Google Architecture

Google is the King of scalability. Everyone knows Google for their large, sophisticated, and fast searching, but they don't just shine in search. Their platform approach to building scalable applications allows them to roll out internet scale applications at an alarmingly high competition crushing rate. Their goal is always to build a higher performing higher scaling infrastructure to support their products. How do they do that?

Platform

1. Linux
2. A large diversity of languages: Python, Java, C++

What's Inside?

The Stats

1. Estimated 450,000 low-cost commodity servers in 2006
2. In 2005 Google indexed 8 billion web pages. By now, who knows?
3. Currently there over 200 GFS clusters at Google. A cluster can have 1000 or even 5000 machines. Pools of tens of thousands of machines retrieve data from GFS clusters that run as large as 5 petabytes of storage. Aggregate read/write throughput can be as high as 40 gigabytes/second across the cluster.
4. Currently there are 6000 MapReduce applications at Google and hundreds of new applications are being written each month.
5. BigTable scales to store billions of URLs, hundreds of terabytes of satellite imagery, and preferences for hundreds of millions of users.

The Stack

Google visualizes their infrastructure as a three layer stack:

1. Products: search, advertising, email, maps, video, chat, blogger
2. Distributed Systems Infrastructure: GFS, MapReduce, and BigTable.
3. Computing Platforms: a bunch of machines in a bunch of different data centers
4. Make sure easy for folks in the company to deploy at a low cost.
5. Look at price performance data on a per application basis. Spend more money on hardware to not lose log data, but spend less on other types of data. Having said that, they don't lose data.

Reliable Storage Mechanism with GFS (Google File System)

1. Reliable scalable storage is a core need of any application. GFS is their core storage platform.
2. Google File System - large distributed log structured file system in which they throw in a lot of data.

3. Why build it instead of using something off the shelf? Because they control everything and it's the platform that distinguishes them from everyone else. They required:
   - high reliability across data centers
   - scalability to thousands of network nodes
   - huge read/write bandwidth requirements
   - support for large blocks of data which are gigabytes in size.
   - efficient distribution of operations across nodes to reduce bottlenecks

4. System has master and chunk servers.
   - Master servers keep metadata on the various data files. Data are stored in the file system in 64MB chunks. Clients talk to the master servers to perform metadata operations on files and to locate the chunk server that contains the needed they need on disk.
   - Chunk servers store the actual data on disk. Each chunk is replicated across three different chunk servers to create redundancy in case of server crashes. Once directed by a master server, a client application retrieves files directly from chunk servers.

5. A new application coming on line can use an existing GFS cluster or they can make your own. It would be interesting to understand the provisioning process they use across their data centers.

6. Key is enough infrastructure to make sure people have choices for their application. GFS can be tuned to fit individual application needs.

Do Something With the Data Using MapReduce

1. Now that you have a good storage system, how do you do anything with so much data? Let's say you have many TBs of data stored across a 1000 machines. Databases don't scale or cost effectively scale to those levels. That's where MapReduce comes in.

2. MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

3. Why use MapReduce?
   - Nice way to partition tasks across lots of machines.
   - Handle machine failure.
   - Works across different application types, like search and ads. Almost every application has map reduce type operations. You can precompute useful data, find word counts, sort TBs of data, etc.
   - Computation can automatically move closer to the IO source.

4. The MapReduce system has three different types of servers.
   - The Master server assigns user tasks to map and reduce servers. It also tracks the state of the tasks.
   - The Map servers accept user input and performs map operations on them. The results are written to intermediate files
   - The Reduce servers accepts intermediate files produced by map servers and performs reduce operation on them.

5. For example, you want to count the number of words in all web pages. You would feed all the pages stored on GFS into MapReduce. This would all be happening on 1000s of machines simultaneously and all the coordination, job scheduling, failure handling, and data transport would be done automatically.
   - The steps look like: GFS -> Map -> Shuffle -> Reduction -> Store Results back into GFS.
   - In MapReduce a map maps one view of data to another, producing a key value pair, which in our example is word and count.
   - Shuffling aggregates key types.
   - The reductions sums up all the key value pairs and produces the final answer.

6. The Google indexing pipeline has about 20 different map reductions. A pipeline looks at data with a whole bunch of records and aggregating keys. A second map-reduce comes a long, takes that result and does something else. And so on.

7. Programs can be very small. As little as 20 to 50 lines of code.

8. One problem is stragglers. A straggler is a computation that is going slower than others which holds up everyone. Stragglers may happen because of slow IO (say a bad controller) or from a temporary CPU spike. The solution is to run multiple of the same computations and when one is done kill all the rest.

