SCALABLE INDEXING AND SEARCHING ON DISTRIBUTED FILE SYSTEMS

BY

ITUA IJAGBONE

DEPARTMENT OF COMPUTER SCIENCE

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved ___________________________
Adviser

Chicago, Illinois
May 2016

ACKNOWLEDGEMENT

I want to thank God for making this possible, for seeing me through to this stage of my life and the opportunity to successfully complete my thesis in graduate school.

I want to thank my advisor, Dr. Ioan Raicu for the opportunity to be a member of his lab and work under him as my thesis advisor.

I want to thank my committee members, Dr. Boris Glavic and Dr. Kevin Jin for taking their time to serve on my committee.

I also want to thank my lab member, Shiva Kumar, who also contributed to the success of my thesis, as well as other students who contributed on various parts of early iterations of this work, namely Kevin Brandstatter, David Pisanski, and Dr. Dongfang Zhao.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Scientific applications and other High Performance applications generate large amounts of data. It’s said that unstructured data comprises more than 90% of the world’s information [IDC2011], and it’s growing 60% annually [Grantz2008]. The large amounts of data generated from computation leads to data been dispersed over the file system. Problems begin to exist when we need to locate these files for later use. For small amount of files this might not be an issue but as the number of files begin to grow as well as the increase in size of these files, it becomes difficult locating these files on the file system using ordinary methods like GNU Grep [8], which is commonly used in High Performance Computing and Many-Task Computing environments.

It is as a result of this problem that we have chosen this thesis to tackle the problem of finding files in a distributed system environment. Our work leverages the FusionFS [1] distributed file system and the Apache Lucene [10] centralized indexing engine as a fundamental building block. We designed and implemented a distributed search interface within the FusionFS file system that makes both indexing and searching the index across a distributed system simple. We have evaluated our system up to 64 nodes, compared it with Grep, Hadoop, and Cloudera, and have shown that FusionFS’s indexing capabilities have lower overheads and faster response times.

CHAPTER 1

INTRODUCTION

According to [4] data explosion is fueled by the growth from a mix of some of the following factors:

- The ability of HPC systems to run data-intensive problems at larger scale, at higher resolution, and with more elements (e.g., inclusion of the carbon cycle in climate ensemble models)
- The proliferation of larger, more complex scientific instruments and sensor networks, from "smart" power.
- The increasing transformation of certain disciplines into data-driven sciences for example Biology.

Many of the data-intensive applications in the research community require large simulations of physical phenomena, some of which are too expensive to setup experimentally or impossible to build. For example, it is impossible to remake climate phenomena in the laboratory, as a result, simulations are used in climate modeling. Also simulations are conducted in high-energy physics to design hardware and software systems needed to process data from experiments, before the actual experiments are conducted which cost billions of dollars. These simulations produce large datasets from long-running parallel computations. The Community Climate System Model (CCSM2) [15] developed at the National Center for Atmospheric Research (NCAR) completed the first 1,000-year control simulation of the present climate [16]. This simulation produced a long-term, stable representation of the earth’s climate.

Unstructured data makes up 90% by volume in the digital space compared to 20% structured data according to IDC [17]. This presents a lot of problems. According to 5, approaches to dealing with unstructured data in the research community from time past includes the following:

- Archival and data management systems convert all of the disparate research data into a single unified proprietary format, with an eye on protecting data for years to come. These solutions did not fit the creative and collaborative needs of the research community. Archival systems and processes are not geared toward actually helping the laboratory perform better, or toward maximizing the efficiency of research. They merely ensure that the data is secure.
- Using corporate structured portal solutions and their complex document management systems used in companies unfortunately don't fit the research community because it's a document management.
- Electronic lab notebooks, used by scientist to document research, experiments and procedures in the laboratory, have generally not performed as expected due to their extreme structure.

As the amount of textual data increases it becomes paramount to store this data in a scalable fault-tolerant environment. Distributed file systems acts as storage mechanisms, storing unstructured textual data, spread over a group of nodes or cluster. There comes a time when we will need to locate and use these files in our cluster, in situations where we are dealing with exascale computing or many storage nodes with corresponding large amount of unstructured textual data as already mentioned, locating these files become a problem especially when we are not just interested in the metadata but the content of the data itself. For example, the U.S. Department of Energy (DOE) Coupled Climate Model Data Archive, makes output data from DOE-supported climate models freely available to the climate-modeling community [18]. The data in these archives have been post-processed from the original output data so that it can be stored and accessed in a database that makes the data more convenient and useful to climate researchers.

1.1 Distributed Search

A distributed search engine is a search engine where there is no central server. Unlike traditional centralized search engines, distributed search [19] is a search engine model in which the tasks of indexing and query processing are distributed among multiple computers and networks in a decentralized manner where there is no single point of control.

One of the many use cases for distributed search is horizontal scaling for enhanced performance. Distributed search facilitates horizontal scaling by providing a way to distribute the indexing and searching of workloads in our case unstructured textual data across multiple nodes, making it possible to search and index large quantities of data.

We discuss in detail the anatomy of a search engine in Chapter 2 where we explain how each part works and how it applies to our work.

In most HPC and Many-Task Computing systems, the common way to search for a file is using GNU Grep. However, from our evaluation and experimental results, it shows it does not scale as the files are large and are spread across the cluster. This work is motivated by and builds on distributed search to provide a more scalable way for locating unstructured textual data in HPC and Many-Task Computing systems.

