

HBase Concrete Architecture

RegionServer

Oct. 31 2018

Daniel McVicar	213027479
Yahya Ismail	213235403
Rafay Sheikh	213033451
Daniel Fortunato	216796443
Adham El Shafie	212951018

Abstract	1
Introduction	1
System Overview	1
The Conceptual Model	1
The Reflexion Model	3
The Derivation Process	4
Subsystem Overview: The Region Server	6
The Write Ahead Log	6
The Memstore	8
The BlockCache	8
The HFile	10
Architecture Analysis	11
HBase	11
RegionServer	12
Design Patterns	12
Concurrency	13
External Interface	13
Use Cases	13
Client Write Use Case	14
Client Read Use Case	14
Lessons Learned	14
Conclusion	15
Appendix A: Naming Conventions	16
Appendix B: Bibliography	17

Abstract

This report investigates the concrete design of the database program Hbase. It investigates the general subsystem structure and takes a detailed look at the RegionServer subsystem. The derivation process used to form this concrete analysis is explained and the differences between the concrete analysis and the conceptual analysis of assignment 1 are discussed. An architectural analysis of Hbase in general and RegionServer specifically is provided. Concurrency and how it is enabled by the architectural model is investigated. Finally, two use cases provide working examples of Hbase and the lessons learned throughout this assignment are summarized.

The key discoveries are focused on the architecture of the system. Hbase turns out to be more interdependent than previously suspected while also holding two new subsystems that suggest a new Model View Controller architecture.

Introduction

Apache Hbase is an open source database system designed to handle a large number of concurrent users. Modeled after Google's Bigtable paper^[1] Hbase provides a horizontally scalable database structure that is ideal for handling large sets of data that are read often but written relatively infrequently. Companies that use Hbase include Facebook and JPMorgan Chase^[2], making it an important and widely accepted standard database system.

System Overview

To start off, we focused on what we had in the previous assignment (i.e. what we already knew and had to work with), which was the conceptual model of HBase. In doing so, we helped ourselves get started in using LSEdit and editing the raw and contain files in order to visualize exactly what it is we needed to do. Once the conceptual model was created, we had to step it up and decide on what the concrete one would look like, which brings us to our 'reflexion' model. These two models can fully represent exactly what it was that we had to understand and learn by the end of this assignment.

The Conceptual Model

In Figure-1, we simplified the already established conceptual model of HBase from the first assignment and re-drew using only two elements (included in the legend of Figure-1). The arrow shows dependency where the head of the arrow points to the depended-on entity, where an entity in Figure-1 represents a subsystem/subcomponent of the top-level of HBase.

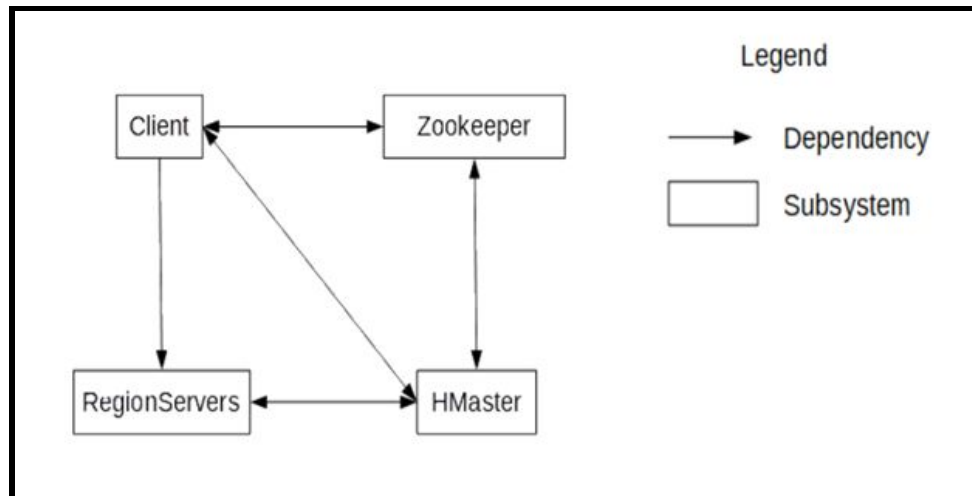


Figure-1 – Simplified version of conceptual architecture discussed in previous assignment

From what we can see, there is a 2-way dependency relationship between Client and Zookeeper, Zookeeper and HMaster, HMaster and Client, and HMaster and RegionServers. The relationship between Client and Zookeeper is there as the client's commands and/or inputs that relate to the data in HBase need to be analyzed by the Zookeeper in order to check for any failed components and/or unavailable data via the metastable (i.e. signal is sent to Zookeeper to check for system functionality, a response is sent back from Zookeeper to Client).

Zookeeper has a relation with HMaster as there are 'heartbeats' that Zookeeper sends to HMaster to know if the current HMaster component is failed or is still up and running in order to execute the user's commands (i.e. Zookeeper sends a 'heartbeat' signal to HMaster, HMaster responds or doesn't and Zookeeper comes to a decision based on the response).

HMaster and Client have a relation that is relatively weak ^[3] but is mainly there for administrative operations ^[4] of which the client gets information on the response of HBase to their inputs. Similarly, HMaster has a relation with RegionServers as HMaster manages the different regions in HBase and regularly checks on the health of the different RegionServers ^[5], hence the 2-way relation between them.

Finally, Client and RegionServers are in a 1-way relationship where the Client depends on RegionServers, unlike the previous one where there were always 2-way relationships between entities. The reason for this is because the RegionServer does not depend on Client in any way. The only dependency here is a result of the Client simply inputting read/write commands and the Client depending on the RegionServer for feedback or a response (via related data) with regards to the command ^[6].

After researching and understanding all of the above, we moved onward and went through with creating the concrete model. From the concrete model we simplified it into a 'neater' form and we came up with the 'reflexion' model.

The Reflexion Model

We recreated the conceptual model in LSEdit and, after placing the components into the four subsystems that we came up with previously, we were left with quite a few subsystems that did not relate to any of them for various reasons. Through vigorous research of what all the components did, we came up with a very brief understanding of the responsibilities of all the other subsystems. Through this understanding, we decided that there were two new subsystems in HBase that were not there in the conceptual model.

Firstly, the main reason we decided on two new components and not just one, or not just filter it into the other four subsystems was due to the fact that they were handling very different responsibilities and aspects of HBase that there weren't many common areas between them. We chose two as that was the smallest number of entities that it would take to organize the rest of the components without much redundancy.

To begin with, we ended up with two new subsystems, namely TestingAndSetUp and BusinessLogic as shown in Figure-2. TestingAndSetUp deals with the initialization of HBase on a new system via hbase-examples^[6] and hbase-archetypes^[7] that come preset with the system to help a new user get accustomed to the workings and functionality of HBase. It also deals with maintaining the healthiness of HBase throughout its operation via testing. Components that deal with this are hbase-annotations^[8] and hbase-testing-util^[9] where annotations are Java annotations that ensure that the annotated method satisfies what the annotation enforces^[10].

