UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Decentralising Big Data Processing

by

Scott Ross Brisbane

Thesis submitted as a requirement for the degree of

Bachelor of Engineering (Software)

Submitted:	October 2016	Student ID:	z3459393
Supervisor:	Dr. Xiwei Xu	Topic ID:	3692

Abstract

Big data processing and analysis is becoming an increasingly important part of modern society as corporations and government organisations seek to draw insight from the vast amount of data they are storing. The traditional approach to such data processing is to use the popular Hadoop framework which uses HDFS (Hadoop Distributed File System) to store and stream data to analytics applications written in the MapReduce model. As organisations seek to share data and results with third parties, HDFS remains inadequate for such tasks in many ways. This work looks at replacing HDFS with a decentralised data store that is better suited to sharing data between organisations. The best fit for such a replacement is chosen to be the decentralised hypermedia distribution protocol IPFS (Interplanetary File System), that is built around the aim of connecting all peers in it's network with the same set of content addressed files.

Abbreviations

API Application Programming Interface

AWS Amazon Web Services

CLI Command Line Interface

DHT Distributed Hash Table

DNS Domain Name System

EC2 Elastic Compute Cloud

FTP File Transfer Protocol

HDFS Hadoop Distributed File System

HPC High-Performance Computing

IPFS InterPlanetary File System

IPNS InterPlanetary Naming System

SFTP Secure File Transfer Protocol

UI User Interface

Contents

Chapter 1

Introduction

As society creates an ever increasing volume of data, both large corporations and government organisations are working to conduct research and analysis on huge quantities of data in order to gain insight and understanding into complex topics. Traditional approaches towards storing and processing large data sets, such as the use of relational databases, have been unable to keep up with the sheer scale and growth in today's corporate and government data collection climate [11]. This has driven the need for, and development of, new technologies such as Google's MapReduce [12], which breaks data processing tasks into seperate Map and Reduce phases that can be executed in parallel across a distributed cluster of machines.

The most popular implementation of MapReduce is the Apache Software Foundation's open source Hadoop library, which is based very closely on the work from Google. Hadoop consists of three core components [8]:

1. MapReduce - a system providing parallel data processing capabilities
2. YARN - a framework for allocating jobs and managing resources
3. HDFS - a distributed and scalable filesystem

The Hadoop Distributed Filesystem, and more largely Hadoop as a whole, is designed to

run on a fully connected and controlled cluster of servers within an organisation. This doesn't allow for easy or seamless sharing of data or result sets between organisations. In this thesis, we aim to address the following concerns around the use of the Hadoop Distributed File System with shared data:

1. *Inefficient Data Ingestion Process*: Data sets that are stored on an alternate filesystem, or otherwise downloaded from some external source, need to be ingested into HDFS before processing. This retrieval and ingestion process adds considerable delay before any analysis can be performed.

2. *Complicated Mechanism Needed To Move Computation*: If we aim to eliminate the data ingestion phase with a shared data set, but still utilise HDFS, a mechanism would be needed to move computation between organisations. Such a mechanism would be complicated to implement and run.

3. *Highly Centralised Data Set Hosting*: Data sets that are made available for use by third parties are typically hosted and controlled by one entity. This poses concerns around the longevity of the data set and usually means that only one version is available. If the entity decides they no longer want to host the data, it will no longer be available for access.

4. *No Versioning Guarantees*: Data sets are typically only available in one version, and often there is no easy way to be certain that the version being accessed is the one that is expected.

These concerns will be addressed by implementing an interface between Hadoop and an existing, decentralised, peer-to-peer storage system. Such an interface will allow MapReduce tasks to be executed directly on data stored in this alternate storage system.

Chapter 2

Background

In this chapter some of the features of the Hadoop Distributed File System are discussed as well as some of the problems it solves, and those it doesn't. Some of the existing work to replace HDFS with alternate file systems, and the motivation of each of these projects, is also discussed. In addition, HTTPFS is mentioned, which is a REST HTTP gateway providing external access to HDFS clusters.

2.1 The Hadoop Distributed File System

The Hadoop Distributed File System [13] is a file system developed largely by Yahoo! that provides data storage for the Hadoop framework. It is a distributed and scalable file system designed to run on large clusters of servers running commodity hardware. One of the key concepts behind HDFS, and more generally Hadoop, is data locality, in which computation is executed as close as possible to the data. HDFS connects the local storage of the compute nodes in a cluster into a single distributed file system.

There were a number of key design decisions behind HDFS that provide certain benefits, but are also the source of some of its short-comings. Some of these decisions include:

1. *File metadata and contents stored separately*: HDFS stores file metadata separately to the actual data itself. This is implemented using the concept of a NameNode and seperate DataNode's. This has the advantage of simplifying the system greatly. The downside to this architecture, however, is that until recently with Hadoop 2.0, there was only a single NameNode. This NameNode became a single point of failure and a significant bottleneck in larger clusters as all requests must first contact the NameNode to determine the location of data blocks.

2. *Naive replication*: HDFS's approach to data replication is to assume that the nodes themselves have no redundancy, and that data replication must be implemented by the file system. The method via which HDFS achieves this is in naive replication of all data blocks across a number of nodes (typically 3). This has benefits in simplifying the compute nodes and allowing data to be distributed in such a way that it is more likely a copy of some data is nearby a compute node wanting to access it. The disadvantage is, however, that storing 3 complete copies of a data set is quite expensive from a storage footprint perspective.

3. *Performance over POSIX compliance*: The interface for HDFS has many similarities to traditional Unix file system interfaces, but is not POSIX compliant. This was deemed to be a worthy tradeoff as not supporting POSIX semantics allowed the development team behind HDFS to increase performance of the file system. This decision has meant that HDFS is well suited to high-performance Hadoop MapReduce applications, as well as other applications in the Hadoop ecosystem, but that alternate applications reliant on POSIX semantics are not supported.

2.2 Existing Work to Replace HDFS

Whilst there have been other attempts at replacing HDFS with an alternate filesystem, the motives behind doing so have been largely different to that of this work, and have mostly been a result of some of the design decisions behind HDFS. Some of the aims of the other projects have been to:

1. Improve scalability - HDFS' single NameNode becomes a bottleneck in larger clusters.

2. Reduce the overall storage footprint - the naive replication strategy used by HDFS results in significant storage footprint overhead.

3. Remove the need for multiple storage systems within an organisation - this need arises from the lack of POSIX compliance provided by HDFS.

The main aim of these projects has been to provide an alternate file system for Hadoop that is relatively similar to HDFS, but improves on it in some way. Some of the projects are now commercially available, whilst others were just intended as a proof of concept.

It should be noted that the work of this thesis does not aim to improve on any of these previous projects and that the motivation here for replacing HDFS is quite different.

2.2.1 Hadoop on LustreFS

Both Intel [14] and Seagate [15] have worked separately to build their own interface between Hadoop MapReduce and the Lustre File System. Lustre is similar in many ways to HDFS in that it is a distributed and scalable file system deployed to a known and controlled cluster of commodity based hardware [4]. It provides a number of advantages over HDFS; most notably, it is more scalable and performs better under certain conditions.

The team at Intel were largely motivated in their work by the need to improve scalability of high-performance computing and to provide tools that can keep up with the ever-increasing size of todays data stores. They felt Lustre was an obvious choice for integration with Hadoop as it is already the file system of choice in a variety of high-performance computing (HPC) applications. In many organisations, this could eliminate the need for two seperate file-systems (HDFS and Lustre for example), and thus remove the overhead of data transfer between the filesystems. This was another motivating factor behind their work.

Seagate's project had much of the same motivation as Intel's project, however they were also interested in reducing the overall storage footprint, and hence cost, of a Hadoop cluster.

2.2.2 Hadoop on GFarm

A small team of Japanese researchers have developed a plugin to allow Hadoop MapReduce jobs to run on top of the GFarm file system. The aim of their work was to provide an alternate file system to HDFS that had comparable performance, but also supported POSIX semantics to allow other applications to run on top of the file system [17]. Results from their own testing show they were successful in achieving their aims, with GFarm performing only slightly worse than HDFS. The group implemented their plugin by implementing Hadoop's FileSystem interface for the GFarm file system.

GFarm is similar to HDFS in many regards, in that it is a scalable, distributed file system with extensive support for data-locality, such that computation is moved to a node containing a replica of the relevant data [16]. GFarm differs from HDFS however in that it is more of a general purpose filesystem with POSIX support allowing it to be used with a wide array of applications.

2.3 HTTPFS

HTTPFS [6] is a REST HTTP gateway that allows external entities to access a HDFS cluster externally. It supports the full set of file system operations and can be used to transfer data between clusters or access data from an external cluster. Whilst this gateway does allow effective sharing of data sets with third parties, it doesn't address the concerns around data centralisation or versioning.

Chapter 3

Analysing Different Options

In this section, some of the options for alternate file systems are presented. In order to best solve the issues raised, a decentralised file system makes the best fit, and so all the options presented here are decentralised file systems. Each of the alternatives solve the concerns raised around data ingestion, movement of computation and centralisation of data, so these aren't discussed here. Instead, the following properties of each are considered:

1. Mechanism for sharing data with peers
2. Data replication and distribution approach
3. Presence of an API on which the FileSystem interface can be built
4. Support for large files
5. Data versioning capabilities
6. Access control mechanisms

3.1 IPFS: Interplanetary File System

The Interplanetary File System (IPFS) is a highly ambitious project that aims to connect all peers with the same set of files [1]. One of the core idea's underpinning the system is the concept of content addressed files, which makes it ideal in addressing the data versioning concerns.