1.2 Contributions

The main contributions of this thesis are as follows:

1. Design and implement a distributed index and search interface to support distributed file systems
2. Performance evaluation of both indexing performance and search performance up to 64 nodes, and comparison to GNU Grep, Hadoop Grep, and Cloudera Search

1.3 Thesis Outline

The thesis is organized in several chapters as follows: Chapter 2 gives background information and related work about FusionFS, Apache Lucene, distributed indexing and searching. Chapter 3 presents the design and implementation of the distributed indexing and search interface. In Chapter 4, we show the evaluation and experimental results. We conclude our work and discuss future work in Chapter 5.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter, covers the following:

- FusionFS, a distributed file system, which serves as our underlying file system
- The Anatomy of a Search Engine
- Apache Lucene, a high-performance, full-featured text search engine library on which we build our distributed search
- Related work

2.1 FusionFS

FusionFS [1] is a distributed file system that co-exist with current parallel file systems in High-End Computing, optimized for both a subset of HPC and Many-Task Computing workloads. As shown in Figure 1, FusionFS is a user-level file system that runs on the compute resource infrastructure, and enables every compute node to actively participate in the metadata and data management. Distributed metadata management is implemented using ZHT [2], a zero-hop distributed hash table. ZHT has been tuned for the specific requirements of high-end computing (e.g. trustworthy/reliable hardware, fast networks, non-existent "churn", low latencies, and scientific computing data-access patterns). The data is partitioned and spread out over many nodes based on the data access patterns. Replication is used to ensure data availability, and cooperative caching delivers high aggregate throughput. Data is indexed, by including descriptive, provenance, and system metadata on each file. FusionFS supports a variety of data-access semantics, from POSIX-like interfaces for generality, to relaxed semantics for increased scalability.

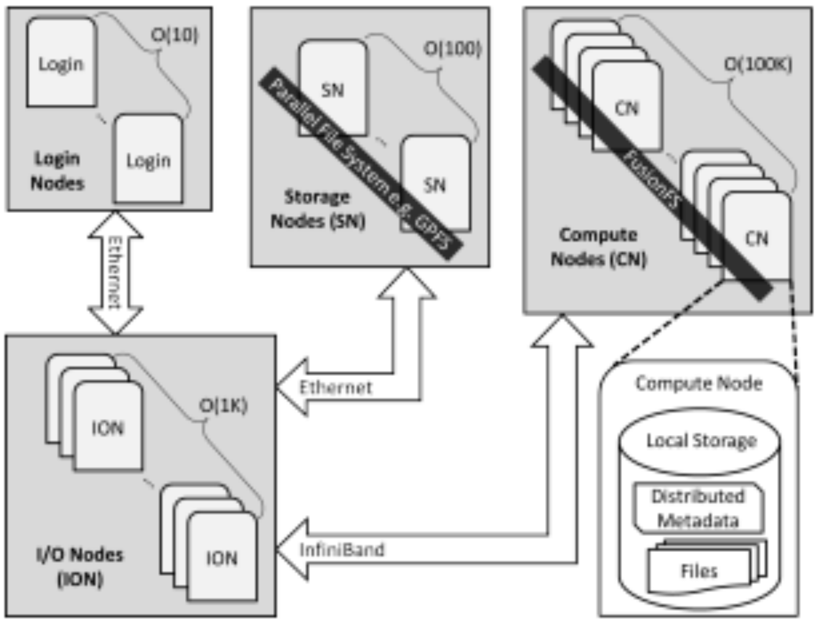

Figure 1. FusionFS deployment in a typical HPC system

From experiments already conducted [1], Figure 2 and Figure 3, show why we chose FusionFS over other HPC storage systems like GPFS [20] and data center storage system like HDFS [3] as our storage system. Figure 2, shows the aggregate write throughput of FusionFS and GPFS on up to 1,024 nodes of Intrepid. As seen, FusionFS shows almost linear scalability across all scales. GPFS scales at a 64-node step because every 64 compute nodes share one I/O node. GPFS is orders of magnitude slower than FusionFS at all scales. Figure 3 shows that the aggregate throughput of FusionFS outperforms HDFS by about an order of magnitude. FusionFS and HDFS was deployed on the Kodiak [42] cluster, a 1024-node cluster at Los Alamos National Laboratory. Each Kodiak node has an AMD Opteron 252 CPU (2.6 GHz), 4 GB RAM, and two 7200 rpm 1 TB hard drives. FusionFS shows excellent scalability, whereas HDFS starts to taper off at

256 nodes, mainly because of the weak write locality as data chunks (64 MB) need to be scattered out to multiple remote nodes. We must remember that our target is scientific applications on HPC machines which HDFS wasn't designed for.