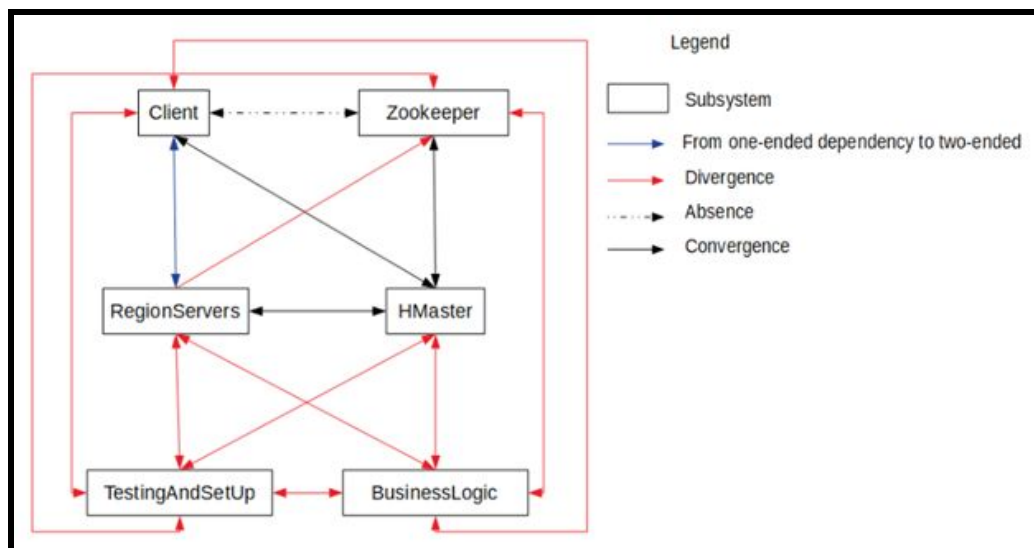


Figure-2 – Reflexion Model with divergences, convergences and absences of dependencies.

The other subsystem that is introduced is the BusinessLogic subsystem. This subsystem takes care of all the background features of HBase. This is how the commands that a Client inputs is translated from whichever supported language it's inputted from to commands that HBase can implement via hbase-thrift^[11] or hbase-rest^[12] which are APIs for HBase. This also takes care of all the metrics, the client and server ends' implementations and also the hadoop1 and hadoop2 compatibility interfaces that HBase was built upon^{[13] [14]}.

Due to how important these new subsystems are in order for the entire system to be healthy and work as intended, they both are in completely two-way relationships with every other component in the system, even each other. The 'TestingAndSetUp' subsystem talks to every other subsystem in order to run checks and set HBase up, and BusinessLogic communicates with all of them as that is how HBase functions as it should in the background.

A few things to note is that these weren't in the conceptual as the conceptual only deals with what HBase does as a whole and not how it completes every individual instruction, which explains that vast amount of divergences in Figure-2.

The other interesting one we found was the two-way relation between RegionServer and Zookeeper. At first, we thought there was an issue with our diagram, but the more we thought about it the more it made sense. The dependency between Zookeeper and RegionServer is there due to the Zookeeper maintaining a list of all the live components in the system and the RegionServer being one of them^[15].

RegionServer and Client got a two-way dependency as, other than the aforementioned dependency of Client on RegionServer, the other way is true as well due to the RegionServer being able to accept commands directly from the client without having to go through any of the other subsystems.

The final difference we found was the absence of a dependency between the Client and the Zookeeper. After thorough investigation of why that could happen, we concluded that it was due to the meaning we gave the Client subsystem to have. Usually it is the implementation on the client's end and all the possible features a client can process and compute on their end, mainly create, read, update and delete (CRUD) operations^[16]. We rearranged it such that, both the client and server implementations are both in BusinessLogic and the Client subsystem handles all the different ways a Client can interact with HBase through API, whether it be HTTP via REST, or programming languages via Thrift. When we understood this, Figure-2 was immediately clarified.

The Derivation Process

Deriving the subsystems was a task that took up a good slot of our time as it had to be done at least twice; one for the top-level subsystems and one for the

subsystem that was assigned. The derivation process consisted of the following steps: first, the known subsystems from the conceptual model were inputted into the contain file, along with any file-paths or directories that were suspected to be a part of those subsystems. Once all of those were organized into their respective entities (i.e. subsystems/subcomponents), the LSEdit display was looked at again. As can be seen in Figure-3, the 4 main subcomponents that were previously discussed are visible with their respective labels. Looking at Figure-3, one can conclude that there are a variety of other subsystems that HBase comes with that do not belong in any of the conceptual ones.

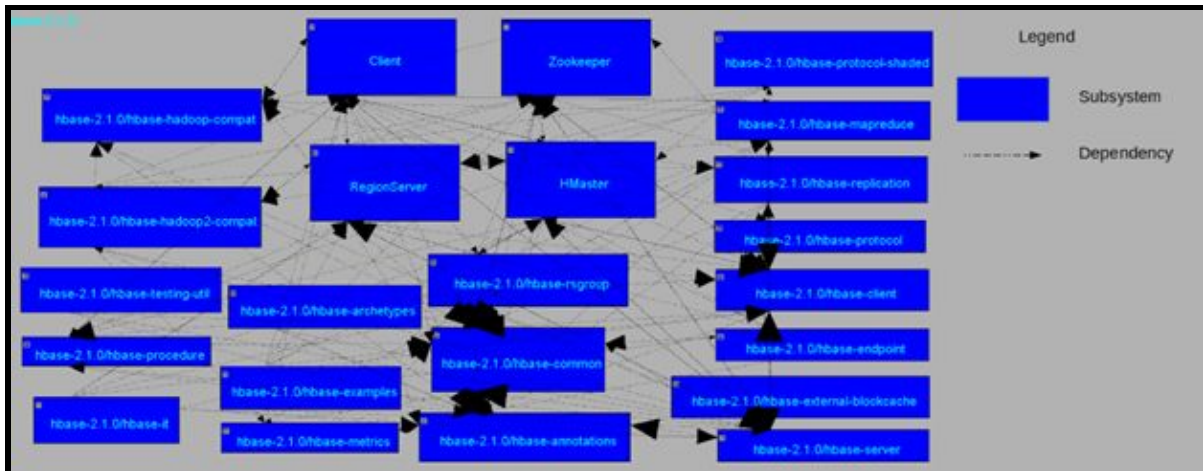


Figure-3 – Main subsystems are organized while the others are left out

Second, a lot of research was done to understand the responsibilities of every other subcomponent seen in Figure-3. Using that research, a consensus was reached through the group that two new subsystems would suffice to group up the rest of the entities. As previously discussed, the two new ones would be TestingAndSetUp and BusinessLogic.

