Storage Infrastructure Behind Facebook Messages Using HBase at Scale

Amitanand Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania
Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan
Nicolas Spiegelberg, Liyin Tang, Madhuwanti Vaidya[*]
Facebook, Inc.

Abstract

Facebook Messages, which combines messages, chat and email into a real-time conversation, is the first application in Facebook to use HBase in production. In this article, we will discuss why we chose HBase for this use case, the early improvements we did to make HBase production ready, engineering and operational challenges encountered along the way, and the continued work we have had to do once in production, to improve HBase's efficiency and reliability. We will also describe some of the other use cases of HBase at Facebook.

1 Introduction

In 2009, discussion started around possible storage solutions for the next version of the Facebook Messages product. This version of the product would bring together chat, email and the earlier version of messages product into one umbrella. Chat messages were about 20 times the number of traditional messages, and email messages were likely to be much larger in terms of raw size. Previously, chat messages were held in memory and stored only for a small number of days, and the non-chat messages were stored in MySQL. The new product requirements meant that we needed a storage system that could efficiently support a 20x increase in the amount of writes, in addition to providing cheap and elastic storage for datasets that we were expecting to grow at 150TB+/month.

2 Why we picked HBase for Facebook Messages

HBase is a large-scale, distributed database built on top of the Hadoop Distributed File System (HDFS). It is an Apache open source project, and is modeled after BigTable. In this section, we'll describe our reasons for choosing HBase as the storage solution for the new Facebook messages.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]In alphabetical order

2.1 High write throughput

For a use case like Messages, clustering the message data on disk by userid as the primary dimension, and time or other properties such as thread id or word in the message (for search index purposes) as the secondary dimension is a reasonable schema choice since we want to able to retrieve the messages for a given user efficiently.

Traditional RDBMSs use a single mutating B-Tree as the on-disk data structure for each index. When backed by a traditional RDBMS, the Messages workload would cause a lot of random writes on the B-Tree index since the index is clustered by user id, and several users will receive messages at any given time. With ever increasing data sets, this approach incurs a substantial penalty as, over time, each write requires updating potentially a different block of the B-Tree. In addition, the in-place modification of the B-Tree incurs a read penalty on every write because the block being updated first needs to be read.

HBase's Log Structured Merge Tree (LSM) approach to updating its on-disk data makes it very efficient at handling random writes. HBase first collects all updates in a memory data structure and then periodically flushes the in-memory structure to disk creating an immutable index-organized data file (or HFile). Over time several immutable indices get created on disk, and they are merged in the background. In this approach, all writes to the disk are large sequential writes; and the saved disk iops can be in turn be used to serve reads!

2.2 Low latency random reads

Even though we planned to front the storage system with an application level caching layer for Facebook Messages, the storage system needed to be able to serve any misses with reasonably low latencies. HBase offered acceptable performance to begin with, and we knew that even the implementation wasn't as efficient out-of-the-box, we would be able to optimize reads a lot further– especially for read-recent type of workload using techniques like bloom filters, time range hints and lazy seeks.

2.3 Elasticity

We expected the Facebook Messages data sets to constantly grow over time. Being able to add incremental capacity to the system easily and without downtime was important. Both HBase and HDFS are systems built with elasticity as a fundamental design principle.

2.4 Cheap and fault tolerant

We expected the data sizes to reach several petabytes pretty quickly. Using cheap disks and commodity hardware would be the most cost effective way to scale. But failures are the norm with commodity hardware. Any solution must be resilient to node failures and should require no operational intervention. HBase over HDFS provides a very cheap, elastic and fault tolerant storage solution.

2.5 Strong consistency within a data center

Strong consistency guarantees, such as being able to read your own writes, or reading a consistent value irrespective of the replica you read from, was a requirement for the Messages product. In fact, for a wide class of applications, this model is much easier to work with. Strong consistency is another attractive property that HBase provides.

2.6 Experience with HDFS

HDFS, the underlying distributed filesystem used by HBase, provides several nice features such as replication, end-to-end checksums, and automatic rebalancing of data. Additionally, our technical teams already had a lot of development and operational expertise in HDFS from data processing with Hadoop.

