Motivating a Distributed System of Commodity Machines[1]

Andrew Kane
arkane@uwaterloo.ca

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada.

Technical Report CS-2009-09
February 18, 2009

[1] This paper was originally created as a course project for 'CS 798 Advanced Research Topics - Information Retrieval' taught by Professor Charles Clarke in Fall 2007.

Motivating a Distributed System of Commodity Machines

Andrew Kane
arkane@uwaterloo.ca

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada.

ABSTRACT

This report examines the price/performance benefit of using a large cluster of commodity machines rather than server level hardware for certain large scale software applications.

A number of tools are presented which make it easier to produce software that runs across large clusters of commodity machines. These tools are the Chubby locking service, the Google file system, MapReduce and BigTable, all written by Google.

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

Keywords

Distributed system, Google, file system, commodity machines, performance.

1. INTRODUCTION

Increasingly software applications must handle very large amounts of data and very large numbers of requests. This is especially true of applications on the world wide web, premiere among them being the web search engine.

Search engine vendors like Google have learned that running on a large cluster of commodity machines can result in a large price performance benefit.

Building software to run on a large cluster of commodity machines is not an easy task. This report presents some software tools written by Google that make software development easier.

2. DESIGNING A WEB SEARCH SYSTEM

An example is used to examine the issues associated with building a large computer system. This example will be used to motivate the use of a distributed system of commodity machines based on a cost benefit analysis.

2.1 Web Search Example

Example #1: How would you build a web search engine such as Google, comprising a cache of the web, an indexing system, a searching system and presentation of results with query specific summaries. The resulting system must scale in disk size, disk performance, and CPU performance. It must also have good redundancy and fault tolerance. Even with all these requirements, the system must have a low total cost of ownership (TCO).

2.2 How Big Is The Web?

In 2005, Google claimed their index contained 8 billion web pages and the indexable web was thought to be 11.5 billion pages [1]. Even at that time, some people thought that Google was indexing much more than 8 billion pages. They are indexing much more than that now, perhaps even 100 billion pages. For this example, we will assume the number is 50 billion pages.

The average web page [2] contains approximately 25kB of HTML and a total size of approximately 130kB including scripts, images, etc. Many of the scripts and images will be shared between multiple web pages, but that effect is ignored here.

The size of the cache of the web in our example is:

50 billion * 130kB = 6,500 TB

The size of the index in our example, assuming the standard 30% of the text input size, is:

50 billion * 25kB * 30% = 375 TB

2.3 Serving a Cache of the Web

The cache of the web in our example is externally visible to the end users. The results pages for searches using Google allow the user to open the cached version of the result, which often loads faster than the original location for that result, but it might not be entirely up to date. Also, this cache allows a user to see the contents that matched their search, even if that data was subsequently removed from the original site. In this usage pattern, the cached result is not processed, it is just returned to the user.

Given this usage pattern, the cache of the web needs very little CPU resources. Also, the number of requests is small relative to the data size, meaning this data does not need extra disk performance via fast disks or replicas.

2.4 Producing the Index

The indexing process must read through all of the data in the cache of the web, process it, and produce the searchable index. This process does not need to run very often and, with slight complications, it can be done incrementally (I.E. the system could index only the data that has changed since the last time it was indexed).

Given this usage pattern, the resource requirements for the indexing process are negligible when compared to the search process resource requirements (below).

2.5 Serving Web Search

In March 2006, Google was serving approximately 91 million searches per day in the United States [3]. That number is much higher now and the number for the entire world would be higher still. However, many of the top searches are repeated many times and the results are served from a cache rather than executed repeatedly. For this analysis, we assume that the search system

must support 200 million searches per day which is 2,315 searches or queries per second (qps).

Disk performance becomes a big issue for the search process. Assuming that the index is split into partitions each fitting on a 500 GB disk, how many copies of each partition are needed? A normal disk (7200 RPM) can support approximately 100 random reads per second (rps). Given 3 terms per query and 2 random reads per term we end up with the number of copies of each partition:

2,315 qps * 3 terms * 2 reads / 100 rps/disk = 139 disks.