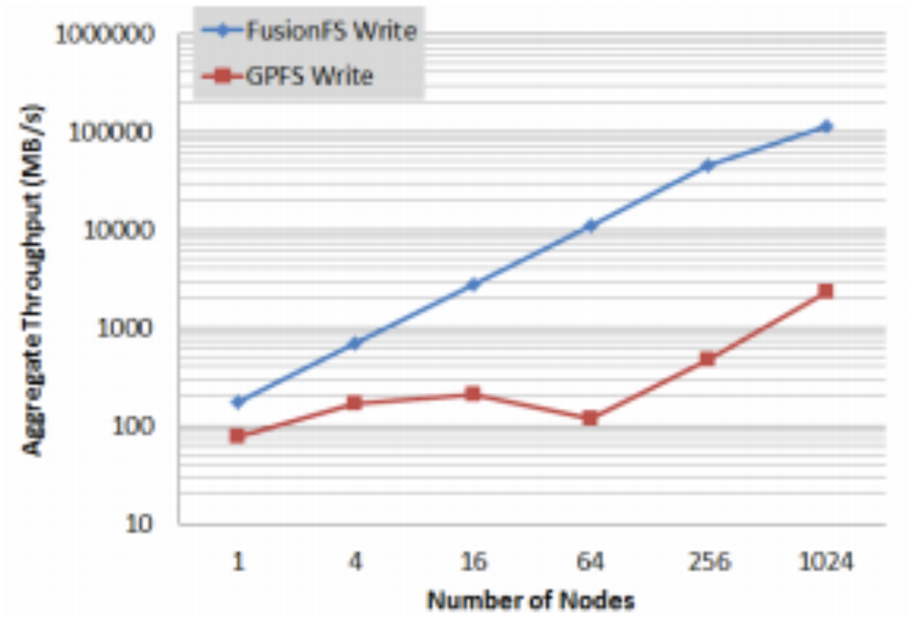

Figure 2. Write throughput of FusionFS and GPFS on Intrepid

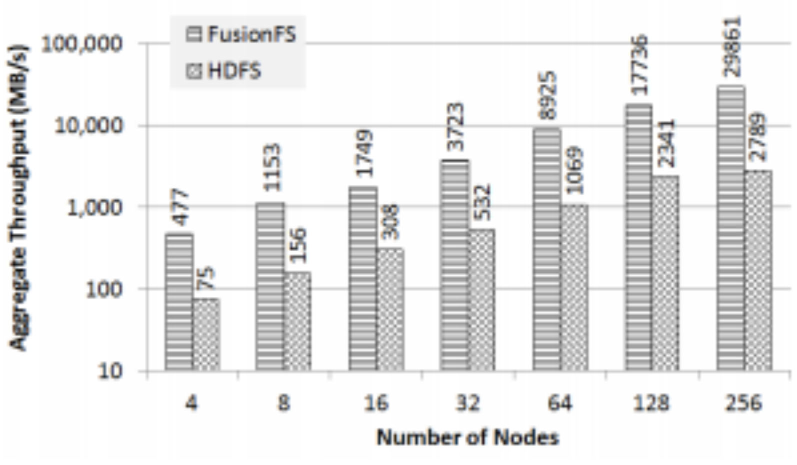

Figure 3. Throughput of FusionFS and HDFS on Kodiak

2.2 Anatomy of a Search Engine

A search engine [21] [22] [23] [24] can be broken down into:

1. Selection and crawling. Using a map of the web graph to select what documents to schedule for fetching, and a crawler to efficiently download and discover these.
2. Index building by inverting the content of documents into efficient inverted indexes for searching.
3. Matching and ranking documents upon user queries.
4. A search engine user interface to handle the dialog with the user.

For our use case as shown in Figure 4, selection, crawling and using a map of web graph to select what documents to index is replaced by the FusionFS (or the storage system). Creating, updating and deleting documents on FusionFS triggers the indexing and de-indexing as the case may be of the content of these documents.

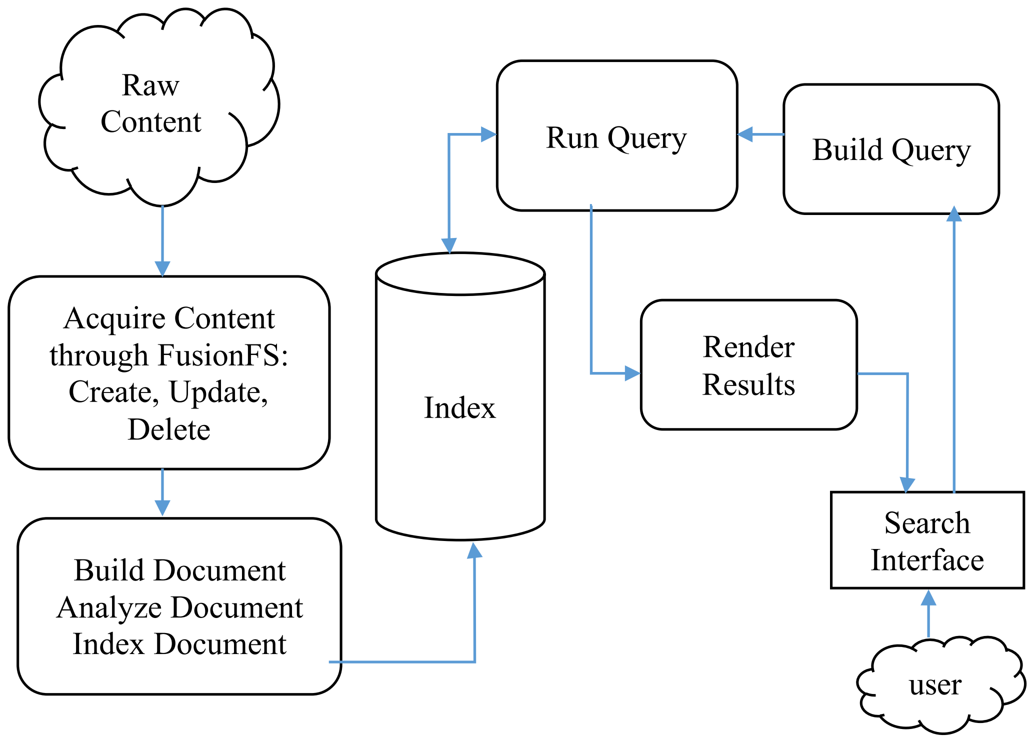
Raw Content
Acquire Content through FusionFS: Create, Update, Delete
Build Document Analyze Document Index Document
Index
Run Query
Build Query
Render Results
Search Interface
user

Figure 4. Typical Components of a Search Application as seen through FusionFS

Without indexing, it will require scanning the content of each file to satisfy a search query. For small amount of documents this may look feasible but for the scale we are targeting as discussed earlier this is not feasible as this will not scale. This is how GNU Grep works, it searches the content of every file in finding a query. Our evaluation in Chapter 4 have shown this to be a flaw.

We store index to improve speed and performance when finding relevant documents during search query. Data to be indexed are converted into formats that allow rapid search. An index can be seen as a data structure inform of a hash table that allows fast random access to words stored in it. Data to be indexed are translated into "units" usually called "documents". A document consists of several separately name field with values which are application specific. Data stored in the index as documents are not stored directly, they are first taken to the phase of tokenization. Data is broken into series of atomic elements called tokens. A token corresponds to a word.

