The Google File System

Summary:

Abstract

The great success of Google Inc. is attributed not only to its efficient search algorithm, but also to the underlying commodity hardware and, thus the file system. As the number of applications run by Google increased massively, Google's goal became to build a vast storage network out of inexpensive commodity hardware. Google created its own file system, named as Google File System.

Google File system is the largest file system in operation. Formally, Google File System (GFS) is a scalable distributed file system for large distributed data intensive applications. In the design phase of GFS, points which were given stress includes component failures are the norm rather than the exception, files are huge in the order of MB & TB and files are mutated by appending data. The entire file system is organized hierarchically in directories and identified by pathnames. The architecture comprises of a single master, multiple chunk servers and multiple clients. Files are divided into chunks, which is the key design parameter. Google File System also uses leases and mutation order in their design to achieve consistency and atomicity. As of fault tolerance, GFS is highly available, replicas of chunk servers and master exists.

Assumptions

In designing a file system for Google's needs, they have been guided by assumptions that offer both challenges and opportunities.

• The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.

• The system stores a modest number of large files. Usually a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported but need not optimize for them.

• The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.

• The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.

• The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. The files are often used as producer consumer queues or for many-way merging. Hundreds of producers, running one per machine, will concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.

• High sustained bandwidth is more important than low latency. Most of the target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write

Google File System Architecture

A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients. The basic analogy of GFS is master maintains the metadata; client contacts the master and retrieves the metadata about chunks that are stored in chunkservers; next time, client directly contacts the chunkservers. Figure 1 describes these steps more clearly.

Each of these is typically a commodity Linux machine running a user-level server process. Files are divided into fixed-size chunks. Each chunk is identified by an immutable and globally unique 64-bit chunk handle assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, three replicas are stored, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers.

The master periodically communicates with each chunkserver in Heartbeat messages to give it instructions and collect its state. GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached.

Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory. Before going into basic distributed file system operations like read, write, we will discuss the concept of chunks, metadata, master, and will also describe how master and chunkservers communicates.

Leases and Mutation:

mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk's replicas. Leases are used to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the primary. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the Heartbeat messages regularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

Write algorithm is similar to Read algorithm, in terms of contacts between client, master, and chunkservers. Google keeps at least three replicas of each chunks, so in Read, we just read from one of the chunkservers, but in case of Write, it has to write in all three chunkservers, this is the main difference between read and write.

Following is the algorithm with related figures for the Write operation.

1. Application originates the request

2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master

3. Master responds with chunk handle and (primary + secondary) replica locations

4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers

Conclusion

While some design decisions are specific to the unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness. Google started work on GFS by reexamining traditional file system assumptions in light of current and anticipated application workloads and technological environment. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequentially), and both extend and relax the standard file system interface to improve the overall system.

The system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, check summing is used to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system. The design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks.

This is achieved by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, centralized master that does not become a bottleneck. GFS has successfully met the storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables Google to continue to innovate and attack problems on the scale of the entire web.