SpeedReader: Read-Optimized Distributed Key Value Store

Alex Dao, Gautam Hathi, Joy Patel

Duke University

14 December 2016

Abstract

SpeedReader is a read-optimized distributed key-value store. It is built on DDDFS, which load balances files based on the number of accesses to files and servers, rather than the storage capacity of servers. This allows for higher throughput for reads to keys that have more read demand. SpeedReader adds the capability to write to the system in a read optimized way using versioned data and asynchronous, store-to-store writes. This allows the system to handle a read-heavy workload in a scalable manner while still accomodating writes without interruption to reads. In this paper, we describe the SpeedReader system and detail an implementation of SpeedReader.

I. Introduction

Read-optimized storage is useful to have in situations with high read workload but low write workload. An example of such a workload is one that a CDN might face when serving a website that requires occaional content updates that can be propagated slowly through the system but which has large numbers of users accessing the website. SpeedReader provides such a service using a multi-tiered system that makes information available for reads while delaying writes to minimize read interference. This is accomplished through two main mechanisms: load-balancing and asynchronous write propagation.

Load balancing allows for read-optimization by distributing read workload across a number of stores to optimize resource use. Many classic load-balancing systems balance load in a familiar way: a central server sends heartbeats to its nodes, which in turn return statistics such as its storage capacity. Using this information, the central server balances replicas such that the storage utilization of its nodes is as balanced as possible. The methods of achieving balancing without a large overhead involves many tactics including in-memory metadata to make operations occurring in the central server fast. We will refer to this well-explored form of load balancing as "storage load balancing".

Storage load balancing works well if there is the assumption that all keys are accessed with similar frequency. In this case, the number of accesses to a server is proportional to the number of keys it stores, so the accesses to the servers will be distributed evenly. However, there are many cases in which certain important keys are accessed more frequently than others. In this case, the servers that store this file will have an increased share of accesses and may become a bottleneck. Thus, to further optimize our system for a read-heavy workload, we use a "performance load balancing" system developed in DDDFS, which aims to balance the location and replication of keys based on performance metrics, such as number of accesses or response times, rather than storage metrics. This will prevent bottlenecks occurring due to many accesses to a single key or server.

However, with replication comes the issue of latency and consistency with writes. SpeedReader is designed to ensure that writes

to a given key do not interfere with reads to that key, especially for keys that are frequently accessed. To account for this, we use versioned writes and implement an algorithm which propogates writes throughout all replicas that hold a given key. This preserves availability and latency for reads while allowing write propagation throughout the system.

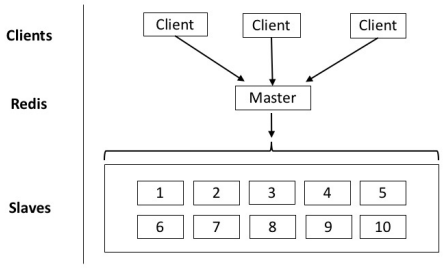
Using Redis, an in-memory data structure store, we have implemented a simulation of a SpeedReader system in DDDFS.

II. Implementation

Infrastructure

SpeedReader is a file system based on HDFS that uses performance load balancing in a write-occasionally read-often environment. Thus, the architecture is focused on read performance. The ideal system will balance both performance load and storage load in order to truly optimize the use of resources. However, by showing that performance load and storage load can be balanced respectively, combining them naturally becomes a question of algorithmic feasibility, which would be left to future research.

Like HDFS, all metadata will be stored in memory on the central server, while the application data is separately stored on follower servers.



The above figure shows the design infrastructure. Clients send file system requests through the master server, which stores all metadata associated with the key-value pairs and followers. Requests are accepted by a Java Spark web API through HTTP requests. On initial write, the master randomly chooses among the known followers to store the file. This server is designated as the "original" server, and the file can never be removed from this server via rebalancing procedures. The original server is kept as a simple guarantee that a file will always exist somewhere in the file system.

In addition to a mapping of each file to its original server, the master server also maintains a mapping of each file to all duplicates, a mapping of each server to all files stored on that server, timestamps per file, and a buffer of the most recent files and servers accessed. Our implementation sets the buffer size at 40 for testing, but in production this can be much higher. SpeedReader was primarily built with speed (reads) in mind and does not plan for frequent updates. Thus, older unread file metadata can be un-cached from Redis memory and eventually garbage-collected (or backed up to disk).

