From Google File System to Omega: a Decade of Advancement in Big Data Management at Google

Jade Yang
Department of Computer Information Technology and Graphics
Purdue University - Calumet
Hammond, IN USA

Abstract— **Since the dawn of the big data era the search giant Google has been in the lead for meeting the challenge of the new era. Results from Google's big data projects in the past decade have inspired the development of many other big data technologies such as Apache Hadoop and NoSQL databases. The study article examines ten major milestone papers on big data management published by Google, from Google File system (GFS), MapReduce, Bigtable, Chubby, Percolator, Pregel, Dremel, to Megastore, Spanner and finally Omega. The purpose of the study article is to help provide a high-level understanding of the concepts behind many popular big data solutions and derive insights on building robust and scalable systems for handling big data.**

Keywords*— Big Data management; distributed systems; parallel processing*

I. Introduction

Internet technology has ushered us in the epoch of big data. Big data have been characterized by the four V's [5]: volume, variety, velocity and value, and they are too big, too fast and too hard for existing traditional database management tools [11]. Systems that store and manage big data should be highly scalable and provide high throughput and availability.

On the path of becoming the biggest search engine on earth, Google has pioneered the work in meeting big data challenges in the last decade or so. It is instrumental to study the technology used at Google since it has exerted a powerful influence on the type of data storage and management systems that global businesses are using and will be using in the future; Google technology has also inspired the creation of many popular big data storage and management tools in use today such as Apache Hadoop and NoSQL databases.

This article reviews a series of Google projects published between 2003 and 2013. In chronological order they are Google file system (GFS), MapReduce, Chubby, Bigtable, Percolator, Dremel, Pregel, Megastore, Spanner and Omega. The paper will examine each of these projects by providing a conceptual understanding of how each system works and by highlighting crucial details that characterize the resulting system. The paper concludes with some lessons gleaned from Google's experience in building systems suitable for the big data universe.

Big data storage and management systems can be classified into three levels from bottom up: (1) file systems (2) databases and (3) programming models [5]. As big data computing is performed under the framework of a distributed or clustered system, the file system, locking mechanism and scheduler are part of the foundation layer of the Google cluster environment. Their databases and programming models are then built on top of the foundation layer. The big data system layers are illustrated in Fig. 1.

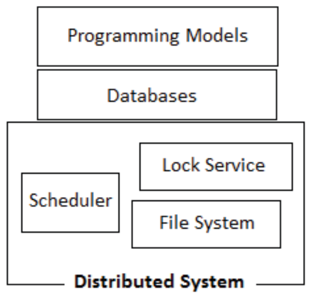

Fig. 1. Big data system layers.

The Google projects examined in this paper cover all three layers. We will review them in the order of the layers from bottom up: the first layer including Google File System, Chubby (lock manager) and Omega (scheduler) will be described in section II; the second layer includes BigTable, Megastore and Spanner in section III; the third layer includes MapReduce, Dremel, Percolator and Pregel in section IV.

II. System Environment

A. Google File System (2003)

The Google file system (GFS) is a distributed file system scalable for applications use large data set [9]. GFS divides a file into chunks and stores them with high redundancy on a cluster of commodity machines. The system is made up of three components:

- *Master* – GFS uses a single master to maintain file system metadata: namespace, mapping from each file to its list of chucks and chuck locations. The master performs system-wide activities such as chunk lease management for placing locks on chunks, garbage collection for freeing unused chunks, chunk migration for copying and moving chunks between chunk servers, and regular heartbeat messages for controlling chunk placement and monitor chunk server status.

For fast operation, metadata which include file and chunk namespaces, file name to chunk mappings and chunk locations are kept in master's memory. For fault tolerance, operation logs which contain a record of metadata changes are stored on the master's local disk. Also for simplicity and high fault tolerance, chunk location information is not stored on master's local disk but is collected from chunk servers at system startup.

