results by the use of table boundaries detection techniques and the use of text post-processing techniques to detect the noise and to correct bad-recognized words.

Appendix

OCR:- Optical character recognition, NCM:- normalized central moments, PA:-Principle angle
E-GOV :- Electronic governance,GUI:- Graphical user interface

Apache Hadoop Goes Realtime at Facebook

***Prof.Komal Shringare**

Abstract

Facebook recently deployed Facebook Messages, its first ever user-facing application built on the Apache Hadoop platform. Apache HBase is a database-like layer built on Hadoop designed to support billions of messages per day. This paper describes the reasons why Facebook chose Hadoop and HBase over other systems such as Apache Cassandra and Voldemort and discusses the applicationBs requirements for consistency, availability, partition tolerance, data model and scalability. I explore the enhancements made to Hadoop to make it a more effective realtime system, the tradeoffs we made while configuring the system, and how this solution has significant advantages over the sharded MySQL database scheme used in other applications at Facebook and many other web-scalecompanies. I discuss the motivations behind my design choices, the challenges that we face in day-to-day operations, and future capabilities and improvements still under development.I offer these observations on the deployment as a model for other companies who are contemplating a Hadoop-based solution over traditional sharded RDBMS deployments.

Keywords: Data, scalability, resource sharing, distributed file system, Hadoop, Hive, HBase, Facebook, Scribe, log aggregation, distributed systems.

Introduction

Apache Hadoop [1] is a top-level Apache project that includes open source implementations of a distributed file system [2] and MapReduce that were inspired by GoogleBs GFS [5] and MapReduce [6] projects. The Hadoop ecosystem also includes projects like Apache HBase [4] which is inspired by GoogleBs BigTable, Apache Hive [3], a data warehouse built on top of Hadoop, and Apache ZooKeeper [7], a coordination service for distributed systems.

At Facebook, Hadoop has traditionally been used in conjunction with Hive for storage and analysis of large data sets. Most of this analysis occurs in offline batch jobs and the emphasis has been on maximizing throughput and efficiency. These workloads typically read and write large amounts of data from disk sequentially. As such, there has been less emphasis on making Hadoop performant for random access workloads by providing low latency access to HDFS. Instead, I have used a combination of large clusters of MySQL databases and caching tiers built using memcached[8]. In many cases, results from Hadoop are uploaded into MySQL or memcached for consumption by the web tier.

The first set of applications requires realtime concurrent, but sequential, read access to a very large stream of realtime data being stored in HDFS. An example system generating and storing such data is Scribe [9], an open source distributed log aggregation service created by and used extensively at Facebook. Previously, data generated by Scribe was stored in expensive and hard to manage NFS servers. Two main applications that fall into this category are Realtime Analytics

[10] and MySQL backups. I have enhanced HDFS to become a high performance low latency file system and have been able to reduce our use of expensive file servers.

Why HADOOP AND HBASE

The requirements for the storage system from the workloads presented above can be summarized as follows (in no particular order):

- Elasticity: We need to be able to add incremental capacity to our storage systems with minimal overhead and no downtime. In some cases we may want to add capacity rapidly and the system should automatically balance load and utilization across new hardware.
- High write throughput: Most of the applications store (and optionally index) tremendous amounts of data and require high aggregate write throughput.
- Efficient and low-latency strong consistency semantics within a data center: There are important applications like Messages that require strong consistency within a data center. This requirement often arises directly from user expectations. For example iunreadB message counts displayed on the home page and the messages shown in the inbox page view should be consistent with respect to each other. While a globally distributed strongly consistent system is practically impossible, a system that could at least provide strong consistency within a data center would make it possible to provide a good user experience. We also knew that (unlike other Facebook applications), Messages was easy to federate so that a particular user could be served entirely out of a single data center making strong consistency within a single data center a critical requirement for the Messages project. Similarly, other projects, like realtime log aggregation, may be deployed entirely within one data center and are much easier to program if the system provides strong consistency guarantees.
- Efficient random reads from disk: In spite of the widespread use of application level caches (whether embedded or via memcached), at Facebook scale, a lot of accesses miss the cache and hit the back-end storage system. MySQL is very efficient at performing random reads from disk and any new system would have to be comparable.
- High Availability and Disaster Recovery: We need to provide a service with very high uptime to users that covers both planned and unplanned events (examples of the former being events like software upgrades and addition of hardware/capacity and the latter exemplified by failures of hardware components). We also need to be able to tolerate the loss of a data center with minimal data loss and be able to serve data out of another data center in a reasonable time frame.
- Fault Isolation: Our long experience running large farms of MySQL databases has shown us that fault isolation is critical. Individual databases can and do go down, but only a small fraction of users are affected by any such event. Similarly, in our warehouse usage of Hadoop, individual disk failures affect only a small part of the data and the system quickly recovers from such faults.
- Atomic read-modify-write primitives: Atomic increments and compare-and-swap APIs have been very useful in building lockless concurrent applications and are a must have from the underlying storage system.

