Apache Hadoop Goes Realtime at Facebook

Dhruba Borthakur
Kannan Muthukkaruppan
Karthik Ranganathan
Samuel Rash

Joydeep Sen Sarma
Nicolas Spiegelberg
Dmytro Molkov
Rodrigo Schmidt

Jonathan Gray
Hairong Kuang
Aravind Menon
Amitanand Aiyer

Facebook

{dhruba,jssarma,jgray,kannan,
nicolas,hairong,kranganathan,dms,
aravind.menon,rash,rodrigo,
amitanand.s}@fb.com

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 application's requirements for consistency, availability, partition tolerance, data model and scalability. We 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-scale companies. We discuss the motivations behind our design choices, the challenges that we face in day-to-day operations, and future capabilities and improvements still under development. We offer these observations on the deployment as a model for other companies who are contemplating a Hadoop-based solution over traditional sharded RDBMS deployments.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous.

General Terms

Management, Measurement, Performance, Distributed Systems, Design, Reliability, Languages.

Keywords

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

1. 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 Google's GFS [5] and MapReduce [6] projects. The Hadoop ecosystem also includes projects like Apache HBase [4] which is inspired by Google's BigTable, Apache Hive [3], a data warehouse built on top of Hadoop, and Apache ZooKeeper [8], 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, we have used a combination of large clusters of MySQL databases and caching tiers built using memcached[9]. In many cases, results from Hadoop are uploaded into MySQL or memcached for consumption by the web tier.

Recently, a new generation of applications has arisen at Facebook that require very high write throughput and cheap and elastic storage, while simultaneously requiring low latency and disk efficient sequential and random read performance. MySQL storage engines are proven and have very good random read performance, but typically suffer from low random write throughput. It is difficult to scale up our MySQL clusters rapidly while maintaining good load balancing and high uptime. Administration of MySQL clusters requires a relatively high management overhead and they typically use more expensive hardware. Given our high confidence in the reliability and scalability of HDFS, we began to explore Hadoop and HBase for such applications.

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 [10], 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 [11] and MySQL backups. We have enhanced HDFS to become a high performance low latency file system and have been able to reduce our use of expensive file servers.

SIGMOD '11, June 12–16, 2011, Athens, Greece.

The second generation of non-MapReduce Hadoop applications needed to dynamically index a rapidly growing data set for fast random lookups. One primary example of such an application is Facebook Messages [12]. Facebook Messages gives every Facebook user a facebook.com email address, integrates the display of all e-mail, SMS and chat messages between a pair or group of users, has strong controls over who users receive messages from, and is the foundation of a Social Inbox. In addition, this new application had to be suited for production use by more than 500 million people immediately after launch and needed to scale to many petabytes of data with stringent uptime requirements. We decided to use HBase for this project. HBase in turn leverages HDFS for scalable and fault tolerant storage and ZooKeeper for distributed consensus.

In the following sections we present some of these new applications in more detail and why we decided to use Hadoop and HBase as the common foundation technologies for these projects. We describe specific improvements made to HDFS and HBase to enable them to scale to Facebook's workload and operational considerations and best practices around using these systems in production. Finally we discuss ongoing and future work in these projects.

2. WORKLOAD TYPES

Before deciding on a particular software stack and whether or not to move away from our MySQL-based architecture, we looked at a few specific applications where existing solutions may be problematic. These use cases would have workloads that are challenging to scale because of very high write throughput, massive datasets, unpredictable growth, or other patterns that may be difficult or suboptimal in a sharded RDBMS environment.

2.1 Facebook Messaging

The latest generation of Facebook Messaging combines existing Facebook messages with e-mail, chat, and SMS. In addition to persisting all of these messages, a new threading model also requires messages to be stored for each participating user. As part of the application server requirements, each user will be sticky to a single data center.

2.1.1 High Write Throughput

With an existing rate of millions of messages and billions of instant messages every day, the volume of ingested data would be very large from day one and only continue to grow. The denormalized requirement would further increase the number of writes to the system as each message could be written several times.

