**Google File Systems**

**Introduction:**

Google’s data processing requirements has led the company to work on a new design process, a new design of file system to meet their demands. They met with the demands like component failures, files with large size, scalable systems and a better Application Interface to work with the file systems. The result is Google file systems.

GFS runs on commodity hardware, that offers horizontal scaling, a novel File system API more convenient to work with, considering fault tolerance systems as a matter of high importance with continual monitoring of the processes. The file storage is done by splitting file into chunks of 64 MB and storing in the file systems.

**Design Overview:**

**Interface:** GFS uses a familiar file system interface and has snapshot, record append features in it.

**Snapshot** -creates a copy of file/directory at low cost. **Record append**- allows multiple clients to append data of each individual client simultaneously maintaining the atomicity as well.

**Architecture:**

**Single Master:** Uses only one master. Client makes a request with the master to know the file location. Master replies with the chunk name and locations. Master does not interact with chunk fileserver for client requests (Data requests).

**Chunk Size:** A 64 MB is the maximum file size of chunks in chunk file server. Huge files are split into small chunks for better accessibility and storage purposes. When multiple clients access a particular chunk, hotspots are created on the chunk. A better improvement is to ask clients contact other clients who are currently using for the chunk.

**Metadata:** Master stores meta data information namely file, chunk namespace, location of each chunk’s replica, mapping from files to chunks.

**In Memory Data Structures:** master stores data in memory and so re replication in the presence of chunk server failures, chunk migration to balance load, disk space usage across chunk servers could be monitored periodically.

**Chunk Locations:** Master uses heart beat messages to verify the status of chunk server and manage the chunk locations efficiently.

**Operation Log:** Provides details about the status, checkpoint, location of chunks at a certain point of execution. This is really helpful in times of failover wherein, we can locate the checkpoint and find the chunk involved with the issue.

**Consistency:** Guarantee by GFS: Master uses namespace locking to achieve atomicity and correctness of information written to the file, during file mutations.

**System Interactions:**

**Leases and Mutation Order:** File modification is called as mutation and it is achieved uniformly across all the replicas using leases. Lease is a 60 second time out state that reduces operational overhead of master as well as provides time for the replicas to update the mutation.

**Data Flow:** Data flows from client to primary and then to other secondaries. This helps utilize full network bandwidth and avoid network bottleneck. A linear distribution helps achieve this unlike a tree.

Atomic Record Appends and snapshot are some of the other system interactions involved in GFS.

**Master Operation:**

Using Namespace management and locking: GFS uses namespace as a lookup table mapping full pathnames to metadata. This helps achieve atomicity and locking.

**Replica Placement:** Chunk servers are distributed in the server racks and so communication between chunks may have latency due to network. In order to avoid this, a replica of the chunk is placed closer to the server. Write traffic is a tradeoff for this scenario, though we achieve high availability and reliability.

Creating, re-replicating, rebalancing: Chunk replicas serve the following purposes efficiently,

1. To place replicas on chunk servers with below average utilizations
2. Limit the number of recent creations on each chunk server.
3. Latency due to network traffic

**Fault Tolerance and diagnosis:**

**High Availability:** As chunk replication is carried out, failover of chunk servers, does not impact the process that run across systems at any point of time.

**Master replication**: Master is replicated for reliability. Its operation logs are replicated to several machines. Shadow masters could be helpful in providing read only access, when a master in GFS fails over.

**Data Integrity:** Checksum operation is carried out to verify the integrity of data in GFS. Thus data corruption and disk failures are identified from data issues.

**Diagnostic Tools:** RPC logs include exact details about the request and responses sent over wire. These logs are written sequentially and asynchronously.

**Related Works:** GFS closely resembles NASD (Network attached Storage Drives).

**Conclusion:** GFS meets the needs of Google as a storage platform, providing a high reliability over fault tolerance, scalability and data storage.

**Map Reduce: Simplified Data processing on large clusters**

**Introduction:**

A Programming model to process large datasets collected from a varied set of sources such as crawled documents, web request logs and so on at a quick pace, is Map Reduce. This involves two major operations namely Map, and Reduce steps where data is split into intermediate Key /Value pair and manipulated and regrouped to write into a single final output file.

Advantages include fault tolerance, load balancing and data distribution that helps process data at rapid pace.

**Programming Model:**

**Input file:** A file/document is input as a value to the map function and it generates intermediate key/value pairs. (Object in terms of any OOPs language).

**Map:** Involves split the input file into sets of key/value pairs. (Typical map function works on more sets than a reduce and vice-versa). (Function in terms of any OOPs language).

**Reduce:** Receives the intermediate key values generated by the map function and reduces the values by manipulating the values in the pair. (Function in terms of any OOPs language).

This can be written as below: map (k1,v1) → list(k2,v2) reduce (k2,list(v2)) → list(v2)

**Examples:** Word count in a document, Count URL visits, Inverted index (tracks position of words in a documents and helps in auto-correct) etc.

