4.1. GOOGLE App Engine:

Google App Engine (GAE) is a Platform-as-a-Service cloud computing model that supports many programming languages.

GAE is a scalable runtime environment mostly devoted to execute Web applications. In fact, it allows developers to integrate third-party frameworks and libraries with the infrastructure still being managed by Google.

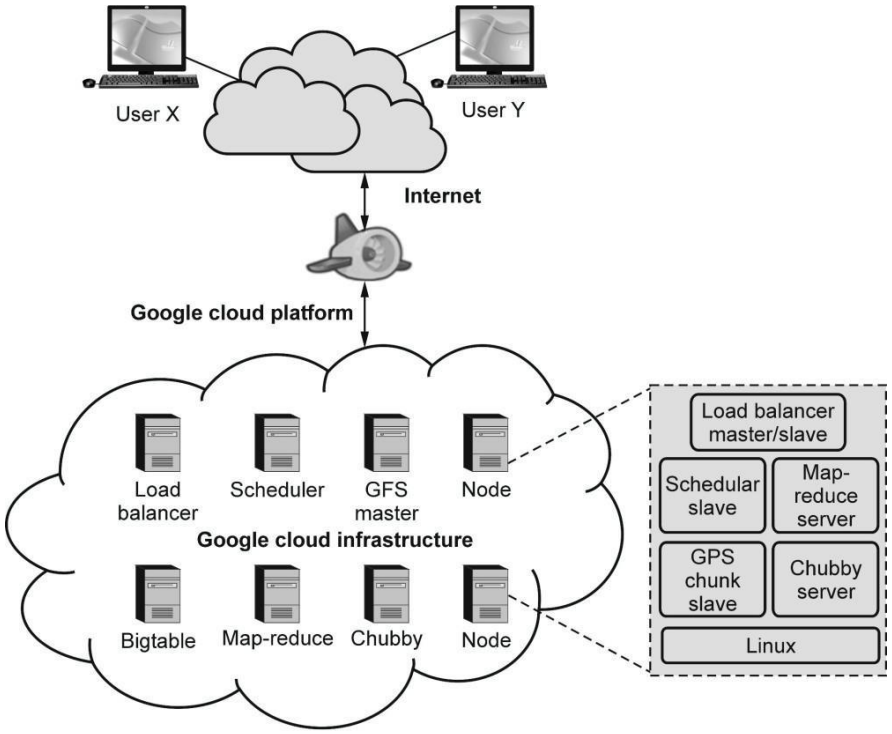
It allows developers to use readymade platform to develop and deploy web applications using development tools, runtime engine, databases and middleware solutions. It supports languages like Java, Python, .NET, PHP, Ruby, Node.js and Go in which developers can write their code and deploy it on available google infrastructure with the help of Software Development Kit (SDK).

In GAE, SDKs are required to set up your computer for developing, deploying, and managing your apps in App Engine. GAE enables users to run their applications on a large number of data centers associated with Google's search engine operations.

Google App Engine uses fully managed, serverless platform that allows to choose from several popular languages, libraries, and frameworks to develop user applications and then uses App Engine to take care of provisioning servers and scaling application instances based on demand.

The infrastructure for google cloud is managed inside datacenter. All the cloud services and applications on Google runs through servers inside datacenter. Inside each data center, there are thousands of servers forming different clusters. Each cluster can run multipurpose servers.

The infrastructure for GAE composed of four main components like Google File System (GFS), MapReduce, BigTable, and Chubby. The GFS is used for storing large amounts of data on google storage clusters. The MapReduce is used for application program development with data processing on large clusters. Chubby is used as a distributed application locking services while BigTable offers a storage service for accessing structured as well as unstructured data. In this architecture, users can interact with Google applications via the web interface provided by each application.



Functional architecture of the Google cloud platform for app engine

The GAE platform comprises five main components like

- Application runtime environment offers a platform that has built-in execution engine for scalable web programming and execution.
- Software Development Kit (SDK) for local application development and deployment over google cloud platform.
- Datastore to provision object-oriented, distributed, structured data storage to store application and data. It also provides secures data management operations based on BigTable techniques.

- Admin console used for easy management of user application development and resource management
- GAE web service for providing APIs and interfaces.