- Range Scans: Several applications require efficient retrieval of a set of rows in a particular range. For example all the last 100 messages for a given user or the hourly impression counts over the last 24 hours for a given advertiser.

Realtime HDFS

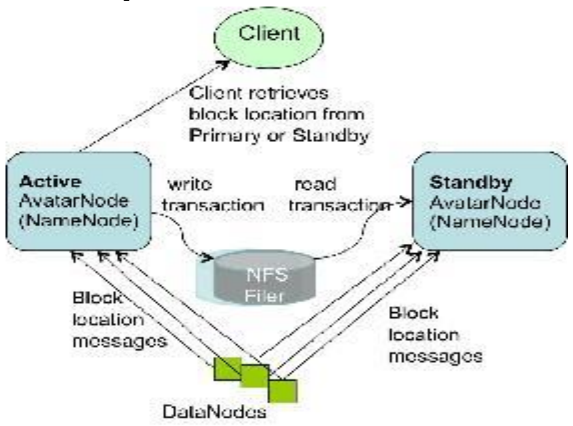
HDFS was originally designed to be a file system to support offline MapReduce application that are inherently batch systems and where scalability and streaming performance are most critical. I have seen the advantages of using HDFS: its linear scalability and fault tolerance results in huge cost savings across the enterprise. The new, more realtime and online usage of HDFS push new requirements and now use HDFS as a general-purposelow-latency file system.

High Availability - AvatarNode

The design of HDFS has a single master b the NameNode. Whenever the master is down, the HDFS cluster is unusable until the NameNode is back up. This is a single point of failure and is one of the reason why people are reluctant to deploy HDFS for an application whose uptime requirement is 24x7. In our experience, I have seen that new software upgrades of our HDFS server software is the primary reason for cluster downtime. Since the hardware is not entirely unreliable and the software is well tested before it is deployed to production clusters, in our four years of administering HDFS clusters, we have encountered only one instance when the NameNode crashed, and that happened because of a bad filesystem where the transaction log was stored.

Hot Standby - AvatarNode

At startup time, the HDFS NameNode reads filesystem metadata from a file called the fsimage file. This metadata contains the names and metadata of every file and directory in HDFS. However, the NameNode does not persistently store the locations of each block. Thus, the time to cold-start a NameNode consists of two main parts: firstly, the reading of the file system image, applying the transaction log and saving the new file system image back to disk; and secondly, the processing of block reports from a majority of DataNodes to recover all known block locations of every block in the cluster. Our biggest HDFS cluster [12] has about 150 million files and I see that the two above stages take an equal amount of time


Client
Client retrieves block location from Primary or Standby
Active AvatarNode (NameNode)
write transaction
read transaction
Standby AvatarNode (NameNode)
NFS Filer
Block location messages
Block location messages
DataNodes

A HDFS cluster has two AvatarNodes: the Active AvatarNode and the Standby AvatarNode. They form an active-passive-hot-standby pair. An AvatarNode is a wrapper around a normal

NameNode. All HDFS clusters at Facebook use NFS to store one copy of the filesystem image and one copy of the transaction log. The Active AvatarNode writes its transactions to the transaction log stored in a NFS filesystem. At the same time, the Standby opens the same transaction log for reading from the NFS file system and starts applying transactions to its own namespace thus keeping its namespace as close to the primary as possible. The Standby AvatarNode also takes care of check-pointing the primary and creating a new filesystem image so there is no separate Secondary Name Node anymore.
The DataNodes talk to both Active AvatarNode and Standby AvatarNode instead of just talking to a single NameNode. That means that the Standby AvatarNode has the most recent state about block locations as well and can become Active in well under a minute. The Avatar DataNode sends heartbeats, block reports and block received to both AvatarNodes. AvatarDataNodes are integrated with ZooKeeper and they know which one of the AvatarNodes serves as the primary and they only process replication/deletion commands coming from the primary AvatarNode. Replication or deletion requests coming from the Standby AvatarNode are ignored.

Enhancements to HDFS transaction logging

HDFS records newly allocated block-ids to the transaction log only when the file is closed or sync/flushed. Since I wanted to make the failover as transparent as possible, the Standby has to know of each block allocation as it happens, so I write a new transaction to the edits log on each block allocation. This allows a client to continue writing to files that it was writing at the moment just before the failover.
When the Standby reads transactions from the transaction log that is being written by the Active AvatarNode, there is a possibility that it reads a partial transaction. To avoid this problem I had to change the format of the edits log to have a transaction length, transaction id and the checksum per each transaction written to the file.

