SocialCom/PASSAT/BigData/EconCom/BioMedCom 2013

Scaling Deep Social Feeds at Pinterest

Varun Sharma
Pinterest Inc.
Email: varun@pinterest.com

Jeremy Carroll
Pinterest Inc.
Email: jeremy@pinterest.com

Abhi Khune
Pinterest Inc.
Email: abhi@pinterest.com

***Abstract*—With the advent of Twitter, the follow model has become pervasive across social networks. The follow model enables users to follow other users i.e. subscribe to content created by other users, thereby, establishing the concept of a following feed for a user. At Pinterest, we continually store, update and serve feeds for millions of users and fan out millions of newly created pins/repins to thousands of followers, leading to billions of operations everyday. We describe the current feed storage solution, backed by Apache HBase, at Pinterest. We describe how we handle data management challenges unique to our scale, in the wake of strict performance and availability requirements. We also present a qualitative comparison to our previous "following feed" architecture, backed by Redis.**

I. Introduction

Pinterest is an online pinboard where people pin images and organize content into self created virtual boards. Curation and discovery of inspirational content are the two primary categories of user activity on Pinterest. The follow model is a construct to aid content discovery. The Pinterest follow model allows users to follow other users or follow boards created by other users. For example, a running enthusiast could choose to either follow the "nike brand" on Pinterest or follow the "shoes" board created by the "nike brand". Hence, each user/board has a list of followers who subscribe to pins created on that board or by that user. Similarly, each user has a list of followees which are boards/users, the user is following. The following feed for a user consists of content aggregated from all the followees of the user. Each time, a user accesses pinterest.com, we render the following feed to the user. A large number of social networks have embraced the follow model, including Tumblr and Instagram. The following feed is a core piece of user experience for the majority of social networks today.

An implementation of the following feed, at Pinterest, needs to be continually updated as users follow/unfollow other users and boards. The corresponding pins need to be added to or removed from the following feed of the user performing the action. New content needs to be pushed out to the following feeds of all followers. Such fanout results in significant write amplification. Lastly, the following feed needs to be retrieved whenever an authenticated user accesses pinterest.com.

II. Architecture

Pinterest uses MySQL for persistent storage of core entities - pins, boards and users. For example, all metadata corresponding to a pin such as the pin's image URL, the pin's title and the pin's description are stored in the MySQL database. The MySQL database is also responsible for unique identifier generation. The dataset is sharded into multiple MySQL databases across the identifier space for scalability.

However, feeds are not stored in MySQL for reasons, we mention in the next section. Figure 1 shows the write path for the following feed. Follow/unfollow actions by users and creation of new content triggers writes into the following feed storage. The frontend receives these actions and asynchronously enqueues a "task" into our message queue system. The frontend does not perform the heavy lifting associated with expensive operations such as fanout. The message queue is responsible for persisting the task and retrying it until it succeeds. If a new pin is created, the follow store is consulted to retrieve the list of followers for the user and the pin is written to the respective following feed(s). If a follow/unfollow action is performed, the recent pins of the user being followed are retrieved and are added to/deleted from the following feed of the user performing the action.

The Feed Storage is responsible for storing only the identifiers associated with the pins in the following feed. As shown in Figure 2, the frontend consults the Feed Storage for the reverse chronologically sorted list of pin ids in a user's following feed. After obtaining the list of pin ids, the frontend consults the MySQL backed pin metadata store to obtain the metadata necessary to render the page to the user.

III. The Storage

Figure 3 shows the read and write requests to the Following Feed cluster on a relative scale. We expect the write volume to be significantly higher due to the fanout incurred during content creation. B-Tree backed stores such as MySQL, are not amenable to a high volume of random writes. Pin creations would require updating feeds of a large number of users during the fanout operation. In a B-Tree implementation[1], this would require updates in different parts of the B-Tree, which in turn, could lead to rearrangements within the tree, due to node splitting etc.