3 Pre-flight preparations

HBase was already a reasonably functional and feature rich system when we started experimenting with it in 2009. For instance, it already had features like automatic load balancing and failover, compression, and map reduce integration. However, there was also a fair amount of correctness, reliability and performance work that we needed to do before we could deploy it at scale. In this section, we'll describe some of the early work we did at Facebook on HBase and HDFS in order to get HBase ready for production use.

3.1 Correctness Fixes

HBase keeps its transaction log in HDFS– but HDFS at the time didn't provide a way to guarantee that the contents appended to a file were flushed/synced to multiple nodes. This was one of the reasons why HBase wasn't popular as a storage solution for online user facing workloads with strict data durability requirements. Adding sync support to HDFS, in order to ensure durability of committed transactions, was one of the first features that we worked on.

HBase promises row-level transactional guarantees. An insert involving multiple column families should be done atomically as long as all the records belong to the same row– either all the changes must persist or none. However, the initial implementation of the commit log allowed this property to be violated. We enhanced the commit log implementation to support atomic multi-column family operations.

We also fixed several bugs in the recovery (commit log replay) code, such as a bug that could cause deleted items to be resurrected after a node failure, or cause transactions in the commit log to be skipped because of incorrect handling of transaction sequence ids.

3.2 New/Pluggable HDFS Block Placement Policy

HDFS, the underlying file system used by HBase, provides several nice features such as replication, end-to-end checksums, failover, and storage elasticity. However, the default block placement policy of HDFS, while rack aware, is minimally constrained. The non-local replicas of the different blocks of a file can pretty much end up spanning the entire cluster. To reduce the possibility of losing data if multiple nodes were to fail around the same time, we implemented a new block placement policy in HDFS that allowed us to constrain the placement of the replicas of the blocks of a file within configurable smaller sized node groups. This meant that there were a lot fewer combinations of multiple node failures that could result in data loss compared to before. Using the new block placement policy, we were able to reduce the theoretical data loss probability by about two orders of magnitude.

3.3 Availability

HBase handles node failures well. Except for the HDFS master (aka namenode) failure, the system automatically handles and recovers from failures of other components (such as the HBase master or regionservers, or HDFS datanodes). Given this, we expected the more common reasons for downtime to be maintenance. Since HBase software was still relatively new at that time, we anticipated that we would likely need to push critical bug fixes and improvements routinely to production clusters, and hence implemented rolling upgrades, whereby we could upgrade the software on the cluster in a rolling fashion without taking the entire database down. We also implemented the ability to alter a table schema (e.g., adding new column families or changing the settings of an existing column family) in an online manner. We also made HBase compactions interruptible so that restarts of a server or the entire cluster didn't have to wait for compactions to finish.

3.4 Performance

HBase's Log Structured Merge Tree (LSM) approach for maintaining on-disk data makes it very efficient at writes. On the read-side, however, because multiple, unmerged, immutable, index-organized data files aka HFiles need to be consulted before a read request can be served, in a naive implementation, the overhead on reads is likely to be higher in the general case compared to a traditional RDBMS.

We implemented several performance features to help reduce this read overhead.

- Bloom filters on keys– These are maintained at a per-HFile level, and help reduce the number of HFiles that need to be looked up on a read.
- Timerange hints– We added timerange hints to HFiles, so that timestamp supplied key lookups or time range queries only needed to look up HFiles that contained data for an overlapping time range.
- Seek optimizations– We also fixed some inefficiencies in the earlier implementation that were causing multi column lookups within a row to become full row scans, where instead an index could have been used to efficiently reseek to the desired columns.

3.5 Shadow Testing

Rather than speculating on the kind of issues we would run into when we put the system in production, or limiting ourselves to running synthetic load against test clusters, we decided to invest effort in shadowing/mirroring traffic for a portion of the users from the older generation of the messages and chat products while the new one was still under development. The shadow cluster setup gave us pretty good insights into what to expect under real traffic conditions, and it also served as a useful platform for testing failure scenarios, and evaluating alternate schema designs.

Once the rollout of the new version of the product reached a significant part of our user base, we switched to a new shadowing setup– one that mirrored the new version of the product. This setup continues to remain in place and helps us iterate on software changes rapidly and in a safe manner. All changes go through shadow testing before they are pushed to production.