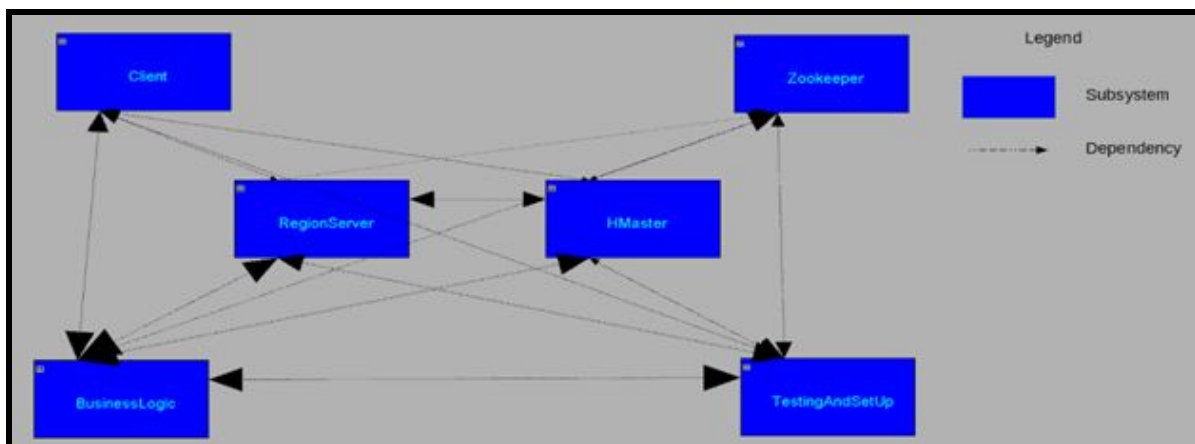


Figure-4 – The final concrete model after the derivation process

All the stranded subcomponents are filed into the two newly added subsystems. This provided the final concrete model for the top-level of HBase as

shown in Figure-4. Once the above model was derived, a reflexion model was created which is shown in Figure-2.

Subsystem Overview: The Region Server

Our goal is now to validate our Reflexion Model (Figure-2) through understanding the external relations each RegionServer subcomponent has with the other HBase components since we already have gone over the interactions that take place inside the RegionServer in assignment 1 and illustrated in the Use Case Diagram (Figure-12).

The Write Ahead Log

The Write Ahead Log, abbreviated WAL, is the first thing that is accessed when trying to persist data in HBase. From assignment 1 we know that the WAL records all data updates in HBase, it is one of the mechanisms who ensures that there is no data loss in the system. If there is a RegionServer crash the WAL of that RegionServer can replay the changes that didn't take effect.

After the derivation process, already explained, we obtained a series of external dependencies illustrated in Figure-5.