978-0-7695-5137-1/13 $26.00 © 2013 IEEE
DOI 10.1109/SocialCom.2013.7

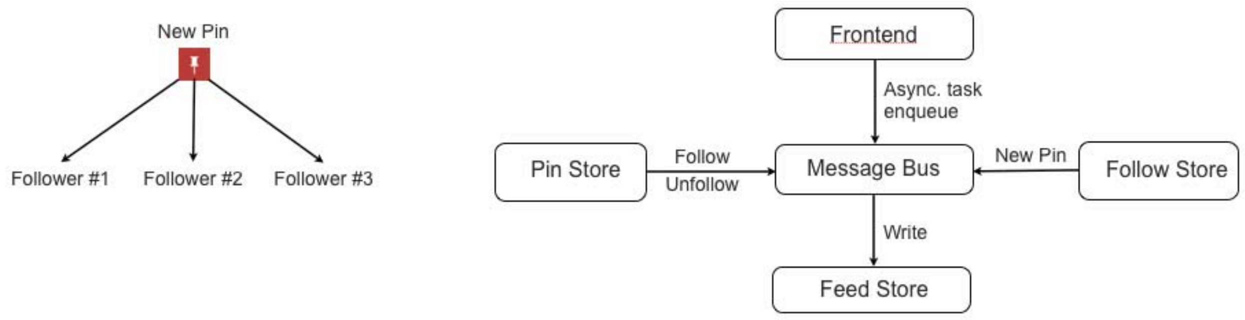

Figure 1. Following feed writes.

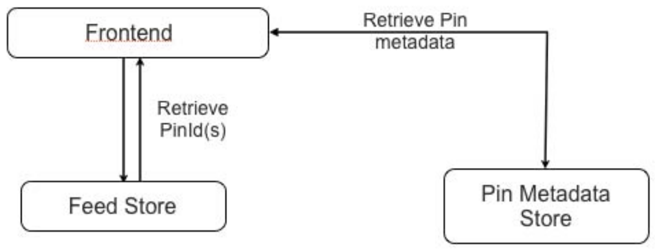

Figure 2. Following feed reads.

Moreover, the high write load due to fanout on the B-Tree would incur a substantial number of read operations. This write amplification factor due to fanout poses a requirement for high write throughput which B-Tree backed systems like MySQL can't handle very well.

On the other hand, a number of NoSQL solutions provide disproportionately higher write throughput and are better suited for our application.

- In memory databases: Redis[2] is an in-memory database which enjoys widespread use and is well known to be extremely performant. It achieves low latency and high throughput by keeping the entire dataset in RAM. It provides weak durability gaurantees and supports Master Slave replication.
- Log Structured Merge Trees/Bigtable like stores: A number of Bigtable[3] like stores have gained popularity over the past few years. Notable examples include HBase[4], Cassandra[5]. The LSM approach[6] allows the system to buffer writes in RAM, until the buffer is exhausted. When the buffer fills up, a file is output as a sorted map of <key, value>pairs. Over time, many such files accumulate. A read operation requires seeking and lazily merging the accumulated sorted files to retrieve the result. Compaction operations run in the background, to merge the files together so that the number of files remains low. This reduces the number of disk seeks required during read operations. Another notable feature with these solutions (HBase/Cassandra) is the ability to manage sharding and machine failures automatically, rather than relying on manual intervention or the application tier to do so.

The current feed infrastructure at Pinterest uses HBase as the backend storage. We describe the system in more detail in the following sections and finally, compare it with a setup backed by Redis. Amongst the wide variety of LSM stores available (HBase, Cassandra, LevelDB), we chose HBase because of its maturity, adoption in the community and its seamless integration with Hadoop and Hive. Also, Facebook already uses HBase for messages[7], an online user facing application. We also had considerable experience operating Hadoop clusters and HBase uses HDFS as its underlying file system.

IV. HBase

Apache HBase is a NoSQL, distributed, versioned database system modeled after Google Bigtable. Similarly, Apache ZooKeeper[8] is a distributed lock service modeled after Google Chubby[9]. HBase uses a ZooKeeper quorum for master election, maintaining cluster state and performing cluster coordination. Though HBase can use any filesystem