1. *Mechanism for sharing data with peers*: Any peer can access any file stored in the network using it's content hash address. Files can be accessed via an API or through a CLI.

2. *Data replication and distribution approach*: Data is replicated throughout the network based on interest. As peers access content, it is kept in a local cache for a short time and can also be actively 'pinned' to ensure that a local copy is persisted.

3. *Presence of an API on which the FileSystem interface can be built*: IPFS has an extensive API that allows file system operations to be invoked from a variety of languages. Most notably, there is a Java API which is ideal for our use case.

4. *Support for large files*: Whilst not tested or documented extensively, there are known cases where IPFS has been used with files in the 10's to low 100's of Gigabyte's, without suffering significant performance degradation.

5. *Data versioning capabilities*: One of the fundamental design concepts underpinning IPFS is it's content addressing, by which files are accessed via a cryptographic hash link of their contents. This is an ideal solution to the data-versioning concerns because as soon as one byte in a file is changed, the hash will also change, which provides certainty around the content being accessed. In addition, the IPFS development community aim to add Git style commit objects that would allow multiple versions of a file to linked. The IPFS paper [1] makes reference to such objects, but they have not yet been fully implemented.

6. *Access control mechanisms*: Whilst IPFS doesn't currently support access control mechanisms, there are plans to add such features in the near future. It is possible, however, to run an IPFS network behind a firewall, which would allow a private network to be created. Currently all peers within an IPFS network have read/write access to all files. Since files are immutable, however, appending data to an existing file creates a new file with a new content address. In this way, files that have already been created are not able to be modified.

3.2 Tahoe-LAFS: Tahoe Least Authority File System

Tahoe Least Authority File System is a long-lived project to develop a decentralised file system with security as the major focus [3].

1. *Mechanism for sharing data with peers*: Tahoe uses a concept of 'caps' (or capabilities) to share files with other users of the system. A 'cap' is a cryptographic identifier that identifies a file and dictates what type of access (read, write etc.) to the file is allowed with the use of this 'cap'. A peer who has a 'cap' for a file can access the file using Tahoe's REST API, CLI or the FTP/SFTP protocols.

2. *Data replication and distribution approach*: Data is split into erasure coded and encrypted 'shares' that are distributed amongst peers in the network. The approach taken with the 'shares' is that only n of k 'shares' are needed to recreate a file, and so some of these 'shares' can be lost without the underlying file being lost.

3. *Presence of an API on which the FileSystem interface can be built*: Tahoe currently supports a RESTful web API and a CLI which would both serve our purposes.

4. *Support for large files*: Whilst not tested or documented thoroughly, there are recorded cases of people using Tahoe with files in the order of 10's of Gigabytes.

5. *Data versioning capabilities*: Files are accessed via 'caps', which are a cryptographic hash derived from the file contents and a 'convergence key', which is by default unique to each user. This provides a user the ability to confirm that the downloaded data is exactly the data they were expecting.

6. *Access control mechanisms*: A user must have the specific 'cap' for a file before it can be accessed. Further to this, a 'cap' also dictates whether the file access is read only or read-write. The system uses RSA key pairs to ensure security of mutable files.

3.3 Sia

Sia is a relatively young project with a more commercial focus. It is aimed at becoming a decentralised cloud alternative (to compete with the likes of Dropbox etc.). It is mainly targeted at enterprise and highlights it's key features as being distributed, customisable, extensible and secure [9] [2].

1. *Mechanism for sharing data with peers*: File sharing in Sia centres around siafile's, which contain data access and location information for retrieving a file. Without the siafile corresponding to a file, no user has the ability to retrieve it. Files are retrieved over a Web API using the information contained in the siafile.

2. *Data replication and distribution approach*: Files are split into erasure coded and encrypted 'shares', which are distributed to peers who are paid in cryptocurrency for their efforts. In this way, data-replication is guaranteed through financial incentive.

3. *Presence of an API on which the FileSystem interface can be built*: Sia currently only supports a RESTful web API.

4. *Support for large files*: There is currently no documented case of Sia being used with any reasonable sized data.

5. *Data versioning capabilities*: Sia currently has no built-in support for data versioning.

6. *Access control mechanisms*: Sia provides access control through its siafiles, which contain the necessary data for retrieving and decrypting a file. Without a siafile, no user is able to access and decrypt the corresponding file.

3.4 Storj

Storj is similar to Sia in that it is also a more commercialised alternative aiming to become an open source and decentralised alternative to the likes of Dropbox, Google Drive etc. It is aimed more at personal users [10].

1. *Mechanism for sharing data with peers*: Storj is not expressly designed to allow sharing of data with peers. It is more focussed on personal storage. It is possible to share data with peers using the web API, however this is not a particularly easy task.

2. *Data replication and distribution approach*: Files are split into erasure coded and encrypted 'shares', which are distributed to peers who are paid in cryptocurrency for their efforts. In this way, data-replication is guaranteed through financial incentive.

3. *Presence of an API on which the FileSystem interface can be built*: Storj provides a RESTful web API for interacting with the network programatically.

4. *Support for large files*: There is currently no documented case of Storj being used with any reasonable sized data.

5. *Data versioning capabilities*: In order to access files using the REST API, the content hash of the desired file is needed. This provides a user the ability to ensure that the data retrieved is the data they expect.

6. *Access control mechanisms*: As Storj is not expressly designed for sharing of data with third parties, access control is not something well defined in the system. In order to retrieve a file via the REST API however, a great deal of information on the file is needed, including it's content hash, address, the ID of the host and a cryptographic signature.

Chapter 4

Chosen Solution

In this chapter, it is determined which File System is the most suitable to replace HDFS for our use case and some of the differences between HDFS and the chosen FileSystem that will be important and relevant to this work are discussed.

4.1 Ideal Choice: The InterPlanetary File System

The InterPlanetary File System (IPFS) has been chosen as the most suitable for this project as it solves the four main issues raised in the following ways:

1. *Inefficient Data Ingestion Process*: With the implementation of a layer between Hadoop MapReduce and IPFS, it will be possible for organisations to store data sets, share this data with third parties, perform analysis on the data and output results all directly on top of IPFS. Any third party interested in this data will also be able to perform analysis on the data directly from IPFS, and output their results straight into IPFS, to potentially be shared with others. This eliminates the need for any data ingestion phase and allows organisations to also share their output with ease.

2. *Complicated Mechanism Needed To Move Computation*: If data is stored directly in IPFS, there is no longer a need to move computation between organisations. Instead, data is moved between them during execution of the job.

3. *Highly Centralised Data Set Hosting*: With data set's stored in IPFS, any party connected to the same IPFS network will be able to retrieve and store this data for themselves by actively 'pinning' the content. As they do so, the data is being replicated throughout the network and becoming more and decentralised as more copies are made. This form of replication is based purely on interest and means that as long as there is one party still interested in 'pinning' the data, it will be persisted. In this way multiple versions will also be available, as long as some party in the network still has interest in the relevant version.

4. *No Versioning Guarantees*: One of the key ideas behind IPFS is that everything is content addressed. This is a perfect solution to the versioning concerns as a user of the system must specify the cryptographic hash of the data they wish to access. This guarantees they will receive the exact version of the data that they are expecting.

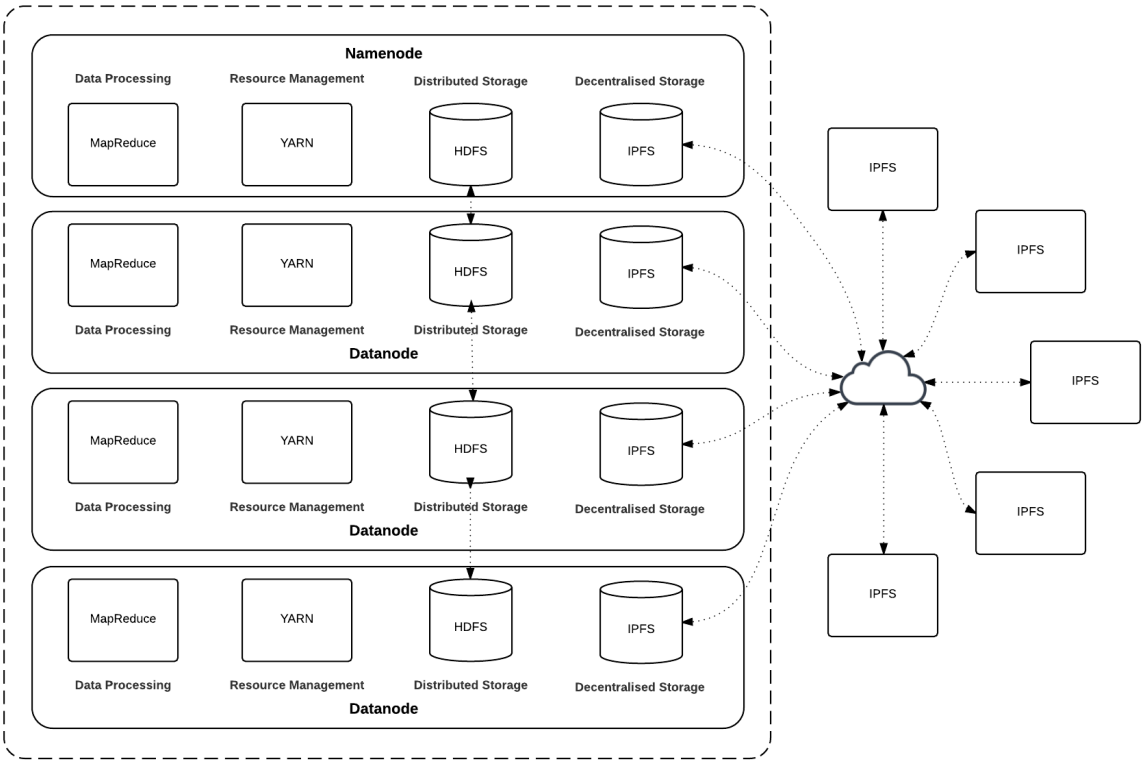
As well as solving the above issues, IPFS also made the most sense as it is a highly ambitious project gaining lots of attention and is in very active development. It's data replication strategy is the best suited to our application in that it doesn't require any party paying for the storage and doesn't make any assumptions about the trustworthiness or authenticity of peers hosting the data. It also has been documented to support large files relatively well, an important consideration for applications in big data.