The number of partitions to support the index is:

375 TB / 500 GB/partition = 750 partitions.

The number of disks required to support the search process is:

750 partitions * 139 disks/partition = 104,250 disks.

Note: Both the query rate and the web size affect this number and both are growing quickly over time. This means the number of disks required for the search process is growing very quickly over time. This could quickly motivate the use of flash memory instead of disk (see Section 3.3).

It is unclear how much CPU resources are needed for each partition, but it seems reasonable to assume that one disk could saturate 1 CPU core given the complex ranking system.

2.6 Serving the Search Results

Producing the search results page does not normally require a lot of resources, but we are producing a query specific summary for each result. That means reading the cached page and processing it for each results. Assuming 10 results per page, the number of random disk reads is:

2,315 qps * 10 results = 23,150 rps.

The cache of the web will be spread over 500 GB drives requiring:

6,500 TB / 500 GB/drive = 13,000 drives.

Assuming the disk reads are spread randomly across the cache of the web, then supporting the query based summary requires:

23,150 rps / 13,000 drives = 1.8 rps/drive.

Normal disks (7200 RPM) can support 100 rps/drive, so the disk resources used by query based summary are negligible.

The CPU resources used by query based summary are assumed to also be negligible.

3. COSTS OF RUNNING THE SYSTEM

There are many costs involved in building and running the system to support our example. These include the cost of purchasing the computer hardware, the cost of powering the hardware and the cost of administrating the system. We assume the administration costs are the same for commodity hardware and server level hardware for our example.

3.1 Hardware Costs

The main costs of the system in our example are the storage of the cache of the web and the resources required by the search process. All the other costs are negligible and can be ignored.

3.1.1 Commodity Hardware

This analysis assumes that each machine has a dual core CPU and that the disks are 500 GB at 7200 RPM.

The cache of the web requires 13,000 drives (Section 2.6), but has low CPU resource requirements. Implementing redundancy requires more space; in this case we assume 3 times the space as used by the Google file system (see Section 4.2), This means we need 39,000 drives. The low CPU resource usage means we can put 4 disks in each machine giving:

39,000 drives / 4 drives/machine = 9,750 machines.

The search process needs 104,250 disks (Section 2.5) and each disk needs 1 CPU core. Each machine has 2 CPU cores so:

104,250 cores / 2 cores/machine = 52,125 machines.

In total, the example application's resource requirements are:

9,750 + 52,125 = 61,875 machines.

39,000 + 104,250 = 143,250 drives.

The current online prices [4] are sufficient for this analysis. A Core 2 Duo CPU (E4500 at 2.2GHz) costs approximately $150, and the rest of the machine is approximately $150. A 500 GB drive at 7200 RPM is approximately $130.

Using these prices, the total cost of hardware for the example application would be:

61,875 * ($150 + $150) + 143,250 * $130

= $18,562,500 + $18,622,500

= $37,185,000

A three year replacement schedule is usually assumed [5], which means the hardware costs for the system are approximately $12.4 million per year.

3.1.2 Server Level Hardware

Server level hardware means large multi-CPU machines attached to large disk systems like Network Attached Storage (NAS) or Storage Area Network (SAN) devices which use RAID architecture to support redundancy and fault tolerance.

It is often difficult to get prices for server level hardware, but in general the larger the system the more it costs per resource. The more expensive systems may be more reliable, have more features or more support. Therefore, using prices for medium sized hardware should be sufficient to approximate the costs for this analysis.

Using Dell prices for hardware [6], the PowerVault MD3000 (PV) RAID system costs $7,500 without disks and can hold 15 disk drives, assuming one parity disk per PV leaves 14 usable disks. For the cache of the web we need:

13,000 drives / 14 drives/PV = 929 PV.

For the search process we need:

143,250 drives / 14 drives/PV = 10,232 PV.

The price for a PowerVault with 15x500 GB drives is:

$7,500 + 15 * $130 = $9,450.