2.1.2 Large Tables

As part of the product requirements, messages would not be deleted unless explicitly done so by the user, so each mailbox would grow indefinitely. As is typical with most messaging applications, messages are read only a handful of times when they are recent, and then are rarely looked at again. As such, a vast majority would not be read from the database but must be available at all times and with low latency, so archiving would be difficult.

Storing all of a user's thousands of messages meant that we'd have a database schema that was indexed by the user with an ever-growing list of threads and messages. With this type of random write workload, write performance will typically degrade in a system like MySQL as the number of rows in the table increases. The sheer number of new messages would also mean a heavy write workload, which could translate to a high number of random IO operations in this type of system.

2.1.3 Data Migration

One of the most challenging aspects of the new Messaging product was the new data model. This meant that all existing user's messages needed to be manipulated and joined for the new threading paradigm and then migrated. The ability to perform large scans, random access, and fast bulk imports would help to reduce the time spent migrating users to the new system.

2.2 Facebook Insights

Facebook Insights provides developers and website owners with access to real-time analytics related to Facebook activity across websites with social plugins, Facebook Pages, and Facebook Ads.

Using anonymized data, Facebook surfaces activity such as impressions, click through rates and website visits. These analytics can help everyone from businesses to bloggers gain insights into how people are interacting with their content so they can optimize their services.

Domain and URL analytics were previously generated in a periodic, offline fashion through our Hadoop and Hive data warehouse. However, this yields a poor user experience as the data is only available several hours after it has occurred.

2.2.1 Realtime Analytics

The insights teams wanted to make statistics available to their users within seconds of user actions rather than the hours previously supported. This would require a large-scale, asynchronous queuing system for user actions as well as systems to process, aggregate, and persist these events. All of these systems need to be fault-tolerant and support more than a million events per second.

2.2.2 High Throughput Increments

To support the existing insights functionality, time and demographic-based aggregations would be necessary. However, these aggregations must be kept up-to-date and thus processed on the fly, one event at a time, through numeric counters. With millions of unique aggregates and billions of events, this meant a very large number of counters with an even larger number of operations against them.

2.3 Facebook Metrics System (ODS)

At Facebook, all hardware and software feed statistics into a metrics collection system called ODS (Operations Data Store). For example, we may collect the amount of CPU usage on a given server or tier of servers, or we may track the number of write operations to an HBase cluster. For each node or group of nodes we track hundreds or thousands of different metrics, and engineers will ask to plot them over time at various granularities. While this application has hefty requirements for write throughput, some of the bigger pain points with the existing MySQL-based system are around the resharding of data and the ability to do table scans for analysis and time roll-ups.

2.3.1 *Automatic Sharding*

The massive number of indexed and time-series writes and the unpredictable growth patterns are difficult to reconcile on a sharded MySQL setup. For example, a given product may only collect ten metrics over a long period of time, but following a large rollout or product launch, the same product may produce thousands of metrics. With the existing system, a single MySQL server may suddenly be handling much more load than it can handle, forcing the team to manually re-shard data from this server onto multiple servers.

2.3.2 *Fast Reads of Recent Data and Table Scans*

A vast majority of reads to the metrics system is for very recent, raw data, however all historical data must also be available. Recently written data should be available quickly, but the entire dataset will also be periodically scanned in order to perform time-based rollups.

3. WHY HADOOP AND HBASE

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

1. **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.

2. **High write throughput:** Most of the applications store (and optionally index) tremendous amounts of data and require high aggregate write throughput.

3. **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 'unread' 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.

4. **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.

5. **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.

6. **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.

7. **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.

8. **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.

It is also worth pointing out non-requirements:

1. **Tolerance of network partitions within a single data center:** Different system components are often inherently centralized. For example, MySQL servers may all be located within a few racks, and network partitions within a data center would cause major loss in serving capabilities therein. Hence every effort is made to eliminate the possibility of such events at the hardware level by having a highly redundant network design.

2. **Zero Downtime in case of individual data center failure:** In our experience such failures are very rare, though not impossible. In a less than ideal world where the choice of system design boils down to the choice of compromises that are acceptable, this is one compromise that we are willing to make given the low occurrence rate of such events.

3. **Active-active serving capability across different data centers:** As mentioned before, we were comfortable making the assumption that user data could be federated across different data centers (based ideally on user locality). Latency (when user and data locality did not match up) could be masked by using an application cache close to the user.

