Big Data Paper Summary

Rebecca Murphy

12/7/14

The Google File System:

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

A Comparison of Approaches to Large-Scale Data Analysis:

Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Michael Stonebraker

Main Idea of First Paper

The Google File system is designed to provide efficient, reliable access to data using large clusters of commodity hardware. Due to the use of cheap commodity computers for clusters, precautions must be taken against the individual nodes high failure rate. The Google File system has a relaxed consistency model that supports Google's highly distributed applications well but remains relatively simple and efficient to implement.

There are several differences between the GFS and most other file systems. First, because of the heap commodity computers, hardware failure is the norm rather than the exception, and this part of the design makes constant monitoring, error detection, fault tolerance, and automatic recovery integral. Second file sizes are huge by traditional standards. Third, most files are mutated or appended with new data instead of being written over with new data. Once written files are only read, and often sequentially, and given this the most appending becomes the focus of performance optimization and atomicity guarantees, while caching data block in the client becomes less useful.

How the idea is implemented

A Google File System cluster consists of multiple nodes. There are two types of nodes: one Master node and a large number of Chunkservers. Each chunk is assigned a unique 64 bit by the Master when their created. The Google File System makes use of large chuck size with lazy allocation. The master that stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas.

The Master server does not usually store the actual chunks, but all the metadata associated with the chunks. There are three types of metadata file and chunk namespaces, file-to-chunk mapping, and the locations of each chunk's replicas. The first two types are also kept persistent by logging mutations to an operation log stored on the Master's local disk and replicated on remote machines. This log allows the master state to update simply, reliably, and without risking inconsistencies in event of a master crash. Permissions for modifications are handed by a system of time-limited, expiring "leases" where the Master server grants permission to a process for a finite period of time. The changes are not saved until all Chunkservers acknowledge, which guarantees the completion and atomity of the operation. This is also known as namespace locking.

Analysis of Idea and Implementation

I find it odd that they chose to use cheap component computers and design around not just compensating for failure, but making failure a part of the system. I understand that it works because the master is constantly updating with the Heartbeat messages, but it's still such an odd concept. I also find it interesting that they co-designed the applications and the file system API, and that the file system doesn't run in the kernel but in the user namespace.

One thing I don't like about the implement is that there is no difference between a normal and an abnormal termination. I thinks good to have a record of abnormal termination, in case it was causing a problem.

The cool thing is that there's high aggregate throughput to many concurrent readers and writers performing a variety of tasks. Which means a lot of users can use the same files are the same time with out everything going wrong.

Comparison

The Google File System makes use of the in cluster design described in the Comparison paper. In MapReduce if a node falls silent for longer than the interval it is expected to report back periodically with completed work and status updates, the master node, which is similar to the Master in the Google File System, records the node as dead and sends out the node's assigned work to other nodes. In the Google File System to compensate for failure, fast recovery, and chunk replication. The master replicas a chunk three times as a default and clones existing replicas as needed. As chunkservers go offline or detect corrupted replications, which are used if the original chunk is dead.

MapReduce is able to recover from faults in the middle of query executing. I believe the Google File System recovery is as automatic as possible but I do not think it is able to recover data if the middle of an operation failed. I think what would happen is the chunk which failed during the operation, the operation would restart on a replicate of the original chunk before it failed (or perhaps the last time the Heartbeat message checked in with the Master?)

Advantages & Disadvantages

Advantages

- Simplified for API and application use.
- Atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them.
- The system is built from many inexpensive commodity components.
- Large chunck size with lazy space allocation reduces clients need to interact with the master, it can reduce network overload, and it reduces the size of metadata stored on master.

Disadvantages

- Component failures are the norm rather than the exception, which means there might not be operation recovery.
- File sizes are huge.
- It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- Does not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process.