3.6 Backups V1

HBase had not been used previously, at scale, for critical user facing applications. So during the initial stages, we were concerned about unknown/potential bugs in HBase, especially ones that could cause data corruptions. So we invested in a backup and recovery pipeline implemented outside of HBase. We used Scribe, Facebook's distributed framework for aggregating logs from thousands of servers, to log all actions against a user's mailbox (such as addition/deletion of message, change to read/unread status, etc.) to an HDFS cluster. But Scribe does buffered forwarding, and is not designed for guaranteed delivery. So to greatly minimize the probability of data loss in this pipeline, we double log to Scribe from both the application server and HBase tiers. The two log streams are then merged/de-duped and stored in HDFS clusters in a local as well as remote data center.

4 Mid-air refueling

Once in production, we had to rapidly iterate on several performance, stability and diagnosability improvements to HBase. Issues observed in production and shadow cluster testing helped us prioritize this work. In this section, we will describe some of these enhancements.

4.1 HFile V2

Few months into production, as datasets continued to grow, the growing size of the index started to become a problem. While the data portion in an HFile was chunked into blocks, lazily loaded into the block cache, and free to age out of the block cache if cold, the index portion, on the other hand, was a monolithic 1-level entity per HFile. The index, cold or otherwise, had to be loaded and pinned in memory before any requests could be served. For infrequently accessed column families or old HFiles, this was sub-optimal use of memory resources. And this also resulted in operations such as cluster restarts and region (shard) load balancing taking much longer than necessary. Bloom filters suffered from the same problem as well.

As a temporary measure, we doubled the data block sizes; this halved the size of the indices. In parallel, we started working on a new HFile format, HFile V2– one in which the index would be a multi-level data structure (much like in a traditional B-Tree) comprising of relatively fixed sized blocks. HFile V2 also splits the monolithic bloom filter per HFile into smaller blooms, each corresponding to a range of keys in an HFile. The bloom and index blocks are now interspersed with data blocks in the HFile, loaded on demand, and cached in the same in-memory LRU block cache that is used for data blocks. A single read operation could now result in paging in multiple index blocks, but in practice frequently accessed index blocks are almost always cached.

In traditional databases, such changes are almost impossible to make without a downtime. Thanks to compactions, in HBase, data format changes can be pushed to production and data gets gradually converted to the new format without any downtime, as long as the new code continues to support the old format. But it wasn't before HFile V2 had a lot of soak time in shadow cluster setup and significant compatibility testing that we felt comfortable pushing the code to production.

4.2 Compactions

Compaction is the process by which two or more sorted HFiles are merge-sorted into a single HFile. This helps colocate all the data for a given key on disk, as well as helps drop any deleted data. If a compaction takes all the existing HFiles on disk and merges them into a single HFile, then such a compaction is called a major compaction. All other compactions are ”minor” compactions, or referred to below as just compactions.

We usually select 3 files of similar sizes and compact them. If the files are of different sizes, we do not consider them viable candidates for a minor compaction. The first issue we found was the case of inefficient cascaded minor compactions. Consider the case when we find 3 files of a similar size and compact them to a single file of three times the size (3x). It could so happen that there are 2 files existing in the store of a similar size (3x) and this would cause a new compaction. Now the result of that could trigger another compaction. These cascaded set of compactions are very inefficient because we have to read and re-write the same data over and over. Hence we tweaked the compaction algorithm to take into account the sum of the data.

The next issue to hit us was the parameter which allows a file to be unconditionally included in a compaction if its size was below a given threshold. This parameter initially was set to 128MB. What this resulted in was for some column families, which had small files flushed to disk, we would compact the same files over and over again until the new files resulted in a 128MB file. So, in order to get to 128MB, we would write a few GB of data, causing inefficient usage of the disk. So we changed this parameter to be a very small size.

Originally, compactions were single threaded. So, if the thread was performing a major compaction which would take a while to complete for a large data set, all the minor compactions would get delayed and this would bring down the performance of the system. And to make this worse, the major compactions would compact every column family in the region before moving on to any other compaction. In order to improve the efficiency here, we did a couple of things. We made the compactions multi-threaded, so one of the threads would work on a ”large” compaction and the other would work on a ”small” compaction. This improved things quite a bit. Also, we made the compaction queue take requests on a per-region, per column family granularity (instead of just per region) so that the column family with the most HFiles would get compacted first.