So the cost of the server level disk system for the example application is:

(929 PV + 10,232 PV) * $9,450 / PV

= $105,471,450

This is already much more than the cost of the commodity hardware system, so we will not continue to calculate the server level machine costs for the example application.

3.2 Power Costs

Power consumption is becoming more of an issue as the price of hardware declines. At these scales the commodity machines are built into racks with very high power densities, and therefore require expensive cooling systems.

Each Core Duo CPU uses about 65W [7] and one motherboard with RAM uses about 25W. Each disk drive uses about 10W. The power supply is usually only 75% efficient. Cooling takes approximately 50W per machine [5]. This means the commodity hardware power consumption in our example is:

(61,875*90W + 143,250*10W) / 0.75 + 61,875 * 50W

= 12,428 kW

Assuming 15 cents per hW-h to cover the power usage, UPS and distribution loss, the power costs for the commodity hardware in our example application would be:

12,428 kW * 24*365 h/year * 0.15 $/kW-h

= $16,330,392 per year

This is a significant cost and is approximately 132% of the cost of the hardware per year. Back in 2003, the power costs were much smaller compared to the hardware costs, approximately 19% of the cost of the hardware per year [5]. This would explain why Google seems to be building new data centres where power is less expensive. As the price of hardware decreases, one would expect this trend to continue. However, the CPU accounts for a large portion of the power consumption and new CPUs are improving their power efficiency (operations per W).

3.3 Flash Memory

Flash memory has been dropping in price very quickly and it is starting to become an interesting option for replacing hard disk drives (HDD) in many applications. Flash memory can execute more than 100 times as many random reads per second as a HDD, and often has faster contiguous read and write performance.

There are hybrid hard drives (HHD) which combine a normal HDD with a large buffer of flash memory. These could be used to reduce the number of random reads serviced by the HDD portion of the drive. In our example, the mapping from a term to the location of the postings list on disk is usually encoded in a B+ tree and accounts for some of the random reads per token. This B+ tree might fit in the flash portion of a HHD, thus reducing the number of random reads per token serviced by the HDD portion.

Flash memory also comes in solid state drives (SSD) which have the same interface as a normal HDD and can be easily plugged into commodity machines. The SSDs are approximately $8 per gigabyte, compared to $0.25 per gigabyte for HDD [8], which is 32 times more expensive. In our example, the indexing portion needed 139 disk copies to service the query stream. Using the 100 times speed up for random reads, SSDs may be able to service the query stream with just the minimum 3 copies (for redundancy). The SSDs would also be cheeper then the HDDs because each SSD replaces 139/3=46 HDDs (which is larger than the price ratio of 32). Unfortunately, 500 GB SSDs do not exist yet, so the calculations from Section 2.5 change more than suggested above. Our CPU resource assumption of one core per 500 GB disk could also be an issue.

In all likelihood, Google is already experimenting with clusters using flash memory, both HHD and SSD. Hopefully such systems will soon be used broadly in production applications.

4. SOFTWARE TOOLS TO BUILD THE SYSTEM

Building software to run across large numbers of commodity machines is a difficult endeavor. Google has released information on a number of the tools they use to build these systems, some of which are presented here.

4.1 Chubby Locking System

Chubby [9] is a lock service in a distributed system. It is intended to supply synchronization between client processes and store small amounts of information used by multiple clients, for example configuration information. One common use of the synchronization features is to elect a “master” or “leader” process from a set of possible processes. This is a distributed consensus problem, solved through the use of the Paxos protocol [10].

Chubby is used within Google as a name service rather than the Domain Name Service (DNS) because it can support faster refresh with less overhead than DNS especially for a large number of clients. Apparently MapReduce [12] uses Chubby as a rendezvous mechanism, though it is not mentioned in the 2004 paper. The Google files system (GFS) [11] and BigTable [13] use Chubby to pick a primary from redundant replicas. Chubby is also use to store configuration files and access control lists (ACLs).