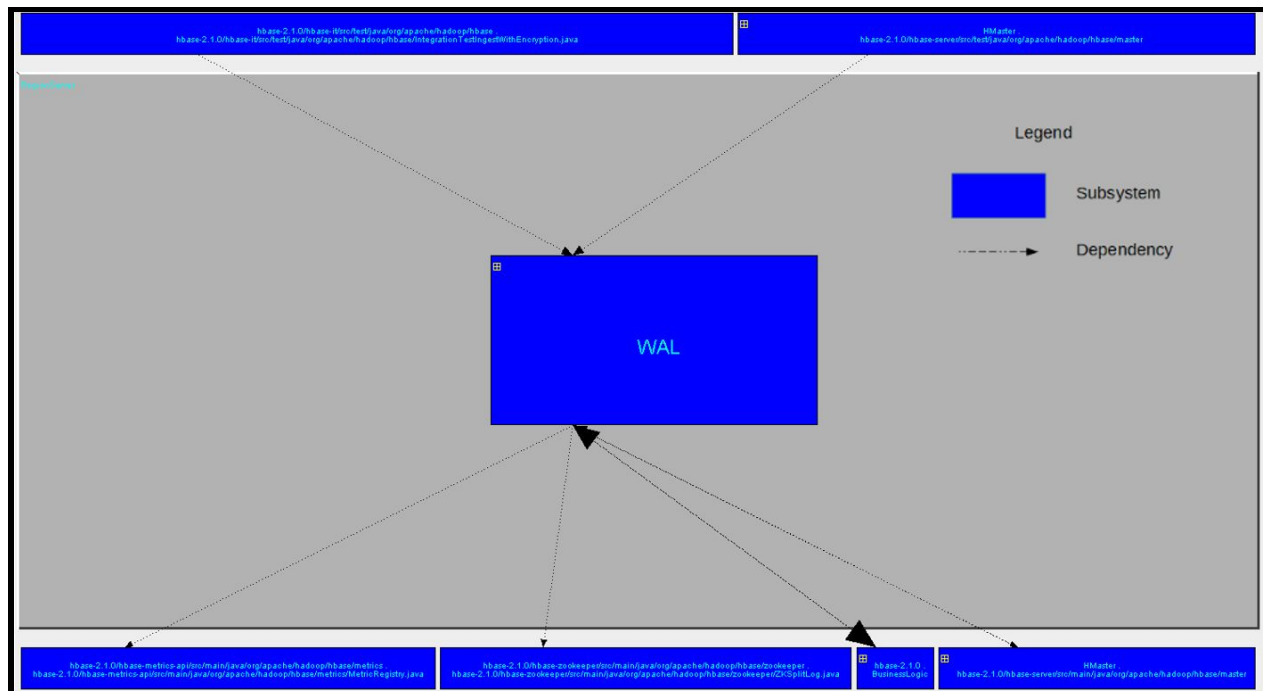


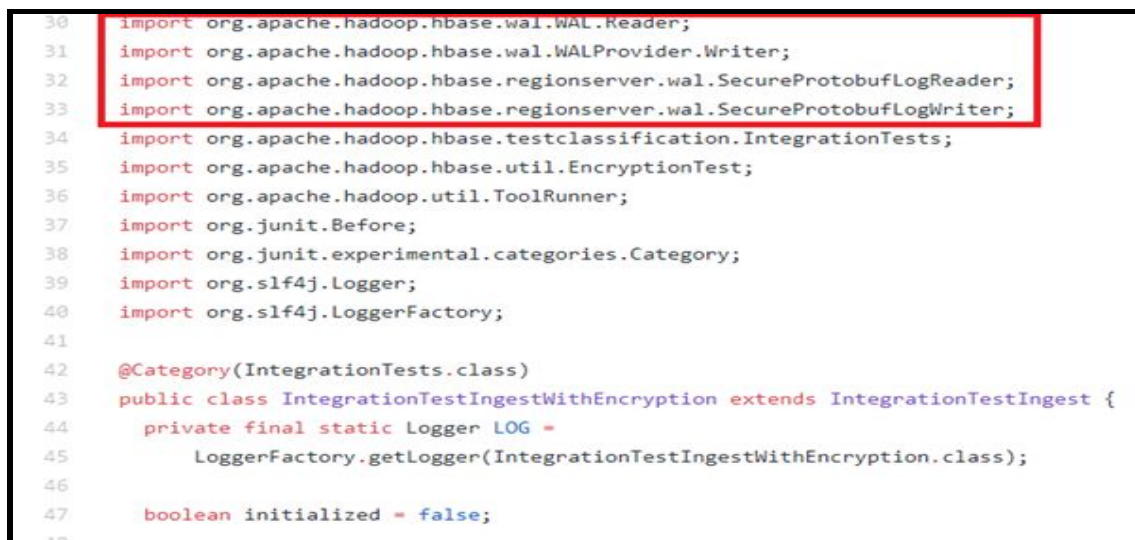
Figure-5 – WAL Dependencies/Relations

As we can see in Figure-5, the blue boxes represent the external dependencies, and the direction of the arrows represent the direction of the dependency. They can be bi-directional, outgoing or ingoing.

The first impressions we got when analyzing Figure-5 is that there weren't a lot of external dependencies, so we thought that perhaps we got something wrong. But after further analyzing the role of the WAL in HBase we changed our minds. As we explained before the main role of the WAL is done inside the RegionServer and it isn't usually used by other components but, as we are going to see, is still used.

We then breakdown each dependency we found, starting from the top-left one until the last one, the bottom right one.

The first two dependencies as the arrows indicate are ingoing, meaning they depend on the WAL to function. Looking at their directory path we are left with /hbase-it/src/test and /hbase-server/src/test. Both represent testing directories, so it only makes sense that the dependency is ingoing, the tests are based on the component not the other way around. For the first one we also have a filename, *IntegrationTestIngestWithEncryption.java* meaning it is the only file dependent on WAL in that directory. To verify our findings, we went to the HBase's GitHub^[17] page and as we can see in Figure-6, highlighted in red are the dependencies. These two dependencies could be placed under the component we called TestingAndSetUp, validating our Reflexion Model (Figure-2).



```
30 import org.apache.hadoop.hbase.wal.WAL.Reader;
31 import org.apache.hadoop.hbase.wal.WALProvider.Writer;
32 import org.apache.hadoop.hbase.regionserver.wal.SecureProtobufLogReader;
33 import org.apache.hadoop.hbase.regionserver.wal.SecureProtobufLogWriter;
34 import org.apache.hadoop.hbase.testclassification.IntegrationTests;
35 import org.apache.hadoop.hbase.util.EncryptionTest;
36 import org.apache.hadoop.util.ToolRunner;
37 import org.junit.Before;
38 import org.junit.experimental.categories.Category;
39 import org.slf4j.Logger;
40 import org.slf4j.LoggerFactory;
41
42 @Category(IntegrationTests.class)
43 public class IntegrationTestIngestWithEncryption extends IntegrationTestIngest {
44     private static Logger LOG =
45         LoggerFactory.getLogger(IntegrationTestIngestWithEncryption.class);
46
47     boolean initialized = false;
```

Figure-6 – Highlighted WAL dependencies in file *IntegrationTestIngestWithEncryption.java*^[17]

The third and fourth dependencies are outgoing. Looking at the directories and filenames, we guessed first and verified later when analysing the code that these contain information used by the WAL to function. The directory *base-metrics-api* contains the settings to program the WAL. The *ZKSplitLog.java* file^[18], found in the Zookeeper component, is used by the WAL to check for the health of a node. As already discussed the metric directory belongs to the *BusinessLogic* component which is also represented in the fifth dependency with two bigger arrow heads representing a strong dependency between the two. All of

this makes sense according to our Reflexion Model (Figure-2) and definition of the BusinessLogic component.

Finally the sixth node represent the 2-way dependency of HMaster and the WAL which, as already explained, are codependent.

The Memstore

The Memstore is where the data is stored after it has been successfully written to the WAL. From assignment 1 we know that it is a write buffer where HBase stores data in memory (volatile memory) before doing a permanent write to an HFile. Following the same thought process as we did for the WAL, Figure-7 shows us the external dependencies we found for the MemStore.

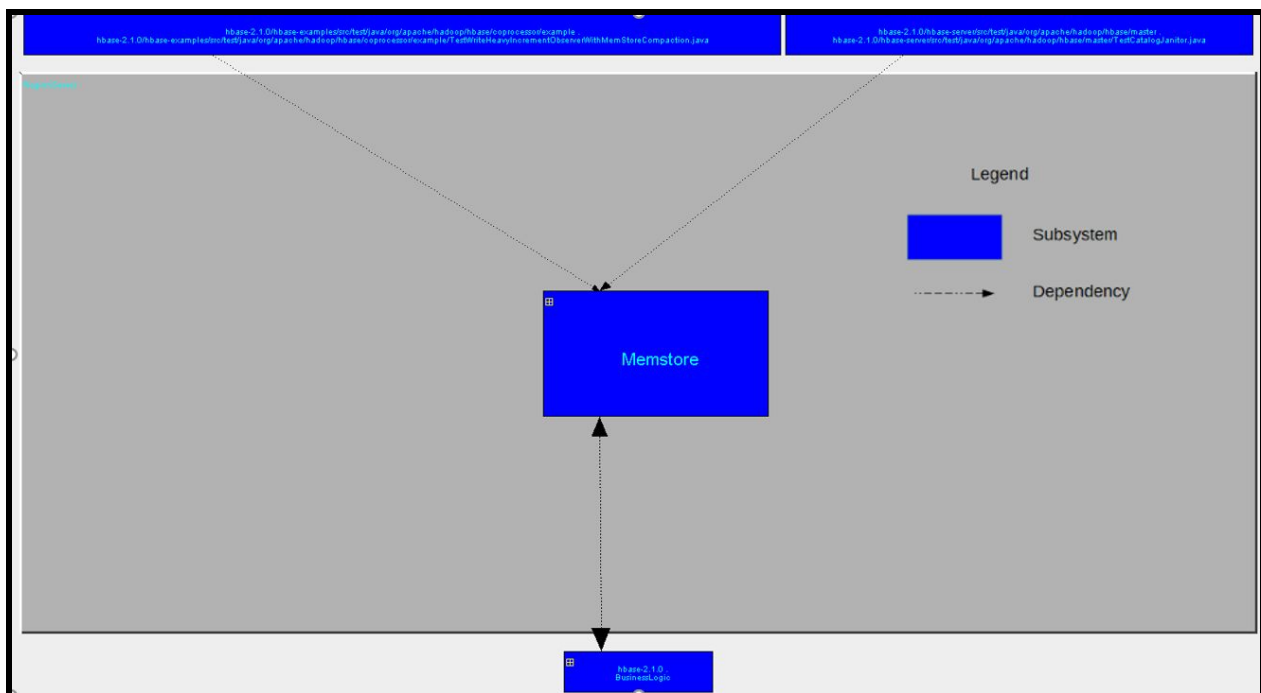


Figure-7 – MemStore Dependencies/Relations

As before we can see in Figure-7 that the first two dependencies are ingoing and are tests. Following the same procedure as for the WAL we went to HBase's GitHub^{[19][20]} and verified that the dependency really existed and belonged to the TestingAndSetUp component which validated our Reflexion Model (Figure-2).