Chapter 5

My Work

In this chapter, the work that has been completed is discussed, as well as how it was completed, some of the challenges that were encountered and how they were dealt with. Some suggestions for future improvements to the work are also discussed.

5.1 System Structure



5.1.1 High-Level Overview

As shown in the above diagram, the system has a similar structure to a conventional Hadoop cluster, except with the addition of an IPFS component on each node. The Namenode and Datanodes that form the Hadoop cluster can be either within one datacenter, or be geographically distributed. Each of the nodes has an IPFS daemon process running on it, which connects to a peer-to-peer IPFS network (either public or private) for file access. This enables the cluster to perform MapReduce tasks on any data stored within the IPFS network.

5.1.2 Key Components

In order to develop my solution, there were a number of pieces that needed to be implemented. These various components and how they fit together are as follows:

1. *IPFSFileSystem Interface*: This is the main body of work, and implements the interface between Hadoop and IPFS. It is compiled as a jar file and placed on each node in the Hadoop cluster.

2. *IPFSOutputStream*: This component is an output stream used in file creation on IPFS. It is used directly by the *IPFSFileSystem* interface, and is compiled as part of it's jar file.

3. *Custom Output Committer*: This is a seperate component that is used by the MapReduce job itself (in our testing a multiple file word count). It enables us to only use IPFS for the final output of the job, and to use HDFS for intermediate files (the reasons for this behaviour are outlined in section 5.2.2). It is compiled as part of the MapReduce application's jar file.

5.1.3 Anatomy of a MapReduce Job

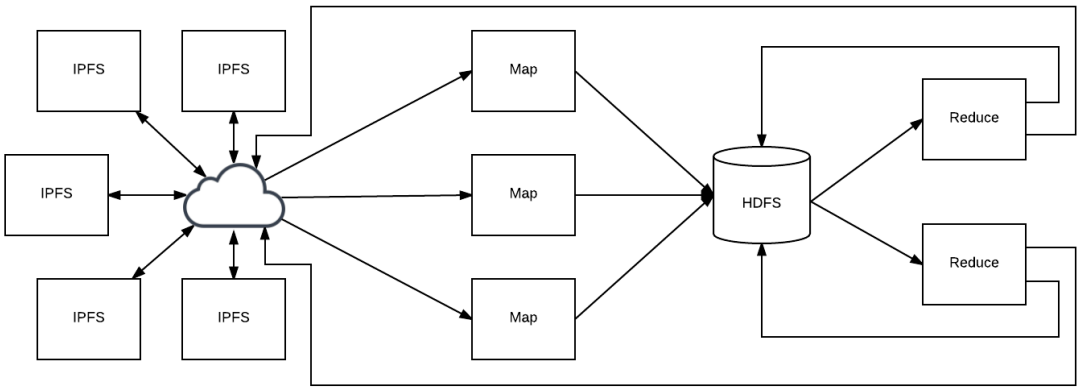
To give a detailed understanding of how the system operates, a typical MapReduce job run on top of the IPFS system will behave as follows:

1. *Data fetched from IPFS*: Metadata for the input data set is retrieved from IPFS and important parameters determined (such as the size of each file, the number of files, etc). From here the data will be split by the MapReduce framework (as in any MapReduce job) and an appropriate number of containers scheduled.

2. *Map Phase*: Each of the conatiners forming the the Map phase will then operate on their own subset of the input data set fetched from IPFS and perform their computation. Each node in the cluster has it's own IPFS API allowing it to connect directly with the IPFS network. Any temporary output files created by the Map phase are stored in HDFS, rather than IPFS, due to performance and concurrency concerns (outlined in section 5.2.2).

3. *Reduce Phase*: The containers forming the reduce phase will then fetch the intermediary results created by the Map phase from HDFS and perform their computation. The output from this phase is committed to both HDFS and IPFS, based on the design of our output committer. This, however, could be changed and output only committed to IPFS. Due to the semantics of mutability on IPFS (see section 5.2.1), the output directory is created via a single node (typically the NameNode).

The above workflow is further illustrated with the following diagram showing the various phases of the job and the input and output locations of each.



In addition to the above workflow, depending on the use case, a MapReduce programmer may decide only to fetch the input data from IPFS and to output to HDFS as normal. They may also decide to do the reverse and fetch data from HDFS, with the output stored in IPFS. A new type of custom output committer could also be developed that stores the final output only on IPFS rather than storing in both HDFS and IPFS.

5.2 Challenges

5.2.1 Mutability and Working Directories

Possibly the biggest challenge for this work was determining how to implement mutability, and even basic file creation, on top of a file system that is inherently immutable and whose file path's are semantically very different to those used in the FileSystem interface for Hadoop. Paths used by Hadoop for all file and directory operations are defined before any content for the files or directories is given. In the case of IPFS, where files and directories are content addressed, these semantics simply don't work. Object paths in IPFS take the form *hash/foo/bar*, where *hash* is a cryptographic hash of the objects contents, and so an absolute Hadoop Path object cannot be simply passed in when performing operations on IPFSobjects, as this hash is not known until after the operation is complete.

To solve this issue, I used IPNS (InterPlanetary Naming System) to implement the concept of a working directory. IPNS can be thought of as a DNS system for IPFS, where every peer has a peer ID which is used as the key for their IPNS entry. This means that each IPFS peer has one IPNS record for themselves, which they can publish any IPFS object hash to.

For implementing working directories and mutability this system works quite well and allowed me to implement file operations on IPFS using simple string paths. When any modification is made to the working directory or any files within it, we first resolve the IPNS record to an object hash, then perform the relevant operations. These operations result in a new object being formed (with a new hash) which is then published to IPNS to replace the old object.

This system works well, but exposes two major issues which are discussed below.

5.2.2 Concurrent Modification of Files Within a Directory

Taking the above mechanism for mutability and IPFS' content addressing into account raises concerns about concurrent modification of any objects within a working directory. More specifically a race condition arises when any two or more nodes attempt to concurrently modify the working directory in any way. This is due to the important fact, and feature of IPFS, that when any element of an object or any of its children is modified, all parent objects of the modified object will have a new hash, up to and including the root object. When two or more nodes are concurrently modifying the working directory, a critical section exists between the time when each of them resolves the IPNS name and when they publish the newly modified hash back to IPNS.

In order to solve this issue, I first considered implementing a locking system to ensure that no two nodes are in the critical section at the same time. This option wasn't appropriate as the loss of performance would be too significant and the complexity of implementing a distributed locking system for Hadoop was outside the scope of this work.

Instead, the best option was to only use IPFS for input for the job and for the final committed output. This meant that any intermediate files created by Hadoop (such as the output of the Map phase) would still need to be stored in HDFS.

Whilst at first this solution may seem like a loss of functionality for the system, it is important to note that the solution still solves all of the issues it set out to solve. From the perspective of a user of the system, this solution is just as effective in solving the defined use case scenarios as a system that ran entirely on IPFS. What matters in our use case is the initial input and final output, not the location of the intermediate files.

5.2.3 IPNS Caching Functionality

During my work with IPNS, I encountered an issue where resolving an IPNS peer ID would return a stale cached result, even when the most recent publish for that peer ID was issued by the peer itself, only moments earlier. This appears to be an oversight by the IPFS team in that publishing a record to IPNS does not set (or even flush) the local cache for that record.

The solution to this issue was thankfully quite straightforward in that I was able to simply pass a TTL (time to live) parameter to the IPFS API. The downside to this however, is that all IPNS resolve requests now need to query the IPFS DHT, which can be quite a noticeable hit to performance.

5.3 The Hadoop FileSystem Interface

In order to allow interoperability between Hadoop and various different file systems, Apache have provided a generic FileSystem interface that all Hadoop MapReduce application programmers are encouraged to use, rather than interacting with HDFS (or its alternatives) more directly. It is this interface that has allowed past projects to replace HDFS with alternate FileSystems, without needing to modify existing MapReduce Applications. The most crucial part of this work was to implement this interface for the InterPlanetary File System (IPFS).

The main functions in this interface, and those that I implemented, are as follows:

```
public FSDataOutputStream append(Path f, int bufferSize, Progressable
    progress);

public void concat(Path trg, Path[] psrcs) throws IOException;

public FSDataOutputStream create(Path f, FsPermission permission, boolean
    overwrite, int bufferSize, short replication, long blockSize,
    Progressable progress) throws IOException;

public boolean delete(Path f, boolean recursive) throws IOException;

public boolean exists(Path f);

public int getDefaultPort();

public FileStatus getFileStatus(Path f) throws IOException;

public String getScheme();

public URI getUri();

public Path getWorkingDirectory();

public void initialize(URI name, Configuration conf) throws IOException;

public boolean isDirectory(Path f) throws IOException;

public boolean isFile(Path f) throws IOException;

public FileStatus[] listStatus(Path f) throws IOException;

public boolean mkdirs(Path f, FsPermission permission);

public FSDataInputStream open(Path f, int bufferSize) throws IOException;
```

```
public boolean rename(Path src, Path dst) throws IOException;

public void setWorkingDirectory(Path new_dir);
```

Some of the more critical and interesting functions will be discussed in more depth below.