Storage Tier

We have designed our storage tier to handle versioned reads and writes issues by the master.

All writes to stores are versioned. When writing a value, clients submit a version number along with the key and value to the master, which then sends the key, value, and version to a randomly-chosen Follower. The master also gives the chosen Follower a list of stores which the write should be propagated to. If the version sent by the client is greater than the latest version seen by chosen Follower, the Follower increments the version number for that key and the changes issues by the client overwrite previous versions. Otherwise, the Follower adds the client write value is added to version list for that key. The Follower then returns a list of currently stored values for the written key to the master.

Once a key/value is written to an individual Follower, that Follower then begins the propagation of the write to all other Followers which have that key. The chosen Follower issues an asynchronous write command to some number of the other "second-level" Followers designated by the master for propagation of

the write. The chosen Follower also splits the remaining stores designated by the master for propagation among the second-level Followers. These second-level Followers then repeat the process until the write is propagated to all designated replicas.

Reads are issued to any store which holds a desired key. Each read returns an object containing all versions of the value for that key as well as the latest version number for that key. Each store may contain a different set of values for a key, depending on which versions the store has seen.

The storage tier also presents an interface for key/value replication and value reconciliation. During key/value replication, each Follower accepts a key with all versions of the value written for that key. If the key is already present on the Follower, it rejects the replication. Value reconciliation works in much the same way as a write.

Master Tier

SpeedReader uses a single master node that stores metadata about each follower and its data. Redis, an open-source and networked server, was chosen for use on the master server two main reasons:

1. In-memory: Many of the rebalancing procedures and metadata maintenance require accesses to a large portion of metadata. By keeping this in-memory, overhead for maintaining master is minimized. In-memory procedures done in master should be overshadowed by the actual I/O to followers, which are known to take much longer to finish.

2. Data structure store: Redis supports storage of much more than just key-value pairs. This simplified our development process, as we could use its built-in set, map, and list operations.

The master node is the bridge between the client tier and the server tier (which consists of n number of followers). Clients send GET and POST requests to the master node, which has the appearance of a monolithic server. We use Java Spark as the framework for creating and controlling the endpoints. The master node then completes these requests by querying the appropriate followers in the server tier. In our implementation, we have implemented this by calling functions in references to *FollowerService* objects, since we are testing locally. In real-world scenarios, this can easily be extended to remote procedure calls with minimal modification. Communication between the master and the server tier is handled via a simple, internal API that allows for reads, writes, and replication.

Since the master node controls the location of all keys-value pairs, it is able to redistribute data across followers without client intervention. An important application of this feature is load balancing, which spreads the burden of handling read requests across multiple followers. In general terms, this works by duplicating heavily-requested data to more followers via read rebalancing. We also move data from busy servers to less busy servers via server rebalancing. Details of the algorithm will be described further in this paper. It is important to note that each individual follower in the server tier has no knowledge of the status of other followers, which allows for SpeedReader to be extremely flexible in adding new followers.

Client Tier

The main purpose of the client is to handle reconciliation. On a given read, the client gets the latest version number of a file along with all the values corresponding to that version number. If there is more than one values, the client must follow up the read with a write that reconciles by picking one of the values, similar to the way Amazon's Dynamo key-value store pushes reconciliation to the client. If concurrent reconciliations occur, the client continues to reconcile until there is only one value. On a given write, the client must pass in the version of the file being written (usually the latest plus one) along with the new value to be written. This makes the assumption that the client has

cached the file with a read before the write.

Load Balancing Design

We have updated the two algorithms for load balancing from DDDFS for SpeedReader: file balancing and server balancing. For file balancing, the basic idea is to predict when a file will be read by many clients and to duplicate that file across many servers prior to that event. By doing so, clients can read in parallel, eliminating constraints imposed by network throughput or server performance. Server balancing focuses on a related idea when files on a specific server will be read by many clients. In this case, files are moved from the busy server to less busy servers. This should lead to the same performance improvements as file balancing.