The third dependency also does not come at a surprise if we take into consideration the role of the MemStore and our definition of BusinessModel, both are codependent in a lot of aspects.

The BlockCache

BlockCache is one of the major subsystems of the four in the region server. It is used mainly in read operations when data is to be read from the HFile in

response to a client's request. Before we examine the external dependencies of BlockCache we need to understand what it is and its functionality. BlockCache is like the memory cache in Memstore in that both Memory cache and BlockCache are in the RAM for cached storage to take place. A block is a single unit of I/O in HBase and it allows for the random reads what makes HBase so efficient. Memstore is used mostly for edits and write operations. The instance of the BlockCache is created during the region server start up and this one instance is retained for the lifetime of the process. Also, there is just one single instance of the BlockCache which means all data from all regions hosted by that server share the same cache pool. The BlockCache is a heap implantation using a LRU (Least recently used) algorithm for pages stored in the cache. Now looking at Figure-8 we can see that BlockCache has two one-way dependencies. One with BlockCache and HFile and BlockCache and Business logic. We will talk about these external relations that we did not see in the conceptual architecture. However, using the system implantation and focusing on the read and write operations we were able to extract these relationships that exists between these subcomponents inside the RegionServers.

The first dependency that we see in Figure-8 is that BlockCache is dependent on HFile. This is because in a read operation when data is not found in the BlockCache it goes straight to the HFile to retrieve the data. Once it gets the desired data it keeps it in the BlockCache for future so data is readily available if called upon.

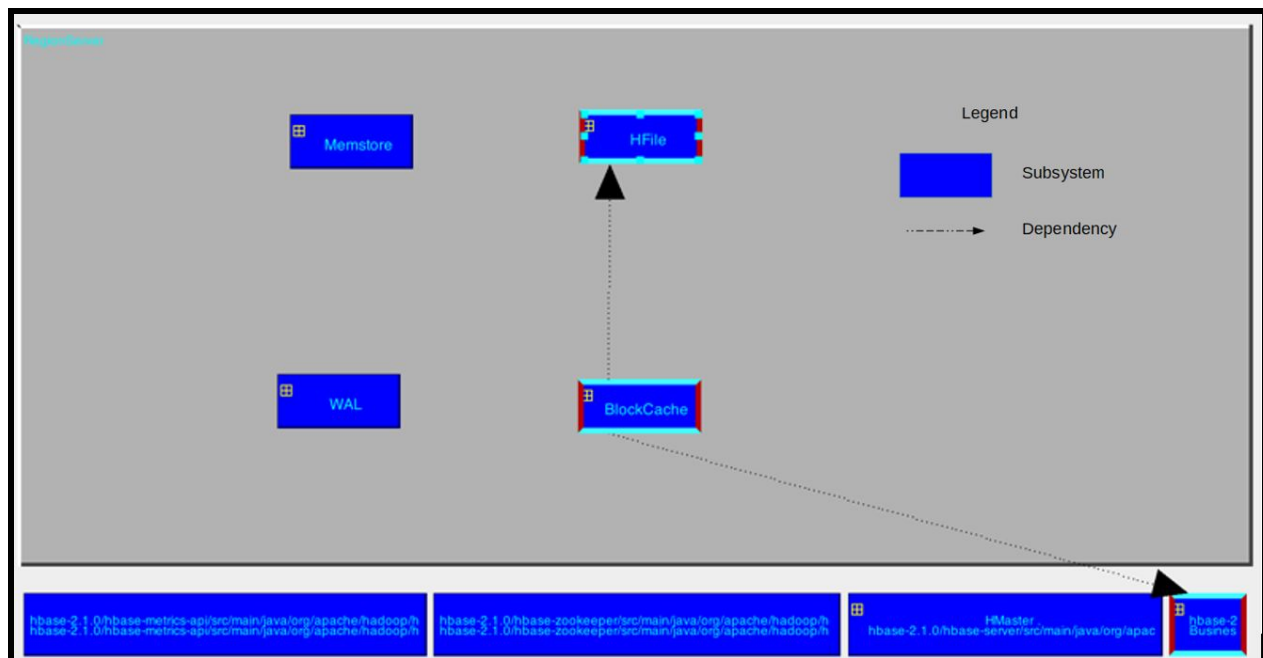


Figure-8 - BlockCache Dependencies/Relations

The other dependency we see in Figure-8 is BlockCache and Business Logic. We have already covered what the Business Logic subcomponent contains.

BlockCache has a depends on relationship with Business Logic because Business Logic contains directory for HBase Server and as explained above BlockCache is a single instance that resides in the server for all other regions to use the same cache pool. So we were aware HMaster has a relation with Zookeeper and RegionServers the concrete architecture allowed us to see the external dependencies of inner subcomponents of the RegionServers to the outer layer.

The HFile

HFile is one of the four subcomponents that resides in the region server main component. HFile is a very simple yet interconnected module used in read and write operations in HBase. This is clear when we realize that HBase HFile is the file-based data structure that stores data in HBase. This file is basically where the tabular information exists in physically form in a row sorted key-value pairs. As we can see in Figure-9 it shows us the dependencies of HFile as a whole inside the region server. The dependencies are relations in the level at which HFile exists with those in higher levels of the architecture structure. The main two relations that we need to look at is the dependencies of HBase HMaster to HFile and Business Logic to HFile. These are dependencies that we didn't realize existed in the conceptual architecture.