4.2 Google File System (GFS)

The Google file system (GFS) is a distributed file system intended to store a relatively small number of large files and to be distributed over a large number of commodity machines. Each file is broken up into 64 MB chunks. Reliability and fault tolerance are achieved by replicating each file chunk on at least 3 machines and actively maintaining the number of replicas. This replication also improves the read performance of the system. The file system is accessed through a client side library which knows about the 64 MB chunk size and talks to a single master process to get locations of desired chunks.

The GFS design is intended to be simple, reliable and have high aggregate performance. There are many optimizations added to the system to balance load across machines and network connections. The 64 MB chunk size was picked to allow the single master to load all chunk information into memory, thus making many tasks simpler to implement and still have good performance.

For our ongoing example, the GFS could be used to store the entire cache of the web, but the number of files would have to be reduced by combining many web pages into a single file. This would create some issues when the cache of the web is updated. This type of usage pattern and many others is supported by the BigTable abstraction (see Section 4.4).

In our example, the search process could not be implemented on top of GFS because of the 64MB chunk size. If the web pages are partitioned to have a subindex fit in a 64 MB chunk, then each chunk would have 2 random disk reads for each query term rather than each 500 GB disk drive as was calculated, resulting in much

higher resource requirements. If the 500 GB partition is simply split into 64 MB chunks then they would appear on different machines using GFS and the disk accesses would have to go across the network, resulting in a bottleneck.

4.3 MapReduce

MapReduce [12] is a programming abstraction that allows a programmer to work on hugh datasets by exploiting parallelism across a large number of machines without needing to implement the parallelization or understand how it happens. The resulting programs are easier to write, understand, modify, and debug. The abstraction requires only two parts to be written by the user; the map function and the reduce function.

The map function takes a <key, value> pair, processes it in some way, then outputs a set of <key, value> pairs. The system then sorts these <key, value> pairs by the key and groups them into <key, list(values)> pairs. The reduce function then takes a <key, list(values)> pair, processes it in some way and produces a list of values. In mathematical notation this would be:

map: <K1, V1> → <K2, V2>

reduce: <K2, list(V2)> → list(V3)

The MapReduce library executes on a triple <map function, reduce function, input dataset>. The library splits the input data into M pieces typically 16 to 64 MB and starts many processes on a cluster of machines which will do the work. One of the processes becomes the master, and the others become the workers. The intermediate results output by the map function are split into R pieces using a partitioning function *hash(key) mod R*. The workers can be assigned a map task, meaning they process one of the M pieces of input data, buffering results in memory, and periodically writing them to disk partitioned into R parts using the partitioning function. The workers can also get assigned reduce tasks, meaning for one reduce partition it collects the corresponding parts output by the map task, and sorts them. When all intermediate results have been collected for a reduce partition, it can execute the user defined reduce function for each key.

The MapReduce system is optimized in many ways. The system tries to run processes on the machines that contain the disk files to reduce network usage. The variables M and R can be static, tuned to the data size or specified by the user. Tasks that are taking too long can be killed and restarted, perhaps on another machine, which often results in a significantly reduced overall completion time. In addition, workers are pinged periodically to check if they have failed.

In our example, the MapReduce abstraction can be used to implement the indexing step. The map function parses each document and produces a list of <word-partition, docid-offset> pairs, then the reduce function takes in a <word-partition, list (docid-offset)> pair and encodes the list(docid-offset) into the on disk index format. A second pass could then combine the postings lists into full index partitions.

4.4 BigTable

BigTable [13] is a distributed system for storing and manipulating structured data. It is similar to a relational database system, but the supported functionality is simplified to allow the system to scale to huge sizes and still support low latency requirements as needed. The trade-off between scale and performance is tunable by the user who has dynamic control over data layout and format. For example, deciding if the data is stored in memory or on disk.

The BigTable data model is a simple map between a <row key, column key, timestamp> and some string value:

<row:string, column:string, time:int64> → string

BigTable stores data in lexicographic order by row key and splits the data into row ranges called tablets which allows data to be easily distributed. All data for a particular row is stored in only one tablet and reads/writes to data in a single row are atomic. Data tablets are stored in GFS using the Google SSTable format. The entire BigTable instance is organized into a B+ style tree with three levels. A pointer to the root of this tree is stored in Chubby. The first level is a single tablet which stores the locations of all tablets in the second level. The second level comprises a special metadata table which stores the location and row key ranges of third level tablets. The third level stores the actual data.