4.3 Backups V2

The V1 backup solution wrote the action logs for the user mailboxes as they arrived into Scribe. To restore a user's data, all the user's action logs had to be read out of scribe and replayed into the HBase cluster. Then, the secondary indices for the user's mailbox had to be regenerated.

As data sizes started growing, the existing backups V1 solution was getting less effective because the size of the action logs that would need to be read and replayed on recovery kept increasing. This made a V2 solution necessary.

The backups V2 solution is a pure HBase backup solution. The backups are split into two parts - the HFile backups (HFiles are HBase data files) and HLog backups (HLogs are HBase write ahead logs). With both of these backed up, and knowing the start and end time of the HFile backup process, it is possible to do point in time restores. Hence, we started taking HFile backups and increased the retention of the HLogs on the production cluster. These backups, which live on the production cluster, are our stage 1 backups. In order to survive cluster failures, we also have stage 2 backups where the data persists in another cluster in the same datacenter. For stage 3 backups, data is also copied to a remote datacenter.

One issue we were hit with pretty quickly while doing in-cluster stage 1 backups was that as the files got larger, we needed more disk iops on the production clusters to do the backups and it was taking longer to do the backups as well. To get around this issue, we modified HDFS to create hardlinks to an existing file's blocks, instead of reading and writing the data to make a copy.

4.4 Metrics

We had started out with a basic set of metrics to being with, for example - HBase get, put and delete ops count per second and latency of each, latencies of the resultant DFS read, write and sync calls, and compaction related metrics. We also tracked system level metrics such as CPU, memory and network bytes in and out. Over time, the need for more insight in order to fix bugs or improve performance made it necessary to add and track various other metrics.

In order to understand how our block cache was doing at a per column family level (so that we can decide not to cache some column families at all or look into possible application bugs), we started tracking per column family cache hit ratios. We also noticed that for some calls, the average times of various operations was low, but there were big outliers - this meant while most users had a good experience, some users or some operations were very slow. Hence we started tracking the N worst times per operation based on machine names so that we could look further into the logs and do an analysis. Similarly, we added a per-get result size so that we can track the size of responses being requested by the application servers.

Once we had HBase backups V2 in production, the total disk usage became an important metric, so we started tracking the DFS utilization as well. In addition, we found that there were quite a few rack switch failures and started tracking those.

These are just a handful of examples of how we added more metrics as we went along.

4.5 Schema Changes

The messages application has complex business logic. In the days leading up to its release and even for a short while afterwards, the product kept evolving. Hence an early decision was made to not invest in a complex schema to store the data in HBase. But loading all the messages into the application server's memory from HBase on a cache miss when a user accessed his mailbox was prohibitively expensive. So the initial solution was to store all the messages of the user in one column family, and maintain an index/snapshot of the in-memory structures required to display the most commonly accessed pages in another column family. This served us well in the days of evolving schema as the snapshot was versioned, and the application server could interpret changes to the snapshot format correctly.

However, as we store messages, chats, and emails, we found that the ”snapshot” column family got several megabytes in size for the active users - both reads and writes were large, and this was becoming a big bottleneck. Hence we had to invest in making the schema finer grained. We went to a ”hybrid” schema where we made the various structures stored in the column families more fine-grained. This helped us update just the more frequently varying data and reduce the number of updates to the infrequently changing data. This helped us make the writes smaller, though the reads were as large as before.

Finally, we looked into the usage patterns of the application and came up with a very fine-grained schema where the reads and writes were done in small units across all the calls. This improved the efficiency of things quite a bit. But since we were changing the entire disk layout of all the data, rolling this out at our scale with no noticeable downtime was a very big challenge. In addition, since the map-reduce jobs that were run to transform the data were very intensive and it was very risky to run the job on the production clusters, we decided to run the transformation jobs on a separate set of machines in a throttled fashion.

4.6 Distributed Log Splitting

To describe what this is briefly, consider the scenario when all or most of the regionservers die (for example, when the DFS namenode is unreachable because of a network partition). Upon restarting the clusters, all the regions would have some entries in the write-ahead logs (HLogs) that would have to be replayed before the regions could be brought online. To do this, all the HLogs belonging to all the regionservers would have to be split and regrouped into edits per region, and the new regionservers opening the regions would have to replay these edits before the regions can be opened to serve reads or take write traffic.