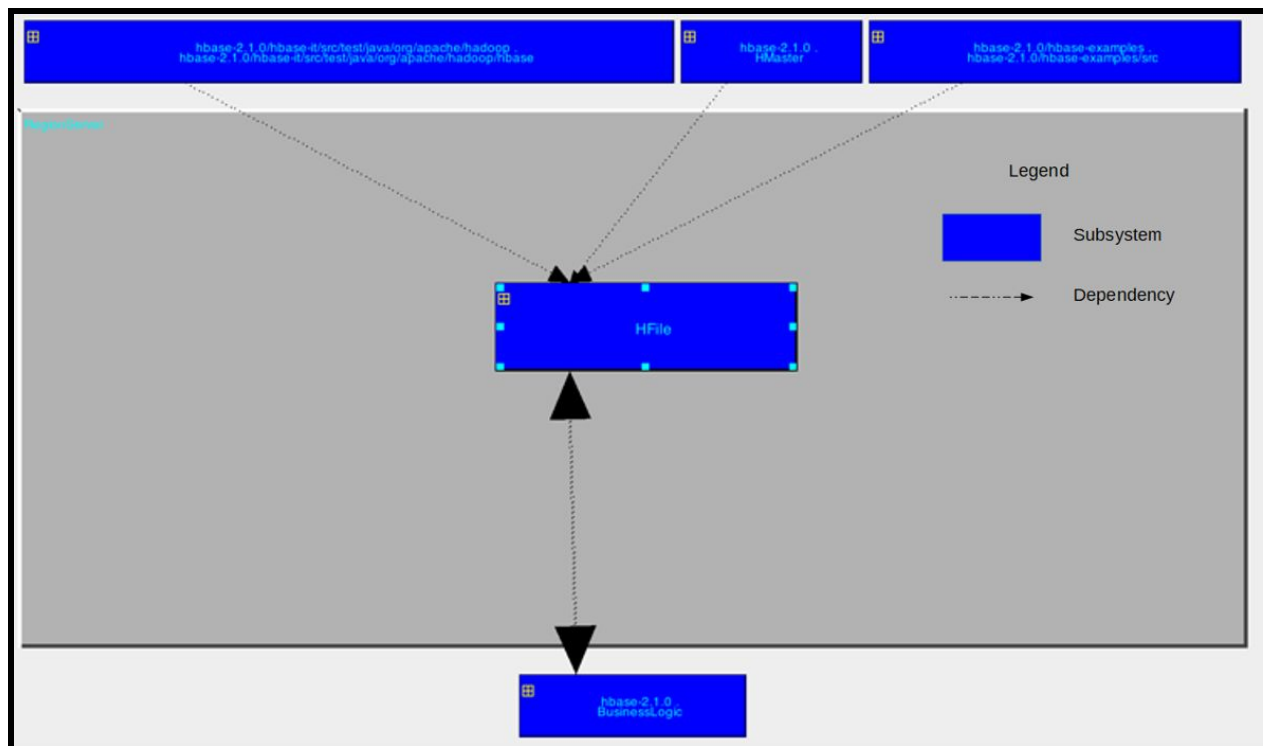


Figure-9 - HFile Dependencies/Relations

The relation between HMaster and HFile is that HFile resides in a region which is encapsulated in a region server. This relation exists because Zookeeper uses meta table to figure out the region and passes this on to HMaster. HMaster is

responsible for assigning regions on start-up and provides the interface for creating, deleting, updating tables (which eventually happen in a HFile). DDL operations are handled by HMaster and they require HFile. Hence the explanations above illustrate Figure-9.

The relation between Business Logic Subsystem and HFile is a two-way dependency as visible in Figure-9. This is because as we have explained above the term and containment of Business Logic subcomponent, we see that Business Logic contains files which encapsulate end user interface in HBase server and HBase client directories while also containing HBase MapReduce for storage. Business Logic depends on HFile because it is responsible for getting information from HFile when it passes it to user interface when client sends operations to table content and the other way around HFile is depended on Business Logic because Business Logic contains HBase MapReduce which allows for a divide and conquer approach to retrieve chunks of data when MapReduce splits into different regions and also helps in putting it back together. The dependencies explained above are illustrated in Figure-9.

Architecture Analysis

HBase

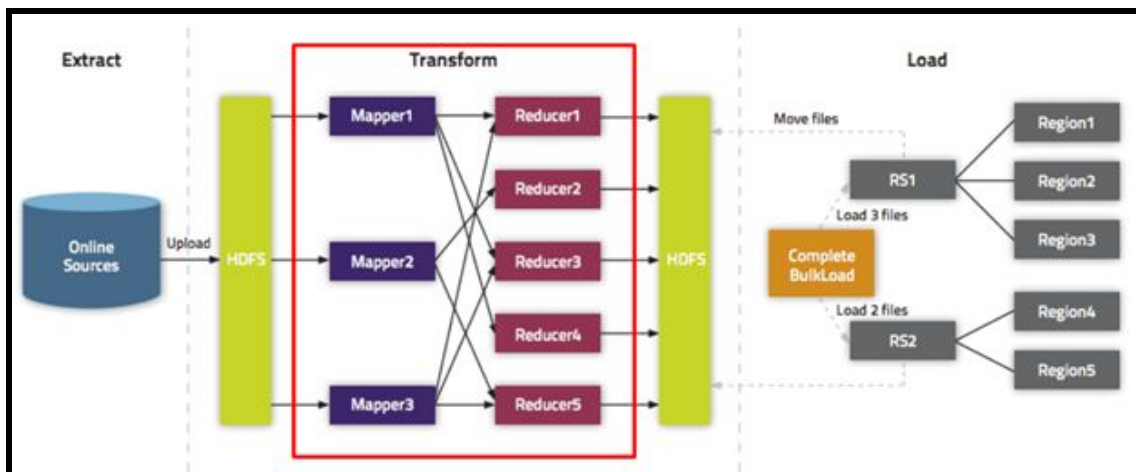


Figure-10 – MapReduce flow chart showing pipe-filter architecture style^[15]

Some architecture styles that were come across during research that have been discussed in class are pipe-filter, repository, layered and client-server styles. Pipe-filter in MapReduce functionality as there is a set order in which data is processed by MapReduce such that the data comes out in a savable format by the HTables as shown in Figure-10 below. Repository architecture style as every server has access to certain information, and on that server, many accounts may have access to that information. This results in there being a big slot of shared information stored that a lot of different users can access, which is the definition of repository architecture style. The layered architecture style is also there but it is not as distinct as it was described in the conceptual as there are two-way relations

between the different components of HBase. The final is Client-Server and this is obvious due to the relationship that hbase-server and hbase-client have.

RegionServer

The RegionServer subsystem is responsible for storing and retrieving information from the physical machines that host Hbase. Figure-11 shows the concrete architecture of the RegionServer subsystem. The three of the four main subsystems contained in RegionServer are arranged in a master/slave relationship: WAL, Memstore, and HFile. Each subsystem is responsible for a small piece of work with no dependencies between the subsystems. BlockCache exists as a layered component with HFile, and is responsible for providing quick access to often called data that would otherwise have to be retrieved from HFile.