5.3.1 create

The *create* function is used to create files at a given Path, returning a stream to which file contents are sent. My implementation of the function first performs a check on the type of file Path it is given. This Path must either be a relative string Path (of the form *foo/bar*) or an IPNS string Path (of the form *hash/foo/bar*). This is due to the way mutability is handled by our implementation.

If the given Path is of the right form, a check is performed to ensure the root hash (either that given in the IPNS string Path or the hash that corresponds to the current working directory) is mutable. If so, the *mkdirs* function is called, as needed, to construct the Path of the file to be created and finally an *FSDataOutputStream* is created from a new *IPFSOutputStream* object, and is returned. The *IPFSOutputStream* is covered in more depth in the next section, but has the basic functionality of accepting writes to the new File and saving the file contents to IPFS once the stream is closed or flushed.

5.3.2 delete

The *delete* function is as advertised, deleting files or directories from the file system. My implementation enforces the same requirements as the *create* function on the Path type it is given. This is again due to the way mutability is handled.

Once it has been determined that the given Path is of the correct format, and that it is mutable, the function checks whether the Path refers to a directory. If so, as long as the recursive flag is true, *delete* is called recursively on all the children of the current Path.

When any recursion has hit the base case, the parent of the given Path is modified to remove the link to the child (which is being deleted). This creates a new hash for the parent, and so the parent of the parent must be modified so that it's link to the parent points to the new parent (with removed link) and not the old parent. This process of updating links continues up the path to the root object until a new object is created for the root. This new object's hash is then published to IPNS such that subsequent operations on the working directory will be operating on the now modified working directory.

5.3.3 exists

The *exists* function determines whether a given file or directory exists. Our implementation simply makes a call to IPFS to retrieve the status of the object at the given Path. If no status is returned, it is assumed that the file doesn't exist. This is quite different to the way most file system's would implement such functionality, and has the downside of being very slow, particularly when the file doesn't exist as a full query of the IPFS DHT is necessary.

5.3.4 getFileStatus

The *getFileStatus* function retrieves certain information and parameters about files in IPFS. It is not able to provide the full information that HDFS provides on its files, as much of this information is not supported by IPFS (such as file mode, symlinks and data locality information). What is provided is the size and block size of the file (as provided by *ipfs object stat*) and whether the Path refers to a file or directory.

5.3.5 isDirectory/isFile

The *isDirectory* and *isFile* functions determine whether a given Path refers to a file or directory. These were not as straight forward to implement with the current functionality of IPFS as one might expect. IPFS doesn't currently provide a simple isDirectory flag in any of its object data methods. Simply inspecting the number of outgoing links from an object is also not suitable for this purpose as large files are chunked and will have outgoing links to the other chunks in the file. Instead, we rely on the fact that the data object in every IPFS directory is 2 bytes in size. Whilst this is not entirely accurate, it is assumed that no file will be 2 bytes long. There is currently no better method in IPFS to determine whether an object represents a file or a directory.

5.3.6 mkdirs

The *mkdirs* function creates one or more directories in a Path, as needed. Our implementation first checks that the given path is of a correct type and is mutable. It then resolves the root hash of the Path, an IPNS name, into its object hash. From here a stack is created of the components of the Path, and is iterated over. For each Path component, we determine whether or not a link exists of the same name from the current child element (beginning with the resolved root object). This information is then used to separate the components into two new stacks, one for components of the Path that already exist and one for those that are being created.

We then create a new object (initially an empty directory) and add a series of links to it using the stack of Path components that do not currently exist. To illustrate, if we had the Path *hash/foo/bar/baz/abc* and in the current working directory the Path *hash/foo* already exists, we would create an object to represent *abc* (an empty directory). We would then create another object, representing *baz*, and create a link from it to our already created object for *abc*. We would then create yet another object, this time representing *bar*, and create a link from it to the object representing *baz*. The result of this is an object representing *bar* with a chain of links to *baz* and *abc*.

Following this, the child object of the components of the Path that already exist is updated with a link to the object we created for the Path components that didn't already exist. In the example above, that would mean that the *foo* object would be patched with a link to the *bar* object. This change creates a new hash for the child object, and so we must update the existing links to point to the updated objects, back up the path to the root object. Once we have the new hash for the updated root object, this is published to IPNS.

5.3.7 open

The *open* function is just as expected, returning a stream containing the file contents. Whilst functions involving mutability proved more difficult to implement on top of IPFS, *open* remained straight forward. The function simply retrieves a byte array of the object contents using *ipfs cat* and creates an *FSDataInputStream*.

5.3.8 rename

The *rename* function renames files or directories. My implementation first checks that the given source and destination Path's are mutable, and whether or not they exist. Following this, it determines whether the destination Path is a directory, and if so calls *rename* recursively with a new destination Path given by concatenating the destination Path with the filename of the source Path. The object at the source Path is then retrieved and the *delete* function called on the source Path.

The Parent object of the destination Path is retrieved, and a link added using the filename from the destination Path and the hash of the original source object. This link addition creates a new parent object (with a new hash) and so the parent to the parent is updated to point to the new parent object. This process repeats up the chain to the root object, which is then published to IPNS.

5.4 IPFSOutputStream

The *IPFSOutputStream* is used in the *create* function to construct the *FSDataOutputStream* it must return. It provides a stream through which files can be created on IPFS via the *IPFSFileSystem* interface. It has the role of acting as a buffer between file data submitted by Hadoop and that published to IPFS.

A custom output stream of this nature is typically not required when implementing the *FileSystem* interface for a new file system, however it was required here because of the way in which mutability is implemented on IPFS. When the stream is closed or flushed, the contents of the buffer must be submitted to IPFS as a new file and the working directory updated accordingly (via publishing to IPNS).

The interface itself is quite small, providing only the following functionality:

```
public void close() throws IOException;

public void flush() throws IOException;

public void write(int b) throws IOException;

public void write(byte[] b) throws IOException;

public void write(byte[] b, int off, int len) throws IOException;
```

5.5 Custom Job Committer

The custom job committer is a seperate class not used by the *IPFSFileSystem*. It functions as a wrapper around the *FileOutputCommitter* provided by Hadoop, and commits the result of the job both to the default FileSystem (specified in the *core-site.xml*) and the FileSystem specified in the output path. Any intermediate files are stored in the default FileSystem rather than that specified in the output path. This

custom committer was necessary as the default committers used in Hadoop will use the file system specified in the output path both for intermediate files and the final commit, which isn't suitable in this case.

The class itself is quite small and only implements a constructor and override for the *commitJob* function. The effort for this component of my work was in reading through the Hadoop source code and determining what needed to be modified and how. It is also worth noting that most other projects to implement alternate file systems on Hadoop would not need a custom committer because they are usually suitable for both intermediate output and the final commit.

5.6 Deployment and Configuration

In order to deploy the solution to AWS, a standard Hadoop/HDFS cluster must first be setup, and then modified to run the IPFS FileSystem alongside HDFS.

5.6.1 Deployment to Amazon Web Services

Aside from the standard deployment of a Hadoop cluster on AWS, the following additional considerations needed to be made to ensure the smooth operation of Hadoop on IPFS:

1. *Instance Size*: Running Hadoop on top of the IPFS daemon process has proved to be fairly CPU intensive at times and so a standard T2 type instance was not sufficient, as instances were running out of CPU credits. A more appropriate instance type is any of the C4 family, that are optimised for computational capacity.

2. *Security Group*: IPFS uses port 4001 for its swarm (data transfer between DHT nodes) and 5001 for the API. Port 4001 for the swarm should be open to all traffic from all IP's within the IPFS network (whether that be a private IPFS network, or the public network). Port 5001 should only be open to the nodes in the Hadoop cluster to ensure that they can access the API on other nodes. If port 5001 is left open to the world, this is a serious security risk for the IPFS network as a third party could compromise the API on one of the nodes and perform operations on its behalf, without its knowledge.

5.6.2 Cluster Configuration

In order to setup a Hadoop cluster to use IPFS, a number of modifications are necessary. These changes are as follows:

1. *Install IPFS*: IPFS must be installed on all nodes (NameNode and DataNodes).

2. *Configure the IPFS API*: In order to allow API requests from other nodes, the IPFS config must be modified on each host. This config is usually located at *~/.ipfs/config* and should contain the following:

```
"Addresses": {
    "Swarm": [
      "/ip4/0.0.0.0/tcp/4001",
      "/ip6/::/tcp/4001"
    ],
    "API": "/ip4/0.0.0.0/tcp/5001",
    "Gateway": "/ip4/127.0.0.1/tcp/8080"
  },
```

3. *Ensure Java 8 is installed*: The *IPFSFileSystem* uses a Java API for IPFS which requires Java 8. All nodes should be running Java 8.

4. *Install the IPFSFileSystem jar file*: The *IPFSFileSystem* jar file needs to be placed on all nodes in the cluster, both Namenodes and Datanodes. It should be placed in a directory that is on the classpath for both the Hadoop master and the containers it spawns on the various nodes.

5. *Modify the core-site.xml*: The *core-site.xml* needs to be modified on all nodes to specify a class to use for files with an *ipfs://* scheme. This file should be modified to contain the following two entries:

```
<property>
  <name>fs.ipfs.impl</name>
  <value>com.sbrisbane.hadoop.fs.ipfs.IPFSFileSystem</value>
</property>
<property>
  <name>fs.AbstractFileSystem.ipfs.impl</name>
  <value>com.sbrisbane.hadoop.fs.ipfs.IPFSDelegate</value>
</property>
```

6. *Increase the hard limit for maximum number of open files*: The system-wide (but enforced per-process) hard limit for number of open files must be increased. The appropriate number for the new limit will depend on the application and data set being used, but the IPFS daemon process makes use of a large number of open file descriptors in many cases. The daemon process is able to manage its own soft limit, but cannot increase this past the global hard limit.