- *Client* – Client code linked into each application implements the file system API, client-master interaction for metadata operations, client-chunk server interaction for data transfers. Client caches metadata, chunk location but not file data because of their huge size and for simplicity. If data to be written by the application is large or straddles a chunk boundary, client code breaks it down into multiple write operations.

- *Chunk servers* –A file is split into a set of chucks and each chuck is replicated multiple (three by default) times and stores on several servers. A chunk server stores 64 MB chunks as local files. A unique chunk handle is assigned by the master to each chunk. They report to the master which file chunks they have stored on their local disks via regular heartbeat messages.

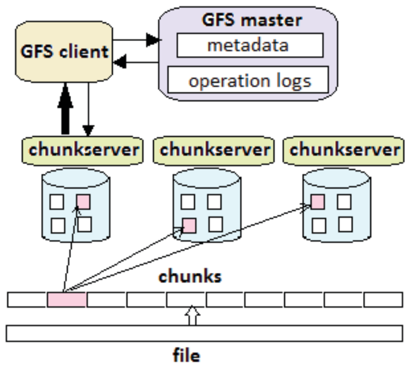

Fig. 2. GFS system architecture

The interaction among the components can be described as the following. For reading, 1) the client contacts the master to obtain the locations of the list of chuck servers that stores the file data, 2) then the client contacts any of the available chuck servers to read the chunk data. For writing, it involves the following sequence of actions:

1) The master grants a chunk lease to one replica called primary and forwards the information to the client. This is to reduce management overhead at the master.
2) The client stores the primary and other replica's location information then pushes data directly to all replicas.
3) The client sends a write request to the primary. The primary assigns the request a serial number to remember the request that is associated with the received data. Primary forwards the same request to other replicas.
4) To maintain consistency across all replicas, the primary picks an order (use the serial numbers) for all mutations to be made onto a chunk. The order is global, so is the same to all other replicas.

The overall characteristics of the system are highlighted as follows:

- Use larger fixed block sizes (64 MB chunks) as files are huge (usually in GB).
- Use append operations instead of overwrites due to the access pattern of most applications.
- Allows many clients append to the same file without locking so it does not guarantee that all replicas are identical but that the data is written at least once atomically.
- Allow for flexibility of system overall with relaxed consistency model and atomic append operation.
- Simplify the design and implementation of chunk placement and replication policies by using one centralized master.
- Separate control flow and data flow to fully utilize each machine's network bandwidth.
- Achieve fault tolerance by keeping master state small, fully replicating master state on other machines, and appending changes to a persistent file.
- Use "shadow" (read-only) masters for scalability and availability.

B. *Chubby(2006)*

In general a lock service synchronizes clients' activities and knowledge about their environment. Built on already well-established ideas and practices for distributed systems, Chubby is a product of an engineering effort in providing a lock service for Google's distributed systems [3]. For example, both GFS and Bigtable use Chubby to elect a master of the system, and Bigtable uses it to enable the master to discover the servers it controls and to allow clients to find the master. Google systems also use Chubby as a well-known location to store small amounts of metadata and as a popular name server.

Chubby is implemented as a distributed lock service library that maintains lock information in small files. Those small files are stored in a replicated database implemented on top of a log layer that runs the distributed consensus algorithm, Paxos. The purpose of Paxos algorithm is not to find the best conclusion but to reach a consensus in a distributed system.

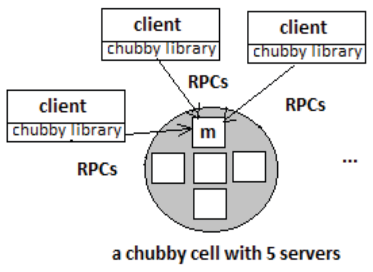

Fig. 3. Chubby system diagram

A Chubby cell which typically consists of a set of five servers can handle thousands of clients simultaneously. The multiple servers in the Chubby cell are replicas for providing fault tolerance. The replicas use the Paxos protocol to elect a master. Chubby clients contact the master for various operations on file logs. The relationship between a Chubby cell and a client is maintained through a session and the time interval during which the session exists is called the lease time. Chubby has the following characteristics:

- Coarse grained – advisory locks granted to files are held for hours or days instead of seconds or less; this way it is more server failure tolerant and saves unnecessarily more frequent communications.
- Consistent caching – lock service deals with small files only thus use client side caching reduces Chubby server load so that the system can maintain its scalability.
- Grantees safety – only one value is chosen and the chosen value is a proposed one. This property allows to create application without worrying that any lock holder would die with the lock.
- Swift name updates – the system can keep a client alive with a keepAlive message and cover all name caches the client is using.