9. Data transferred between map and reduce servers is compressed. The idea is that because servers aren't CPU bound it makes sense to spend on data compression and decompression in order to save on bandwidth and I/O.

Storing Structured Data in BigTable

1. BigTable is a large scale, fault tolerant, self managing system that includes terabytes of memory and petabytes of storage. It can handle millions of reads/writes per second.

2. BigTable is a distributed hash mechanism built on top of GFS. It is not a relational database. It doesn't support joins or SQL type queries.

3. It provides lookup mechanism to access structured data by key. GFS stores opaque data and many applications needs has data with structure.

4. Commercial databases simply don't scale to this level and they don't work across 1000s machines.

5. By controlling their own low level storage system Google gets more control and leverage to improve their system. For example, if they want features that make cross data center operations easier, they can build it in.

6. Machines can be added and deleted while the system is running and the whole system just works.

7. Each data item is stored in a cell which can be accessed using a row key, column key, or timestamp.
8. Each row is stored in one or more tablets. A tablet is a sequence of 64KB blocks in a data format called SSTable.
9. BigTable has three different types of servers:
   - The Master servers assign tablets to tablet servers. They track where tablets are located and redistributes tasks as needed.
   - The Tablet servers process read/write requests for tablets. They split tablets when they exceed size limits (usually 100MB - 200MB). When a tablet server fails, then a 100 tablet servers each pickup 1 new tablet and the system recovers.
   - The Lock servers form a distributed lock service. Operations like opening a tablet for writing, Master aribtration, and access control checking require mutual exclusion.
10. A locality group can be used to physically store related bits of data together for better locality of reference.
11. Tablets are cached in RAM as much as possible.

Hardware

1. When you have a lot of machines how do you build them to be cost efficient and use power efficiently?
2. Use ultra cheap commodity hardware and built software on top to handle their death.
3. A 1,000-fold computer power increase can be had for a 33 times lower cost if you you use a failure-prone infrastructure rather than an infrastructure built on highly reliable components. You must build reliability on top of unreliability for this strategy to work.
4. Linux, in-house rack design, PC class mother boards, low end storage.
5. Price per wattage on performance basis isn't getting better. Have huge power and cooling issues.
6. Use a mix of collocation and their own data centers.

Misc

1. Push changes out quickly rather than wait for QA.
2. Libraries are the predominant way of building programs.
3. Some are applications are provided as services, like crawling.
4. An infrastructure handles versioning of applications so they can be release without a fear of breaking things.

Future Directions for Google

1. Support geo-distributed clusters.
2. Create a single global namespace for all data. Currently data is segregated by cluster.
3. More and better automated migration of data and computation.

4. Solve consistency issues that happen when you couple wide area replication with network partitioning (e.g. keeping services up even if a cluster goes offline for maintenance or due to some sort of outage).

Lessons Learned

1. **Infrastructure can be a competitive advantage**. It certainly is for Google. They can roll out new internet services faster, cheaper, and at scale at few others can compete with. Many companies take a completely different approach. Many companies treat infrastructure as an expense. Each group will use completely different technologies and their will be little planning and commonality of how to build systems. Google thinks of themselves as a systems engineering company, which is a very refreshing way to look at building software.
2. **Spanning multiple data centers is still an unsolved problem**. Most websites are in one and at most two data centers. How to fully distribute a website across a set of data centers is, shall we say, tricky.
3. **Take a look at Hadoop if you don't have the time to rebuild all this infrastructure from scratch yourself. Hadoop is an open source implementation of many of the same ideas presented here.**
4. **An under appreciated advantage** of a platform approach is junior developers can quickly and confidently create robust applications on top of the platform. If every project needs to create the same distributed infrastructure wheel you'll run into difficulty because the people who know how to do this are relatively rare.
5. **Synergy isn't always crap**. By making all parts of a system work together an improvement in one helps them all. Improve the file system and everyone benefits immediately and transparently. If every project uses a different file system then there's no continual incremental improvement across the entire stack.
6. **Build self-managing systems that work without having to take the system down**. This allows you to more easily rebalance resources across servers, add more capacity dynamically, bring machines off line, and gracefully handle upgrades.
7. **Create a Darwinian infrastructure**. Perform time consuming operation in parallel and take the winner.
8. **Don't ignore the Academy**. Academia has a lot of good ideas that don't get translated into production environments. Most of what Google has done has prior art, just not prior large scale deployment.
9. **Consider compression**. Compression is a good option when you have a lot of CPU to throw around and limited IO.