The implementation to achieve this was to have the HBase master list all the logs and split them using a few threads. The issue while running larger clusters - say with 100 regionservers - was that it would take more than 2 hours in order for the single master process to split the logs for all the servers. Since it was unacceptable to go into production in this state, the first and simple version of distributed log splitting was to have a script that would parallelize this across all the regionservers before the cluster was started. This did achieve the intended effect of making the log splitting times in the 5-10 minute range.

The issue with the script-based approach was that it was transparent to the HBase master. This required the cluster to be completely shutdown until the log splitting process was completed, so it was not suitable for the cases when a subset of regionservers (say, the ones on a rack) exited because they were partitioned from the DFS. Hence we implemented an automatic distributed log splitting feature into the HBase master itself - where the HBase master uses Zookeeper as a task queue and distributes the job of log splitting to all the regionservers. Once the log splitting is done the regions are opened as usual.

4.7 Data locality

To be available in spite of rack failures, we do not want all the replicas of a file to be on the same rack. For a very data intensive application, like messages, this causes a lot of network traffic between racks. As the data size grew larger, the network traffic kept increasing until it was causing the top-of-rack to become the bottleneck. So, to reduce network bandwidth during reads, locality of data became extremely important. In order to maintain locality of data, the regions have to assigned to the region servers that contained most of the data on the local disk. We solved this by computing the machine with the most data locality for each region at startup time and assigned the regions accordingly at the HBase startup time.

4.8 Data Block Encoding

In several applications, including Facebook messages, there is a lot of redundancy between adjacent records (KVs) within a data block. And hence it is desirable to have an in-memory format for blocks that takes advantage of this; a format that allows us to pack more records per block, yet one in which records can be efficiently searched. We implemented a ”data block encoding” feature in HBase, which is a lightweight, pluggable compression framework that’s suited for in-memory use. It increases the holding capacity of the block cache, without much impact on performance. We have implemented a few of the encoding schemes, and we’re seeing block cache capacity wins of up to 1.1x-5x. On disk too, now that we encode the blocks first and then compress, these schemes are able to add about 1.1x-2x savings on top of LZO.

4.9 Operational aspects and issues

Failure is the norm when running large-scale clusters. Hence disk failures, machine failures, per-node network partitions, per-rack network failures and rack switch reboots were all recurring events. We wanted to automatically detect and handle the more frequent of these. Early on, we had decided that we should automatically handle most of the single node issues. Hence we had monitoring scripts that would disabled and send to repair those machines which were not network reachable for an extended period of time, those which had root-disk failures, and so on. In fact, the process of machines being retired and reclaimed from the repair pool is a constant one, with about 2-3% of the machines always in the queue.

We had a lot of alerts that paged the relevant people on a variety of failure scenarios. For example, if the number of errors observed by the application server exceeded a certain steep threshold then it triggered alerts. In addition, if the error rate was not steep enough to trigger alerts but stayed persistently over a time window, then too an alert would get triggered (prolonged errors even if lower in volume). Similarly, the master nodes (”controller” nodes as we call them) would alert on issues, as they needed a higher availability than the other nodes. In addition, there is a HBase health checker that is run constantly and would alert if it fails the health checks.

In the light of issues we experienced, operator error is a non-trivial cause of some of the issues we faced. There was one such incident where upon accidentally unplugging a power cable caused about 60% of the nodes in the cluster to go down. The cluster was restored in a few minutes and there was no data loss.

5 New use cases

As HBase matures, its usage inside Facebook, for both serving and user-facing workloads as well as real-time and batch processing workloads, has grown. In this section, we will describe some of these new use cases.

5.1 Puma: Real-time Analytics

Puma is a stream processing system designed for real-time analytics. Using Puma, content publishers/advertisers can, for example, get real-time insights into the number of impressions, likes and unique visitors per domain/URL/ad, and a breakdown of these metrics by demographics. Such analytics was previously supported in a periodic, batch-oriented manner using Hadoop and Hive, and hence available for consumption only several hours later. With Puma, data is collected in real-time via Scribe, read via ptail and processed/aggregated by Puma nodes and written out to HBase. Puma nodes also act as a cache for serving the analytic queries, deferring any cache misses to HBase.

