

Google Case Study

Disclaimer: Contents of this file is not generated by me borrowed from books. References are mentioned at the end of the document.

Objective:

- Explain Google Case Study of designing distributed systems.

Google (Traditionally, a US-based search engine company) uses various distributed computing services in its offering to customers such as cloud computing platforms, analytics services, AI/ machine learning services, navigation services, search engines, etc. To satisfy extremely demanding requirements, Google uses different mechanisms of distributed systems to meet the performance, scalability, reliability, and openness. The two most popular services that are provided by Google are a search engine and cloud

services, Google search is one of the most popular search engines in the world. It takes a keyword (or set of keywords) as an input to search on the web as a query and returns an ordered list of the most relevant results that match the given query for a keyword over the web. The crawling, indexing and ranking are the important mechanisms that are used in search engines. The crawling mechanism is used for locating and fetching the contents of the searched keywords over the web and pass those to the indexing system.

The indexing mechanism is used for creating an index for the searched contents, while the ranking mechanism is used for providing the ordered list of searched contents that are based on their popularity.

Google is also a pioneer in providing cloud computing services. They support two popular service models, namely Software as a service and Platform as a service. The Software as a service is provided through various Google apps that are intended for on-demand delivery of applications or software to their users with utility-based pricing. The popular Google Apps are Gmail, Google Docs, Google Sites, Google Talk and Google Calendar. The Platform as a service provides distributed system APIs to support the development and hosting of web applications through the Google app engine.

The physical infrastructure of Google is very large which consists of many numbers of commodity PCs to provide distributed storage and computations in a cost-effective manner. The commodity PCs are arranged in racks. These racks accommodate both PCs and Ethernet switches with internal and external connectivity across the rack. They are again organized into clusters in a data center.

1. Components of Google Infrastructure

The basic requirements of Google's infrastructure are scalability, reliability, performance and openness. Its components are shown in Figure 1. It is composed of three layers, namely communication paradigm, data storage and coordination, and distributed computations.

These layers provide a set of distributed services to offer communication paradigms and core functionality to developers. The bottommost layer of the infrastructure is communication paradigms which have two sub-components, namely protocol buffers and publish-subscribe service. They are used for both remote invocation and indirect communication. The protocol buffers

provide communication patterns in the form of serialization of requests and replies for remote invocation, while the publish-subscribe service supports the efficient distribution of events to potentially large numbers of subscribers. The second layer is data and coordination services that provide storage for unstructured and semi-structured data along with services to support coordinated access to the data. It has three components, namely GFS, Big tables and Chubby. The third layer is distributed computation services, which is used for providing a platform for carrying out parallel and distributed computation over the physical infrastructure. It also has two components, namely MapReduce and Sawzall.

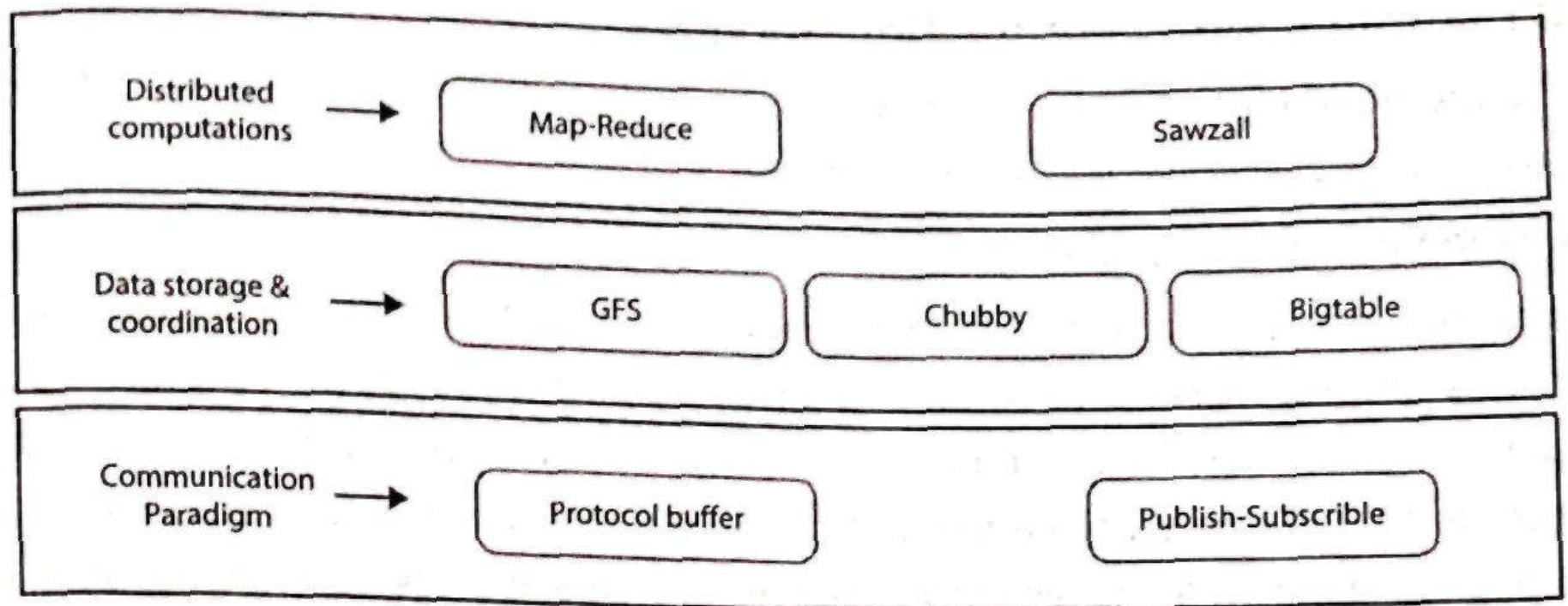


Figure 6.12: Google Infrastructure

The detailed description of the above three layers is as follows.

Communication Paradigm

The Communication paradigm(s) is a crucial component of the overall system design of the Google infrastructure. It provides inter-process communication over the socket using remote invocation services like request-reply protocol (RR Protocol), remote procedure calls (RPC) or remote method invocation (RMI). It also supports client-server interactions over the indirect communication paradigm such as distributed shared memory approaches, distributed event-based approaches, group communications, etc. It has two sub-components, namely Protocol buffer and Publish-subscribe service.