Programming Environment for Google App Engine

The Google provides programming support for its cloud environment, that is, Google Apps Engine, through Google File System (GFS), Big Table, and Chubby. The following sections provide a brief description about GFS, Big Table, Chubby and Google APIs.

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.

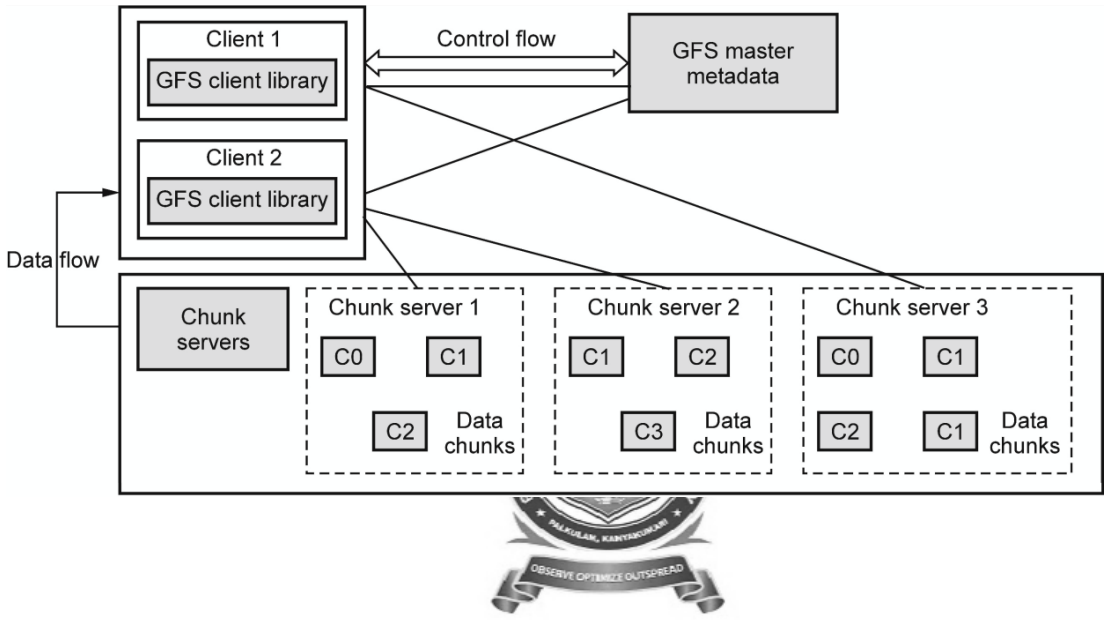
e) Semantics that are defined well are implemented.

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. Fig. 4.2 shows the architecture of the GFS clusters.



Architecture of GFS clusters

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.

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.

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.

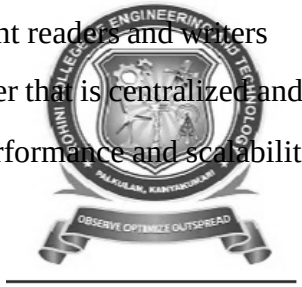
Clients or servers do not perform the caching of file data. Due to the presence of the

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.



Big Table

Googles 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 user's 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.

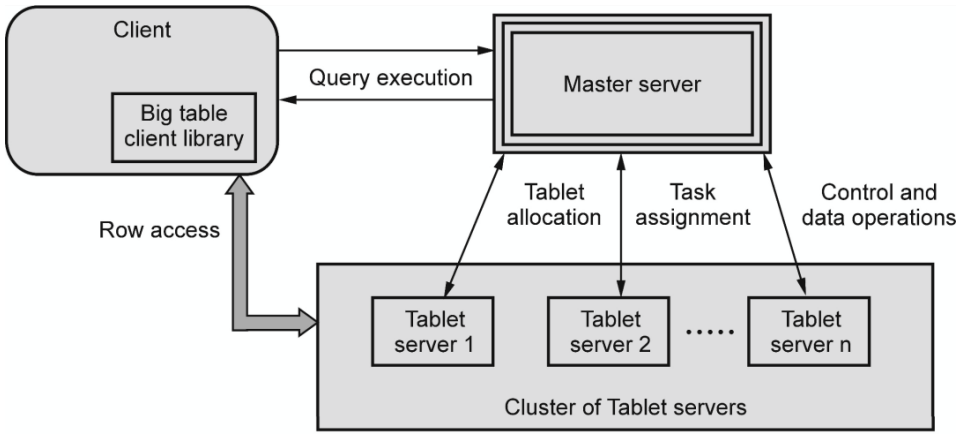
It is required for concurrent processes that can update various data pieces so that the most recent data can be accessed easily at a fast speed. The design requirements of Big Table are as follows:

