture you need to spend a lot of money for incremental bits of write performance.
- One of their solutions was prioritize traffic by splitting the data into two clusters: a video watch pool and a general cluster. The idea is that people want to watch video so that function should get the most resources. The social networking features of YouTube are less important so they can be routed to a less capable cluster.

2. The later years:
   - Went to database partitioning.
   - Split into shards with users assigned to different shards.
   - Spreads writes and reads.
   - Much better cache locality which means less IO.
   - Resulted in a 30% hardware reduction.
   - Reduced replica lag to 0.
   - Can now scale database almost arbitrarily.

Data Center Strategy

1. Used manage hosting providers at first. Living off credit cards so it was the only way.

2. Managed hosting can't scale with you. You can't control hardware or make favorable networking agreements.
3. So they went to a colocation arrangement. Now they can customize everything and negotiate their own contracts.
4. Use 5 or 6 data centers plus the CDN.
5. Videos come out of any data center. Not closest match or anything. If a video is popular enough it will move into the CDN.
6. Video bandwidth dependent, not really latency dependent. Can come from any colo.
7. For images latency matters, especially when you have 60 images on a page.
8. Images are replicated to different data centers using BigTable. Code looks at different metrics to know who is closest.

Lessons Learned

1. **Stall for time**. Creative and risky tricks can help you cope in the short term while you work out longer term solutions.
2. **Prioritize**. Know what's essential to your service and prioritize your resources and efforts around those priorities.
3. **Pick your battles**. Don't be afraid to outsource some essential services. YouTube uses a CDN to distribute their most popular content. Creating their own network would have taken too long and cost too much. You may have similar opportunities in your system. Take a look at Software as a Service for more ideas.
4. **Keep it simple!** Simplicity allows you to rearchitect more quickly so you can respond to problems. It's true that nobody really knows what simplicity is, but if you aren't afraid to make changes then that's a good sign simplicity is happening.

5. **Shard**. Sharding helps to isolate and constrain storage, CPU, memory, and IO. It's not just about getting more writes performance.
6. **Constant iteration on bottlenecks**:
   - Software: DB, caching
   - OS: disk I/O
   - Hardware: memory, RAID
7. **You succeed as a team**. Have a good cross discipline team that understands the whole system and what's underneath the system. People who can set up printers, machines, install networks, and so on. With a good team all things are possible.

Todd Hoff | 47 Comments | Permalink | Share Article Print Article Email Article

in Apache, CDN, Example, Linux, MySQL, Python, Shard, lighttpd