Transparent Failover: DAFS

I developed a DistributedAvatarFileSystem (DAFS), a layered file system on the client that can provide transparent access to HDFS across a failover event. DAFS is integrated with ZooKeeper. ZooKeeper holds a zNode with the physical address of the Primary AvatarNode for a given cluster. When the client is trying to connect to the HDFS cluster (e.g. dfs.cluster.com), DAFS looks up the relevant zNode in ZooKeeper that holds the actual address of the Primary AvatarNode (dfs-0.cluster.com) and directs all the succeeding calls to the Primary AvatarNode. If a call encounters a network error, DAFS checks with ZooKeeper for a change of the primary node. In case there was a failover event, the zNone will now contain the name of the new Primary AvatarNode. DAFS will now retry the call against the new Primary AvatarNode. I do not use the ZooKeeper subscription model because it would require much more resources dedicated on ZooKeeper servers. If a failover is in progress, then DAFS will automatically block till the failover is complete. A failover event is completely transparent to an application that is accessing data from HDFS.

Hadoop RPC compatibility Early on, clear that I will be running multiple Hadoop clusters for our Messages application. I needed the capability to deploy newer versions of the software on different clusters at different points in time. This required that I enhance the Hadoop clients to be able to interoperate with Hadoop servers running different versions of the Hadoop software. The various server process within the same cluster run the same version of the software. I enhanced the Hadoop RPC software to automatically determine the version of the software running on the server that it is communicating with, and then talk the appropriate protocol while talking to that server.

Block Availability: Placement Policy

The default HDFS block placement policy, while rack aware, is still minimally constrained. Placement decision for non-local replicas is random, it can be on any rack and within any node of the rack. To reduce the probability of data loss when multiple simultaneous nodes fail, I implemented a pluggable block placement policy that constrains the placement of block replicas into smaller, configurable node groups. This allows us to reduce the probability of data loss by orders of magnitude, depending on the size chosen for the groups. Our strategy is to define a window of racks and machines where replicas can be placed around the original block, using a logical ring of racks, each one containing a logical ring of machines. More details, the math, and the scripts used to calculate these numbers can be found at HDFS-1094[11].I found that the probability of losing a random block increases with the size of the node group. In our clusters, I started to use a node group of (2, 5), i.e. a rack window size of 2 and a machine window size of 5. I picked this choice because the probability of data loss is about a hundred times lesser than the default block placement policy.

Performance Improvements for a Realtime Workload

HDFS is originally designed for high-throughput systems like MapReduce. Many of its original design principles are to improve its throughput but do not focus much on response time. For example, when dealing with errors, it favors retries or wait over fast failures. To support realtime applications, offering reasonable response time even in case of errors becomes the major challenge for HDFS.

RPC Timeout

One example is how Hadoop handles RPC timeout. Hadoop uses tcp connections to send Hadoop-RPCs. When a RPC client detects a tcp-socket timeout, instead of declaring a RPC timeout, it sends a ping to the RPC server. If the server is still alive, the client continues to wait for a response. The idea is that if a RPC server is experiencing a communication burst, a temporary high load, or a stop the world GC, the client should wait and throttles its traffic to the server. On the contrary, throwing a timeout exception or retrying the RPC request causes tasks to fail unnecessarily or add additional load to a RPC server.
However, infinite wait adversely impacts any application that has a real time requirement. An HDFS client occasionally makes an RPC to some Dataode, and it is bad when the DataNode fails to respond back in time and the client is stuck in an RPC. A better strategy is to fail fast and try a different DataNode for either reading or writing. Hence, I added the ability for specifying an RPC-timeout when starting a RPC session with a server. Recover File Lease
Another enhancement is to revoke a writerBs lease quickly. HDFS supports only a single writer to a file and the NameNode maintains leases to enforce this semantic. There are many cases when an application wants to open a file to read but it was not closed cleanly earlier. Previously this

was done by repetitively callingHDFS-*append* on the log file until the call succeeds. The append operations triggers a fileBs soft lease to expire. So the application had to wait for a minimum of the soft lease period (with a default value of one minute) before the HDFS name node revokes the log fileBs lease. Secondly, the HDFS-*append* operation has additional unneeded cost as establishing a write pipeline usually involves more than one DataNode. When an error occurs, a pipeline establishment might take up to 10 minutes.
To avoid the HDFS-append overhead, I added a lightweight HDFS API called recoverLease that revokes a fileBs lease explicitly. When the NameNode receives a recoverLease request, it immediately changes the fileBs lease holder to be itself. It then starts the lease recovery process. The recoverLease rpc returns the status whether the lease recovery was complete. The application waits for a success return code from recoverLeasebefore attempting to read from the file.