C. *Omega(2013)*

As Google's "next generation cluster management system" Omega is a parallel scheduler architecture built around shared state using lock-free optimistic concurrency control [16]. It is designed to support a wide range of policies for a heterogeneous mix of workloads. A cluster's workload can be roughly categorized as two types: service jobs and batch jobs. Service jobs are long running jobs, whereas batch jobs are short computations and require fast turnaround.

Other alternative cluster scheduling models are monolithic and two-level. Monolithic architecture uses only one scheduler for both types of jobs. It suffers from the "head of line blocking" problem due to the lack of parallelism, therefore it is hard to scale. Although resources are distributed among multiple schedulers with two-level schedulers, they offer pessimistic concurrency control that limits parallelism. For example, in Mesos each scheduler can only see available resources offered by a centralized resource allocator. Only one scheduler examines an available resource at a time. This is equivalent to holding a lock on that resource for the duration of the scheduling decision time, which may lead to deadlock.

Omega uses a shared state approach. Each scheduler has full access to the entire cluster and is allowed to compete for resources. Optimistic concurrency control is used to mediate clashes when they update the cluster state. This increases the parallelism and visibility of resources but incurs a potential cost of redoing the work when the optimistic concurrency assumptions are incorrect. Through simulation results presented in the paper, however, such a cost seems to be justified.

The Omega system maintains a master copy of the resource allocations in the cluster called cell state. Each scheduler has a local copy of the cell state that it uses for job scheduling. The scheduler can see and claim any cluster resources even for those that have been acquired by other schedulers, as long as it has the permission and priority to do so. When the scheduler makes a decision, it submits a transaction to update the master copy of the cell state. Whether the transaction succeeds or fails (in case of a scheduling conflict), the scheduler re-syncs its local copy of cell state with the master copy and tries again if necessary.

Performance is evaluated by running simulations with lightweight simulators and comparing simulation results on all three models: monolithic, two-level, and Omega. The paper shows that Omega architecture scales the best, can tolerate long decision times and is free from the "head of line blocking" issue. Therefore, all this performance gain outweighs the extra cost of redoing resulted from optimistic concurrency control.

III. DATABASES

A. *Bigtable(2006)*

Bigtable is a distributed storage system for large amounts of structured data. It is similar to a database but has a different interface and does not support a full relational data model. It is a system that maps the combination of a row key, a column key, and a timestamp to an un-interpreted string [4].

The Bigtable maintains data in lexicographic order by row keys. A table is dynamically partitioned into a set of tablets. Each tablet corresponds to a range of rows. It serves as the unit of distribution for load balancing. The partition also facilitates data access as it provides locality. For example, if web pages are stored in the order of their domain names, pages from the same domain will likely be found in the same tablet or nearby tablets.

A column key is formed by using the *family : qualifier* structure. Column families are created as units for access control. Timestamps are used to store the most recent several fetched versions of the same page.

Bigtable can be implemented with GFS framework: clients with the library code linked into each, a single master that assigns tablets to tablet servers and a cluster of tablet servers that handles read and write operations for the tablets they store.