Protocol buffer provides the specification of messages that has a description of subsequent serialization of data. It provides a language and platform-independent way to serialize the data in a simple, highly efficient and extensible manner. Such serialized data can be used for subsequent storage transmitted using an underlying communications protocol. It uses

RPC exchanges across the network with two abstract interfaces such as RpcChannel and RpcController.

The publish-subscribe system is used when real-time distributed events need to be disseminated with reliability. The RPC system is inappropriate to use for this kind of interaction. The example of a publish-subscribe system is the Google Ads system where appropriate advertisements are displayed that are based on the query on certain websites. A publish-subscribe system has a built-in time and space uncoupling and offers support for the failure and recovery of subscribers. It is implemented in the form of a tree where the root is the publisher, a leaf is a subscriber and the whole tree represents the topic.

Data storage and Co-ordination Services

There are three components in this layer, namely Google File System (GFS), Chubby and Big tables which are described as follows.

The Google File System (GFS)

Google has designed a distributed file system, named GFS, for meeting its

exacting demands off processing a large amount of data. Most of the objectives of designing the GFS are similar to those of the earlier designed distributed systems. Some of the objectives include availability, performance, reliability, and scalability of systems. GFS has also been designed with certain challenging assumptions that also provide opportunities for developers and researchers to achieve these objectives. Some of the assumptions are listed as follows.

- a. Automatic recovery from component failure on a routine basis.
- b. Efficient storage support for large-sized files as a huge amount of data to be processed is stored in these files. Storage support is provided for small-sized files without requiring any optimization for them.
- c. With the workloads that mainly consist of two large streaming reads and small random reads, the system should be performance conscious so that the small reads are made steady rather than going back and forth by batching and sorting while advancing through the file.
- d. The system supports small writes without being inefficient, along with the usual large and sequential writes through which data is appended to files.

- e. Semantics that are defined well are implemented.
- f. Atomicity is maintained with the least overhead due to synchronization.
- g. Provisions for sustained bandwidth is given priority rather than a reduced latency.

Google takes the aforementioned assumptions into consideration, and supports its cloud platform, Google Apps Engine, through GFS. Figure 2 shows the architecture of the GFS cluster.

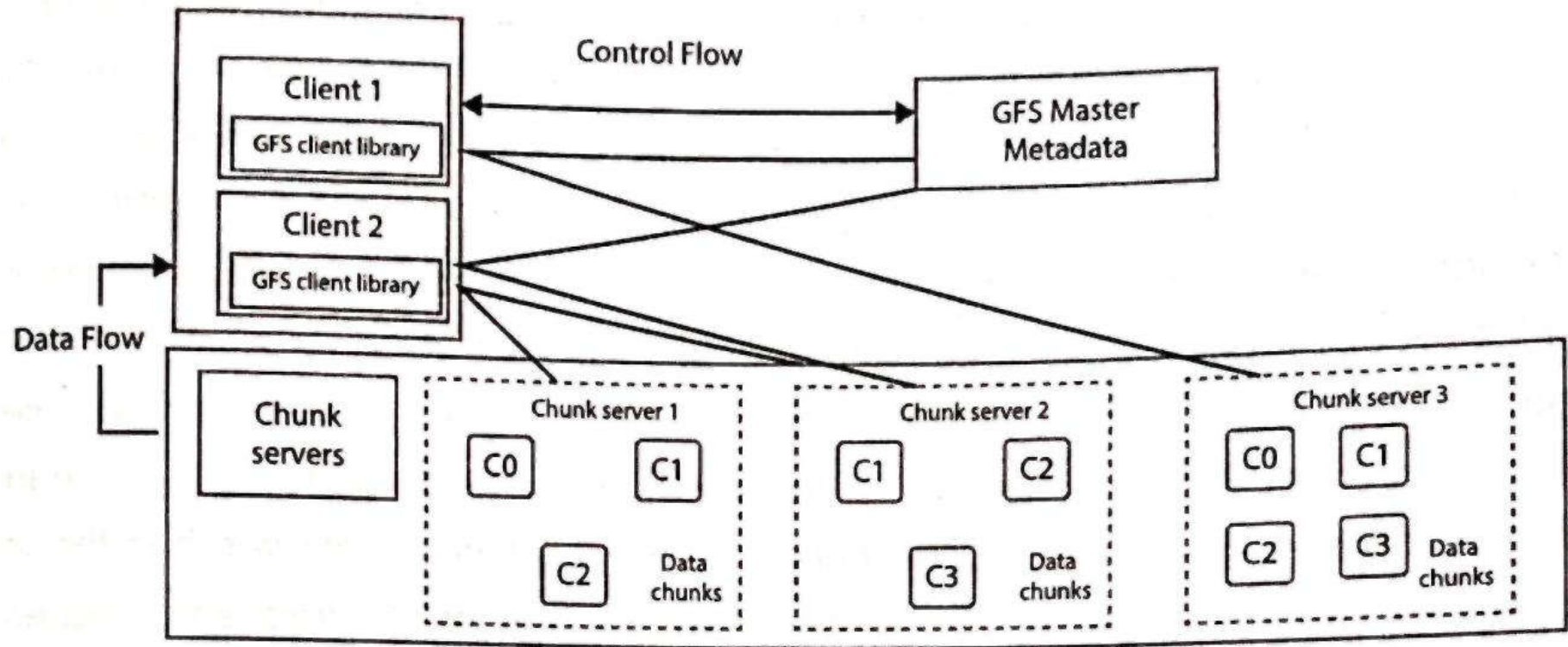


Figure 2. Architecture of GFS Cluster

GFS provides a file system interface and different APIs for supporting different file operations such as create to create a new file instance, delete to delete a file instance, open to open a named file and return a handle, close to close a given file specified by a handle, read to read data from a specified file and write to write data to a specified file.

It can be seen from Figure 2, that a single GFS Master and three chunk servers are serving to two clients comprise a GFS cluster. These clients and servers, as well as the Master, are Linux machines, each running a server process at the user level. These processes are known as user-level server processes.