Some less tangible factors were also at work. Systems with existing production experience for Facebook and in-house expertise were greatly preferred. When considering open-source projects, the strength of the community was an important factor. Given the level of engineering investment in building and maintaining systems like these – it also made sense to choose a solution that was broadly applicable (rather than adopt point solutions based on differing architecture and codebases for each workload).

After considerable research and experimentation, we chose Hadoop and HBase as the foundational storage technology for these next generation applications. The decision was based on the state of HBase at the point of evaluation as well as our confidence in addressing the features that were lacking at that point via in-house engineering. HBase already provided a highly consistent, high write-throughput key-value store. The HDFS NameNode stood out as a central point of failure, but we were confident that our HDFS team could build a highly-available NameNode in a reasonable time-frame, and this would be useful for our warehouse operations as well. Good disk read-efficiency seemed to be within striking reach (pending adding Bloom filters to HBase's version of LSM[13] Trees, making local DataNode reads efficient and caching NameNode metadata). Based on our experience operating the Hive/Hadoop warehouse, we knew HDFS was stellar in tolerating and isolating faults in the disk subsystem. The failure of entire large HBase/HDFS clusters was a scenario that ran against the goal of fault-isolation, but could be considerably mitigated by storing data in smaller HBase clusters. Wide area replication projects, both in-house and within the HBase community, seemed to provide a promising path to achieving disaster recovery.

HBase is massively scalable and delivers fast random writes as well as random and streaming reads. It also provides row-level atomicity guarantees, but no native cross-row transactional support. From a data model perspective, column-orientation gives extreme flexibility in storing data and wide rows allow the creation of billions of indexed values within a single table. HBase is ideal for workloads that are write-intensive, need to maintain a large amount of data, large indices, and maintain the flexibility to scale out quickly.

4. 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. We 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-purpose low-latency file system. In this section, we describe some of the core changes we have made to HDFS to support these new applications.

4.1 High Availability - AvatarNode

The design of HDFS has a single master – 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, we 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.

4.1.1 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 [16] has about 150 million files and we see that the two above stages take an equal amount of time. In total, a cold-restart takes about 45 minutes.

The BackupNode available in Apache HDFS avoids reading the fsimage from disk on a failover, but it still needs to gather block reports from all DataNodes. Thus, the failover times for the BackupNode solution can be as high as 20 minutes. Our goal is to do a failover within seconds; thus, the BackupNode solution does not meet our goals for fast failover. Another problem is that the NameNode synchronously updates the BackupNode on every transaction, thus the reliability of the entire system could now be lower than the reliability of the standalone NameNode. Thus, the HDFS AvatarNode was born.

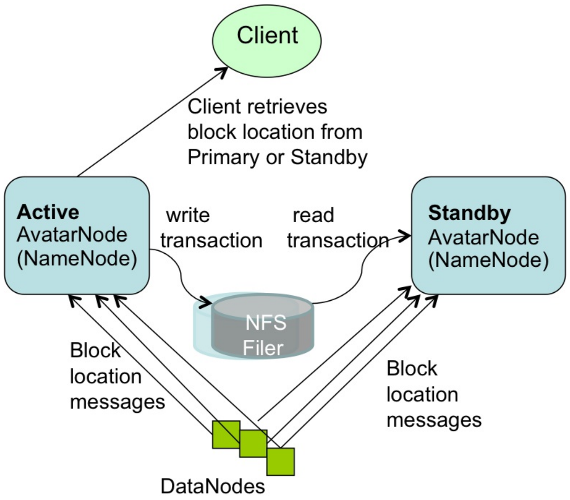
Client
Client retrieves block location from Primary or Standby
Active AvatarNode (NameNode)
write transaction
read transaction
Standby AvatarNode (NameNode)
NFS Filer
Block location messages
Block location messages
DataNodes

Figure 1

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 SecondaryNameNode 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.

4.1.2 *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 we wanted to make the failover as transparent as possible, the Standby has to know of each block allocation as it happens, so we 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 we 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.

4.1.3 *Transparent Failover: DAFS*

We 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. We 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.