Searching, upon receiving user queries involves looking up words in the index to find documents where they may appear, for our case on a FusionFS node. Precision and recall play a significant role in determining the quality of a search. Precision measures how well the system filters out the irrelevant documents. Recall measures how well the search system finds relevant documents.

User queries are translated into query objects. This queries may contain Boolean operations, phrase queries (in double quotes) or wildcard terms. After building the query, the index is searched, retrieving documents that match the user query. The search interface is usually what the user sees inform of a web browser or desktop tool or mobile device or in our case a command line utility.

2.3 Apache Lucene

Lucene [10] is a high performance, scalable Information Retrieval (IR) library. Information Retrieval [24] refers to the process of searching for documents, information within documents or metadata about documents. Lucene provides search capabilities to an application. It's a mature, free, open-source project implemented in Java. Lucene provides a powerful core API that requires minimal understanding of full-text indexing and searching. It can index and make searchable any data that in which text can be extracted from.

In Lucene, Documents are atomic unit of indexing and searching. It can be seen as a box holding one or more Fields. This Fields contain the content of the data. For each Field, it has a name identifier and series of detailed options that describe what Lucene should do with the Field's value when documents are added to the index. Raw content sources are first translated into Lucene's Documents and Fields before they are added to the index. During search, it is the values of these fields that are searched.

When a field is indexed in Lucene, tokens are created from its text value using a process called tokenization. A Field's value can also be stored. When this happens an un-analyzed/un-tokenized value (a copy of the value) is stored in the index so that it can be retrieved later. An example is presenting the abstract of a paper that has been indexed to a user.

Database store and retrieve content but there are important differences between Lucene and a database:

- **Flexible Schemas**: Lucene has no notion of a fixed global schema which a database does. This means that each document in Lucene can be different from documents before it. i.e. it does not need to have the same fields as the previous document added. This gives an iterative approach to build the index. There is no necessity for pre-designing the schema.

- **Denormalization**: A database can have an arbitrary number of joins through the primary and secondary keys that relate tables to one another. However, Lucene documents are flat.

When a Lucene index is queried, an instance containing an ordered array of references pointing to the documents that match the query are returned. Lucene provides different Query implementations. These implementations can be combined to provide complex querying capabilities along with information about where matches took place in the document collection. When a query has been created and submitted for searching, the scoring process begins. At the heart of a query is the Lucene scoring. Lucene finds the documents that need to be scored based on boolean logic in the Query specification, and then ranks this subset of matching documents. Scoring is dependent on the way documents are indexed. Lucene scoring works on Fields and then combines the results to return Documents. If two Documents have the same content, but one has the content in two Fields and the other in one Field, different scores are returned for the same query due to length normalization. We can influence search results in Lucene by "boosting" at different times. One can be before the document is added to the index and the second is during query were we apply a boost to the query.

It's interesting to note that during search, we can only search the index as it existed at the time search was instantiated. If indexing is happening concurrently as searching is going on, new documents added to the index will not be visible to the searches. To see the new documents, we must commit the new changes to the index and instantiate a new search.

2.4 Related Work

Indexing text based files for searching is a very common practice, and there are utilities for searching for content in a file on a single machine like GNU Grep. However, there are very few distributed indexing and searching implementations. One common example of distributed search systems is Google [23]. However, these search engines are primarily web based which means their index is built using link crawling and aggregation, and requires enormous processing power to build and keep up to date. With this project, we have more modest goals of simply indexing the files that are stored in FusionFS. Cloudera Search [12], developed by Cloudera [11] uses Apache Solr [13], an enterprise search version of Apache Lucene. Cloudera Search includes a highly scalable indexing workflow based on MapReduce. A MapReduce workflow is launched onto specified files or folders in HDFS, and the field extraction and Solr schema mapping is executed during the mapping phase. Reducers use Solr to write the data as a single index or as index shards, depending on system configuration and preferences. Once the indexes are stored in HDFS, they can be queried using standard Solr mechanisms. Apache Solr as mentioned earlier is an open source enterprise search platform built on Apache Lucene. It takes Lucene a step further by providing distributed indexing, replication and load-balanced querying, automated failover and recovery and centralized configuration.

Chapter 4 shows our work in comparison to the standard command line GNU Grep, Hadoop Grep and Cloudera Search.

CHAPTER 3

PROPOSED SOLUTION

In this chapter we explain how we designed and implemented our indexing and search interface on FusionFS.

Figure 5 shows an overview of the index and search interface on FusionFS in a typical HPC system. As soon a file is written to FusionFS, the indexing process begins. When the file content has been indexed, it becomes searchable. Each compute node has a user facing search client utility/interface that makes searching the indexes on each of the compute nodes possible. Every compute node has a search sever that receives search request from the search client. The search server which runs as a daemon, has access to the index stored in FusionFS, queries their respective index and returns results of the request back to the search client. The search server tells the client if it has files matching the query and where such files can be located. In subsequent sections of this chapter, we explain the different components in the mentioned figure.