It is necessary for SpeedReader to keep track of recent reads by clients in its metadata in order to perform these two operations. The file balancing operation determines the number of desired replicas for each file using the recent read metadata, and duplicates or deletes files accordingly. Servers designated to store duplicates are chosen at random among the cluster. In the simplest algorithm, a file's desired number of replicas is proportional to the number of recent reads. In our implementation, file balancing occurs every 10 seconds and server balancing occurs every 22 seconds. These intervals can be modified to fit the specific needs and read requests of each application. Thus, SpeedReader has only eventual consistency, depending on both the balancing operations and when each follower schedules its actual file transactions. Such a model is fitting for a system with a large number of reads of unlinked data. By implementing these two operations with low performance and space overhead, we hope to show that performance load balancing should be a consideration for future distributed file systems.

Like HDFS, all metadata will be stored in memory on the central server, while the application data is separately stored on child nodes. Additionally, we will model our file movements and replications after HDFS not only for simplicity, but also as proof that such movements and replications can be done quickly (as shown by HDFS). By showing the viability of the low overhead "performance load balancer" in DDDFS, we aim to elucidate the possibility of using such a load balancer in existing file systems such as HDFS.

III. Difficulties

A difficulty encountered while using Redis was to find a memory and time-efficient algorithm for calculating how many duplicates were desired for a file. A naive algorithm would be to just keep access counts for every file from the creation of the file but that potentially uses memory exceeding the system (practical limit of around 100 GB).

IV. Results

Our implementation's source code can be found at https://github.com/alexdao/SpeedReader. To demonstrate the functionality of SpeedReader, a test suite is provided to simulate write, read, and update requests for a number of transactions. In the following test cases, files correspond to keys.

I. Simple test case

Two clients perform writes for two different files: file1 and file2 of version 0.

```
Client id 1 : Write file1 {version: 0,
    values: [2]}
Client id 1 : Read file1 {version: 0,
    values: [2]}
Client id 2 : Write file2 {version: 0,
    values: [4]}
Client id 2 : Read file1 {version: 0,
    values: [2]}
```

To simulate a concurrent write to file2, we perform another write with the same version number. We use version numbers as a proxy for time in our application, to handle concurrency. If a client writes with a version number that is

equal to or less than the current version number in the datastore, then we know that the client is writing without up-to-date knowledge. For this reason, we store all values of the current version number in a list, and resolve the inconsistency when a client reads the value later. As you can see below, file2 then gets mapped to both values 4 and 5, which is eventually reconciled by the client.

```
Client id 1 : Read file2 {version: 0,
    values: [4]}
Client id 1 : Write file2 {version: 0,
    values: [4, 5]}
Client id 1 : Reconciling file2 {version:
    0, values: [4, 5]}
```

To demonstrate performance load balancing, we perform 10 reads in a row on file2. After the balancing operation, file2 gets duplicated to 6 more servers (out of 10), because it is such a heavily-requested file.

```
File locations with (serverNum, value)
    pairs:
file2: (0,[10]) (1,[10]) (2,[10]) (4,[10])
    (6,[10]) (7,[10]) (9,[10])
file1: (4,[2]) (7,[2])
```

II. Testing write propagation and concurrent writes

To demonstrate concurrent writes on file systems, we first do 100 reads that 10 clients that copy file1 10 times. Then, each of the clients does a write. Due to the same version vectors, all the values for these writes are stored.

```
File locations with (serverNum, value)
    pairs:
file1: (0,[0]) (1,[0]) (2,[0]) (3,[0])
    (4,[0]) (5,[0]) (6,[0]) (7,[0]) (8,[0])
     (9,[0])

Client id 1 : Write file1 {version: 0,
    values: [0]}
Client id 2 : Write file1 {version: 0,
    values: [0, 1]}
Client id 3 : Write file1 {version: 0,
    values: [0, 1, 2]}
Client id 4 : Write file1 {version: 0,
    values: [0, 1, 2, 3]}
Client id 5 : Write file1 {version: 0,
    values: [0, 1, 2, 3, 4]}
Client id 6 : Write file1 {version: 0,
    values: [0, 1, 2, 3, 4, 5]}
Client id 7 : Write file1 {version: 0,
    values: [0, 1, 2, 3, 4, 5, 6]}
Client id 8 : Write file1 {version: 0,
    values: [0, 1, 2, 3, 4, 5, 6, 7]}
Client id 9 : Write file1 {version: 0,
    values: [0, 1, 2, 3, 4, 5, 6, 7, 8]}
Client id 10 : Write file1 {version: 0,
    values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}
```