5.2 Operational Data Store (ODS)

ODS is Facebook’s internal monitoring system which collects a variety of system and application level metrics from every server for real-time monitoring, anomaly detection, alerting and analysis. The workload, consisting of several tens of millions of timeseries data, is pretty write intensive. More than 100B individual data points are written per day. And the reads, both user-issued and from time rollup jobs, are mostly for recent data.

ODS was previously implemented on top of MySQL, but recently switched to an HBase backend implementation for manageability and efficiency reasons. The constant need for re-sharding and load balancing existing shards, as the data sets grew in unpredictable ways, caused a lot of operational inconvenience in the MySQL setup. Also, from an efficiency point of view, HBase is a really good fit for this write intensive, read-recent type of workload. HBase provides a much higher throughput because it converts random record updates into sequential writes at the disk subsystem level. And, because HBase naturally keeps old and newer data in separate HFiles, for applications like ODS, which mostly read recent data, the data can be cached a lot more effectively.

ODS makes heavy use of the HBase's tight integration with MapReduce for running the time-based rollup jobs periodically. ODS also uses HBase's TTL (time-to-live) feature to automatically purge raw data older than a certain number of days. There is no longer a need to implement and manage additional cron jobs for this purpose.

5.3 Multi-tenant key/value store

Several applications within Facebook have a need for a large-scale, persistent key/value store. A per-application key/value store backed by HBase is now offered as a service within Facebook and some of the applications now use this solution for their storage needs. A shared HBase tier provides controlled multi-tenancy for these applications, and relieves the application developers from the usual overheads associated with capacity planning, provisioning and infrastructure setup, and scaling as their data sets grow.

5.4 Site integrity

Facebook's site integrity systems gather data and perform real-time analytics to detect and prevent spam and other types of attacks. One of these systems now uses an HBase backed infrastructure to track new user registrations and surfaces several fake accounts every day. HBase was chosen because it provides high write throughput, elasticity, a data model that allows running efficient scans on lexicographically sorted row keys, and primitives that support aging out old data automatically.

5.5 Search Indexing

HBase is being used to implement a generic framework to build search indices across many verticals. Multiple document sources are fed into HBase constantly and asynchronously. HBase stores the document information, and is used to join the information coming from the various sources for each document. On demand, per-shard search indices are built using MapReduce to read the relevant data from HBase. The index servers then load these shards.

6 Future work

HBase is still in its early stages, and there is a lot of scope and need for improvement.

On the reliability and availability front, HDFS Namenode HA (making HDFS tolerate namenode failures), better resilience to network partitions, data locality aware node failover and load balancing of regions, rolling restarts for HBase/HDFS during software upgrades, and fast detecting single node failures are some of the current focus areas.

We are also working on cross data center replication and point-in-time restore from backups so that every application does not have to implement its own custom solution. Native support for hardlinks in HDFS is being implemented to enable super fast snapshotting of data in HBase. On the efficiency front, we are looking at keeping inlined checksums for data to reduce the disk overheads of verifying checksums, improving HBase's ability to make optimum use of large memory machines, reducing HBase's reliance on OS page cache, adding support for compressed RPCs, optimizing scan performance, commit log compression and so on.

As new applications for HBase emerge, new requirements continue to arise. Better support for C++ clients, improved Hive/HBase integration, support for secondary indices and coprocessors and adapting HBase for use in mixed storage (flash+disk) or pure flash environments are also things on the radar.

7 Acknowledgements

We would like to thank our HDFS engineering, HBase operations and Messaging teams, as well as the Apache HBase community without whose help and involvement, a lot of this work would not have been possible.

References

[1] Apache HBase. http://hbase.apache.org.

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

[3] Kannan Muthukkaruppan. The Underlying Technology of Messages. http://www.facebook.com/-note.php?note_id=454991608919, Nov. 2010.

[4] Dhruba Borthakur et al. Apache Hadoop Goes Realtime at Facebook. In *Proc. SIGMOD*, 2011.

[5] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. The Google File System. In *Proc. SOSP*, 2003.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, 2004.

[7] Fay Chang et al. BigTable: A Distributed Storage System for Structured Data In *Proc. OSDI*, 2006.

[8] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX*, 2010.

[9] Alex Himel. Building Realtime Insights. http://www.facebook.com/note.php?note_id=10150103900258920.

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

[11] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). In *Acta Inf.* 33(4): 351-385 (1996).

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

[13] Scribe. https://github.com/facebook/scribe/wiki.