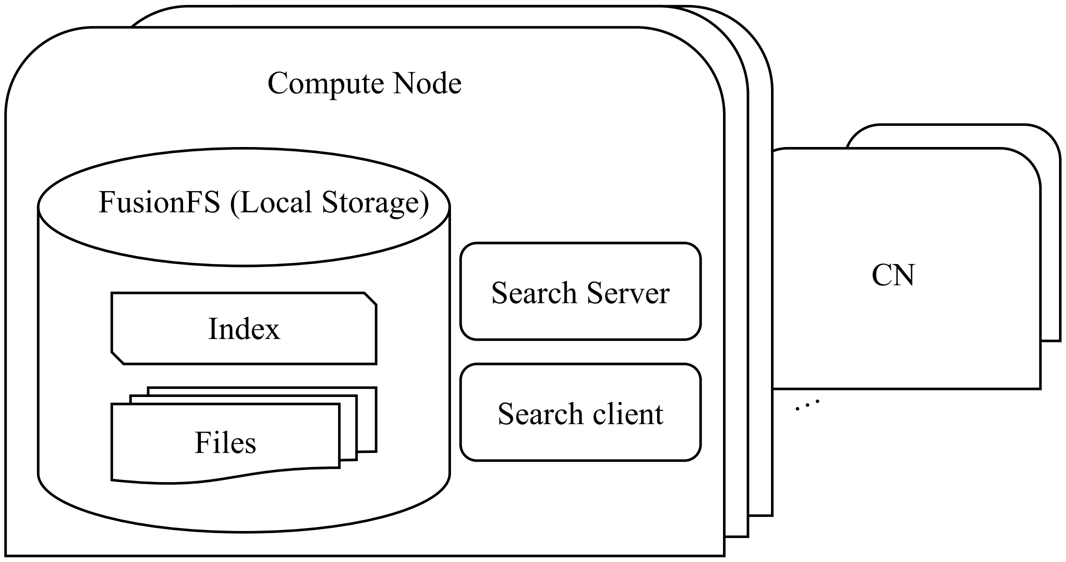

Figure 5. Index and Search interface deployment on FusionFS in a typical HPC system

3.1 CLucene

Since the Lucene core is written in Java, it isn't feasible to use it directly, this is largely due to the fact that FusionFS is written in C/C++. We are currently using a port of Lucene in C++. This port is called Clucene [9]. Though been several versions behind the current Java implementation of Lucene, it provides many of the features we need to implement a high-performance, distributed text search engine. As at the time the port was made, it was said according to the contributors [9], to be faster than Lucene since it is written in C++.

3.2 Library Abstraction

In order to make integration into the file system very simple, we built a set of core library functions. This keeps the methods of indexing and searching separate from the file system. The benefits of this are that if at any point we want to change how to index documents, we only need to modify the library routines, and the file system needs not be aware of the changes.

3.3 FusionFS User Library Extension

At first we attempted to build the indexing functionality directly into the fuse module. This worked well for local file operations, as all the data is local and it only required additional function calls. However, this proved difficult to scale to multiple node deployments as the module had no easy way to operate on remote files. This wasn't a problem for indexing, since all indexing happens locally, but removing a file from a remote node's index proved unfeasible. Thus to address this issue, we decided to extend the file transfer service in FusionFS, to also handle requests for index and de-index operations. FusionFS provides a user library for applications to directly interact with their files. We were able to use much of the same logic as these file operations, since the interface is very similar. Because index operations can take a long time to complete if the input file is large, we don't want the index operations to prevent a file operation from occurring. Therefore, the index operations are received by a separate server process than the file operations. Thus file operations can still be processed while an index is occurring. Furthermore, since all indexing happens on a single process, it alleviates the issue of contention over the index and maintains the order of operations.

3.4 Local File Indexing

Since the files are distributed among the nodes that comprise FusionFS, we decided it would be easiest for each node to maintain the index of the files that reside on it. This is possible because FusionFS is designed to give applications local read and writes. Therefore, each node has a scratch location of all files that are stored on it. To build the index, we use FusionFS to translate the absolute path of each file in the directory to the FusionFS relative path as the index key. Then using the Clucene library, we add each file to the local index. We can do this because each file resides in whole on a particular node, and is not segmented into blocks or chunks as it is on some other distributed storage systems.

3.5 File De-Indexing

File de-indexing occurs in two cases. The first case is the case of a file being removed from the system. The second case is of a file being relocated to a local node for writing. In either case the same process can be taken. Since the file will be removed by a message to FusionFS remote node’s daemon, we simply add another message to be sent prior to that node’s de-index daemon. Thus the file is removed from the remote node’s index, and then removed from the file system. Finally, in the case of a relocation, the file that now resides in the local node will be added to the local node's index upon completion of the write.

3.6 Optimizing the Index

In CLucene, the core elements of an index are segments, documents, fields, and terms. In summary, every index consists of one or more segments. Each segment contains one or more documents. Each document has one or more fields, and each field contains one or more terms. Each term is a pair of Strings representing a field name and a value. A segment consists of a series of files. The number of files making up segment varies from index to index, and depends on the number of fields that the index contains. All files belonging to the same segment share a common prefix and differ in the suffix. CLucene adds segments as new documents are added to the index. It merges segments every so often. To ensure optimal indexing performance and improve search performance, we need to be careful about adding and merging segments while we index documents.

In CLucene, when new documents are added to its index, it by default stores it in memory before writing it to disk. This is done for performance reasons. CLucene has a feature called the merge factor that tells it how many documents to store in memory before writing them to the disk. By controlling this feature, we improve our indexing performance. There is no standard value, the value is selected through experiments, finding which is the sweet spot. Increasing the merge factor allows us to write data to disk less frequently, which speeds up the indexing process. The downside to this approach is that we use more RAM. Given the state of the art of HPC and ManyTask computing systems, this does not pose serious problem as this system come with large RAM sizes. Another side effect of increasing the merge factor too much, results in the decrease of search performance since there are more segments in the index to search over.

3.7 Update on Close

The final piece to effective indexing for searching is to keep the index up to date. In order to do this, we need only modify the index when a file changes. Since this is integrated into the file system, we can issue an index update whenever a file is closed. Clucene provides an update function which under the scene it deletes the document and re-adds it to the index. The other case to consider is that a file may be moved from one node to another. In this case, we have triggers that delete the document from the sending node's index, and add it to the receiving node's index upon completion of transfer.

Finally, since FusionFS keeps track of whether or not a file is written to (for file transfer purposes), we utilize the same information to prevent indexing a file that has not been modified. Thus reading a file will not trigger an index and will prevent the additional overhead from being incurred.