In GFS, the metadata is managed by the GFS Master that takes care of all the communication between the clients and the chunk servers. Chunks are small blocks of data that are created from the system files. Their usual size is 64 MB. The clients interact directly with chunk servers for transferring chunks of data. For better reliability, these chunks are replicated across three machines so that whenever the data is required, it can be obtained in its complete form from at least one machine. By default, GFS stores three replicas of the chunks of data. However, users can designate any levels of replication. Chunks are created by dividing the files into fixed-sized blocks. A unique immutable handle (of 64-bit) is assigned to each chunk at the time of their creation by the GFS Master. The data that can be obtained from the chunks, the selection of which is specified by the unique handles, is read or written on local disks by the chunk servers. GFS has all the familiar system

interfaces. It also has additional interfaces in the form of snapshots and appends operations. These two features are responsible for creating a copy of files or folder structure at low costs and for permitting a guaranteed atomic data-append operation to be performed by multiple clients of the same file concurrently.

Applications contain a specific file system, Application Programming Interface (APIs) that are executed by the code that is written for the GFS client. Further, the communication with the GFS Master and chunk servers are established for performing the read and write operations on behalf of the application. The clients interact with the Master only for metadata operations. However, data-bearing communications are forwarded directly to chunk servers. POSIX API, a feature that is common to most of the popular file systems, is not included in GFS, and therefore, Linux vnode layer hook-in is not required. Client or servers do not perform the caching of file data. Due to the presence of the streamed workload, caching does not benefit clients, whereas caching by servers has the least consequence as a buffer cache that already maintains a record for frequently requested files locally.

The GFS provides the following features.

- Large-scale data processing and storage support
- Normal treatment for components that stop responding
- Optimization for large-sized files (mostly appended concurrently and read sequentially)
- Fault tolerance by constant monitoring, data replication, and automatic recovering
- Data corruption detections at the disk or Integrated Development Environment (IDE) subsystem level through the checksum method
- High-throughput for concurrent readers and writers
- Simple designing of the Master that is centralized and not bottlenecked
- GFS provides caching for the performance and scalability of a file system and logging for debugging and performance analysis.

Chubby

Chubby is the crucial service in the Google infrastructure that offers storage

and coordination for other infrastructure services such as GFS and Bigtable. It is a coarse grained distributed locking service that is used for synchronizing distributed activities in an asynchronous environment on a large scale. It is used as a name service within Google and provides reliable storage for file systems along with the election of coordinator for multiple replicas. The Chubby interface is similar to the interfaces that are provided by distributed systems with advisory locks. However, the aim of designing Chubby is to provide reliable storage with consistent availability. It is designed to use with looser/ coupled distributed systems that are connected in a high-speed network and contain several small-sized machines. The lock service enables the synchronization of the activities of clients and permits the clients to reach a consensus about the environment in which they are placed. Chubby's main aim is to efficiently handle a large set of clients by providing them a highly reliable and available system. Its other important characteristics that include throughput and storage capacity are secondary. Figure 3 shows the typical structure of a Chubby system.