4.2 Hadoop RPC compatibility

Early on, we were pretty clear that we will be running multiple Hadoop clusters for our Messages application. We needed the capability to deploy newer versions of the software on different clusters at different points in time. This required that we 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. We 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.

4.3 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, we 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[14]. We found that the probability of losing a random block increases with the size of the node group. In our clusters, we started to use a node group of (2, 5), i.e. a rack window size of 2 and a machine window size of 5. We picked this choice because the probability of data loss is about a hundred times lesser than the default block placement policy.

4.4 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.

4.4.1 *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, we added the ability for specifying an RPC-timeout when starting a RPC session with a server.

4.4.2 *Recover File Lease*

Another enhancement is to revoke a writer's 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 calling HDFS-*append* on the log file until the call succeeds. The append operations triggers a file's 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 file's 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, we added a lightweight HDFS API called *recoverLease* that revokes a file's lease explicitly. When the NameNode receives a *recoverLease* request, it immediately changes the file's 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 *recoverLease* before attempting to read from the file.

4.4.3 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, we 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.

4.5 New Features

4.5.1 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 we 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.

4.5.2 Concurrent Readers

We 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 block's 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. We solve the problem by re-computing the checksum of the last chunk of data on demand.

5. PRODUCTION HBASE

In this section, we'll describe some of the important HBase enhancements that we have worked on at Facebook related to correctness, durability, availability, and performance.

5.1 ACID Compliance

Application developers have come to expect ACID compliance, or some approximation of it, from their database systems. Indeed, strong consistency guarantees was one of the benefits of HBase in our early evaluations. The existing MVCC-like read-write consistency control (RWCC) provided sufficient isolation guarantees and the HLog (write ahead log) on HDFS provided sufficient durability. However, some modifications were necessary to make sure that HBase adhered to the row-level atomicity and consistency of ACID compliance we needed.

5.1.1 Atomicity

The first step was to guarantee row-level atomicity. RWCC provided most of the guarantees, however it was possible to lose these guarantees under node failure. Originally, multiple entries in a single row transaction would be written in sequence to the HLog. If a RegionServer died during this write, the transaction could be partially written. With a new concept of a log transaction (WALEdit), each write transaction will now be fully completed or not written at all.

5.1.2 Consistency

HDFS provides replication for HBase and thus handles most of the strong consistency guarantees that HBase needs for our usage. During writes, HDFS sets up a pipeline connection to each replica and all replicas must ACK any data sent to them. HBase will not continue until it gets a response or failure notification. Through the use of sequence numbers, the NameNode is able to identify any misbehaving replicas and exclude them. While functional, it takes time for the NameNode to do this file recovery. In the case of the HLog, where forward progress while maintaining consistency and durability are an absolute must, HBase will immediately roll the log and obtain new blocks if it detects that even a single HDFS replica has failed to write data.

HDFS also provides protection against data corruption. Upon reading an HDFS block, checksum validation is performed and the entire block is discarded upon a checksum failure. Data discard is rarely problematic because two other replicas exist for this data. Additional functionality was added to ensure that if all 3 replicas contain corrupt data the blocks are quarantined for post-mortem analysis.

5.2 Availability Improvements

5.2.1 HBase Master Rewrite

We originally uncovered numerous issues during kill testing where HBase regions would go offline. We soon identified the problem: the transient state of the cluster is stored in the memory of the currently active HBase master only. Upon losing the master, this state is lost. We undertook a large HBase master rewrite effort. The critical component of this rewrite was moving region assignment information from the master's in-memory state to ZooKeeper. Since ZooKeeper is quorum written to a majority of nodes, this transient state is not lost on master failover and can survive multiple server outages.

5.2.2 Online Upgrades

The largest cause of cluster downtime was not random server deaths, but rather system maintenance. We had a number of problems to solve to minimize this downtime.

First, we discovered over time that RegionServers would intermittently require minutes to shutdown after issuing a stop request. This intermittent problem was caused by long compaction cycles. To address this, we made compactions interruptible to favor responsiveness over completion. This reduced RegionServer downtime to seconds and gave us a reasonable bound on cluster shutdown time.