3.8 Search and Multithreaded Server

Our distributed search is implemented using a multithreaded server and client approach. A multithreaded server solves the problem of blocking communication. It allows for concurrent search on the same node. As a result, it gives room for scalability. As shown in Figure 6, when a server receives a search/query requests, it queues the request. One of its worker thread takes the request from the queue and performs a search on the local index. Each worker thread knows the location of the local index, meaning that worker threads perform searches concurrently without interfering with one another as Lucene/CLucene allows concurrently read on an index. Also another design problem been solved is the fact that the search starts running once the server starts running, hence the startup cost of running the search has been reduced.

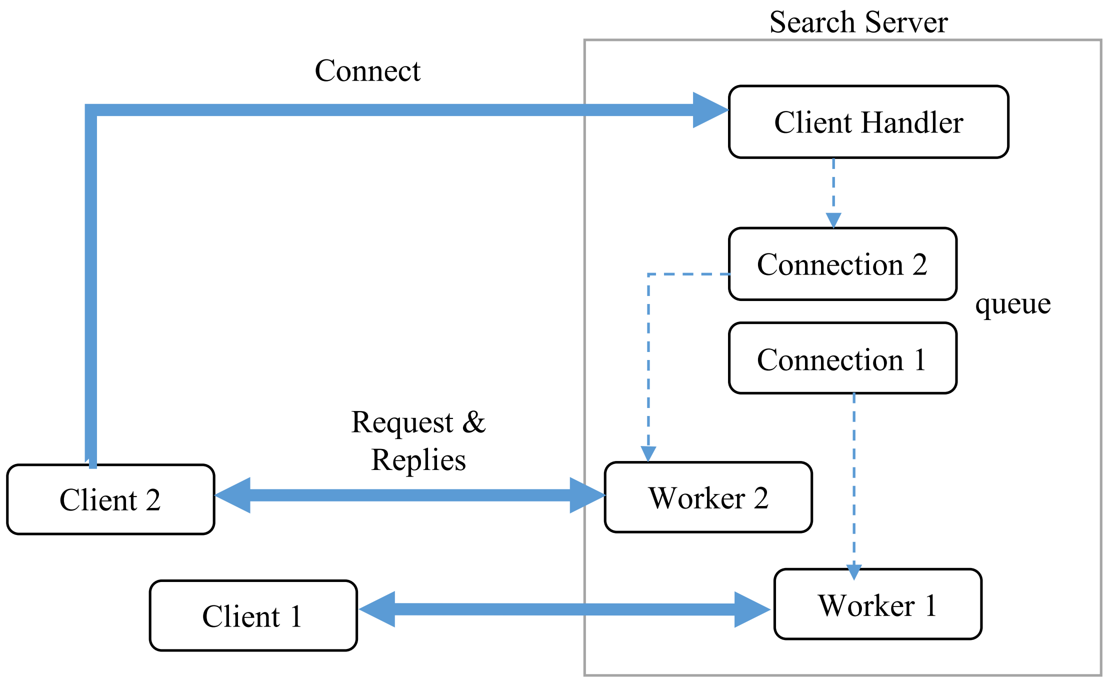

Figure 6. Search Server receives and replies to client queries.

3.9 Multithreaded Client

The multithreaded client solves the problem of connecting to multiple nodes at the same time. As shown in Figure 7, the client has worker threads that know the location of nodes containing the search servers. The nodes are read from a configuration file and are stored in a queue. When a user makes a query, worker threads pick nodes from the queue, establish communication with the search server on that node and sends the query request. Having a multithreaded client makes the application more flexible and also more modular as the client only focuses on sending of query request to the server.

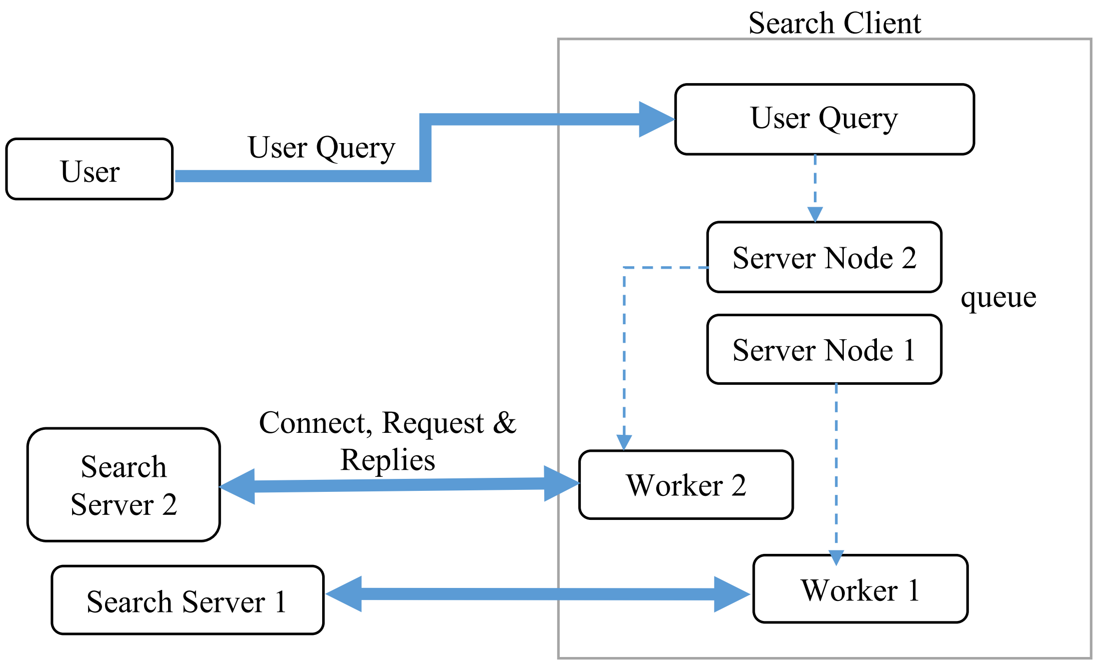

Figure 7. Search Clients receive queries and make search request to Search Servers.