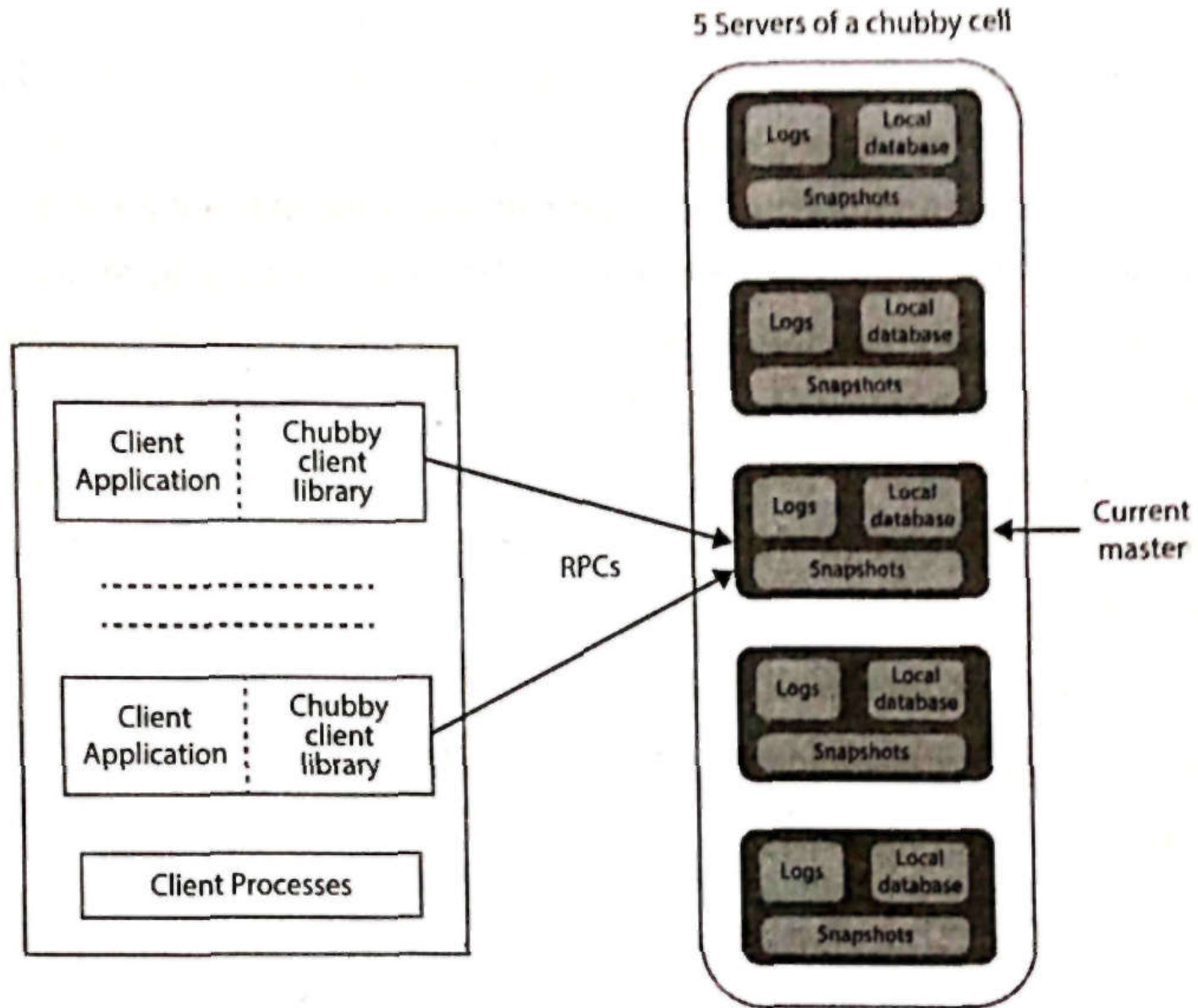


Figure 3. Structure of a Chubby system

It can be seen in Figure 3 that Chubby involves two primary components, namely server and client library. Both the components communicate through a Remote Procedure Call (RPC). However, the library has a special purpose, i.e., linking the clients against the chubby cell. A Chubby cell contains a small set of servers. The servers are also called replicas, and usually, five servers are used in every cell. The Master is elected from the five replicas through a distributed protocol that is used for consensus. Most of the replicas must vote for the Master with the assurance that no other Master will be elected by replicas that have once voted for one Master for a duration. This duration is termed as a Master lease.

Chubby supports a similar file system as Unix. However, the Chubby file system is simpler than the Unix one. The files and directories, known as nodes, are contained in the Chubby namespace. Each node is associated with different types of metadata. The nodes are opened to obtain the Unix file descriptors known as handles. The specific for handles include check digits for preventing the guess handle for clients, handle sequence numbers, and mode information for recreating the lock state when the Master changes. Reader and writer locks are implemented by Chubby using files and

directories. While exclusive permission for a lock in the writer mode can be obtained by a single client, there can be any number of clients who share a lock in the reader's mode. The nature of locks is advisory, and a conflict occurs only when the same lock is requested again for an acquisition. The distributed locking mode is complex. On one hand, its use is costly, and on the other hand, it only permits numbering the interactions that are already using locks. The status of locks after they are acquired can be described using specific descriptor strings called sequencers. The sequencers are requested by locks and passed by clients to servers in order to progress with protection.

Another important term that is used with Chubby is an event that can be subscribed by clients after the creation of handles. An event is delivered when the action that corresponds to it is completed. An event can be:

- a. Modification in the contents of a file
- b. Addition, removal, or modification of a child node
- c. Failing over of the Chubby Master
- d. Invalidity of a handle

- e. Acquisition of lock by others
- f. Request for a conflicting lock from another client

In Chubby, caching is done by a client that stores file data and metadata to reduce the traffic for the reader lock. Although there is a possibility for caching of handles and files locks, the Master maintains a list of clients that may be cached. The clients, due to caching, find data to be consistent. If this is not the case, an error is flagged. Chubby maintains sessions between clients and servers with the help of a keep-alive message, which is required every few seconds to remind the system that the session is still active. Handles that are held by clients are released by the server in case the session is overdue for any reason. If the Master responds late to a keep-alive message, as the case may be, at times, a client has its own timeout (which is longer than the server timeout) for the detection of the server failure.

If the server failure has indeed occurred, the Master does not respond to a client about the keep-alive message in the local lease timeout. This incident sends the session in jeopardy. It can be recovered in a manner as explained in the following points:

- The cache needs to be cleared.
- The client needs to wait for a grace period, which is about 45 seconds.
- Another attempt is made to contact the Master.

If the attempt to contact the Master is successful, the session resumes and its jeopardy is over. However, if this attempt fails, the client assumes that the session is lost. Figure 4 shows the case of the failure of the Master.

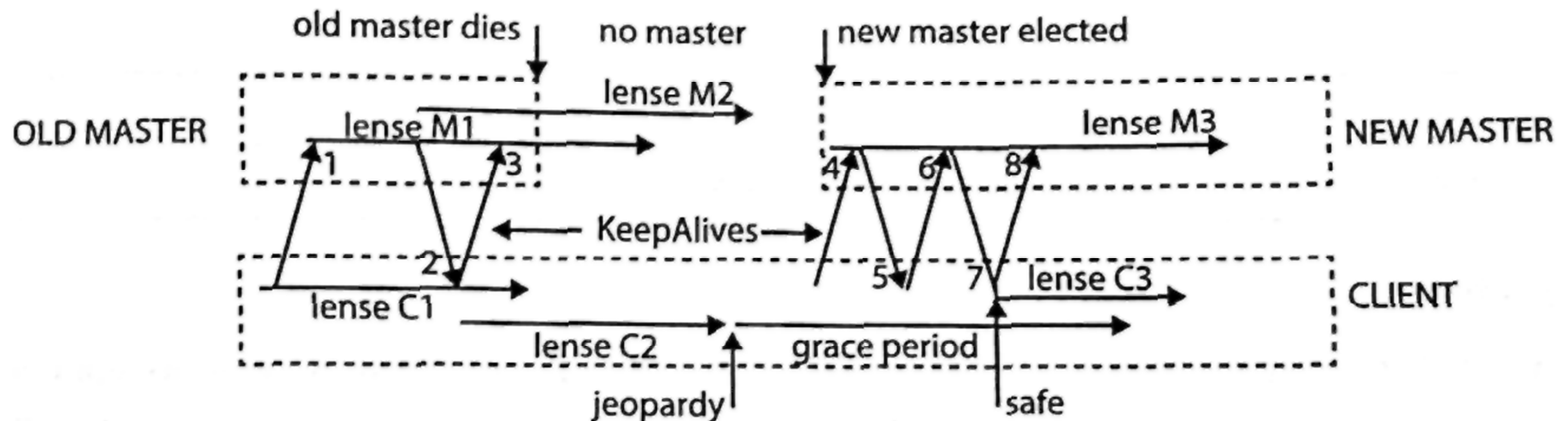


Figure 4. Case of failure of Master Server

Chubby offers a decent level of scalability, which means that there can be any (unspecified) number of the Chubby cells. If these cells are fed with heavy loads, the lease timeout increases. This increment can be anything between

12 seconds and 60 seconds. The data is fed in a small package and held in the Random-Access Memory (RAM) only. The Chubby system also uses partitioning mechanisms to divide data into smaller packages. All of its excellent services and applications included, Chubby has proved to be a great innovation when it comes to storage, locking, and program support services.