Reads from Local Replicas

There are times when an application wants to store data in HDFS for scalability and performance reasons. However, the latency of reads and writes to an HDFS file is an order of magnitude greater than reading or writing to a local file on the machine. To alleviate this problem, I implemented an enhancement to the HDFS client that detects that there is a local replica of the data and then transparently reads data from the local replica without transferring the data via the DataNode. This has resulted in doubling the performance profile of a certain workload that uses HBase.

New Features: HDFS sync

Hflush/sync is an important operation for both HBase and Scribe. It pushes the written data buffered at the client side to the write pipeline, making the data visible to any new reader and increasing the data durability when either the client or any DataNode on the pipeline fails. Hflush/sync is a synchronous operation, meaning that it does not return until an acknowledgement from the write pipeline is received. Since the operation is frequently invoked, increasing its efficiency is important. One optimization I have is to allow following writes to proceed while an Hflush/sync operation is waiting for a reply. This greatly increases the write throughput in both HBase and Scribe where a designated thread invokes Hflush/sync periodically.

Concurrent Readers

I have an application that requires the ability to read a file while it is being written to. The reader first talks to the NameNode to get the meta information of the file. Since the NameNode does not have the most updated information of its last blockBs length, the client fetches the information from one of the DataNodes where one of its replicas resides. It then starts to read the file. The challenge of concurrent readers and writer is how to provision the last chunk of data when its data content and checksum are dynamically changing. I solve the problem by re- computing the checksum of the last chunk of data on demand.

Future work

The use of Hadoop and HBase at Facebook is just getting started and we expect to make several iterations on this suite of technologies and continue to optimize for our applications. As we try to use HBase for more applications, we have discussed adding support for maintenance of secondary indices and summary views in HBase. In many use cases, such derived data and views can be maintained asynchronously. Many use cases benefit from storing a large amount of data in HBaseBs cache and improvements to HBase are required to exploit very large physical memory. The current limitations in this area arise from issues with using an extremely large heap in Java and we are evaluating several proposals like writing a slab allocator in Java or managing memory via JNI. A related topic is exploiting flash memory to extend the HBase cache and we are exploring various ways to utilize it including FlashCache [18].

References

- *Apache Hadoop. Available at http://hadoop.apache.org*
- *Apache HDFS. Available at http://hadoop.apache.org/hdfs*
- *Apache Hive. Available at http://hive.apache.org*
- *Apache HBase. Available at http://hbase.apache.org*
- *The Google File System. Available athttp://labs.google.com/papers/gfs-sosp2003.pdf*
- *MapReduce: Simplified Data Processing on Large Clusters. Available at http://labs.google.com/papers/mapreduce- osdi04.pdf*
- *ZooKeeper: Wait-free coordination for Internet-scalesystems. Available at http://www.usenix.org/events/usenix10/tech/full_papers/Hun t.pdf*
- *Memcached. Available at http://en.wikipedia.org/wiki/Memcached*
- *Scribe. Available at http://github.com/facebook/scribe/wiki*
- *Building Realtime Insights. Available at http://www.facebook.com/note.php?note_id=101501039002 58920*
- *HDFS-1094. Available athttp://issues.apache.org/jira/browse/HDFS-1094.*

Pull out the magnet in you through communication

Prof Ambika Arvind

Abstract

Communication is an instinct of all living things. The most important bearings of communication are best understood when there is a lack of it. The following article discusses how important communication is and why it plays such a vital role in our daily lives in making you a dynamic person.

Introduction

There is more to communication than just talk and gesture. Listening, understanding and interpreting are as much integral to communication as words - verbal, written or gestured. Yes, even gestures in communication play a crucial role in conveying and interpreting the message! Similarly, how we communicate or express ourselves goes a great way towards determining how our expressions are interpreted. To quote Karl Popper, "It is impossible to speak in such a way that you cannot be misunderstood". Faulty or incomplete communication can completely mark the purpose of communicating and may result in damaging consequences. This is where understanding how important communication is and communicating the right way comes into picture. Not everyone is equally endowed with the ability to effectively express himself and this is where the significance of communication skills can be truly fathomed. Communicating the right way is equally important in every walk of like, be it in personal, professional or social life. Every one of us wants to learn how to make people like us. Ever since kindergarten, you would come up with a lot of different ways just to get accepted by other kids. Unfortunately, not all of us have the same success rate. As we grow older, the desire to fit in doesn't fade away. For many people, getting liked by others is as important as the food they eat. Let me tell you a few tips to