Another availability improvement was rolling restarts. Originally, HBase only supported full cluster stop and start for upgrades. We added rolling restarts script to perform software upgrades one server at a time. Since the master automatically reassigns regions on a RegionServer stop, this minimizes the amount of downtime that our users experience. We fixed numerous edge case issues that resulted from this new restart. Incidentally, numerous bugs during rolling restarts were related to region offlining and reassignment, so our master rewrite with ZooKeeper integration helped address a number of issues here as well.

5.2.3 Distributed Log Splitting

When a RegionServer dies, the HLogs of that server must be split and replayed before its regions can be reopened and made available for reads and writes. Previously, the Master would split the logs before they were replayed across the remaining RegionServers. This was the slowest part of the recovery process and because there are many HLogs per server, it could be parallelized. Utilizing ZooKeeper to manage the split tasks across RegionServers, the Master now coordinates a distributed log split. This cut recovery times by an order of magnitude and allows RegionServers to retain more HLogs without severely impacting failover performance.

5.3 Performance Improvements

Data insertion in HBase is optimized for write performance by focusing on sequential writes at the occasional expense of redundant reads. A data transaction first gets written to a commit log and then applied to an in-memory cache called MemStore. When the MemStore reaches a certain threshold it is written out as an HFile. HFiles are immutable HDFS files containing key/value pairs in sorted order. Instead of editing an existing HFile, new HFiles are written on every flush and added to a per-region list. Read requests are issued on these multiple HFiles in parallel & aggregated for a final result. For efficiency, these HFiles need to be periodically compacted, or merged together, to avoid degrading read performance.

5.3.1 Compaction

Read performance is correlated with the number of files in a region and thus critically hinges on a well-tuned compaction algorithm. More subtly, network IO efficiency can also be drastically affected if a compaction algorithm is improperly tuned. Significant effort went into making sure we had an efficient compaction algorithm for our use case.

Compactions were initially separated into two distinct code paths depending upon whether they were minor or major. Minor compactions select a subset of all files based on size metrics whereas time-based major compactions unconditionally compact all HFiles. Previously, only major compactions processed deletes, overwrites, and purging of expired data, which meant that minor compactions resulted in larger HFiles than necessary, which decreases block cache efficiency and penalizes future compactions. By unifying the code paths, the codebase was simplified and files were kept as small as possible.

The next task was improving the compaction algorithm. After launching to employees, we noticed that our put and sync latencies were very high. We discovered a pathological case where a 1 GB file would be regularly compacted with three 5 MB files to produce a slightly larger file. This network IO waste would continue until the compaction queue started to backlog. This problem occurred because the existing algorithm would unconditionally minor compact the first four HFiles, while triggering a minor compaction after 3 HFiles had been reached. The solution was to stop unconditionally compacting files above a certain size and skip compactions if enough candidate files could not be found. Afterwards, our put latency dropped from 25 milliseconds to 3 milliseconds.

We also worked on improving the size ratio decision of the compaction algorithm. Originally, the compaction algorithm would sort by file age and compare adjacent files. If the older file was less than 2x the size of the newer file, the compaction algorithm with include this file and iterate. However, this algorithm had suboptimal behavior as the number and size of HFiles increased significantly. To improve, we now include an older file if it is within 2x the aggregate size of all newer HFiles. This transforms the steady state so that an old HFile will be roughly 4x the size of the next newer file, and we consequently have a steeper curve while still maintaining a 50% compaction ratio.

5.3.2 Read Optimizations

As discussed, read performance hinges on keeping the number of files in a region low thus reducing random IO operations. In addition to utilizing comapctions to keep the number of files on disk low, it is also possible to skip certain files for some queries, similarly reducing IO operations.

Bloom filters provide a space-efficient and constant-time method for checking if a given row or row and column exists in a given HFile. As each HFile is written sequentially with optional metadata blocks at the end, the addition of bloom filters fit in without significant changes. Through the use of folding, each bloom filter is kept as small as possible when written to disk and cached in memory. For queries that ask for specific rows and/or columns, a check of the cached bloom filter for each HFile can allow some files to be completely skipped.