The Chubby is implemented using the following APIs:

1. Creation of handles using the `open()` method
2. Destruction of handles using the `close()` method

The other important methods include `GetContentsAndStat()`, `GetStat()`, `ReadDir()`, `SetContents()`, `SetACI()`, `Delete()`, `Acquire()`, `TryAcquire()`, `Release()`, `GetSequencer()`, `SetSequencer()`, and `CheckSequencer()`. The commonly used APIs in chubby are listed in Table IV.

Table 1. APIs in Chubby

API	Description
Open	Opens the file or directory and returns a handle
Close	Closes the file or directory and returns the associated handle
Delete	Deletes the file or directory
ReadDir	Returns the contents of a directory
SetContents	Writes the contents of a file
GetStat	Returns the metadata
GetContentsAndStat	Writes the file contents and return metadata associated with the file
Acquire	Acquires a lock on a file
Release	Releases a lock on a file

Big table

Google's Big table is a distributed storage system that allows storing huge volumes of structured as well as unstructured data on storage mediums. Google created Big Table with an aim to develop a fast, reliable, efficient and scalable storage system that can process concurrent requests at a high speed. Millions of users access billions of web pages and many hundred TBs of satellite images. A lot of semi-structured data is generated from Google or web access by users. This data needs to be stored, managed, and processed to retrieve insights. This required data management systems to have very high scalability.

Google's aim behind developing Big Table was to provide a highly efficient system for managing a huge amount of data so that it can help cloud storage services. It is required for concurrent processes that can update various data pieces so that the most recent data can be accessed easily at a fast speed. The design requirements of Big Table are as follows.

1. High speed
2. Reliability

3. Scalability
4. Efficiency
5. High performance
6. Examination of changes that take place in data over a period of time

Big Table is a popular, distributed data storage system that is highly scalable and self-managed. It involves thousands of servers, terabytes of data storage for in-memory operations, millions of read/write requests by users in a second and petabytes of data stored on disks. Its self-managing services help in dynamic addition and removal of servers that are capable of adjusting the load imbalance by themselves. It has gained extreme popularity at Google as it stores almost all kinds of data, such as Web indexes, personalized searches, Google Earth, Google Analytics, and Google Finance. It contains data from the Web is referred to as a Web table. The generalized architecture of Big table is shown in Figure 5.

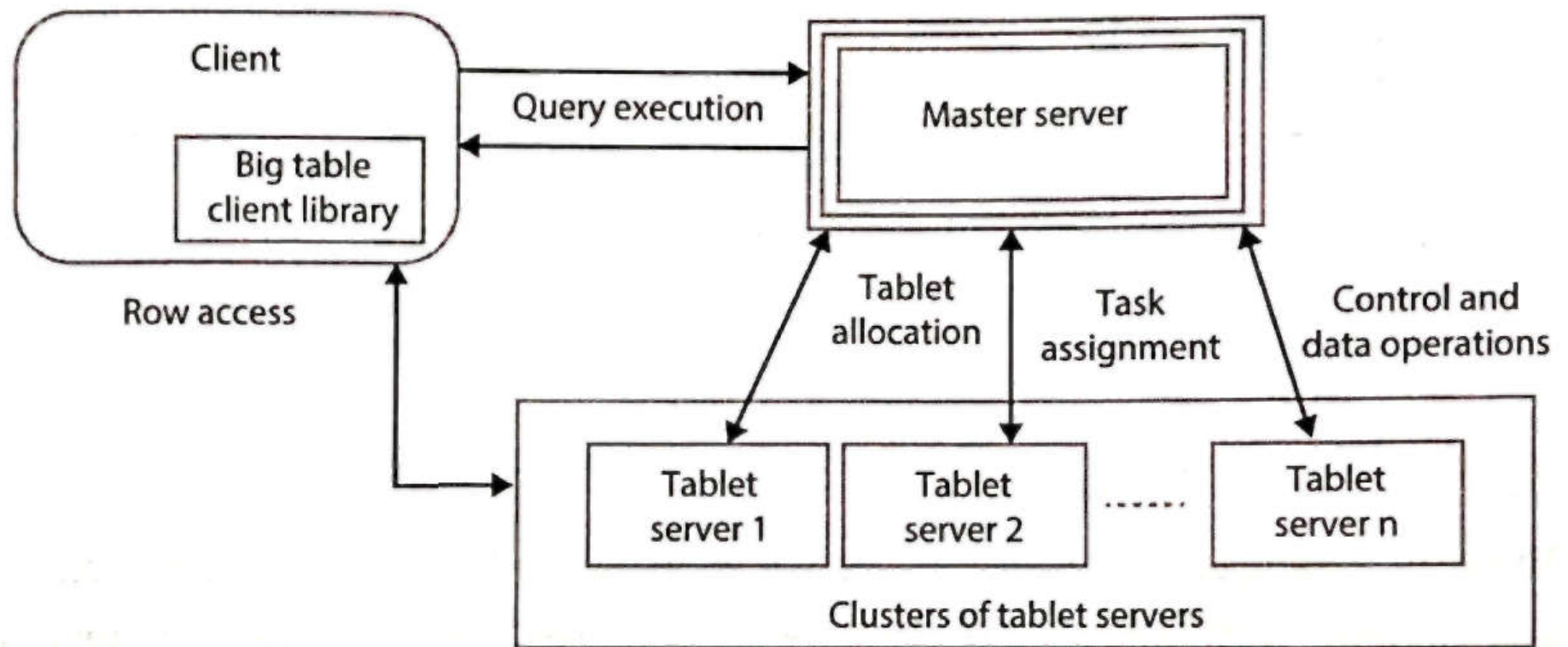


Figure 5. Generalized architecture of Big table

It is composed of three entities, namely Client, Big table master and Tablet servers. Big tables are implemented over one or more clusters that are similar to GFS clusters. The client application uses libraries to execute Big table queries on the master server. Big table is initially broken up into one or more slave servers called tablets for the execution of secondary tasks. Each tablet

is 100 to 200 MB in size.