Chapter 6

Evaluation

In order to determine the success of the thesis work and analyse its merits, the project was evaluated based on its performance characteristics, its functionality for given predefined use cases and how adaptable it is for use with Apache Spark.

6.1 Performance

Whilst performance may appear to be an ill-fitted evaluation criteria for this thesis, mainly due to the apparent loss of data locality benefits, performance is still an important criteria. The performance of a sample MapReduce job, a multiple file word count on a large data set, will be compared on 4 different types of cluster:

1. *Local HDFS*: A standard Hadoop/HDFS cluster running within a single AWS Availability Zone to mimic the cluster being contained within one data centre.

2. *Local IPFS*: A Hadoop cluster running on top of IPFS with the interface developed as part of this work. The nodes will also be kept within the same AWS Availability Zone to mimic the cluster being contained within one data centre.

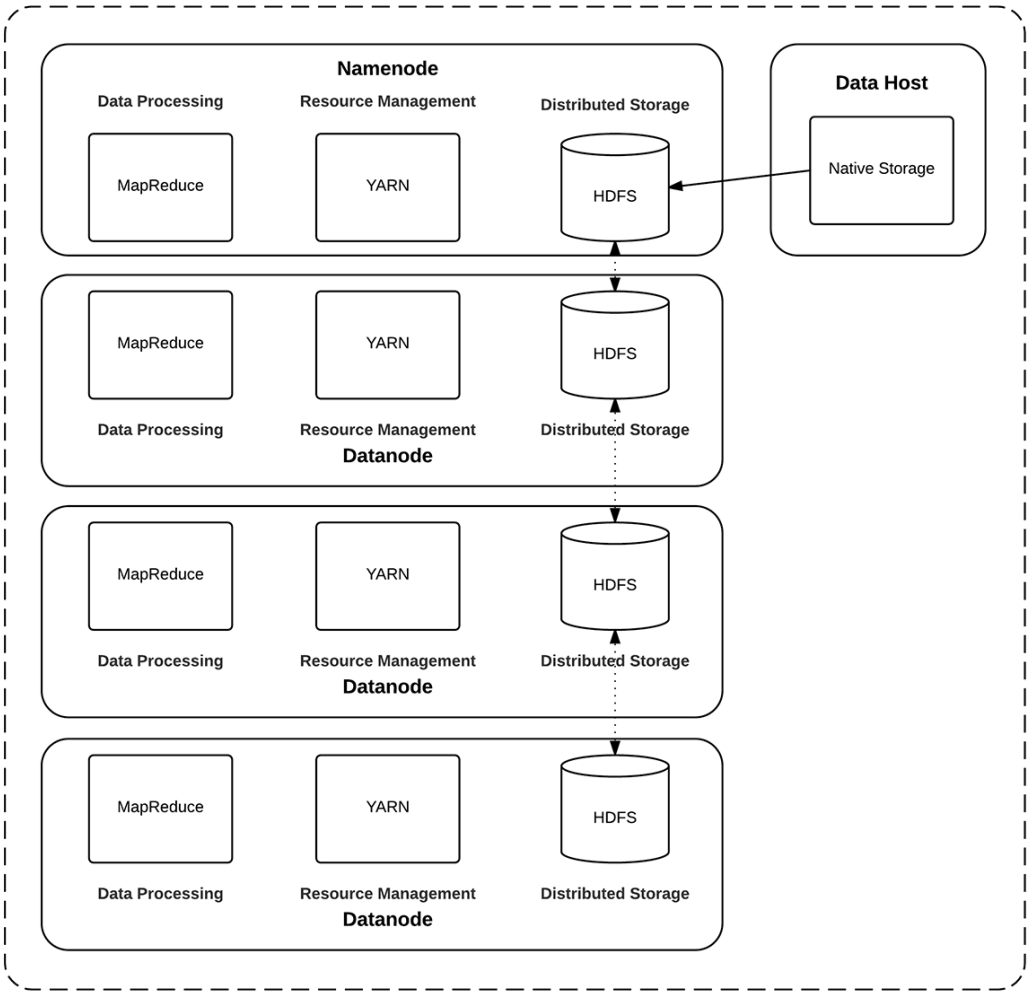
3. *Cross-Region HDFS*: A standard Hadoop/HDFS cluster running across multiple AWS regions to mimic a distributed HDFS based cluster running at the scale of the web.

4. *Cross-Region IPFS*: A Hadoop cluster running on top of IPFS with the interface developed as part of this work. The nodes will be separated between multiple AWS regions to mimic a distributed and decentralised cluster running at the scale of the web.

6.1.1 Local HDFS

As a baseline for comparing the performance of HDFS and IPFS on local clusters, a basic multiple file word count program was run on a 1.2GB data set of over 3000 text files stored on HDFS. The text files contained a corpus of various English language literature pieces obtained from Project Gutenburg. The program was run on a small cluster of 4 nodes (1 Namenode and 3 Datanodes), with each node being a c4.large AWS EC2 instance. All instances were located within the same region, us-west-2. On each iteration of the job, the data set was fetched from a seperate 5th instance, in the same region, using the scp command and then ingested into HDFS, to simulate the data set being owned by the party conducting the analysis.

The system structure for this experiment is further illustrated with the following diagram:



For the purposes of analysing the performance, two different timing measurements were used, as follows:

1. *Unix time command on copy, ingestion and execution*: A simple script was created to copy the data from the 5th instance, ingest it into HDFS, and then perform the map reduce job. This script's execution was timed using the unix time command. This measurement indicates the real-world time that it took to fetch the data set from nearby storage, ingest it into HDFS and then perform the map-reduce job.

2. *Hadoop web UI reported task time*: In addition to the full time for copy, ingestion and execution, the time taken by the MapReduce job itself (as reported by the Hadoop web UI) is also measured. This measurement indicates the time taken for the actual analysis.

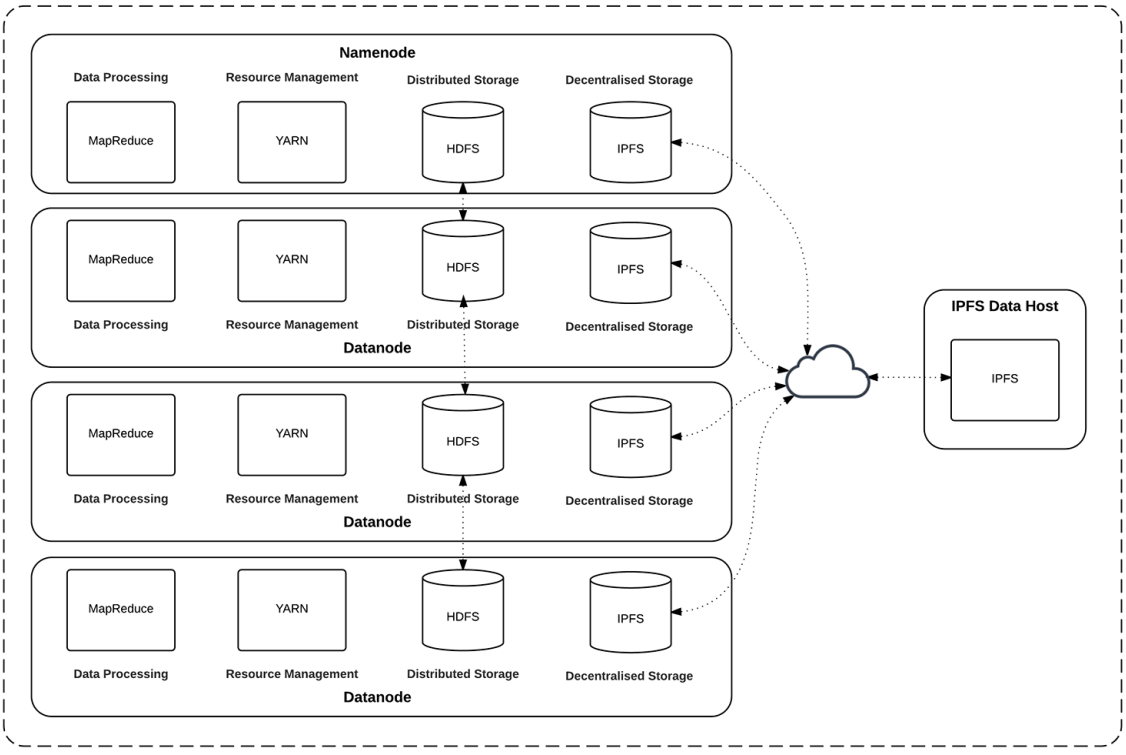
The test was repeated 20 times and an average taken. The data is given in the table below.

Attempt	Copy, Ingestion and Execution Time	MapReduce Task Time
1	262.60	160
2	256.31	162
3	251.39	161
4	253.82	159
5	249.55	159
6	260.14	162
7	261.83	161
8	252.06	162
9	256.93	162
10	259.32	162
11	257.61	161
12	254.49	160
13	253.50	159
14	250.18	161
15	253.41	163
16	255.82	164
17	253.03	160
18	257.26	164
19	265.22	164
20	254.61	161
AVERAGE	255.95	161.35

6.1.2 Local IPFS

In order to compare the performance of the IPFS based solution and HDFS on a local cluster, the same experiment was conducted as above, using the same data set, on the same EC2 cluster. The difference being that rather than using *scp* to transfer the data set to the NameNode and then ingesting it into HDFS, the 5th instance would host the data set over IPFS. The Hadoop job would instead access this data set over IPFS directly, using the solution developed as part of this work. The aim of this setup is to simulate a similar scenario to the above experiment, where the party conducting the analysis owns the data set, but has it hosted in IPFS (potentially to allow them to share this data with other parties).

