DATA ENGINEERING

Assignment 1
Feb 20, 2022

Table of Contents

Suganya Arumugam Manickam

The Google File System

The Google File System (GFS), a scalable distributed file system for large distributed data-intensive applications, mainly designed to handle rapidly growing large datasets being accessed by number of clients in parallel. Multiple GFS clusters are currently deployed for different purposes and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

Features: Reliability, Increased Fault tolerance, Scalability, Availability, Clustered data storage, Diskspace utilization, Diagnostic Tools, optimized Performance by using Atomic append operation.

Components:GFSCluster consists of Single Master, multiple chunkservers, multiple client, Chunkdata

Pros: can run in lower reliability Linux machine with acceptable application code.

Cons: Hotspot was developed when GFS handling requests sequential to simultaneous. As interim solution, suggested to have more file replicas & delay application start times. Potential long-term solution is in place which will be allowing clients to read data from other clients.

Workflow:

Master has all metadata of file system such as namespace, access control information, mapping from files to chunks, current location of chunks. Master executes all namespace operations, manages chunk replicas by creating new chunk, balance load across all chunkservers, garbage collection and chunk lease management, snapshot operations. Master also periodically sends Heartbeat messages and checks chunkserver state and take action accordingly to the state, like replication, deletion.

Client creates chunk index and sends master both filename & index. Master replies with corresponding chunk handle & locations of replicas. Client sends read/write requests to nearest replica. In case of write request, nearest replica in turn sends request to other replicas for write operation. Client then, contacts primary replica to acknowledge the completion write process. Primary replica asks other replicas to acknowledge if the mutation is completed. Other replicas reply to primary, which in turn takes the control to client. On failure, client retries this mutation for failed chunkservers.

Existing Issues and GFS Approaches to overcome:

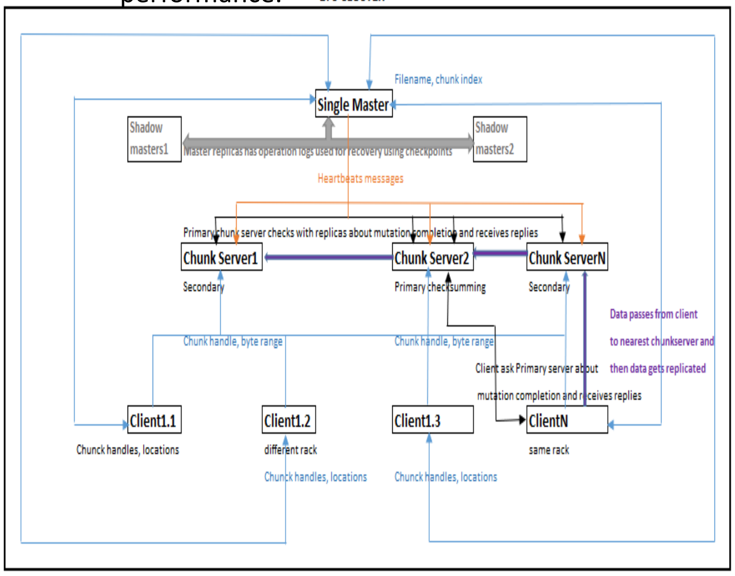
1. Component failures and constant monitoring, error detection, automatic recovery & replication must be integral to the system.