The master server is responsible for allocating tablets to tasks, clearing garbage collections and monitoring the performance of tablet servers. The master server splits tasks and executes them over tablet servers. The master server is also responsible for maintaining a centralized view of the system to support optimal placement and load-balancing decisions. It performs separate control and data operations strictly with tablet servers. Upon granting the tasks, tablet servers provide row access to clients.

Figure 6 shows the structure of Big table.

Big Table is arranged as a sorted map that is spread in multiple dimensions and involves sparse, distributed, and persistence features. The Big Table's data model primarily combines three dimensions, namely row, column, and timestamp. The first two dimensions are string types, whereas the time dimension is taken as a 64-bit integer. The resulting combination of these dimensions is a string type.


Column families and qualifiers					
	Column_Family 1	Column_Family 2: q1	Column_Family 2: q2	Column_Family 3: q1	Column_Family 3: q2
Row 1					
Row 2			 T-9 T-6 T-3	Timestamp	
Row 3					
Row 4					
Row 5					

Figure 6. Structure of Big table

Each row in Big table has an associated row key that is an arbitrary string of up to 64 KB in size. In Big Table, a row name is a string, where the rows are ordered in a lexicological form. Although Big Table rows do not support the relational model, they offer atomic access to the data, which means you can access only one record at a time. The rows contain a large amount of data about a given entity such as a web page. The row keys represent URLs that contain information about the resources that are referenced by the URLs.

The naming conventions that are used for columns are more structured than those of rows. Columns are organized into a number of column families that logically groups the data under a family of the same type. Individual columns are designated by qualifiers within families. In other words, a given column is referred to use the syntax `column_family: optional_qualifier`, where `column_family` is a printable string and `qualifier` is an arbitrary string. It is necessary to provide an arbitrary name to one level which is known as a column family, but it is not mandatory to give a name to a qualifier. The column family contains information about the data type and is actually the unit of access control. Qualifiers are used for assigning columns in each row. The number of columns that can be assigned in a row is not restricted.

The other important dimension that is assigned to Big Table is a timestamp. In Big table, the multiple versions of data are indexed by timestamp for a given cell. The timestamp is either related to real-time or can be an arbitrary value that is assigned by a programmer. It is used for storing various data versions in a cell. By default, any new data that is Inserted into Big Table is taken as current, but you can explicitly set the timestamp for any new write operation in Big Table. Timestamps provide the Big Table lookup option that returns the specified number of the most recent values. It can be used for marking the attributes of the column families. The attributes either retain the most recent values in a specified number or keep the values for a particular time duration.

Big Table supports APIs that can be used by developers to perform a wide range of operations such as metadata operations, read/write operations, or modify/update operations. The commonly used operations by APIs are as follows:

- Creation and deletion of tables
- Creation and deletion of column families within tables

- Writing or deleting cell values
- Accessing data from rows

Associate metadata such as access control information with tables and column families The functions that are used for atomic write operations are as follows:

- Set () is used for writing cells in a row.
- DeleteCells () is used for deleting cells from a row.
- DeleteRow() is used for deleting the entire row, i.e., all the cells from a row are deleted.

It is clear that Big Table is a highly reliable, efficient, and fan system that can be used for storing different types of semi-structured or unstructured data by users.

Distributed computation services

Distributed computation services in Google's infrastructure are intended to perform computations and data processing operations over real-time and stored data. It has two components, namely MapReduce and Sawzall. MapReduce

Google supports the MapReduce programming model that is provided by Hadoop that allows expressing distributed computations on a huge amount of data. It provides an easy scaling of data processing over multiple computational nodes or clusters. In this model, the data processing primitives that are used are called mapper and reducer. Every MapReduce program must have at least one mapper and reducer subroutines. A mapper has a map method that transforms input key-value pairs into any number of intermediate key-value pairs, while a reducer has a reduce method that transforms intermediate key value pairs that are aggregated into any number of output key, value pairs.

MapReduce keeps all the processing operations separate for parallel executions where a complex and extremely large-sized problem is

decomposed into subtasks. These subtasks are executed independently from each other. After that, the result of all independent executions is combined together to get the complete output. The unit of work in MapReduce is a job. During the map phase, input data is divided into input splits for analysis where each split is an independent task. These tasks run parallelly across the Hadoop clusters. The reducer phase uses the result that is obtained from the mapper as an input to generate the final result. MapReduce takes a set of input <key, value> pairs and produces a set of output <key, value> pairs by supplying data through a map and reduce functions. The typical map reduce process is shown in Figure 7.