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 Fig. 4.3



Generalized architecture of Bigtable

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

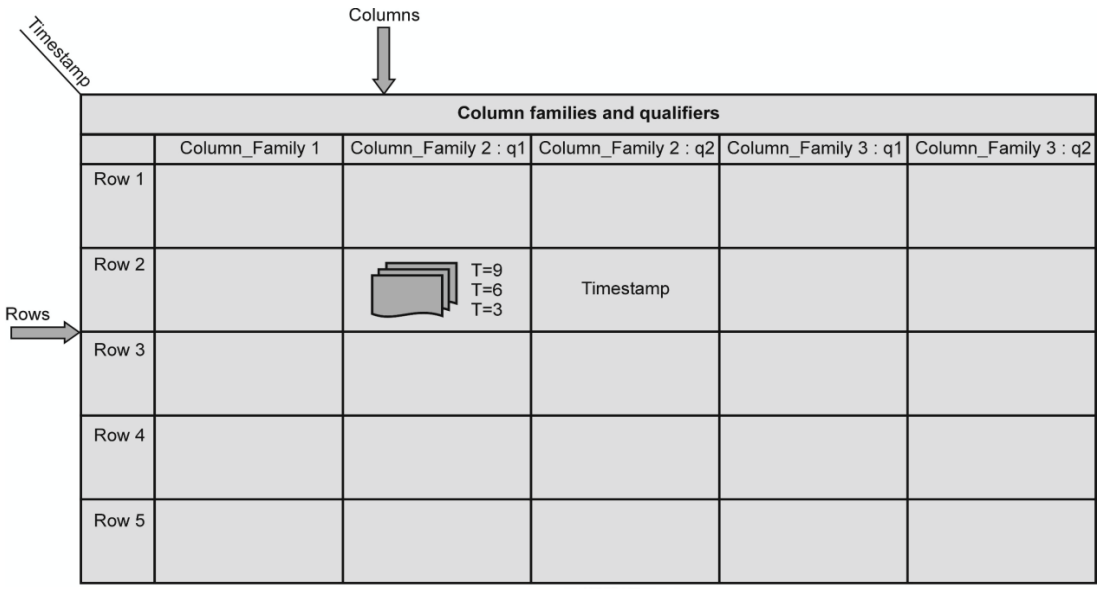
and monitoring the performance of tablet servers. The master server splits tasks and executes them over tablet servers. The master server is also responsible for maintaining a centralized view of the system to support optimal placement and load- balancing decisions.

It performs separate control and data operations strictly with tablet servers. Upon granting the tasks, tablet servers provide row access to clients. Fig. 4.6.3 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 time*stamp. 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.

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 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.



Structure of Big table

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.