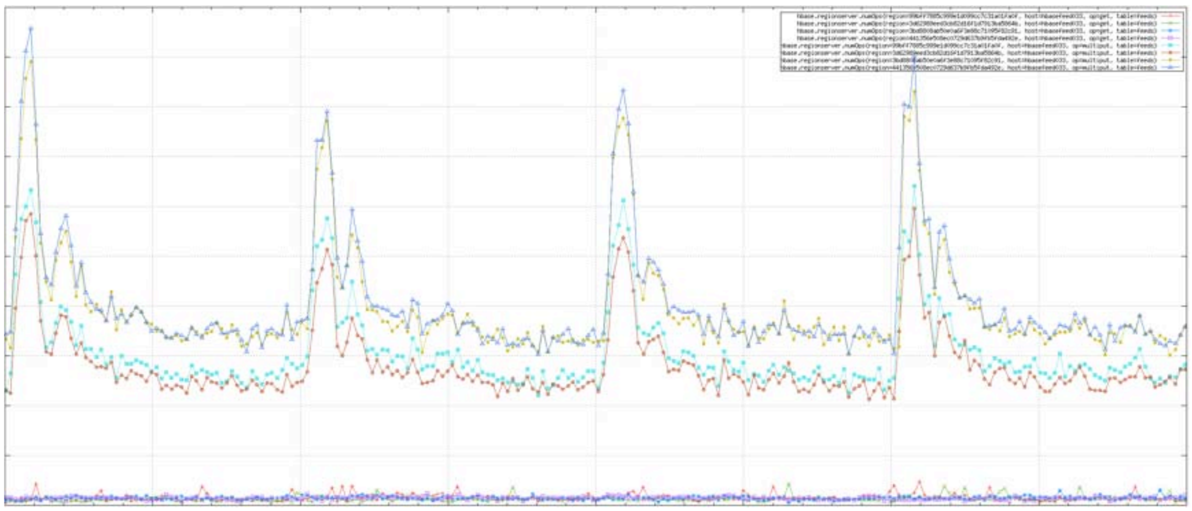

Figure 3. Read write ratio for the following feed. The lines at the bottom denote reads across multiple servers. The lines above denote writes.

underneath the hood, a typical HBase deployment uses HDFS[10], the Hadoop Distributed File Sytem. HDFS is modeled after GFS[11] and uses a master slave model. Files are divided into large sized blocks which are replicated 3 times across the HDFS slaves or datanodes. HDFS master or the namenode is responsible for cluster coordination, block placement, replication and recovery from machine failures. The HDFS master also stores the mapping from files to data blocks and the machines holding the block replicas. By block replicas, we mean the 3 copies of each data block. HBase achieves fault tolerance through HDFS replication. Failure of a single machine or disk drive does not render the data unavailable since it can be restored from a replica.

In HBase, tables are divided into regions and regions are distributed evenly across region servers. The HBase region servers and HDFS datanodes run on the same set of servers. Like Bigtable, HBase stores data as a three dimensional sorted map from $(row, column, timestamp) \mapsto value$. The data is sorted along the row dimension and regions split the row space into contiguous ranges. The HBase master is responsible for distributing regions across region servers and moving regions when servers fail. Each table consists of column families, akin to locality groups in Bigtable.

V. Following Feed on HBase

In the following subsections, we motivate the schema design and our customizations to extract performance, improve Mean Time To Recovery (MTTR) and availability for our HBase deployment.

A. Schema

HBase keeps rows in lexicographic order and columns within rows are also lexicographically sorted. A column based schema (also called wide schema) would look like:

```
                 feed:(rCreationTs:pinId)
userId                   <empty>
```

Note that the value is empty as we have absorbed everything into the row and the column. The rCreationTs is the reverse ordered creation timestamp of a pin, thus, enforcing reverse chronological ordering. The columns are stored in a column family ”feed” as depicted above.

A tall schema, on the other hand, would absorb all the fields into the row. Underneath the hood, both schemas result in very similar storage patterns. However, we choose the wide schema because of the following desirable properties:

- Atomicity: HBase provides transaction support at the row level, thus providing the ability to make atomic updates to a user’s feed.
- Locality: Since a table is divided into regions based on rows, a user’s data lies within the same region and hence on the same region server.

VI. Performance and Availability

A. Performance

We have different requirements for reads and writes. The read volume is comparatively lower but keeping latencies low is very important. On the other hand, writes can be latent since they are processed by the message queue workers. However, the sheer volume of writes requires the system to be high throughput. The predominant component of writes is the fanout of new pins to batches of followers. We realized that since writes accounted for majority of the system load, an important component of optimizing writes was to minimize their interference with reads.