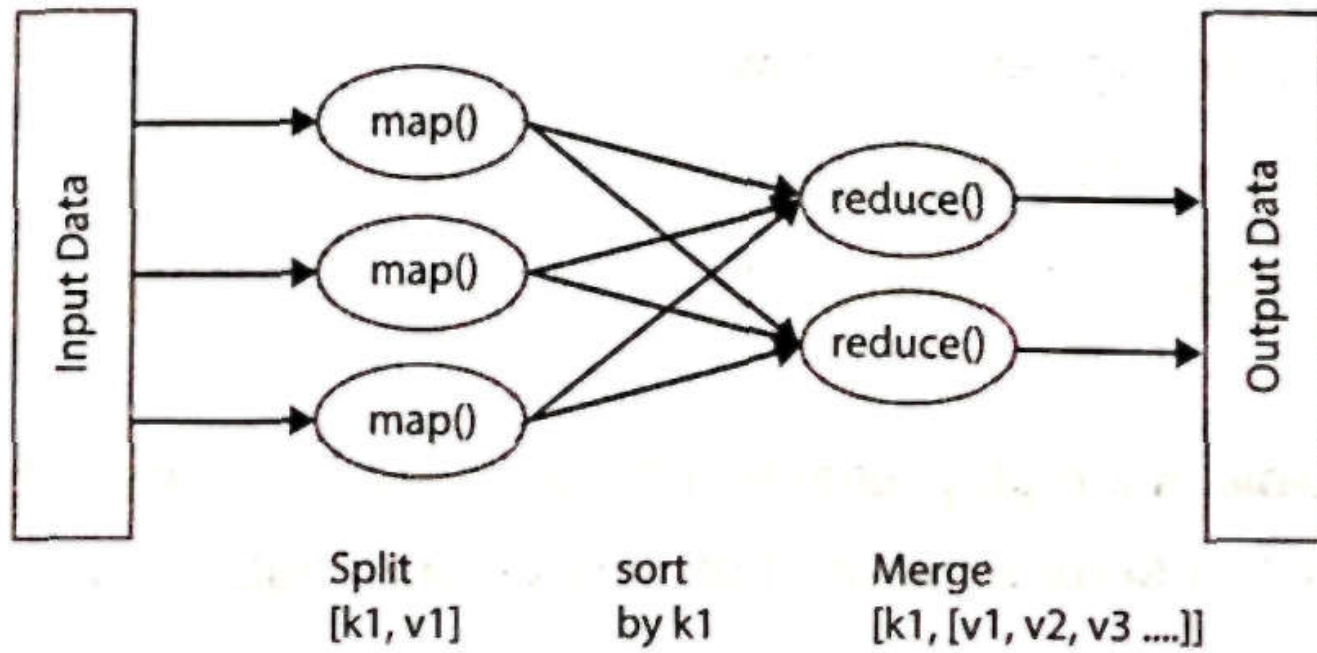


Figure 7. MapReduce Process

Every map-reduce program undergoes different phases of execution. Each phase has its own significance in map-reduce framework. The different phases of execution in map reduce are shown in Figure 8.

In the input phase, the large data set in the form of <key, value> pair is provided as standard input for a MapReduce program. The input files used by MapReduce are kept on the HDFS (Hadoop Distributed File System) store which has a standard input format that is specified by the user.

Once the input file is selected, then the split phase reads the input data and divides them into smaller chunks. These smaller chunks are then given to a mapper.

The map operations extract the relevant data and generate intermediate key value pairs. They read the input data from a split using record reader and generate intermediate results. They are used for transforming the input key, value list data to output key, and value list which is then passed to the combiner.

A combiner is used with a mapper and a reducer to reduce the volume of data transfer. It is also known as a semi-reducer which accepts input from a

mapper and passes output key, value pair to a reducer. The shuffle and sort are the components of reducer. Shuffling is a process of partitioning and moving a mapped output to the reducer where intermediate keys are assigned to the reducer. Each partition is known as a subset and so each subset becomes an input to the reducer. In general shuffle, the phase ensures that partitioned splits reached appropriate reducers where the reducer uses the HTTP protocol to retrieve their own partition from a mapper. The sort phase is responsible for sorting the intermediate keys on a single node automatically before they are presented to the reducer.

The shuffle and sort phases occur simultaneously where the mapped output is being fetched and merged.