The system structure for this experiment is further illustrated with the following diagram:



Similar to the above experiment, the following timing measurements were used:

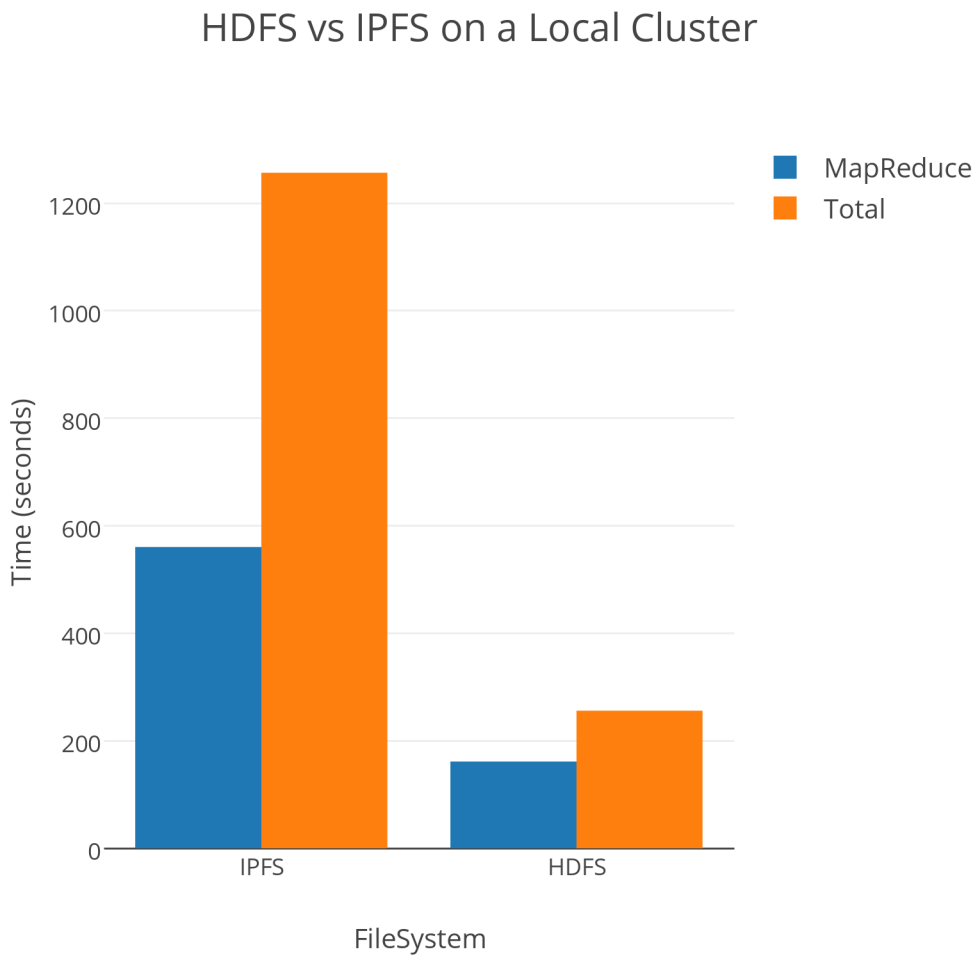
1. *Unix time command on the Hadoop job*: The unix time command was used to time the execution of the Hadoop job in real world terms. This time includes all aspects of the job including setup and the initial accessing of metadata on the data set, via IPFS, as well as the MapReduce task itself.

2. *Hadoop web UI reported task time*: In addition to the time for complete execution of the job, the time taken by the MapReduce task itself (as reported by the Hadoop web UI) is also measured. This measurement indicates the time taken for the actual analysis.

The test was again repeated 20 times and an average taken. The data is given in the table below.

Attempt	Total Time	MapReduce Task Time
1	1289.50	559
2	1097.22	404
3	1236.77	522
4	1245.91	541
5	1247.13	536
6	1425.63	712
7	1198.11	498
8	1147.35	444
9	1149.04	431
10	1337.72	634
11	1259.87	562
12	1318.65	627
13	1312.35	596
14	1216.71	506
15	1288.49	579
16	1326.26	643
17	1462.82	762
18	1148.27	453
19	1392.93	712
20	1212.10	490
AVERAGE	1256.64	560.55

6.1.3 Local Cluster Comparison



As the above graph shows, in a local cluster scenario, HDFS is significantly more performant than IPFS. This is an expected result due to the way in which both file systems operate. HDFS is a purpose-built file system designed specifically for local Hadoop clusters operating in a single data center. IPFS, in contrast, is a much more generalised file system designed for decentralised applications where performance is not a key concern.

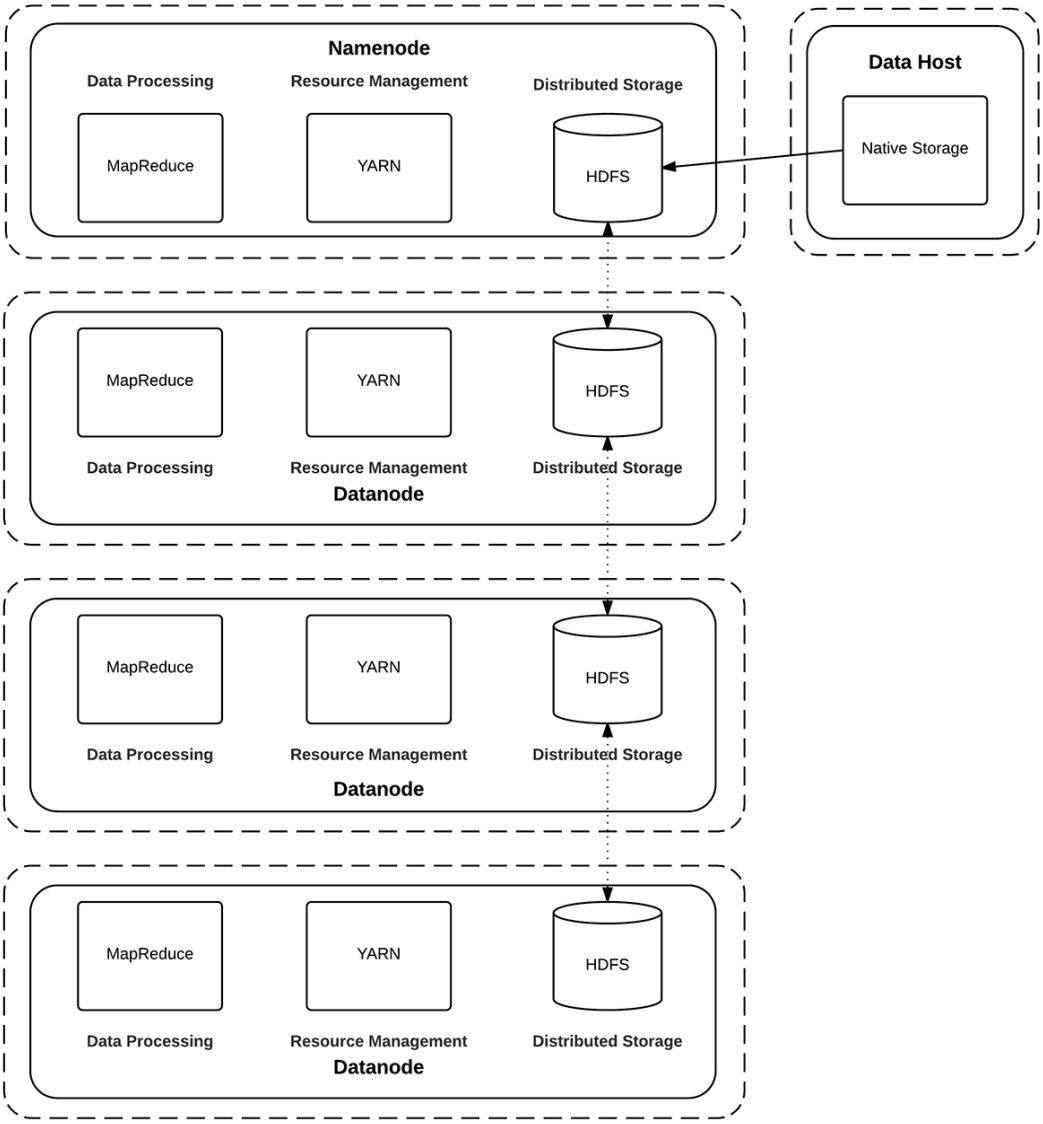
Further to this, HDFS and Hadoop, in combination, are able to take advantage of data locality information and schedule MapReduce computations to take place as close to the data as possible. IPFS provides no such information and in our case the data is on a seperate host meaning all the Hadoop nodes are streaming the entirety of their split of the data set over the network.

In addition, before Hadoop is able to determine how to split the input data, the NameNode must obtain file metadata for the data set to inform its decision. This metadata retrieval is trivial in the case of HDFS, as the metadata is all contained on the NameNode already. However, in the case of IPFS, it must retrieve metadata over the IPFS DHT. In our case, the node with the information is not located very far away (it is within the same AWS region), but this data transfer must occur for every file in the data set (in this case over 3000 files) which becomes a significant burden. One potential solution for this would be for IPFS to support getting file metadata for an entire directory more easily, such that it could occur in one API call.