Finally, on reads, concurrent reconciles occur as the multiple values are reduced down to just one value.

```
Client id 1 : Reconciling file1 {version:
    0, values: [0, 1, 2, 3, 4, 5, 6, 7, 8,
    9]}
Client id 1 : Read file1 {version: 1,
    values: [0]}
Client id 2 : Write file1 {version: 1,
    values: [0, 1]}
Client id 3 : Reconciling file1 {version:
    1, values: [0, 1]}
Client id 3 : Read file1 {version: 2,
    values: [1]}

File locations with (serverNum, value)
    pairs:
file1: (0,[1]) (1,[1]) (2,[1]) (3,[1])
    (4,[1]) (5,[1]) (6,[1]) (7,[1]) (8,[1])
     (9,[1])
```

V. Future Work

SpeedReader keeps open a number of avenues for further improvements in systems for read-optimized workloads. The DDDFS paper itself notes that the performance read balancing algorithms implemented by DDDFS and used here are quite simple and can be improved. Work can be done to benchmark performance increases and compare with existing systems. SpeedReader also currently does not provide full eventual consistency for writes and our current implementation is a simulation rather than an actual distributed implementation.

Workload Analysis

The GFS paper claims that the disproportionate access of a single file occurred in some rare cases, but that this was not a concern [2]. By benchmarking performance of existing distributed file systems and comparing the performance difference between a "regular" workload, a workload consisting of evenly distributed file accesses, and a workload with concentrated file accesses, we can see how much of a bottleneck concentrated file accesses can cause.

Rebalancing Algorithm

The rebalancing algorithms implemented for the DDDFS are extremely simple, as the DDDFS paper points out. It is possible to use different algorithms, or improve upon the existing ones so inefficient changes do not occur. For example, moving a key x from server 1 to server 2, then replicating key x on server 1 would be inefficient. More work should be spent on avoiding such cases. An example of a completely different algorithm would be using heuristics to predict future accesses and the number of desired replicas. Currently, in server rebalancing, we move a random key from the most busy server to the least busy. Instead of a randomly chosen key, we could move the most heavily accessed files (in near $O(1)$ find speed). In addition, read balancing and server balancing are currently both performed at fixed intervals. The balancing should ideally be performed during lulls of activity. As the actual balancing process across follower servers is expensive (due to I/O), balance detection and calculation should be performed often but only executed when deemed necessary.

Consistency

SpeedReader currently handles failures by writing to live servers and waiting for subsequent writes. This does not provide full eventual consistency, since if a Follower misses a write (either because of a failure or a network issue) it will not eventually get that update. We believe that the system we have will work for most scenarios, since writes will be written to most replicas and new writes after a failure will be written to all replicas. However, a future effort could implement full eventual consistency for SpeedReader.

Implementation

We have currently implemented SpeedReader as a simulation on a single machine with limited asynchronous components. The system could be implemented on an actual distributed system in the future.

References

[1] Alex Dao, Jiawei Zhang, Danny Oh Detailed Diagnostic Distributed File System *CS510: Graduate Operating Systems* Duke. 2016.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels Dynamo: Amazon's Highly Available Key-value Store *SOSP 2007, October 14?17, 2007* All Things Distributed.

[3] Valeria Cardellini, Michele Colajanni, Philip S. Yu Dynamic Load Balancing on Web-server Systems *1999 Internet Computing vol. 3, no. 3, pp. 28-39, May-June 1999* IEEE. 1999.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung The Google File System *ACM Symposium on Operating Systems Principles* Google. 19 October 2003.

[5] Java-Spark http://sparkjava.com/documentation.html

[6] Jeffrey J. Hanson An introduction to the Hadoop Distributed File System *developerWorks* IBM. 1 February 2011.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler The Hadoop Distributed File System *MSST*

'10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies Yahoo!. 2010.

[8] Redis http://redis.io/