**Implementation:** Different implementation types are small shared memory machine, NUMA multi-processor, that depends on the required environment.

Distributed systems at Google are commodity hardware, interconnected to perform parallel operation on huge datasets like queries ran per day etc.

**Execution Overview:** The Map function is invoked on the distributed machines by automatically splitting the file into M pieces. These are parallel processed by the machines and the intermediate keys are generated.

The intermediate keys are passed over to a partitioning function and that gives the R splits to be parallel processed by Reduce function.

1. **Process:**  The input file is split into M pieces of size ranging from 16MB to 64 MB. This is copied to other clusters to create a replica.
2. One of the copies of program present in the cluster, is special- the master and the rest are workers. The master picks up idle workers (other machines in the cluster) and assigns each a map tasks and a reduce task.
3. The worker picks up its corresponding split file and passes the file into map function. The intermediate Key/Value pairs are buffered om memory.
4. At regular intervals, these values are written into a local disk with R partitions created by the partitioning function. These locations are passed to master, who shares it with workers to perform reduce tasks.
5. Through remote procedure calls, the workers fetch data from local disks, once instructed by master. Workers passes the data into reduce function, generates intermediate keys that are later sorted and for new intermediate key, the value is passed to reduce function.
6. This output from reduce function is appended to final output file in the worker to mitigate disk partitioning.
7. After completion of MapReduce program, master returns to user code. Now the workers together have R part files that are written to a single file and passed on to another mapreduce function.

**Master Data Structures:** Master keeps track of state (idle, in-progress, completed) and identity of each workers performing a map and reduce tasks separately. For each completed map tasks, the location of files is shared with master, so that workers with in-progress reduce tasks are updated on the next files.

**Worker Failure:** Master pings workers at regular intervals to keep track of the state. If a worker failure happens, the state of worker is changed to idle and the work is re-executed by a different worker. All worker that perform reduce tasks are notified on the status as re-execution, thus resilient to large scale failovers.

**Master Failure:** Master writes to the checkpoints of the operation using master data structures. Even if a master fails, the operation can be started from the last completed checkpoint, however, since there is only one master in this model, the programmer need to abort and restart the operation from the latest checkpoint.

**Semantics in the presence of failures:**

Atomic commits are used to derive deterministic values from the mapreduce execution.

If same reduce task is executed on multiple workers, multiple rename calls will be posted on final output file

**Locality:**  Mapreduce master contains the location information of all the files bits present across all nodes. By assigning the worker the location of reducer file, I/O bandwidth is save and so improves performance.

**Task Granularity:**  More maps than reduces, helps achieve task granularity.

**Backup Tasks:** Straggler is the delay that is caused due to I/O, CPU performance, multiple tasks etc. on a machine that slows the map reduce operation. Bugs in machine initialization code that has disabled cache, caused a difference time about a factor of 100. Running back up tasks marks a task as complete if the primary or back up task completes. This method saves about 44% of total run time.

**Refinements:** Some improvements to the code such as including a partitioning function, combing function, order guarantees help speed up the process significantly. partitioning function involves Hashing the input to reduce tasks, combining function is about doing reduce operations even before passing the values over the reducer. Local execution is nothing but creating a sequential execution of all mapreduce operations on a single local machine. This helps in debugging, small scale testing and profiling of data. The master generates the status of all the operations and shares it via a HTTPS server, which is helpful in understanding the failure tasks and workers affected.

**Experience:** Applications of MapReduce at Google involved computation of large scale machine learning problems, largescale graph computations.

**Related Work:** Sorting facility, part of MapReduce library is similar to NOW-sort. Re-execution system is similar to TACC.

**Bigtable: A Distributed Storage System for Structured Data**

**Introduction:** Bigtable is a distributed storage system designed to store structured data. Bigtable does not support a relational model but uses a data model. Data is indexed using row and column names that can be arbitrary strings.

**Data Model:** A sparsely distributed, persistent sorted multi-dimensional map, indexed by a row key, column key and timestamp. Each value in the map is uninterpreted array of bytes.

**Rows:** Follow a lexicographic order by row key, which are arbitrary strings. The row range in bigtable is dynamically partitioned. Each row range is called as tablets and is the reason for load balancing.

**Column Families**: Column keys are stored as families. This helps in the access control, disk as well as memory accounting.

**Timestamps:** 64 bit integers that help in picking up a unique record in a bigtable.

**API:** Big table API provides functions for creating and deleting tables and column families. Allows cells to be used as integer counters, permits single row transactions that helps perform read-write-modify transactions on any single row key.

**Building blocks:** Google SSTable file format is used to store bigtable data. Bigtable relies on highly available distributed lock service, chubby. Bigtable uses chubby for variety of tasks like, to ensure at most one master is active at any point of time, to store bigtable schema information, bootstrap the location of bigtable data.