6.1.4 Cross-Region HDFS

In order to better simulate the types of scenarios that this thesis is targeted towards, the above experiments were also conducted on a geographically distributed cluster to simulate decentralised analytics at the scale of the web. The cluster consisted of 1 Namenode and 3 Datanode's, all of which were c4.large Amazon EC2 instances. The Namenode was hosted out of the Oregon region, with one Datanode in each of the North Virginia, Ireland and Seoul regions. For sake of performing a fair comparison, the same multiple file word count program was used and the experiment conducted in the same way as the local HDFS experiment above. The data set was again hosted on a seperate 5th instance and copied over to the Namenode as part of the ingestion process. This 5th instance was located in the North California region to again simulate decentralisation and distribution of the data set.

The system structure for this experiment is further illustrated with the following diagram:



Whilst identical to those used in the local HDFS test above, the timing measurements were as follows:

1. *Unix time command on copy, ingestion and execution*: A simple script was created to copy the data from the 5th instance, ingest it into HDFS, and then perform the map reduce job. This script's execution was timed using the unix time command. This measurement indicates the real-world time that it took to fetch the data set from nearby storage, ingest it into HDFS and then perform the map-reduce job.

2. *Hadoop web UI reported task time*: In addition to the full time for copy, ingestion and execution, the time taken by the MapReduce job itself (as reported by the Hadoop web UI) is also measured. This measurement indicates the time taken for the actual analysis.

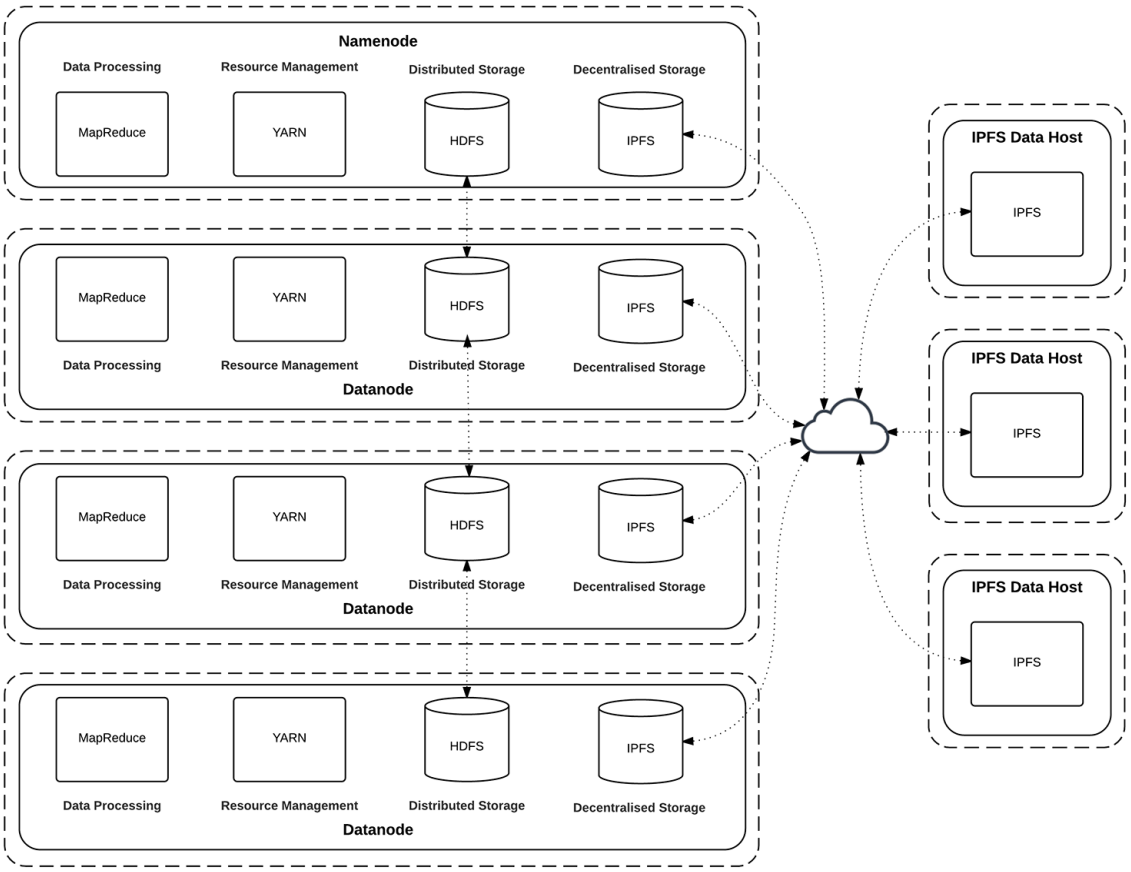
This test took significant time to complete and so was only repeated 5 times (largely due to AWS billing concerns with data transfer between regions). The raw data and an average is given in the table below.

Attempt	Copy, Ingestion and Execution Time	MapReduce Task Time
1	8075.57	327
2	7928.32	250
3	8176.31	311
4	8000.76	257
5	8134.79	305
AVERAGE	8063.15	290

6.1.5 Cross-Region IPFS

With a similar motive to above, in wanting to test my solution in a scenario that is closer to those the work was intended for, a cross-region IPFS test was conducted using the same data set, application and cluster as described above. The difference this time however being that the 5th instance hosting the data set was replaced by 3 instances hosting the data set over IPFS. These 3 instances are located in three different regions (that are also different to those hosting the Hadoop cluster) to simulate the data set being distributed across multiple hosts around the world. These instances were placed in the North California, Frankfurt and Tokyo regions. As above (in the local IPFS test), the Hadoop job will access the data set over IPFS directly, using the solution developed as part of this work. Which of the 3 IPFS hosts the various Hadoop nodes are retrieving their data fromis not known. This is handled by IPFS and is transparent to users of the system.

The system structure for this experiment is further illustrated with the following diagram:



Whilst identical to those used in the local IPFS test above, the timing measurements used were as follows:

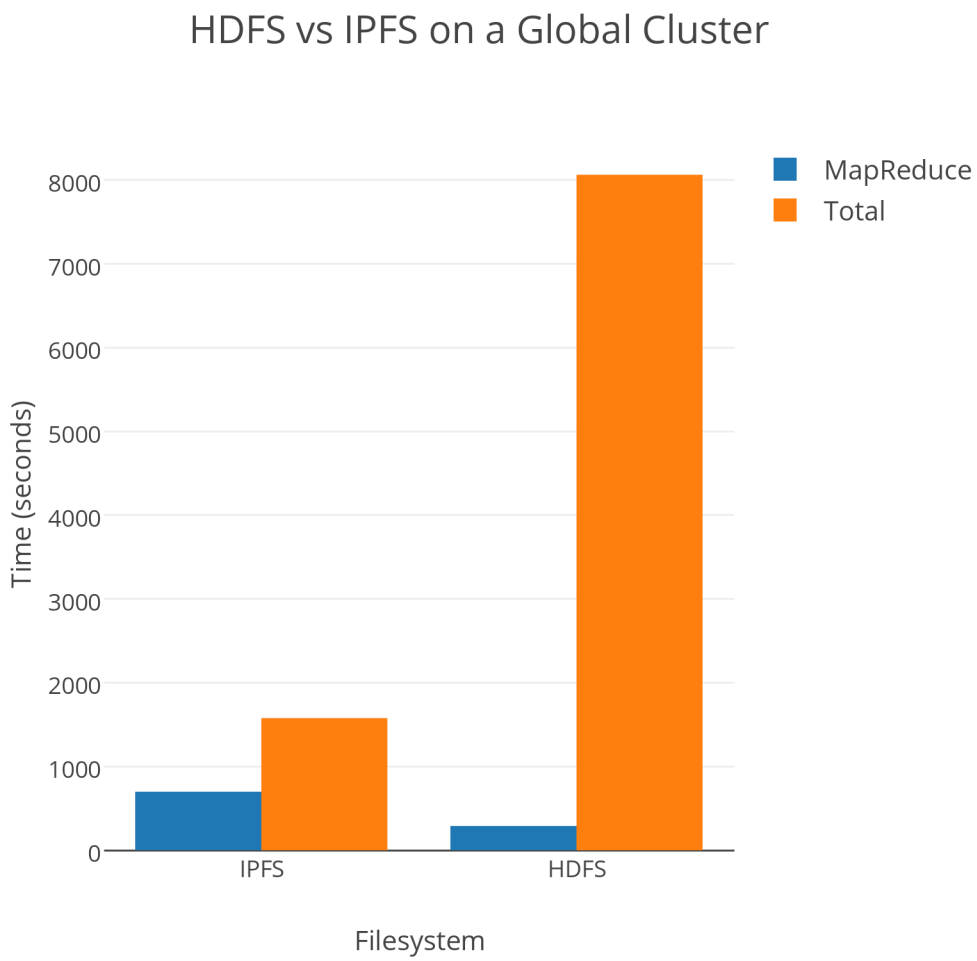
1. *Unix time command on the Hadoop job*: The unix time command was used to time the execution of the Hadoop job in real world terms. This time includes all aspects of the job including setup and the initial accessing of metadata on the data set, via IPFS, as well as the MapReduce task itself.

2. *Hadoop web UI reported task time*: In addition to the time for complete execution of the job, the time taken by the MapReduce job itself (as reported by the Hadoop web UI) was also recorded. This measurement indicates the time taken for the actual analysis.

This test also took a reasonable time to complete and was only repeated 10 times (largely due to AWS billing concerns with moving excessive data between regions). The raw data and an average is given in the table below.