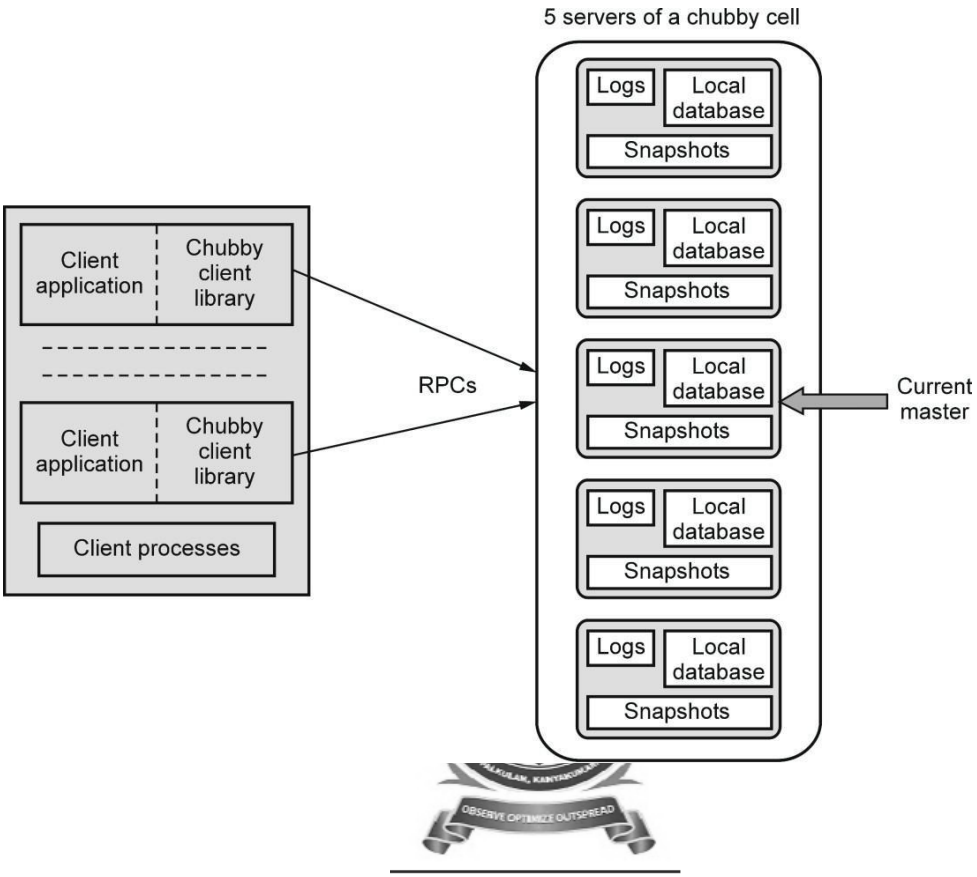
- DeleteCells () is used for deleting cells from a row.
- DeleteRow() is used for deleting the entire row, i.e., all the cells from a row are deleted.

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.

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 loosely 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. Fig. 4.5 shows the typical structure of a Chubby system:



Structure of a Chubby system

The chubby architecture 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. 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. 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.

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

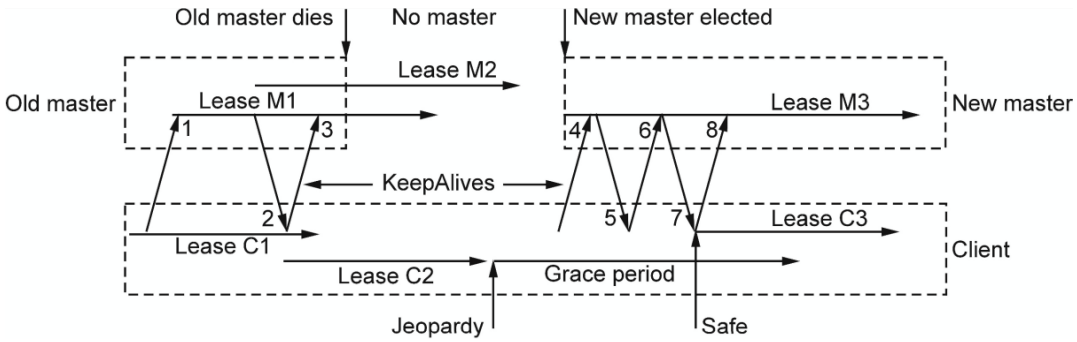
f. Request for a conflicting lock from another client

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.

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.



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(), SetACl(), Delete(), Acquire(), TryAcquire(), Release(), GetSequencer(), SetSequencer(), and CheckSequencer().

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

Release	Releases a lock on a file

APIs in Chubby

Google APIs

Google developed a set of Application Programming Interfaces (APIs) that can be used to communicate with Google Services. This set of APIs is referred as Google APIs. and their integration to other services.

A set of libraries, tools, and capabilities that can be used to generate client libraries and APIs from an App Engine application is known as Google Cloud Endpoints. It eases the data accessibility for client applications. We can also save the time of writing the network communication code by using Google Cloud Endpoints that can also generate client libraries for accessing the backend API.