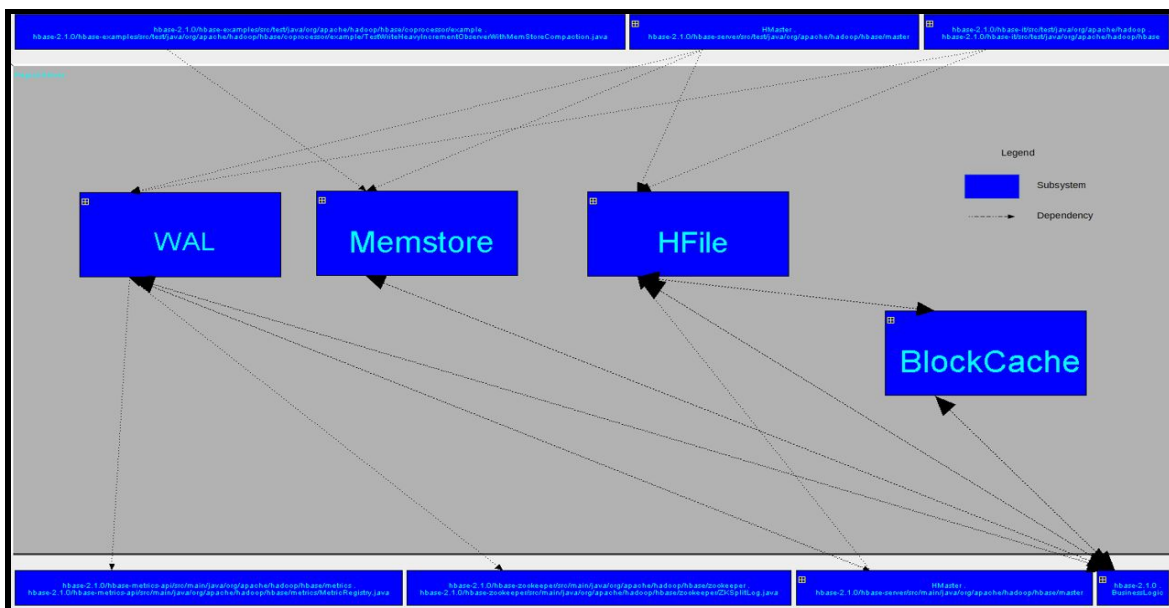


Figure-11 - RegionServer Dependencies/Relationships

RegionServer shares a large biconditional dependency with BusinessLogic, reinforcing the MVC architecture where RegionServer is part of the model and BusinessLogic acts as a controller for that model.

Design Patterns

While researching the different stranded subcomponents of HBase, a few design patterns, which were discussed in class, were come across. A few main ones to note are the adapter design pattern, façade design pattern, MVC (Model-View-Controller) design pattern and the Master-Slave design pattern as well. The adapter design pattern is used in the implementation of hbase-thrift as it is responsible for 'adapting' the input from the user into input that HBase can actually work with. The façade design pattern is used for the hbase-mapreduce functionality as it is built on top of Hadoop's MapReduce implementation with no way to edit or access the inner variables of hbase-mapreduce. The MVC design pattern is there as

the user uses a model of HBase (i.e. one of the interfaces) and then the model will, depending on what the input was, search through the model of HBase to satisfy the user's command and then view the result of the command to the user. The Master-Slave design pattern is there as well due to how HMaster interacts with the regionServers and regions as discussed above.

Concurrency

Supporting concurrency is a major requirement for any database system that needs to handle requests from many thousands of users. Hbase has two main features that allows it to support concurrency without compromising data integrity.^[21]

First, the RegionServer subsystem is designed to have multiple instances split between different physical machines. This naturally partitions the data, allowing concurrent read and write operations on separate RegionServers without any complications.

When two pieces of data are being manipulated inside a single RegionServer Hbase makes use of its second concurrency feature: row locks. To write to a row a row lock must first be obtained from the RegionServer holding the data. This row lock prevents any other attempts to read or write from the row until the lock is released. This ensures the data is never read part way through a write operation when the data is not consistent and two writes never fight to put data in a single row. Reads do not require a row lock to proceed so multiple reads can be processed at the same time. Writes must be processed sequentially. This is one of the reasons Hbase is best suited to data that is written rarely but read often.

External Interface

As mentioned before, HBase has a variety of different APIs that handle different languages. The main ones being hbase-thrift which is a locally integrated apache Thrift framework that can support languages such as C++, Java, Python, PHP, Ruby and many more ^[11]. The other main API that is integrated into HBase is REST (**RE**presentational **S**tate **T**ransfer), which makes HBase a RESTful system ^[12]. This allows HBase to support HTTP inputs from the user via request making implemented by the REST server.

Use Cases

Two use cases are presented to illustrate the concrete architecture that has been discussed in this document thus far. These two use cases are a successful client write use case, and a successful client read use case. These use cases were chosen as they give a very good overview of how RegionServer works as almost every major component is utilized in some way.

Client Write Use Case

A successful client write use case is shown in the upper portion of the sequence diagram in Figure-12. When the client requests a write, there are essentially four steps that the RegionServer follows. First the WAL is immediately updated in order to ensure disaster recovery is possible. Next, the request is communicated to the Regions inside the Region Server until a suitable Region is found that can store the data. The data is then communicated to the MemStore. Later down the line, the data will be flushed from MemStore into HFile, but this happens at a much later time and is decoupled from the actual write request.

Client Read Use Case

A successful client read use case is in comparison much simpler, and is illustrated in the lower portion of the sequence diagram in Figure-12. When the client requests a read, access is given to either the MemStore or the HFile, depending on where the data is stored. WAL is never updated, and in this sense the large difference between the two operations can be seen. While WAL must always be updated first in a write operation, a read operation is much more silent by comparison.

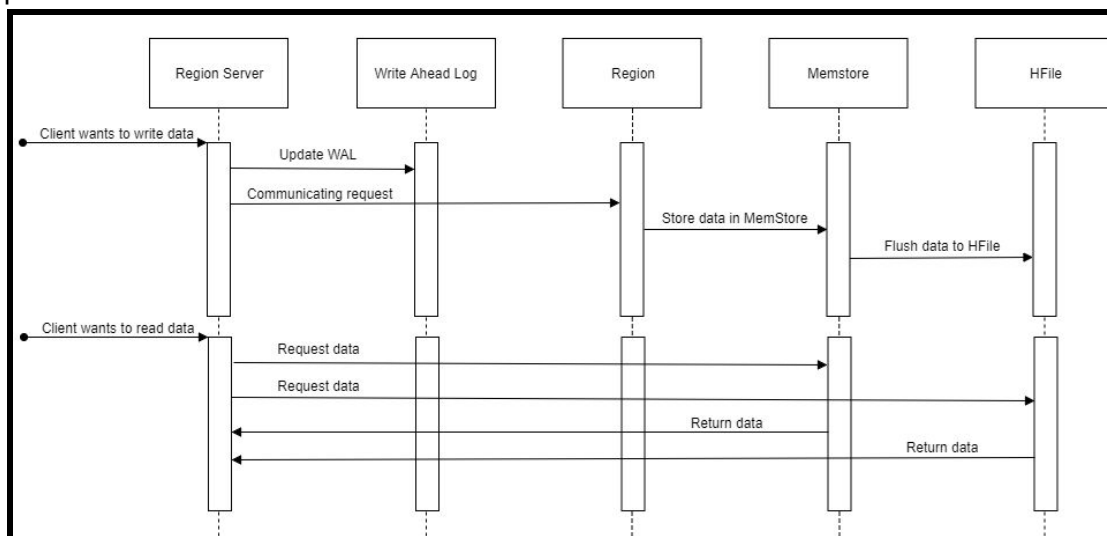


Figure-12 - Sequence Diagram Detailing Read/Write Use Cases

Lessons Learned

In comparison with the conceptual architecture analysis, this concrete architecture analysis yielded more robust lessons. These lessons could not have been learned without delving deep into the implementation of HBase. They can be summarized into three succinct lines of thought.