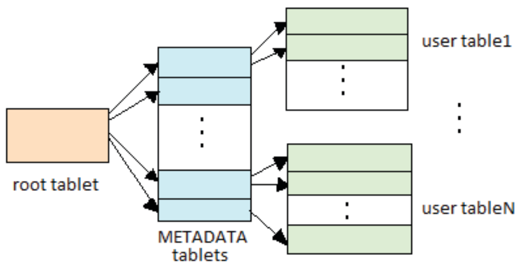

Fig. 4. Bigtable tablet location information

Tablet location information is stored in a three level hierarchical B+ tree like structure as shown in Figure 4. They are from top to bottom: Root tablet, METADATA tablets, and User tablets. The root tablet contains the locations of the tablets in the METADATA table and each METADATA tablet stores the locations of a set of tablets from user tables. In turn the root tablet location is stored in Chubby. This three level hierarchy has the capacity to address 2^{34} tablets with 128 MB tablet size. This tablet location information is cached in the client library.

Newer updates to a tablet are stored in a so called *memtable* in the tablet server's memory; older updates are stored in SSTables (Sorted String tables) in GFS as persist records. An SSTable consists of a set of redo points associated with the tablet. To recover a tablet, a tablet server first reads its metadata from the METADATA table then locates the corresponding SSTable that contains all the committed updates (redo points) to reconstruct the *memtable* for the lost tablet. Some highlight features in Bigtable are:

- Bigtable data are internally stored as SSTables which provide an ordered and persistent map of key-value pairs. Both keys and values can be arbitrary byte strings.
- Bigtable relies heavily on Chubby the lock service for a variety of tasks. Besides for discovering live tablet servers and storing Bigtable schema, tablet assignment is done by the master who scans the server directory in Chubby to find live servers. So it is of vital importance to have a highly available and reliable Chubby system.
- Bigtable client caches tablet locations. As a result most clients never communicate with the master. Therefore, the master is very lightly loaded and the system is more scalable.
- The client communicates directly with tablet servers for reads and writes as in GFS.

B. *Megastore(2011)*

Online services (such as Gmail, Google+, Google Play) have been a major source of big data. These online services require low latency, high availability and scalability, data consistency. Megastore has been created as a storage system to meet the requirements. Built upon the Bigtable, it blends the scalability of NoSQL with the convenience of traditional RDBMS [1].

Megastore scalability is achieved through data partitioning. Under the assumption that the data for most internet services can be suitably partitioned, Megastore partitions data store and replicates each partition separately then provides full ACID semantics within partitions. Each partition is stored in a NoSQL data store.

Megastore achieves data consistency, high availably and low latency through synchronous data replication across distant data centers. The heart of the replication scheme lies in a low latency implementation of Paxos. Systems with dedicated master usually have limited flexibility as reads and writes must be done near the master, thus create uneven workloads between the master and slaves. Master failover is also user visible. The Paxos algorithm used by Megastore does not require a distinguished master thus has reduced latency caused by a machine (master) failover. Each data center stores a replica of data in a Bigtable. Multiple replicated logs are also created over a wide geographic area, each governing its own partition of the dataset.

In addition, Megastore achieves fast reads with coordinator servers that uses Chubby lock service to detect network partitions and node availability; fast writes are performed with "leaders" at different log positions instead of dedicated masters.

In summary, Megastore uses data partition to achieve scalability, synchronous replication for high availability and data consistency through the novel use of Paxos algorithm to replicate primary user data across data centers on every write.

C. *Spanner(2012)*

Spanner has been introduced as the latest globally distributed database system [6] that has improved performance on writes, a more normal SQL like query interface, and a tighter consistency than Megastore. The most distinctive feature of Spanner is that it assigns globally meaningful timestamps to transactions. The key enabler for doing so is the implementation of a new True Time API that exposes clock uncertainty. If the uncertainty is large, Spanner slows down to wait out the uncertainty. Google's implementation keeps uncertainty small by using multiple modern clock references such as GPS and atomic clocks at each data center.

Spanner is organized as a set of zones. Each zone has a zone master and a set of spanservers, which is analogous of the deployment of Bigtable servers. Replication and distributed transactions are layered onto the Bigtable-based implementation.

A Spanserver is responsible for between 100 and 1000 instances of a data structure, similar to a tablet in Bigtable.