**Implementation:** Bigtable components are Client library, master and tablet server.

Master takes care of assigning tablets to tablet servers, keeping track of tablet servers that are active. Tablet server takes care of splitting tablets that have grown large and read/writes that happen on the tablet.

**Tablet Location:** There is a three level hierarchy in storage of tablets. First is a storage file called root tablet present in chubby. This file has Metadata rows about all the tablets. It contains the location of row key, which is an encoding of tablet’s table identifier in end row.

Client library caches tablet locations. If it does not have the location/ store incorrect, it moves up the tablet location hierarchy recursively. Secondary information such as server start up logs and operation logs are also stored in the METADATA table to look for help in debugging and analytics.

**Tablet Assignments**: The Master tablet uses chubby to place a lock on one of the cells. The master tablet later scans for available cells, assign operations and keeps track of status of the operations by checking with the tablet servers. If any of the tablet server is dead, the master reassigns the operation and ensures availability of data is maintained without any trouble.

**Compactions:** As write happens, size of memTable increases. After a limit, the table size is limited and a new SSTable is created. A merging compaction that rewrites all SSTables into a single SSTable is called as major compaction. These major compactions allow bigTable to reclaim all the resources used by deleted data. The deleted data is removed from the system at regular intervals, which is crucial when we store sensitive data.

**Refinements:** Compression, caching for read performance, locality groups, bloom filters, commit log implementation, speeding up tablet recover are some of the refinements possible with Bigtable.

**Related Works:** C-store and bigtable share many characteristics in common; shared nothing architecture

**Chubby Lock Service for loosely coupled Distributed Systems**

**Introduction:**

Chubby is a lock service that allows clients to synchronize to their activities and stay informed about the basic environment. Reliability and availability are the key areas focused in this service. It’s client interface is similar to basic file system that performs whole read/write with advisory locks, sending notification for file modification. Google’s BigTable is most benefited out of the chubby service as availability of systems were managed efficiently.

**Design**:

* **Rationale:**

1. Choosing a master using a lock service is simple and easy to do when compared to the conventional way in which we use a consensus protocol, particularly solving compatibility issues.
2. Chubby could be used in place of a name server for client to server data read/write. Doing this saves client caching, client dependency on servers as well as a need for choosing a cache timeout (DNS).
3. Programmers are familiar with lock-based interface and so this could be used on distributed systems for building a reliable decision making in distributed system.

* **System Structure:** Chubby uses a set of servers called as chubby cell and a chubby library that mediates between clients and chubby cell. The servers in chubby cell elect a master unanimously for a period of time called a master lease. They carry on operations performed as directed by master. If one of the replicas fail (in the chubby cell), the master updates its table with an active replica to serve information.
* **Files, Directories and handles:** File system interface of chubby is similar but simpler to UNIX.
* This file system interface is used as APIs in other file systems like GFS.
* **Locks and Sequencers:**  Chubby uses advisory locks. Clients can access the file in read/ write mode. If more than one client is accessing the file for write mode, a lock is held preventing the write access. Introducing a sequence numbers is more complex in a distributed system and so chubby uses a set of sequence numbers for those interactions that make use of locks.
* **Events:** Chubby clients could be notified on event changes, if they subscribe for events. The chubby library takes care of sending an up-call to its clients after the action is performed
* **Caching:** To mitigate traffic, chubby clients cache file data, node meta data. When a change has to be performed, the master issues a “invalidate cache” call to all its replicas. After the call is acknowledge, the master proceeds to modify the data.
* **Sessions and KeepAlives:** Chubby clients establish a session; the first time they make contact with the cell. This is kept valid by checking the session status using calls sent to the clients, called as KeepAlives. KeepAlives are also used for passing events and cache invalidations, that makes clients mandatory to acknowledge cache invalidations to refresh session lease.
* **Failover:** When a master fails, the in-memory state of all session, handles and locks are lost. Once a new master is elected, clients contact the new master for information.
* **Backups:** For every few hours, the master provides a snapshot of database to a GFS file server, located at a different building. This ensures impact prevention of building damage/ location related issues.
* **Database Implementation:** Initial version of chubby used Berkeley DB.

**Mechanism of Scaling:**  Chubby uses a number of proxies and partitioning to handle huge number of clients

**Lessons Learned:** Developers rarely consider availability as a concern; The Single fail over has consequence of about hundred times the conventional one both in time as well as number of machines affected. However, observed availability of for a given client is less than that for a client’s local chubby cell. Hence, clients hardly feel a difference between service outage and availability.

Fine grain locking was not a necessity, as avoiding the unnecessary communication that happens in the chubby cells through implementing the library provides a coarse grain locking.

**Related Works:** Chubby is built with ideas from different resources. The concept of sessions and tokens are derived from Echo, VMS uses lock service system; Similar to VMS caching model is chubby’s API, that uses a file system like name space as more convenient to just files, is based on file system model.