1) Reads: Intuitively, we expect only a subset of the users to be active at a particular instant and we expect temporal locality for their following feed accesses. We hypothesized that an LRU based caching scheme would be very effective. On our HBase region servers, we increased the percentage of heap memory occupied by the HBase Block cache from 25 to 40 percent. In production, we were able to validate our hypothesis by observing cache hit rates upwards of 85 percent with a modest amount of cache memory.

Another important observation was that all the data was part of the key $(row, column, timestamp)$ in the underlying HFiles written by HBase. Hence, a prefix encoding scheme would effectively compress data. Apart from saving disk space, it would allow us to cache larger amounts of recently accessed data. Fortunately, HBase came equipped with prefix compression. Enabling prefix compression lead to a 4X reduction in data size, hence more data being cached and an improvement in read latency.

We enabled other standard read optimizations for HBase. We reduced the block size for HFiles from 64K to 16K since back of the envelope calculations suggested that a user's feed would have fit within roughly 16 kilobytes. The HFile block cache operates at the HFile block level and the block size controls the granularity of caching. We wanted blocks to be close to the size of the following feed for a particular user so that we populated and evicted an entire user's data. We also enabled short circuit reads so that the HBase region server does not read data from HDFS datanode over a socket when data is local. Since the node that runs a region server also runs the datanode, data is available locally unless there have been machine failures causing HBase regions to get redistributed.

2) Writes: HBase provided very high write throughput out of the box. However, as writes kept coming and data was flushed to create HFiles, we found that compaction activity became more and more intense. The I/O generated by compactions started impacting read latencies. By default, whenever a region's in memory write buffer, also called the memstore, reached 128M, a flush was initiated. We observed that with prefix compression, a 128M memstore was resulting in a tiny HFile of size 8M. Therefore, we upped the memstore size to 512M, a number we found apt given the amount of heap memory we could provision for all the memstores. This resulted in less frequent HFile flushes, hence fewer HFiles and less compaction activity. The pressure on the I/O subsystem was immediately relieved.

We use the Concurrent Mark Sweep collector[12] to minimize GC pauses for HBase. The heavy writes generate a lot of garbage along with surviving objects which need to be promoted. We realized that our tail latency for reads was as good as our new generation GC pauses. Decreasing the size of the new generation resulted in more frequent, smaller pauses and an immediate improvement in tail latency.

B. MTTR

Mean Time To Recovery from region server failures was a major concern for us, since a failure would cause an availability loss for a subset of users. The HBase Master discovers region server failures through ZooKeeper. Each region server registers with ZooKeeper and negotiates a configurable session timeout. We followed the standard practice of lowering this timeout to 30 seconds. We found that this was insufficient to guarantee a low MTTR. Instead, we found recovery times upwards of 10 minutes from region server failures. The issue was that even though HBase discovered that a region server was dead, the HDFS namenode would only mark that node as dead, after 10 minutes. We thought that reducing this timeout would solve our MTTR problems. However, this practice was strongly discouraged, since upon reaching this timeout, the HDFS namenode would actively start replicating the now underreplicated data blocks which belonged to the failed node. This would place additional load on an already wounded cluster.

After looking more closely at the recovery process and more recent work in HDFS, we did a few customizations to optimize our HDFS setup for a low MTTR. Each region server writes edits to a Write Ahead Log (WAL) which is triplicated on HDFS. During recovery, the WAL needs to be read and then split to recover edits for each region on the dead region server. The edits are replayed to recover the in memory state of the regions on the dead region server. We examine our customizations in context of the recovery steps.