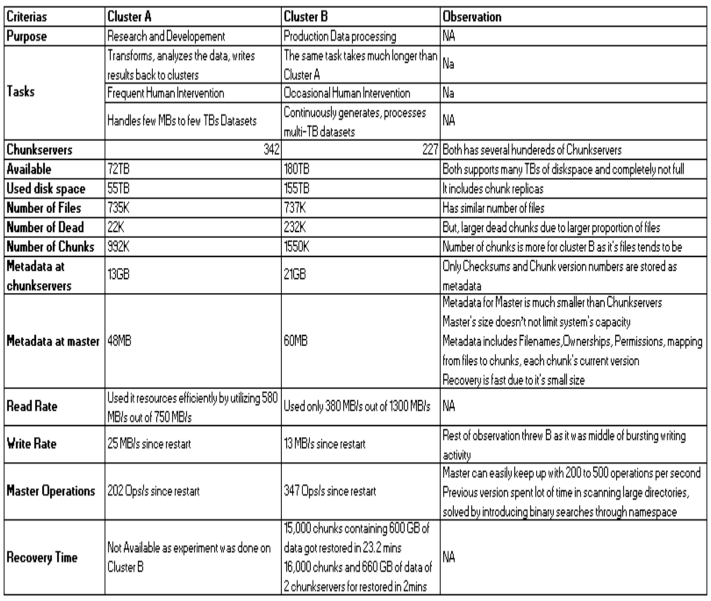
2. Data with different format & size to be handled: Design must be equipped to handle both large and small workloads. Design, Parameters and Block sizes must be revisited and so on.

A Real Life Use Case: Two real clusters at Google in use and its data handling

Google completely controls both GFS and its applications. Cluster A is used for research and development by hundreds of engineers and, Cluster B is used for production data processing and performance.

GFS CLUSTER


Filename, chunk index
Single Master
Shadow masters1
Master replicas has operation logs used for recovery using checkpoints
Shadow masters2
Heartbeats messages
Primary chunk server checks with replicas about mutation completion and receives replies
Chunk Server1
Chunk Server2
Chunk ServerN
Secondary
Primary checksumming
Secondary
Data passes from client to nearest chunkserver and then data gets replicated
Chunk handle, byte range
Chunk handle, byte range
Client ask Primary server about mutation completion and receives replies
Client1.1
Client1.2
Client1.3
ClientN
Chunck handles, locations
different rack
same rack
Chunck handles, locations
Chunck handles, locations



Criterias	Cluster A	Cluster B	Observation
Purpose	Research and Developement	Production Data processing	NA
Tasks	Transforms, analyzes the data, writes results back to clusters	The same task takes much longer than Cluster A	Na
	Frequent Human Intervention	Occasional Human Intervention	Na
	Handles few MBs to few TBs Datasets	Continuously generates, processes multi-TB datasets	NA
Chunkservers	342	227	Both has several hundereds of Chunkservers
Available	72TB	180TB	Both supports many TBs of diskspace and completely not full
Used disk space	55TB	155TB	It includes chunk replicas
Number of Files	735K	737K	Has similar number of files
Number of Dead	22K	232K	But, larger dead chunks due to larger proportion of files
Number of Chunks	992K	1550K	Number of chunks is more for cluster B as it's files tends to be
Metadata at chunkservers	13GB	21GB	Only Checksums and Chunk version numbers are stored as metadata
Metadata at master	48MB	60MB	Metadata for Master is much smaller than Chunkservers Master's size doesn't not limit system's capacity Metadata includes Filenames,Ownerships, Permissions, mapping from files to chunks, each chunk's current version Recovery is fast due to it's small size
Read Rate	Used it resources efficiently by utilizing 580 MB/s out of 750 MB/s	Used only 380 MB/s out of 1300 MB/s	NA
Write Rate	25 MB/s since restart	13 MB/s since restart	Rest of observation threw B as it was middle of bursting writing activity
Master Operations	202 Ops/s since restart	347 Ops/s since restart	Master can easily keep up with 200 to 500 operations per second Previous version spent lot of time in scanning large directories, solved by introducing binary searches through namespace
Recovery Time	Not Available as experiment was done on Cluster B	15,000 chunks containing 600 GB of data got restored in 23.2 mins 16,000 chunks and 660 GB of data of 2 chunkservers for restored in 2mins	NA

MapReduce: Simplified Data Processing on Large Clusters

A **MapReduce** is a programming model used for processing and generating large datasets. A Map function is used to generate a set of key/value pair and process it. A Reduce function is used to merge all values associated with same keys. This program partitions the input data, schedules execution, distributes data in parallel across several machines, handles failures, and manages inter-machine communication on large cluster to finish execution in reasonable amount of time. It processes different set of data such as crawled documents, web request logs, etc and provides analysis outputs like inverted indices, visual representation of web documents, and set of most frequent queries in a given day etc. MapReduce functions are efficiently used in scenarios with Distributed Grep, Distributed Sort, Count of URL Access Frequency, Reverse Web-Link Graph, Term-Vector per Host, Inverted Index. MapReduce can be implemented for different environments such as small shared-memory machine, larger collection of networked machines or even for a large NUMA multi-processor.