Whereas a Bigtable tablet is a contiguous partition of the row space in lexicographical order, a Spanner tablet may contain multiple partitions of the row space, so that it is possible to collocate multiple directories, or partitions that are frequently accessed together. Unlike Bigtable, Spanner assigns timestamps to data instead of a row in a Bigtable. So Spanner is more of a multi-version database than a key-value store.

Each spanserver implements a Paxos state machine on top of each tablet which stores the metadata and log for the corresponding tablet. Each spanserver uses a lock table for concurrency control and a transaction manager for distributed transactions. In essence Spanner is a database that shards data across many sets of Paxos state machines in data centers all over the world.

TrueTime implementation has enabled Spanner to guarantee that a whole database audit read at a timestamp *t* will see exactly the effects of every transaction that has committed as of *t*. This is achieved through the monotonicity invariant and disjoint invariant of timestamps. Spanner uses commit wait to ensure there is no overlap of read or write event time intervals.

In conclusion, Spanner achieves a tight global data consistency by using commit timestamps with the underlying Paxos algorithm.

IV. Programming Models

A. *MapReduce (2004)*

MapReduce is a programming model for processing and generating large data sets [7] [8]. This programming paradigm is designed for when computations involved are simple but input is large and needs to be distributed over hundreds or thousands of machines to complete in a reasonable amount of time. The MapReduce workflow (as illustrated in figure 5) is outlined as the following:

1) MapReduce splits the input into M pieces or chunks of data then starts up many copies of the program on a cluster of machines.

2) One of the copies is the master that assigns M map tasks and R reduce tasks to worker machines. It is expected that M and R are much larger than the numer of workers. The workers can run in parallel to process data chunks.

3) For a map task, a worker machine tranforms its input into the intermediate key/value pairs and stores them on the local disks. The key/value pairs are partitioned into R regions by the partitioning function. The locations of the key/value pairs are passed back to the master who forwards the information to the reduce workers.

4) A reduce worker remotely reads intermediate key/value pairs on disks of map workers, sorts and groups them so there is one value for each unique key. The result is appended to a final output file for this reduce partition.

5) After completion, there are R output files.

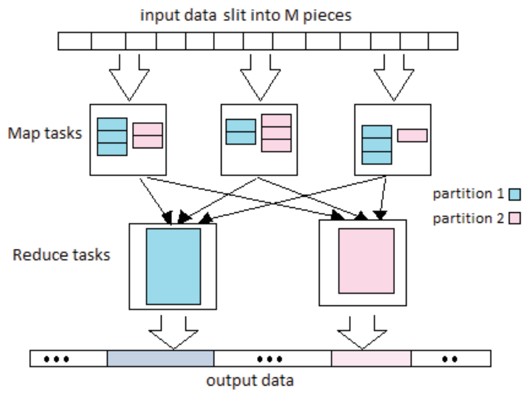

Fig. 5. MapReduce workflow

Main features of MapReduce are:

- Fault tolerance is implemented through re-executing work done by the failed worker node.
- Read and write operations are pushed into processing units that are close to local disks. This reduces the amount of data sent across the network.
- It partitions the problem into a large number of fine grained tasks which are dynamically scheduled on available workers so that faster workers process more tasks.
- While parallel database computing is another tool for processing large sets of data, the following table highlights some differences between parallel database programming and MapReduce [10].

	Data	Query interface	Job granularity	Strength
Parallel Database	structured	SQL	entire query	performance
MapReduce	unstructured	Java, C++, scripts	data storage block size	scalability

TABLE I. PARALLEL DB VS MAPREDUCE

B. *Dremel(2010)*

Dremel is a scalable interactive query system for read-only non-relational or unstructured data [14]. Often data used in web documents and scientific computing is non-relational. Dremel is not a replacement but a complement of MapReduce designed to be more efficient for querying data with nested representation.