First, analyzing the concrete architecture uncovers many hidden subsystems that could not have been found otherwise. These are notably the BusinessLogic and

TestingAndSetUp. These two components might have been guessed at by examining the conceptual architecture, but without actively analyzing the underlying implementation, one could not have hoped to uncover the depth of these systems otherwise.

Secondly, conceptual subsystems can have components that spread throughout the code hierarchy, making it difficult to actually pinpoint where they lie in the concrete architecture. This made organizing and identifying subcomponents in the concrete architecture more difficult than we could have imagined. Ultimately, while the abstraction is helpful in identifying basic functionality, it makes it more difficult to label subsystems later on.

Finally, we learned that extracting a concrete architecture is extremely difficult, much more so than analyzing a conceptual architecture. The documentation that exists to aid our extraction ranges in quality, and many resources conflict. One could devote multiple months to this very activity and keep unearthing new information by the day.

Conclusion

In this assignment we have investigated the concrete architecture of Hbase, compared it to the conceptual architecture researched in assignment 1, and discussed the significance of the architecture discovered. Using the derivation process detailed in this report we were also able to look more closely at the RegionServer subsystem. The effects of RegionServer on concurrency and Hbase in general was examined. Hbase has proved to be a complex and intricate program, with dependencies and relationships discovered in the concrete analysis that were not apparent in the conceptual analysis.

Appendix A: Naming Conventions

Abbreviation	Explanation
WAL	W rite A head L og. A subsystem of RegionServer that logs transactions before they are entered into the database. Ensures the system can recover from crashes without data loss.
RAM	R andom A ccess M emory. Volatile memory used in a computer to store information currently being used. Faster to read and write than long term storage, but much more expensive.
LSEdit	A program used to visualise the layout of a program by dividing it into subsystems.
LRU	L east R ecently U sed. An algorithm for deciding what element of a group should be removed to make room for a new element. As the name suggests, the least recently accessed element is the one removed.
DDL	D ata D efinition L anguage. A language used by a database management like HBase.
MVC	M odel- V iew- C ontroller. Architectural pattern commonly used in developing client-facing applications.
REST	R epresentational S tate T ransfer. A software architecture style that defines a set of constraints when creating web services

Appendix B: Bibliography

[1] Chang, et al. (2006). Bigtable: A Distributed Storage System for Structured Data

[2] "Apache Hbase Commands 7.53% Market Share in Big Data." *IDatalabs - Actionable Marketing Intelligence*, 2017, idatalabs.com/tech/products/apache-hbase.

[3] "HBase Conceptual Architecture." *Amazon*, Amazon, s3.amazonaws.com/files.dezyre.com/images/blog/Overview+of+HBase+Architecture+and+its+Components/HBase+Architecture.jpg.

[4] "HMaster vs Zookeeper - HBase." *Stack Overflow*, stackoverflow.com/questions/31216606/hmaster-vs-zookeeper-hbase.

[5] "The Apache Software Foundation Blogging in Action." *HBase - Who Needs a Master? : Apache HBase*, blogs.apache.org/hbase/entry/hbase_who_needs_a_master.

[6] Chanan, Gregory. "The Apache Software Foundation Blogging in Action." *Apache HBase Internals: Locking and Multiversion Concurrency Control : Apache HBase*, 11 Jan. 2013, blogs.apache.org/hbase/entry/apache_hbase_internals_locking_and.

[7] Apache. "Apache/Hbase." *GitHub*, 10 May 2018, github.com/apache/hbase/tree/master/hbase-archetypes.

[8] "Maven Repository: Org.apache.hbase » Hbase-Annotations." *MavenRepository*, mvnrepository.com/artifact/org.apache.hbase/hbase-annotations.

[9] "Maven Repository: Org.apache.hbase » Hbase-Testing-Util." *MavenRepository*, mvnrepository.com/artifact/org.apache.hbase/hbase-testing-util.

[10] "Java Annotation." *Wikipedia*, Wikimedia Foundation, 11 June 2018, en.wikipedia.org/wiki/Java_annotation.

[11] "Apache Thrift - Home." *Apache Thrift - Home*, thrift.apache.org/

[12] "What Is REST?" *Codecademy*, www.codecademy.com/articles/what-is-rest.

[13] "Maven Repository: Org.apache.hbase » Hbase-Hadoop-Compat." *MavenRepository*, mvnrepository.com/artifact/org.apache.hbase/hbase-hadoop-compat.

[14] "Maven Repository: Org.apache.hbase » Hbase-hadoop2-Compat." *MavenRepository*, mvnrepository.com/artifact/org.apache.hbase/hbase-hadoop2-compat.

[15] "What Are HBase Znodes?" *Cloudera Engineering Blog*, 17 Jan. 2014, blog.cloudera.com/blog/2013/10/what-are-hbase-znodes/.

[16] tutorialspoint.com. "HBase Client API." *Www.tutorialspoint.com*, www.tutorialspoint.com/hbase/hbase_client_api.htm.

[17] Apache. "Apache/Hbase." *GitHub*, github.com/apache/hbase/blob/master/hbase-it/src/test/java/org/apache/hadoop/hbase/IntegrationTestIngestWithEncryption.java.

[18] Apache. "Apache/Hbase." *GitHub*, github.com/apache/hbase/blob/master/hbase-zookeeper/src/main/java/org/apache/hadoop/hbase/zookeeper/ZKSplitLog.java.

[19] Apache. "Apache/Hbase." *GitHub*, github.com/apache/hbase/blob/master/hbase-server/src/test/java/org/apache/hadoop/hbase/master/TestCatalogJanitor.java.

[20] Apache. "Apache/Hbase." *GitHub*, github.com/apache/hbase/blob/master/hbase-examples/src/test/java/org/apache/hadoop/hbase/coprocessor/example/TestWriteHeavyIncrementObserverWithMemStoreCompaction.java.

[21] Chanan, Gregory. "The Apache Software Foundation Blogging in Action." *Apache HBase Internals: Locking and Multiversion Concurrency Control : Apache HBase*, 11 Jan. 2013, blogs.apache.org/hbase/entry/apache_hbase_internals_locking_and.