For data stored in HBase that is time-series or contains a specific, known timestamp, a special timestamp file selection algorithm was added. Since time moves forward and data is rarely inserted at a significantly later time than its timestamp, each HFile will generally contain values for a fixed range of time. This information is stored as metadata in each HFile and queries that ask for a specific timestamp or range of timestamps will check if the request intersects with the ranges of each file, skipping those which do not overlap.

As read performance improved significantly with HDFS local file reads, it is critical that regions are hosted on the same physical nodes as their files. Changes have been made to retain the assignment of regions across cluster and node restarts to ensure that locality is maintained.

6. DEPLOYMENT AND OPERATIONAL EXPERIENCES

In the past year, we have gone from running a small HBase test cluster with 10 nodes to many clusters running thousands of nodes. These deployments are already serving live production traffic to millions of users. During the same time frame, we have iterated rapidly on the core software (HBase/HDFS) as well as the application logic running against HBase. In such a fluid environment, our ability to ship high quality software, deploy it correctly, monitor running systems and detect and fix any anomalies with minimal downtime are critical. This section goes into some of the practices and tools that we have used during this evolution.

6.1 Testing

From early on in our design of an HBase solution, we were worried about code stability. We first needed to test the stability and durability of the open source HBase code and additionally ensure the stability of our future changes. To this end, we wrote an HBase testing program. The testing program generated data to write into HBase, both deterministically and randomly. The tester will write data into the HBase cluster and simultaneously read and verify all the data it has added. We further enhanced the tester to randomly select and kill processes in the cluster and verify that successfully returned database transactions were indeed written. This helped catch a lot of issues, and is still our first method of testing changes.

Although our common cluster contains many servers operating in a distributed fashion, our local development verification commonly consists of unit tests and single-server setups. We were concerned about discrepancies between single-server setups and truly distributed scenarios. We created a utility called HBase Verify to run simple CRUD workloads on a live server. This allows us to exercise simple API calls and run load tests in a couple of minutes. This utility is even more important for our dark launch clusters, where algorithms are first evaluated at a large scale.

6.2 Monitoring and Tools

As we gained more experience with production usage of HBase, it became clear that our primary problem was in consistent assignment of regions to RegionServers. Two RegionServers could end up serving the same region, or a region may be left unassigned. These problems are characterized by inconsistencies in metadata about the state of the regions that are stored in different places: the META region in HBase, ZooKeeper, files corresponding to a region in HDFS and the in-memory state of the RegionServers. Even though many of these problems were solved systematically and tested extensively as part of the HBase Master rewrite (see Section 5.2.1), we were worried about edge cases showing up under production load. To that end, we created HBCK as a database-level FSCK [17] utility to verify the consistency between these different sources of metadata. For the common inconsistencies, we added an HBCK 'fix' option to clear the in-memory state and have the HMaster reassign the inconsistent region. Nowadays we run HBCK almost continuously against our production clusters to catch problems as early as possible.

A critical component for cluster monitoring is operational metrics. In particular, RegionServer metrics are far more useful for evaluating the health of the cluster than HMaster or ZooKeeper metrcs. HBase already had a number of metrics exported through JMX. However, all the metrics were for short-running operations such as log writes and RPC requests. We needed to add metrics to monitor long-running events such as compactions, flushes, and log splits. A slightly innocuous metric that ended up being critical for monitoring was version information. We have multiple clusters that often have divergent versions. If a cluster crash happens, we need to understand if any functionality was specific to that cluster. Also, rolling upgrades mean that the running version and the installed version are not necessarily the same. We therefore keep track of both versions and signify when they are different.

6.3 Manual versus Automatic Splitting

When learning a new system, we needed to determine which features we should utilize immediately and which features we could postpone adopting. HBase offers a feature called automatic splitting, which partitions a single region into 2 regions when its size grows too large. We decided that automatic splitting was an optional feature for our use case and developed manual splitting utilities instead. On table creation, we pre-split a table into a specific number of equally sized regions. When the average region size becomes too large, we initiate rolling splits of these regions. We found a number of benefits from this protocol.

Since our data grows roughly uniform across all regions, it's easy for automatic splitting to cause split and compaction storms as the regions all roughly hit the same data size at the same time. With manual splits, we can stagger splits across time and thereby spread out the network IO load typically generated by the splitting process. This minimizes impact to production workload.