Interactive query processing requires interoperation between the query processor and other data management tools. It involves a high performance common storage layer such as GFS and an interoperable data management tool, a shared storage format. Dremel uses an algorithm to preserve structural information of nested data and to reconstruct records from any subset of fields of a data structure. The goal is to store all values of the fields along the path of the record structure (from the top node down to the leaf node) contiguously in order to achieve retrieval efficiency.

The algorithm assigns each field value a repetition and a definition level so that two values with the same field name can be distinguished. Because Google datasets are usually sparse, it is common to see many missing fields in the data records. As a result, to split records into column strips efficiently means to process missing fields as cheaply as possible. Dremel uses a tree of field writers whose structure matches field hierarchy in the schema. A child synchronizes to its parent's levels only when a new value is added so that the parent's state does not propagate down to the child unnecessarily.

To reconstruct records from a given set of fields, Dremel uses a finite state machine (FSM) that reads the field values and levels for each field and appends the values sequentially to the output records. A FSM state corresponds to a field reader for each selected field and state transitions are labeled with repetition levels. Once a reader fetches a value, it looks at the next repetition level to decide what next reader to use.

In summary, the key feature of Dremel is that it uses a column striped storage representation which saves the cost of reading more data from secondary storage, and in turn less data compression and higher efficiency.

Whereas the traditional RDBMS stores and manages data as rows in a table (row store), column-oriented storage system (column store) as used in Dremel stores data as a set of columns. Depending on the application, each type of storage has its strengths and weaknesses. In many cases, they are complementary of each other [10]. This is summarized in the following table:

	Column store	Row store
Pro	good for reading data; good data compression rate; efficient for range queries	good for writing data; efficient for random access as indices are at record level
Con	multiple column access to change a record causing extra disk I/O	read unnecessary columns causing extra disk I/O; low compression rate

TABLE II. COLUMN STORE VS ROW STORE

C. Percolator(2010)

It is prohibitive to recreate the search index on big data when there is an update, such as maintaining a web search index. Percolator is a system for updating a large data set by transforming the large set of data via small independent changes incrementally [15].

Percolator achieves incremental transformation by setting up as a series of observers; observers are user defined pieces of code that are invoked by the system whenever a table column changes. Each observer completes a task and creates more work for downstream observers by writing to a table.

A percolator system consists of three binaries that run on every machine in a cluster: a Percolator worker, a Bigtable tablet server and a GFS chunk server. All observers are linked into the percolator worker. They scan the Bigtable for changed columns upon which notifications are created and trickled "downstream." Notifications trigger transactions. Transactions are performed by sending read/write RPCs to Bigtable tablet servers, which in turn send read/write RPCs to GFS chunk servers. Percolator uses Chubby again to determine the aliveness of a transaction.

Because many threads run concurrently on many machines to transform the data set, Percolator keeps track of the state of the repository through the use of ACID compliant transactions and keeps track of the state of incremental computation through the use of observers.

In summary, by updating the system incrementally, Percolator drastically reduces average latency from processing changes and new documents as they are crawled. Built on Bigtable, Percolator adds locks and transactions on rows and tables. It uses notifications whenever there is an update in the tables. The notification messages trigger a sequence of transactions, thus percolating the update through the database.

D. Pregel(2010)

Pregel provides a distributed programming framework for processing large web graphs, those with billions of vertices and trillions of edges [12]. Pregel computation consists of a sequence of "supersteps." During each superstep a user defined function is invoked for each vertex. The same function runs for all vertices in parallel. The function defines the behavior at a single vertex V, reads messages send to V from the previous superstep and sends messages to other vertices that will be received at the next superstep. The function can also modify the state of V and its outgoing edges.

Pregel chooses a pure message passing model over other alternatives because it is not only sufficient for existing graph algorithms but also simpler and gives a better performance than for example, passing the entire state of the graph from one state to the next as a chain of MapReduce calls.

Pregel implementation involves dividing a graph into partitions where each partition consists of a set of vertices and their outgoing edges. Copies of the user programs are deployed on a cluster of machines with one of them being the master and the others as workers. The master determines which partition will be assigned to a worker machine and instructs each worker to perform a superstep. The master maintains the information for the list of workers information. It is noteworthy that the size of the data structures is proportional to the number of partitions, not the number of vertices or edges of the graph. This way, a master can coordinate computation for very large graphs.