Features: Automatic parallelization and distribution for large-scale computations, High performance, Highly Scalable, Fault tolerance, Atomic commits, Handling Strangler.

Components: A Cluster, Processor, Commodity networking hardware, IDE disks per machine, A distributed file system, Scheduling system.

Workflow:

A map function partition input data into splits, and then starts Master/Worker programs on cluster. Master stores state and identity or worker machines. So, Master picks idle worker and assigns map or reduce tasks. Map Worker reads content and assigns key/value pairs and passes each pair to user-defined Map function in which these key/values pairs are buffered in memory. These Buffered pairs are written into partitioned disks by a partition function and it's locations are passed to Master. Master forwards these locations to reduce workers. Reduce workers are notified and read all data from local disk of map workers using remote procedure calls. Then Reduce workers sort data based on intermediate key and group all values under same keys. External sort is used if the data is too large. Reduce Worker passes key and corresponding values to Reduce Function and this function appends the result to final output file. MapReduce then, calls user program to return to user code.

Master provides status pages to show the progress of computations and also detailed status of an execution which helps engineers to diagnose bugs.

A Real Life Use Case: Production indexing system

Map Reduce program runs on Google Clusters every day. Production indexing system that produces the data structures used for the Google web search service which is rewritten using MapReduce Jobs. This system retrieves large datasets from crawling system and stores into GFS files. 5 to 10 MapReduce connections are executed to handle over 20 TB of data. 3800 lines from previous C++ code got reduced to 700 lines of MapReduce Code as all parallelization, distribution, fault tolerance locality optimization, and load balancing are hidden in libraries. Updating code has become easier than before. Scaling has become more feasible. Performances of workloads are improved by automatic fault tolerance handled by MapReduce.

Other real-time use cases within Google are Extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist), Extraction of properties of web pages for new experiments and products and Large-scale graph computations.

Bigtable: A Distributed Storage System for Structured Data

Bigtable is a distributed storage system for managing structured data at a very large scale like petabytes of data across thousands of commodity servers. This supports client applications directly to store data. Applications have different demands for Bigtable in terms of data size to backend high throughput bulk processing to real-time data serving. It provides simpler datamodel with various layouts & formats where a client can choose a schema that supports their data locality. Big table has two data structures, one for recent writes, and another for storing long-lived data.

On-going Features: Flexibility, Scalability, Availability, High performance solutions, Deploying Bigtables as Services with clusters to product groups.
Upcoming Features: Support for secondary indices, Cross data center having replicated Bigtables with multiple master replicas.
Components: Chubby- a distributed lock service, GFS, Distributed Data Storage Model, Bigtable Clusters – Master, Tabletserver, Tablet, Tablet(row range), memtable, SSTables, Tables, Data rows, Blocks, Block Index, Row Keys, Column Family Keys and Timestamp.
Workflow:
Bigtable implementation has three components: Master server, many tablet servers, a library that is linked with every client. Master is responsible for detecting the addition, expiration of tablet servers, assigning tablets to tablet servers, balancing load, collecting garbage files, saving schema changes.Tablet servers can be added or removed to cluster to accommodate changes in workloads. Tabletserver manages read, write requests to the tablets. When a tablet is grown, it gets split. Client communicates directly with tabletserver for workload and not with master, thus master workload is reduced. Hierarchy of table location stored in Chubby File ->Roottablet->METADATA tablets->and, then found in tables. Bigtable uses GFS to store log and data files. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

A Real Life Use Case: Google Analytics

Many applications from Google uses Bigtable combined with MapReduce Framework to store data for more than 60 Google products, including web indexing, Google Earth, Google Finance, Google Analytics. Google Analytics is a service to help webmasters analyze traffic patterns at their websites. It provides aggregate statistics on number of unique visitors per day, page views per URL per day, site-tracking reports, % of users purchased from site.