Clients are independent in that they do not need to concern themselves with the location of the index and can be installed on any node as long as it has a list of all the search servers. There is room for improvement in this area. One improvement is using a peer-to-peer pattern, were clients search neighboring nodes and so on. This has an advantage in cases where there is a limitation on the amount of open ports that are allowed on a particular operating system.

3.10 Metadata Indexing and Searching

To improve the search functionality of FusionFS, we added metadata search. Aside from indexing the content of a file, we also index the filename, the file extension as well as the last time the file was accessed and when the file was modified. This improves the search experience. Most times we search for files not based on the content of the file but based on the name or extension of the file.

CHAPTER 4

EVALUATION AND EXPERIMENTAL RESULTS

In this chapter we talk about the experimental environments (hardware and software) including metrics used to measure performance and the workloads used to evaluate our implementation. We talk about about our study and deployment of the indexing and searching feature of FusionFS in house before deploying to a cloud environment. We conclude by showing results conducted in the cloud, comparing it with some related work similar to ours.

4.1 Experiment Environment, Metrics, Workloads

4.1.1 Testbed. Our index and search interface was written in the C/C++ programming language. CLucene is in C/C++ as well as FusionFS. We had two different machine environments for testing and evaluation:

- **Local Virtual Machine:** We have explained why we did this in Section 4.2. Our testbed for this environment was a 64-bit Virtual Machine with 2 vCPU and 1.5GB of RAM.
- **Amazon Elastic Compute:** For deployment to a cluster and comparing it to other similar implementations as explained in Section 4.3, we used Amazon's Elastic Compute Units. We deployed our FusionFS implementation, Grep and Hadoop Grep on a High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors with 2 vCPU, 7.5GB of RAM and 32GB SSD storage for the 10GB data cluster. For validation against Cloudera Search, Cloudera Search ran on the same Intel Xeon with 8 vCPU, 30GB of RAM and 160GB SSD storage.

4.1.2 Metrics and Workloads. Our testing metrics covered the following: Writing Throughput, Index Throughput, Search Throughput and Search Latency. We explain each of these metrics in Section 4.2. Our workloads were divided into two sections.

- **Local Workloads**: This is workload we ran on the local virtual machine. Our testing data was generated from an English Dictionary. The total size of the data was 1GB but was split into 100MB files (10 files in total). In other words, our workload for this experiment was based on weak scaling, keeping the node constant and increasing file sizes from 100MB to 1000MB (1GB). Experiments for the search latency and search throughput were repeated at least three times. The reported numbers are the average of all runs.
- **Cluster Workloads**: This workload was run on the Amazon Cluster. Our testing data is a 10GB Wikipedia dataset [25]. The 10GB dataset are in chunks of 64MB, this chunks are distributed evenly among our 10GB cluster. This workload was applied to our implementation, Grep, Hadoop Grep and Cloudera Search. For search latency and throughput, we ran a 1000 queries where we found the average.

4.2 Studying Indexing and Searching in FusionFS

As a sanity check to ensure we were moving in the right direction, as mentioned at the beginning of this chapter we deployed our implementation on a local virtual machine. For this study our testing metrics covered writing, indexing and searching throughput as well as search latency as we scaled the data size from 100MB to 1000MB (1 GB). For the search throughput, we scaled the number of concurrent client request searching the local server from 1 to 16.

4.2.1 Writing and Indexing Throughput. The writing throughput is the size of the file in MB we can push to FusionFS per second as the file size increases when indexing is enabled. The index throughput on the other hand is the size of file in MB we can index in FusionFS per second as the size increases.

From Figure 8, our writing throughput shows a steady increase up to about 130MB/s at the 800MB mark. It begins to drop afterwards showing there cannot be any more performance improvement. As expected the index throughput is very low, this is because of the time spent indexing each file. As the file sizes increases the compute time spent processing a file increases. As stated this is expected. However, this can be improved by increasing the compute power of the test bed (see Section 4.3).

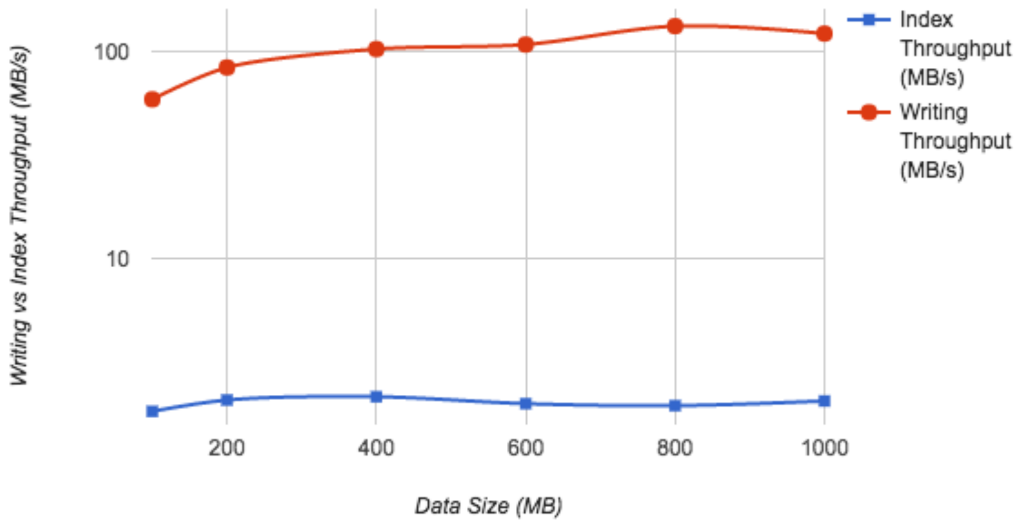

Figure 8. Writing and Indexing throughput in a Local Virtual Machine.