- WAL read: The HDFS namenode marks a datanode as stale if it has not heartbeated for a configurable time period. This came from recent work done by the open source community on HDFS. Earlier, the WAL read would first hit the datanode which was already dead and then there would be a socket timeout which would take 60 seconds or a connect timeout which would take 45 retries of 20 seconds each. We enabled the stale node setting with a timeout of 20 seconds and reduced the socket and connect timeouts to 3 seconds. This means that the WAL read step would avoid hitting the dead node, and even if it did, timeout quickly and move on to the next functional replica.
- Lease recovery: This was one of the least understood features of HDFS. A file that was being written to, in HDFS, such as the WAL needed to be closed before reading it. A close operation would initiate a lease recovery and a block recovery. For block recovery, the namenode would choose one of the replica datanodes as the primary datanode for the last block of the file i.e. the block being written to. The primary datanode would then reconcile the size of the block against other replica datanodes and finalize the state of the block with the namenode. We found two issues with the process. During recovery, the namenode would choose the dead datanode as the primary datanode. Secondly, when instructing the primary datanode to reconcile block replicas, it would include the dead datanode as

a candidate replica for reconciliation. We contributed HDFS 4721 to integrate lease recovery with stale node detection. In our setup, stale nodes are completely ignored during the lease recovery process.

- WAL splitting: The WAL needs to be split to recover the edits for each region which went offline during the region server failure. This is because there is 1 WAL per region server. Each split gets written out as a separate file. Before stale node detection, the namenode could suggest the dead datanode as one of the replicas for the split. This would then timeout with a default socket timeout of 480 seconds thus significantly delaying the recovery. However, with stale node detection, the dead node is avoided in this step as well. Also, we reduced the socket write timeout to 5 seconds, in case we do hit a dead node.

With these customizations, we were able to consistently achieve an MTTR of less than 2 minutes. We thoroughly tested this by suspending the region server and datanode processes and also by blackholing a host using iptables. This was critical to achieving a highly available system.

C. Single Points of Failure

We run our systems on EC2 and it is advised to not keep all of one's data and machines/instances within the same EC2 availability zone. We could have split our HBase cluster across multiple availability zones and used HDFS rack awareness feature to replicate data across availability zones. However, our load tests showed significant performance difference due to having to replicate data across availability zones. Hence, we decided to house our HBase cluster within the same availability zone. To be able to survive availability zone outages, we setup a replica cluster in another availability zone. This was expensive, however, it is fairly standard practice for highly available HBase setups to have 2 clusters.

Since we setup two clusters, we went with a simple setup for the namenode. The namenode is a single point of failure for HDFS. The namenode is configured to write to ephemeral instance storage and over the network to Elastic Block Store (EBS) volumes. This ensures persistence of the namenode fsimage even if we lose the namenode EC2 instance and allows us to restore the cluster.

To ensure consistency across the 2 clusters, we considered HBase replication. However, it was not heavily used/tested at our scale. So, we decided to implement dual writes from our message queue. The message queue would be responsible to persist the writes and retry until a write is successful to both clusters.

D. Challenges at scale

Certain challenges unique to the Following Feed problem emerge as the scale increases to millions of users and hundreds of millions of follow relationships.

1) Data consistency: The write path depicted by Figure 1 shows that user actions are enqueued to the message queue. Workers are responsible for dequeuing and issuing the writes. However, our message queue may not preserve time ordering of user actions. It is possible for writes to be issued out of order and result in a following feed, inconsistent with user actions. Such a situation occurs when user A follows user B and unfollows user B quickly afterwards. Here AB_{follow} represents the user action and $M(AB_{follow})$ represents the writes being issued by the message queue.

$$\begin{gathered} AB_{follow} \mapsto t_1 \\ AB_{unfollow} \mapsto t_2 \\ t_1 < t_2 \\ M(AB_{follow}) \mapsto t_1' \\ M(AB_{unfollow}) \mapsto t_2' \\ t_1' > t_2' \end{gathered}$$

This situation becomes more prevalent when the difference between t_1 and t_2 is of the order of milliseconds or when there are unexpected message queue delays. Note that within HBase, the unfollow action results in a delete operation which inserts delete markers to mask values. To resolve these inconsistencies, we use t_1 and t_2 as cell timestamps for the follow/unfollow actions. In the above scenario, the follow action will insert pins at t_1 which would still be masked by delete markers inserted by the unfollow action at t_2. Hence, we are able to enforce a mutation ordering by making use of cell based timestamps in HBase.

2) Unbounded data growth: As new pins are created and pushed to follower's feeds, the dataset grows exponentially. The dataset growth is a function of the rate of incoming content and the cardinality of follow relationships in the follow graph. We cap the number of pins in a user's following feed to a certain threshold.