As with most systems at Google, there are many optimizations built into the system to improve performance in specific areas. Data location lookup is optimized in many ways, including caching tablet location in memory and prefetching tablet locations in batches. Update/commit logs are shared between all tablets on a machine to optimize for small mutations. SSTable files are immutable so they can be accessed without locking, and easily cached in memory potentially on multiple machines, meaning tablets have to merge and compact SSTable and commit log information. Users can specify compression formats for data.

There is an integration to allow MapReduce to be easily executed over BigTable data, creating a very powerful system.

In our example, BigTable can be used to store the cache of the web without having to combine multiple web pages into one file as required when using GFS directly (see Section 4.2). Updates to the cache of the web would also be handled easily. Using the MapReduce integration, the data stored in the cache of the web could be processed into partition indexes as described in Section 4.3.

BigTable could also be used to calculate PageRank scores [14] by using MapReduce to extract the links from the web page, ordering them by the referenced links, and then calculating the PageRank value from that.

The search process cannot use BigTable to store its index structures for the same reasons GFS cannot be used (see Section 4.2).

5. CONCLUSIONS

This report has shown the price/performance benefit of using a large cluster of commodity machines to solve certain large scale problems, rather than using conventional server level hardware. The price/performance comparison was illustrated in detail using a web search engine as an example.

The report presented a description of some tools built at Google. These tools make it easier to create programs that run on large clusters of commodity machines. Possible uses of these tools are presented in context of the previously introduced web search engine example.

6. REFERENCES

[1] Gulli A. and Signorini A. The indexable web is more than 11.5 billion pages. In Anonymous WWW '05: Special interest tracks and posters of the 14th international conference on

World Wide Web. (Chiba, Japan). ACM, New York, NY, USA, 2005, 902-903.

[2] http://www.optimizationweek.com/reviews/average-web-page/

[3] http://searchenginewatch.com/showPage.html?page=2156461

[4] http://www.tigerdirect.ca/

[5] Barroso L. A., Dean J. and Holzle U. Web search for a planet: the google cluster architecture. IEEE Micro, 23, 2 (2003), 22-28.

[6] http:/www.dell.ca/

[7] http://en.wikipedia.org/wiki/CPU_power_dissipation/

[8] http://en.wikipedia.org/wiki/Solid_state_disk/

[9] Burrows M. The Chubby lock service for loosely-coupled distributed systems. (2006), 335-350.

[10] Lamport L. The part-time parliament. ACM Trans.Comput.Syst., 16, 2 (1998), 133-169.

[11] Ghemawat S., Gobioff H. and Shan-Tak Leung . The Google file system. In Anonymous Operating Systems Review,vol. 37,no.5,pp.29-43,Dec. 2003; SOSP'03. 19th ACM Symposium on Operating Systems Principles, 19-22 Oct. 2003, Bolton Landing (Lake George), NY, USA. Assoc. for Comput. Machinery Special Interest Group on Operating Syst. (12). USA, 2003, 29-43.

[12] Dean J. and Sanjay Ghemawat . MapReduce: simplified data processing on large clusters. In Anonymous Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04); Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04), 6-8 Dec. 2004, San Francisco, CA, USA. USENIX Assoc, Berkeley, CA; USA, 137-149.

[13] Chang F., Dean J., Ghemawat S., Hsieh W. C., Burrows D. A. W. M., Chandra T., Fikes A. and Gruber R. E. Bigtable: a distributed storage system for structured data. In Anonymous (2006). Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06) (pp.205-218). Berkeley, CA: USENIX Assoc.. 396pp.; Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), 6-8 Nov. 2006, Seattle, WA, USA. USA, 205-218.

[14] Page, L., Brin, S., Motwani, R., & Winograd, T. (1998). The PageRank citation ranking: Bringing order to the web (Tech. Rep.). Stanford, CA: Stanford Digital Library Technologies Project.