Since the number of regions is known at any given point in time, long-term debugging and profiling is much easier. It is hard to trace the logs to understand region level problems if regions keep splitting and getting renamed.

When we first started using HBase, we would occasionally run into problems with Log Recovery where some log files may be left unprocessed on region failover. Manual post-mortem recovery from such unexpected events is much easier if the regions have not been split (automatically) since then. We can go back to the affected region and replay unprocessed logs. In doing this, we also leverage the Trash facility in HDFS that retains deleted files for a configurable time period.

An obvious question emerges: doesn't manual splitting negate one of the main benefits of HBase? One of the advantages with HBase is that splitting is logical not physical. The shared storage underneath (HDFS) allows easy reassignment of regions without having to copy or move around large datasets. Thus, in HBase, an easy way to shed load isn't to create more regions but to instead add more machines to the cluster. The master would automatically reassign existing regions to the new RegionServers in a uniform manner without manual intervention. In addition, automatic splitting makes sense in applications that don't have uniform distribution and we plan to utilize it in the future for these.

6.4 Dark Launch

Migrating from a legacy messaging system offered one major advantage: real-world testing capability. At Facebook, we widely use a testing/rollout process called "Dark Launch" where critical back-end functionality is exercised by a subset of the user base without exposing any UI changes to them [15]. We used this facility to double-write messaging traffic for some users to both the legacy infrastructure and HBase. This allowed us to do useful performance benchmarks and find practical HBase bottlenecks instead of relying purely on artificial benchmarks and estimations. Even after product launch, we still found many uses for Dark Launch clusters. All code changes normally spend a week running on Dark Launch before a production push is considered. Additionally, Dark Launch normally handles at least 2x the load that we expect our production clusters to handle. Long term testing at 2x load allows us to weather multiple traffic spikes and verify that HBase can handle outlier peak conditions before we vertically scale.

6.5 Dashboards/ODS integration

We have metrics being exported by JMX, but we needed an easy way to visualize these metrics and analyze cluster health over time. We decided to utilize ODS, an internal tool similar to Ganglia, to visualize important metrics as line graphs. We have one dashboard per cluster, which contains numerous graphs to visualize average and outlier behavior. Graphing min/max is vital because it identifies misbehaving RegionServers, which may cause the application server processing queue to congest. The greatest benefit is that we can observe statistics in realtime to observe how the cluster reacts to any changes in the workload (for example, running a Hadoop MapReduce job or splitting regions).

Additionally, we have a couple different cross-cluster dashboards that we use for high-level analysis. We place vital stats of all clusters in a single overview dashboard to provide a broad health snapshot. In this dashboard, we currently display four HBase-specific graphs: Get/Put Latency, Get/Put Count, Files per Store, and Compaction Queue size. We also realized after exceeding a half-dozen clusters that we needed some way to visualize our version differences. We display the HMaster version, HDFS Client version, NameNode version, and JobTracker version for each cluster on 4 different heat maps. This allows us to scan our versions for consistency and sorting allows us to identify legacy builds that may have known bugs.

6.6 Backups at the Application layer

How do we take regular backups of this large dataset? One option is to copy and replicate the data from one HDFS cluster to another. Since this approach is not continuous, there is a possibility that data is already corrupt in HDFS before the next backup event. This is obviously not an acceptable risk. Instead, we decided to enhance the application to continuously generate an alternate application log. This log is transported via Scribe and stored in a separate HDFS cluster that is used for web analytics. This is a reliable and time-tested data capture pipeline, especially because we have been using the same software stack to capture and transport huge volumes of click-logs from our web application to our Hive analytics storage. The records in this application log are idempotent, and can be applied multiple times without any data loss. In the event of a data loss problem in HBase, we can replay this log-stream and regenerate the data in HBase.

6.7 Schema Changes

HBase currently does not support online schema changes to an existing table. This means that if we need to add a new column family to an existing table, we have to stop access to the table, disable the table, add new column families, bring the table back online and then restart the load. This is a serious drawback because we do not have the luxury of stopping our workload. Instead, we have pre-created a few additional column families for some our core HBase tables. The application currently does not store any data into these column families, but can use them in the future.