Process Flow: Webmasters embed a javascript program on their site, this program is invoked whenever this site is visited, records information such as user identifier, information about the page that is being fetched. Google Analytics summarizes these data and produces it to webmasters.
Rowkey: (analytics.google.com)
Tables Used: Raw click table, Summary table
Rawclick table contains row for each user session with rowname as rowkey and time which session was created. Summary table is generated from Rawclick table which contains various predefined summaries for each website. This table loaded periodically by scheduled MapReduce Jobs.

Spark: Cluster Computing with Working Sets

Spark framework allows users to reuse working datasets across multiple parallel operations, supports many iterative machine learning algorithms and produces interactive data analysis tool while retaining MapReduce features like scalability, fault tolerance. Spark can outperform Hadoop by 10x times while performing machine learning algorithms and using interactive query for huge datasets. Spark also overcame MapReduce's limitation as they use acyclic data flow (loading from disk) which is not effective for re-loading working datasets across multiple parallel operations. Spark integrates with Scala, a programming language, which allows these interactive operations.

Features: Parallel Processing, Reliability, Scalability, Load balancing, Fault tolerance, Locality, Scheduling data loads.

Components: ResilientDistributeDatasets(RDD), ParallelOperations, Meos-a cluster operating system

Workflow:

Spark creates a task to process each partition [Block Ids in HDFS] when parallel operation is initiated and sends these tasks to (preferred) worker nodes [Block locations in HDFS].On reaching worker node, each task calls "getIterator" to start reading its partition. If a node fails, partition re-reads its parent datasets. Finally, shipping closures to workers nodes and it carry outs reduce function.

There are two types of variables: Broadcast Variable where data is distributed only once to worker, and Accumulators are used for operations like parallel sums.

Spark uses RDD, a read-only collection of partitioned objects across. RDD in memory can be cached across machines and reuse it in MapReduce jobs. RDDs stores lineage information, which is helpful to recover just the failed partition. Parallel Operations like combine, reduce, foreach are performed on RDDs. In Spark, each object of RDD is represented by a scala object. Each RDD object has three operations: getPartitions, getIterator, getPreferredLocations. RDDs can be constructed in 4ways: directly from a file in shared file system like HDFS; by parallelizing; by transforming an existing RDD; by changing the persistence of an existing RDD by two ways - cache action, save action.

An Experimental Use case: Execution of Logistic Regression Model for ML using Spark

Components Used: 29 GD Datasets on 20 "m1.xlarge"EC2 nodes with 4 crores each

First Iteration took 174s in Spark and following, other iteration took only 6s as they reused cache. Spark ran 10x faster than hadoop. On 30th iteration Hadoop registered running almost near 4000s where as Spark ran below 500s. A node was killed during 10th iteration; even then the recovery time was much higher than hadoop.

A Real Life Use Case: Healthcare

Healthcare workers are getting valuable information by using Spark in data analytics which helps them to take efficient clinical decision by better image processing in order to provide required care, treatments to patients. It also reduces administrative costs, faster fraud deduction and streamlined data exchange. Since Spark is used on HDFS, special data migration is not required here.

Environment: Spark(Apache Spark 0.8.0) integrated HDFS (Intel Distribution for Apache Hadoop software version 3.0.2)and Apache Shark4 (Apache Shark 0.8.0), large-scale DWH system that runs on Spark cluster.

Components Required: Intel Xeon processor E5-2680, 128GB RAM, 300GB SSD, 2TB HDDNiantic X520-SR2 10GBase-SR PCI-e Dual Port, E10G42BFSR or E10G42BFSRG1P5 Duplex Fiber Optic, Cent OS 6.4 (Developers Workstation version),Scala 2.9.3

Reference: https://www.intel.com/content/dam/documents/white-papers/big-data