Attempt	Total Time	MapReduce Task Time
1	1604.09	766
2	1600.97	705
3	1538.48	673
4	1479.02	628
5	1594.68	704
6	1639.61	760
7	1595.50	684
8	1543.64	676
9	1555.73	654
10	1620.41	745
AVERAGE	1577.21	699.50

6.1.6 Cross-Region Cluster Comparison



As the above graph shows, comparing HDFS and IPFS on a cross-region cluster is quite a different story to the local cluster comparison. When looking at the MapReduce time alone, HDFS is still a clear winner. This was an expected result due to the way in which HDFS is purpose built for Hadoop and supports data locality. By supporting data locality, Hadoop is able to determine on which nodes in the cluster a particularly block is kept and schedule computation on that block to occur where it is hosted. This significantly reduces network traffic, and in doing so significantly reduces the effects of greater latency (brought about in this case due to the geographical separation of the Hadoop nodes) on the job time. In contrast, IPFS does not support data locality, and

so our system built on IPFS does not benefit from reduced network traffic.

When the overall time (including ingestion) is considered, however, a very different picture is obtained. HDFS spends a very significant amount of time ingesting the data set. The exact reason for this is not clear, but it is likely related to the replication HDFS performs. IPFS is much quicker overall, as it does not need to perform any ingestion.

6.2 Functionality

In addition to performance, the interface was tested to ensure it solves the problems outlined in this work, as well as to ensure it allows users of Hadoop to perform MapReduce jobs directly on data stored in IPFS, with only minimal modification to the jobs.

6.2.1 Remove the Need for Inefficient Data Ingestion Processes

As the performance results above show, performing MapReduce jobs on data sets hosted in IPFS in a geographically distributed cluster is significantly more efficient than ingesting the same data set into HDFS before the analysis is performed. Whilst the MapReduce job itself is slower, if the data set in question is not already hosted on HDFS, the overall time is significantly reduced.

However, if the cluster is not geographically distributed, HDFS remains a more performant alternative. The time needed to ingest data into HDFS is less than the overhead of running Hadoop on top of IPFS, in a local cluster scenario.

6.2.2 No Complicated Mechanism Needed To Move Computation

With data stored on IPFS, organisations are able to easily share data sets and perform MapReduce computations on their shared data sets, without a need for a mechanism to move computation between them. The Hadoop clusters of the different organisations

can be completely seperate, even in different parts of the world, and operate on exactly the same data set. This requirement was tested in the cross-region IPFS tests above, as the data set used in the test was hosted in a completely seperate region to the nodes in the Hadoop cluster.

6.2.3 Decentralised Data Set Hosting

Storing and accessing data sets directly on IPFS eliminates the typically decentralised nature of shared data sets, provided there is adequate interest in the content. As people access a data set, they can choose to pin it (keep a local copy), which can then be accessed by other peers in the network. As more people pin the content, it becomes more and more decentralised.

Whilst a system like this may raise security and quality concerns in that various parties 're-hosting' the data could modify their version and serve incorrect data, this isn't possible in IPFS as all files are content hash addressed. Further, this content addressing means that if a data set is updated or modified it will have a new address. This further decentralises data set hosting and ensures that as long as someone is interested in a version of some data set, it will still be accessible in IPFS.

This aspect of the functionality has not been explicitly tested, but is implicit in the way IPFS itself is designed.

6.2.4 Strong Versioning Guarantees

Data stored on IPFS is accessed using its content hash. For this reason, as long as the original hash given to a party does indeed refer to the data set they are interested in, they can be sure that they are accessing the data they expect. Any change in a file means a new hash must be created, and any change in a string path component for a file means the base hash must also be regenerated. This ensures that when a Hadoop user specifies an IPFS file as input, they know exactly what data they are getting.

This aspect of the functionality has also not been explicitly tested, but is implicit in the way IPFS itself is designed.

6.3 Adaptability

Whilst Hadoop still plays an important role in the big data processing space, Apache Spark proves to be a more generalised framework allowing a greater range of analytics tasks be performed efficiently on large data sets. As such, it would be of great value to be able to repurpose this work for use with Apache Spark. This evaluation criteria is less experimental in nature, and was conducted through researching what would be involved in using the created Filesystem interface with Spark.

On investigation, Spark has explicit support for 'any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc' [7]. This claim is quite broad, and doesn't provide any indication of how this support works. On further investigation, it was determined that Spark has support for the OpenStack Swift Filesystem [5]. This support is better documented and relies on the *org.apache.hadoop.fs.swift.snative.SwiftNativeFileSystem* class being specified through a *core-site.xml* file. This class extends the same *FileSystem* interface as that of the main FileSystem class developed in this work. As such, Spark would be able to run on the IPFS FileSystem in much the same way.

6.4 Limitations of the Evaluation Design

Whilst the above experiments and evaluation criteria are a representative means of evaluating the merits of this work, it is important to note a few limitations.

6.4.1 Single Job Type

In the above experiments, only one type of job was used (a multiple file word count). This job alone is not sufficient to cover the various types of jobs that run on Hadoop, but is adequate in proving that the system is functional and achieves its goals. In order to improve the evaluation of the work, a number of different types of jobs could be run, including iterative machine learning jobs.

6.4.2 Small-Scale Cluster

The size of the cluster used in the above experiments was also quite small, largely due to AWS cost concerns. A more realistic cluster would consist of many more nodes. The cluster used is still adequate in proving the solution works, however, and that it works in a distributed environment.

Chapter 7

Future Improvements

In this section, some of the future efforts that could be undertaken to improve this work in various ways are outlined. Some of these are improvements on my work directly, and others are proposed improvements to IPFS.

7.1 Improve Production Readiness

As with most research work, this solution was developed as a proof of concept and is, as such, not production-ready. The code quality of the work could be improved and performance tweaks made to ready the solution for real-world use in a production setting.

7.2 Improve Testing Approach

In developing this work, I used an end-to-end testing approach. This approach can be argued to have its flaws, and will often be unable to test edge-cases and all logic paths. To combat this and improve the testing of the work, unit tests could be added to test out all different logic paths and potential edge cases. This would also help in making

the work production ready.

7.3 Retrieve File Metadata for an Entire Directory

During the initial phase of a job, Hadoop will perform a *listStatus* operation on its input Path to determine how to split the data. In the current implementation, this operation must individually retrieve object metadata over the IPFS network for each file in the input Path. This is quite a considerable performance loss.

A better approach would be to retrieve this metadata in one go for the entire directory (object). This is, however, not currently supported by IPFS, but would be a useful addition to the system.

7.4 Improve IPNS Caching Mechanism

As outlined in section 5.2.3, the caching mechanism for IPNS records could be improved. Most notably, when publishing a new hash to a peer's own record, the cached version of their record is not updated or flushed. This means that running *ipfs name publish hash* and *ipfs name resolve* on the same peer can mean that the resolve command returns a hash different to that which was just published.

A simple solution to this issue would be to either invalidate the cache for one's own record when *ipfs name publish* is executed. Or, more optimally, for *ipfs name publish* to update one's own cache with the new value.

Chapter 8

Conclusion

This work set out to provide a means for performing big data analysis, at web scale, over decentralised storage. In doing so, it aimed to solve four key problems in the current state of affairs in sharing of data sets. The approach taken was to implement an interface between Hadoop and the InterPlanetary File System. On evaluation of the finished work, these aims of removing the need for inefficient data ingestion, eliminating the need for a complicated mechanism to move computation, decentralising data set hosting and providing data versioning guarantees were all met. Further to this, on analysing the performance of the solution, there are scenarios in which such a system can be significantly more efficient than a HDFS based system.

As society creates an ever increasing volume of data it is hoped that decentralised analytics techniques, such as that proposed by this work, will enable new ways for meaning to be derived and knowledge gained from data sets that may not have been suitable for more traditional analytics platforms. It is also hoped that this system, and potential future systems like it, will enable greater sharing of data, and therefore knowledge transfer, between organisations.

Bibliography

[1] Juan Benet. Ipfs - content addressed, versioned, p2p file system (draft 3), 2015.

[2] Luke Champine David Vorick. Sia: Simple decentralized storage, 2014.

[3] The Tahoe-LAFS Developers. Tahoe-lafs 1.x documentation. http://tahoe-lafs.readthedocs.io/en/latest/, accessed 14/05/2016.

[4] Intel High Performance Data Division. Lustre - the high performance file system, 2013.

[5] Apache Software Foundation. Accessing openstack swift from spark. http://spark.apache.org/docs/latest/storage-openstack-swift.html, accessed 26/09/2016.

[6] Apache Software Foundation. Hadoop hdfs over http - documentation sets. https://hadoop.apache.org/docs/stable/hadoop-hdfs-httpfs/ServerSetup.html, accessed 11/05/2016.

[7] Apache Software Foundation. Spark programming guide. http://spark.apache.org/docs/1.2.0/programming-guide.html, accessed 26/09/2016.

[8] Apache Software Foundation. Welcome to apache hadoop! http://hadoop.apache.org, accessed 14/05/2016.

[9] Nebulous Inc. Sia. http://sia.tech, accessed 14/05/2016.

[10] Storj Labs Inc. Storj - decentralised cloud storage. https://storj.io/index.html, accessed 14/05/2016.

[11] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[12] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters, 2004.

[13] Sanjay Radia Robert Chansler Konstantin Shvachko, Hairong Kuang. The hadoop distributed file system, 2010.

[14] Omkar Kulkarni. Hadoop mapreduce over lustre, 2013.

[15] Seagate Technology LLC. Hadoop workflow accelerator, 2015.

[16] Noriyuki Soda Satoshi Matsuoka Osamu Tatebe, Youhei Morita. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing, 2004.

[17] Osamu Tatebe Shunsuke Mikami, Kazuki Ohta. Hadoop mapreduce on gfarm file system.