One approach to achieve this would be to trim the feed in real time during writes. Since the majority of writes stem from the fanout operation, we would need to read the excess pins on every write and issue deletes to remove these pins. This would result in a high random read volume and would completely defeat the utility of an LSM tree. Another possibility would be to run a mapreduce job to trim the feeds. The mapreduce job would likely impact the performance of the online serving cluster. Also both these approaches would simply insert delete markers into the system rather than truly deleting data.

We came up with a coprocessor[13] based approach to this problem. We implemented a coprocessor which would hook into the major and minor compactions and would only retain the required number of columns for each row. Since the compaction sees columns within each row in sorted order, we would end up retaining the most recent pins in the following feed. This solved the problem without introducing any additional overhead into the system.

VII. Comparison with Redis

Previously, Pinterest deployed a Redis backed storage for the Following Feed. Since Redis is an in-memory store, it provided high throughput and low latencies. We used Redis sorted sets as the underlying data structure with pin creation timestamp as the sorting key and pin id as the object. We used Redis master slave replication for redundancy and fault tolerance. Moving to HBase came with a number of benefits.

- Scalability: The Redis deployment was manually sharded and required application level support for failing over to slaves in the event of machine failures. Adding capacity and machines was more cumbersome and required manual steps. On the other hand, HBase came with built-in support for fault tolerance, automatic sharding and load balancing.
- Deeper Feeds at Lower Costs: With HBase, we were able to keep the hot/recently accessed feeds in memory and exploit the temporal nature of retrieval queries. The rest of the data could reside on disk. On the other hand, we were reluctant to keep all the data in memory for Redis. Hence, HBase enabled us to significantly increase the length of the Following Feed for our users. In fact, we were able to cut down our machine footprint and save costs.
- Data consistency: As we discussed in the previous section, the mutation ordering support provided by HBase allowed us to resolve data inconsistencies.
- Durability: Redis buffers edits in memory and fsync's a second's worth of edits to disk for durability. In the case of a single machine crash, some data loss is inevitable. On the other hand, HBase requires a flush to at 3 replicas before acking that a write is complete. Even though durability was not a strict requirement, it became a nice to have feature for the new system.

VIII. Conclusion

The follow graph and the following feed are an integral component of social networks today. Each poses its own data management challenges. In this work, we have focused on the following feed problem. We have described the current following feed architecture at Pinterest and shared our experiences with building a system at scale, which exploits the write heavy nature of the problem.

We have also described a user facing and mission critical application on top of Apache HBase and HDFS. HBase is widely used for powering analytics, machine learning and other offline big data applications. However, there are only a few examples using Hadoop[7][14] for powering user facing, highly available applications. Our HBase deployment of the following feed is one such example.

We feel that this work would be useful to social networks, as an example for scaling up their feed storage and serving systems. We have also included a systemic description of how we achieved a low MTTR with HBase - a topic not particularly well understood in industry, but indispensable for building highly available applications on top of Hadoop. Such knowledge should be generally useful for people looking at Hadoop for real time serving.

References

[1] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, June 1979.

[2] "Redis," http://redis.io.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Computer Systems*, vol. 26, no. 2, November 2008.

[4] "Apache hbase," http://hbase.apache.org.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, April 2010.

[6] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," vol. 33, no. 4, pp. 351–385, June 1996.

[7] A. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya, "Storage infrastructure behind facebook messages using hbase at scale," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2012.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *7th Symposium on Operating Systems Design and Implementation*, 2010.

[9] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 335–350.

[10] "Apache hdfs," http://hadoop.apache.org/hdfs.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Symposium on Operating Systems Principles*, vol. 37, no. 5, December 2003, pp. 29–43.

[12] "Java se 6 hotspot[tm] virtual machine garbage collection tuning," http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html.

[13] J. Dean, "Designs, lessons and advice from building large distributed systems," in *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[14] D. Borthakur, K. Muthukkaruppan, K. Ranganathan, S. Rash, J. S. Sarma, N. Spiegelberg, D. Molkov, R. Schmidt, J. Gray, H. Kuang, A. Menon, and A. Aiyer, "Apache hadoop goes realtime at facebook," in *SIGMOD*, June 2011, pp. 1071–1080.