A worker machine maintains the state of its portion of the graph in memory. A state of a vertex consists of its current value, outgoing edges, an incoming message queue and a flag showing if the vertex is active. During each superstep, the worker loops through all its active vertices. Computations continue until all nodes have voted to halt.

Pregel achieves its scalability through the "vertex-centric" approach, similar to MapReduce where computation for a very large dataset is performed by a collection of independent locally processed jobs.

V. Conclusion

Since the birth of Google File System more than a decade ago, a multitude of applications has been created relying on this framework. Despite of a loose consistency and the potential bottleneck caused by a single master design [13], GFS has accommodated the needs of many big data projects and many projects have been built around GFS. MapReduce and Bigtable are the other two projects that have laid the foundation for most existing big data solutions. It is also inspiring to see how Google was going against conventional wisdom when they implemented transactions on NoSQL databases in Percolator and achieved global consistency on distributed systems with Spanner [2].

Through these projects, Google has demonstrated how to create a global, reliable and scalable system with a cluster of inexpensive commodity computers where hardware failures are normal. First of all, high fault tolerance is an integral part of all Google projects. For example, Megastore binds the machines into a unified ensemble offering greater performance and reliability through replication and partition. In all these projects, one common goal is to reduce network traffic as much as possible in order to optimize performance. One way to achieve such a goal is to bring computations close to data, as was done in GFS and MapReduce. Another way is to avoid unnecessary communications as in Chubby that has implemented coarse grained locking. Another common theme from the series of Google project experiences is that a simple design is more favorable when dealing with large amount of data.

As all Google projects are designed and implemented on faulty hardware, they have proved that it is possible to build powerful although not perfect big data processing systems out of ordinary machines.

References

[1] Jason Baker et al. “Megastore: Providing Scalable, Highly Available Storage for Interactive Services”. Proc. of CIDR. 2011, pp. 223–234.

[2] M. Braun, “Big data beyond MapReduce: Google’s Big Data papers” [blog post]. Retrieved from http://blog.mikiobraun.de/2013/02/big-data-beyond-map-reduce-googles-papers.html.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In OSDI ’06: Proceedings of the 7th symposium on Operating systems design and implementation, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[4] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. ACM TOCS 26.2 (June 2008), 4:1–4:26.

[5] M.Chen, S.Mao,and Y. Liu, “Big Data: a Survey”, Mobile Newtorks and Applications, April 2014, vol. 19, issue 2, pp171-209.

[6] James Corbett et al. “Spanner:Google’s Globally-Distributed database”. Proc. Of OSDI 2012.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, 2004.

[8] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: a flexible data processing tool”. Commun. ACM 53.1 (Jan. 2010), pp. 72–77.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In SOSP, 2003.

[10] P. Kieun, “Database Technology for Large Scale Data” [blog post]. Retrieved from http://www.cubrid.org/blog/dev-platform/database-technology-for-large-scale-data/

[11] Sam Madden, “From Databases to Big Data”, Internect Computing, IEEE Computer Society, 2012.

[12] Grzegorz Malewicz et al. “Pregel: A system for large-scale graph processing”. SIGMOD’10, ACM, June 2010.

[13] K. McKusick, S Quinlan, “GFS: evolution on fast-forward.” ACM Queue 7(7):10, 2009.

[14] Sergey Melnik et al. “Dremel: interactive analysis of web-scale datasets”. Proc of the 36th Int’l Conf on VLDB 2010, pp330-339.

[15] Daniel Peng and Frank Dabek. “Large-scale incremental processingusing distributed transactions and notifications”. Proc. of OSDI. 2010, pp. 1–15.

[16] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek and J. Wilkes. “Omega: flexible, scalable schedulers for large compute clusters”. SIGOPS European Conference on Computer Systems (EuroSys), ACM, 2013,pp351-364.