6.8 Importing Data

We initially imported our legacy Message data into HBase by issuing normal database puts from a Hadoop job. The Hadoop job would saturate the network IO as put requests were sent across servers. During alpha release we observed that this method would create over 30 minutes of severe latency as the import data intermixed with the live traffic. This kind of impact was not acceptable—we needed the ability to import data for millions of users without severely affecting latencies for production workload. The solution was switching to a bulk import method with compression. The Bulk Import method partitions data into regions using a map job and the reducer writes data directly to LZO-compressed HFiles. The main cause of network traffic would then be the shuffle of the map output. This problem was solved by GZIP compressing the intermediate map output.

6.9 Reducing Network IO

After running in production for a couple months, we quickly realized from our dashboards that we were network IO bound. We needed some way to analyze where our network IO traffic was coming from. We utilized a combination of JMX statistics and log scraping to estimate total network IO on a single RegionServer for a 24-hour period. We broke down the network traffic across the MemStore flush (15%), size-based minor compactions (38%), and time-based major compactions (47%). We found a lot of low-hanging optimizations by observing these ratios. We were able to get 40% network IO reduction by simply increasing our major compaction interval from every day to every week. We also got big gains by excluding certain column families from being logged to the HLog. Best effort durability sufficed for data stored in these column families.

7. 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 HBase's 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]. Finally, as we try to use Hadoop and HBase for applications that are built to serve the same data in an active-active manner across different data centers, we are exploring approaches to deal with multi data-center replication and conflict resolution.

8. ACKNOWLEDGEMENTS

The current state of the Hadoop Realtime Infrastructure has been the result of ongoing work over the last couple of years. During this time a number of people at Facebook have made significant contributions and enhancements to these systems. We would like to thank Scott Chen and Ramkumar Vadalli for contributing a number of enhancements to Hadoop, including HDFS AvatarNode, HDFS RAID, etc. Also thanks are due to Andrew Ryan, Matthew Welty and Paul Tuckfield for doing a lot of work on operations, monitoring and the statistics setup that makes these tasks easy. Thanks are also due to Gautam Roy, Aron Rivin, Prakash Khemani and Zheng Shao for their continued support and enhancements to various pieces of the storage stack. Acknowledgements are also due to Patrick Kling for implementing a test suite for HDFS HA as part of his internship at Facebook. Last but not the least, thanks are also due to the users of our infrastructure who have patiently dealt with periods of instability during its evolution and have provided valuable

feedback that enabled us to make continued improvements to this infrastructure.

9. REFERENCES

[1] Apache Hadoop. Available at http://hadoop.apache.org

[2] Apache HDFS. Available at http://hadoop.apache.org/hdfs

[3] Apache Hive. Available at http://hive.apache.org

[4] Apache HBase. Available at http://hbase.apache.org

[5] The Google File System. Available at http://labs.google.com/papers/gfs-sosp2003.pdf

[6] MapReduce: Simplified Data Processing on Large Clusters. Available at http://labs.google.com/papers/mapreduce-osdi04.pdf

[7] BigTable: A Distributed Storage System for Structured Data. Available at http://labs.google.com/papers/bigtable-osdi06.pdf

[8] ZooKeeper: Wait-free coordination for Internet-scale systems. Available at http://www.usenix.org/events/usenix10/tech/full_papers/Hunt.pdf

[9] Memcached. Available at http://en.wikipedia.org/wiki/Memcached

[10] Scribe. Available at http://github.com/facebook/scribe/wiki

[11] Building Realtime Insights. Available at http://www.facebook.com/note.php?note_id=10150103900258920

[12] Seligstein, Joel. 2010. Facebook Messages. Available at http://www.facebook.com/blog.php?post=452288242130

[13] Patrick O'Neil and Edward Cheng and Dieter Gawlick and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree)

[14] HDFS-1094. Available at http://issues.apache.org/jira/browse/HDFS-1094.

[15] Facebook Chat. https://www.facebook.com/note.php?note_id=14218138919

[16] Facebook has the world's largest Hadoop cluster! Available at http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html

[17] Fsck. Available at http://en.wikipedia.org/wiki/Fsck

[18] FlashCache. Available at https://github.com/facebook/flashcache