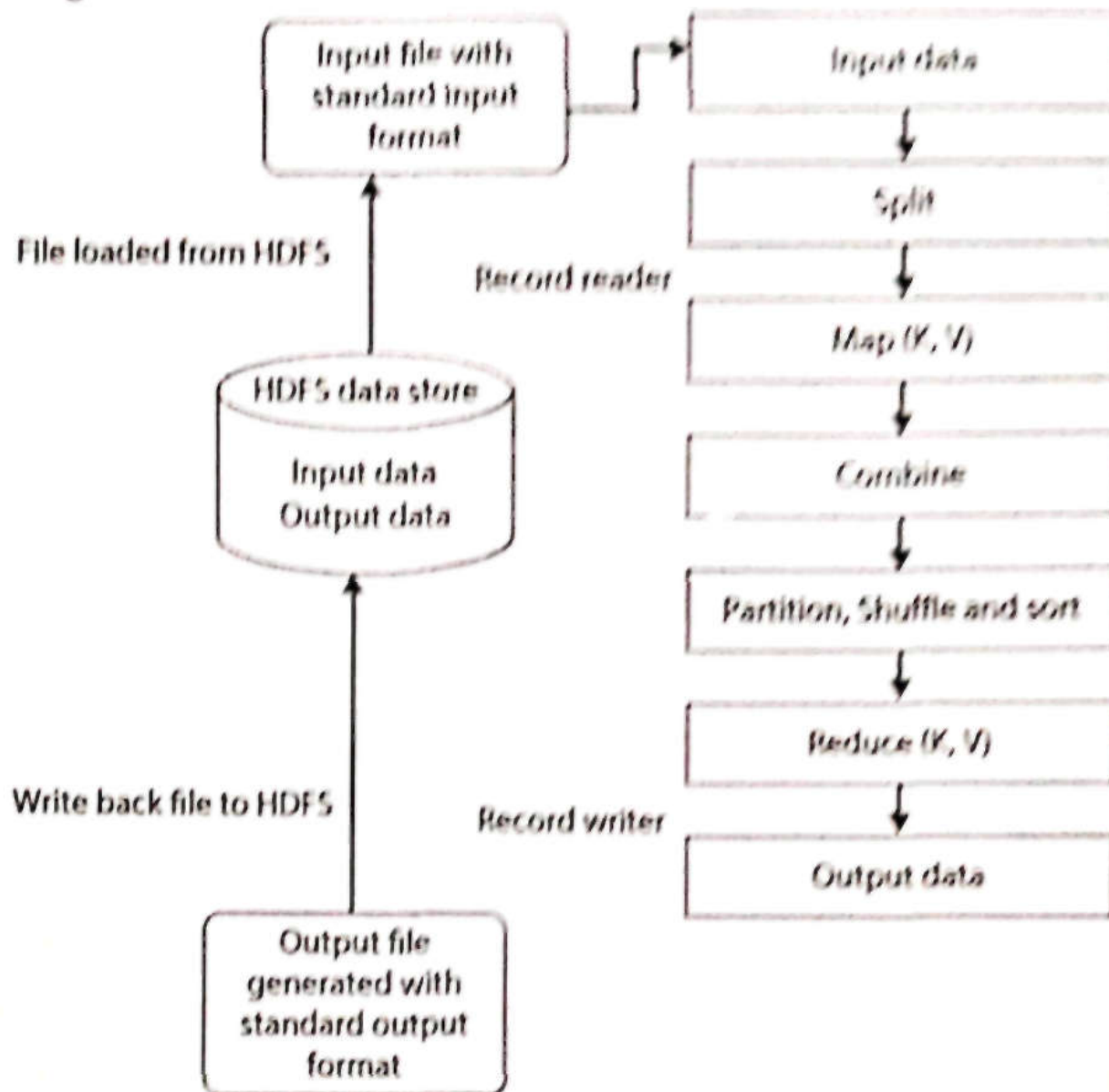


Figure 8. Different phases of execution In MapReduce

The reducer reduces a set of intermediate values that share unique keys with a set of values. The reducer uses sorted input to generate the final output. The final output is written using a record writer by the reducer into the output file with standard output format. The final result of each MapReduce program is generated with key value pairs that are written in the output file which is written back to the HDFS store. MapReduce is implemented by a library, which is built on the top of the Google infrastructure and provides RPC for communication and GFS for the storage of intermediary values such as HDFS. It supports parallelization and distribution of tasks across different MapReduce processes. Google provides master and worker servers to execute the MapReduce jobs. The intermediate servers that are used for executing map and reduce tasks called workers. The workers are assigned by the master server. The overall operation of the MapReduce program on Google infrastructure is shown in Figure 9.

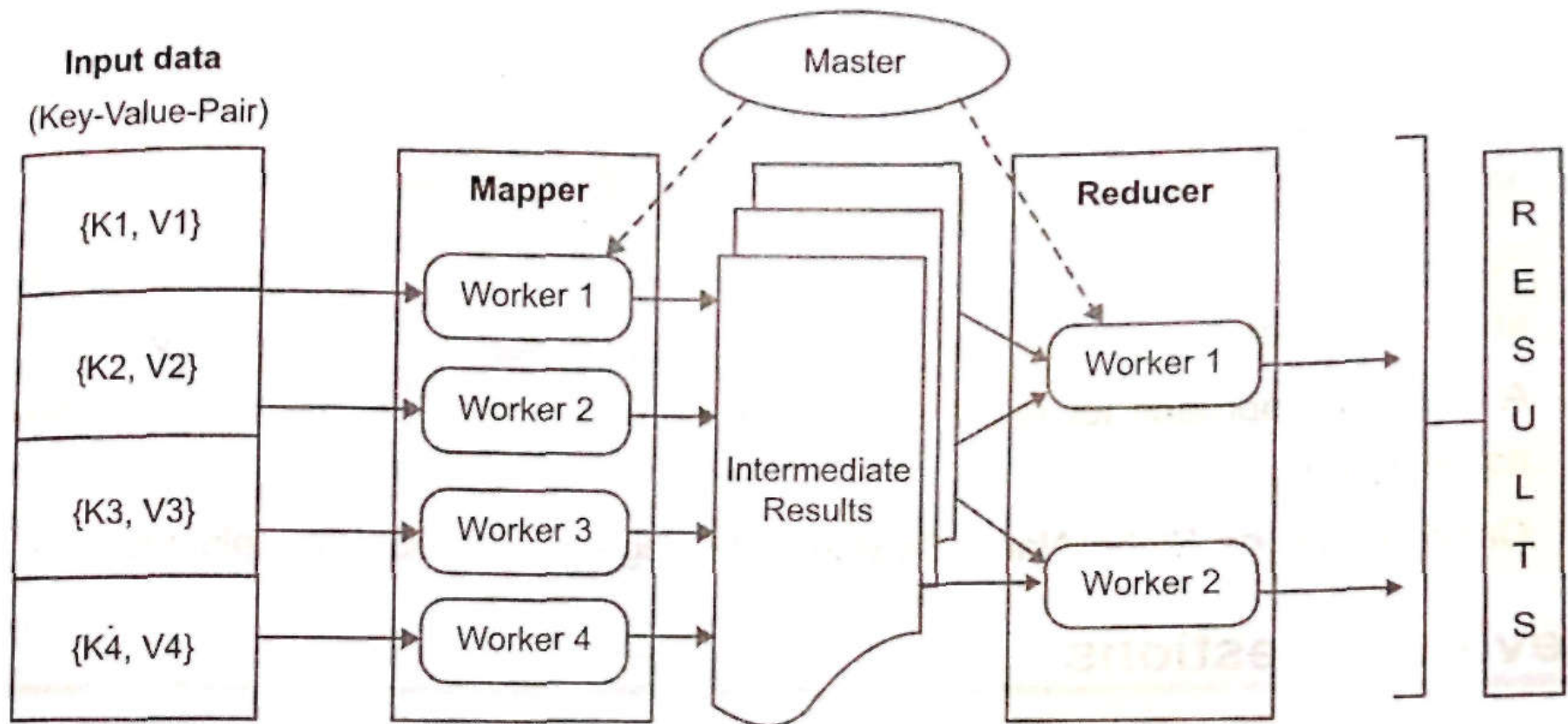


Figure 9. Overall operation of the MapReduce program on Google infrastructure

Sawzall

The Sawzall is an interpreted programming language that is used by Google to perform parallel data analysis over large datasets. It is mostly used as a map-reduce for a distributed data processing. It works in a distributed environment where filters and aggregators are used for fetching data from datasets. In general, the aggregator operations are associative while filter operations are commutative across all records. Unlike MapReduce, Sawzall programs perform the filter operations in the map phase of MapReduce and aggregator operations in the reduce phase.

References:

- 1] George Coulouris, Jean Dollimore, Tim Kindberg, “Distributed Systems: Concepts and Design”, 4th Edition, Pearson Education, 2005.
- 2] S. Tanenbaum and M. V. Steen, “Distributed Systems: Principles and Paradigms”, Second Edition, Prentice Hall, 2006.
- 3] Nupur Giri, Lata Ragha, Bhushan Jadhav, “Distributed Computing”, StarEdu, 2020.