4.2.2 Search Latency. The search latency is the time it takes for the server to respond to a search request from the client. As shown in Figure 9, our chat consists of the latency when caching is turned off and when the caching is turned on. It is expected as the file sizes increases, the segments through which we search increases causing an increase in latency but we have kept the latency very low. Our figure shows that our latency did not, at most, go past the 0.3 seconds mark. This shows that our search scales as the size of the files increases. More impressive is when we allow caching. The latency significantly drops by at least 10x.

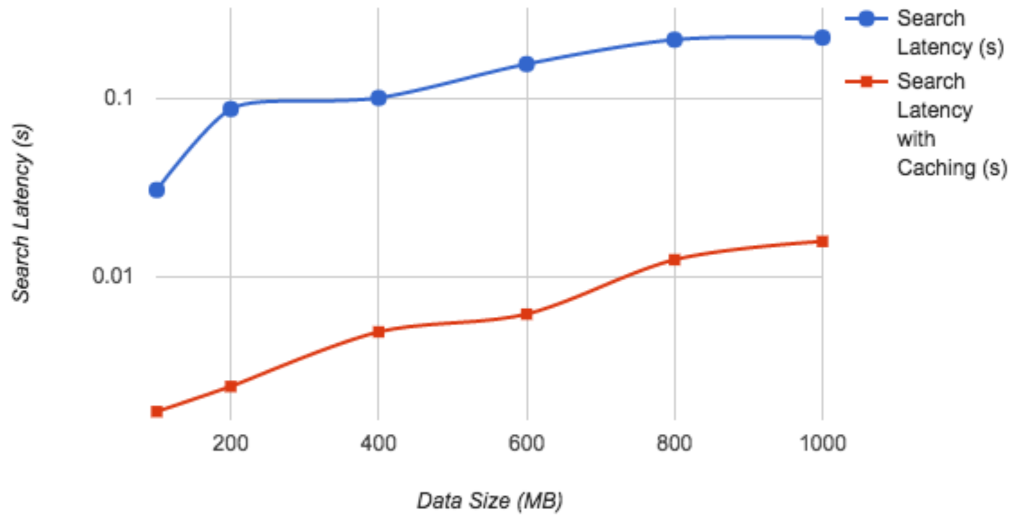

Figure 9. Search Latency with caching and without catching on local virtual machine.

4.2.3 Search Throughput. The search throughput is the the number of concurrent clients that the server can respond to per second as the number of concurrent clients' increase. In carrying out this evaluation we kept the data size constant (1GB). We increased the number of clients, this is made possible since our client is multithread and as a result can make concurrent request given a list of server nodes. Since this is a local experiment done on one system, we only have one server, we made the client concurrently search the server by replicating the server nodes in the clients' configuration (see Chapter 3 for more on this). We also increased the number of worker threads on the server that handle incoming request. We turned off and on caching for this evaluation. From Figure 10 our search throughput when caching isn't allowed does poorly. This is somewhat expected since the search does not cache previous request, thereby repeating the same search process. This is an example where it appears we are penalizing our results heavily. On the other hand, when caching is allowed, our throughput shows positive growth as the number of clients increases. This is encouraging since in real life we need the IO Cache to help improve our performance**.**

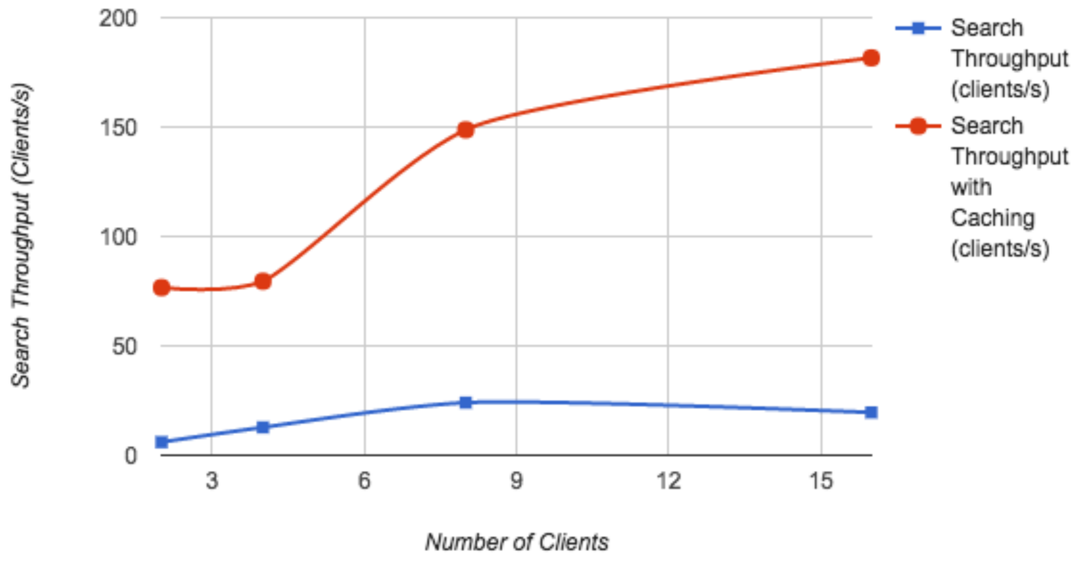

Figure 10. Search Throughput with Caching vs. without Caching on local virtual machine

4.3 Index and Search on FusionFS Vs. Grep, Hadoop Grep and Cloudera Search

In this section we scale our implementation of indexing and searching on FusionFS as mentioned in section 4.1.1 and compare it to some existing, well known implementations similar to our work. We have already discussed the workload of this evaluation in section 4.1.2. Our goal was to see how well FusionFS performed in a cluster from 4 to 64 nodes as compared to Grep, Hadoop Grep and Cloudera Search. We knew we were going to do better than Grep since Grep is not a distributed tool. However, we were more curious to see how our implementation performed as compared to Hadoop Grep and Cloudera Search. Before we talk about the experimental results, we give a brief overview on how the search mechanism works in Grep and Hadoop Grep, and how the indexing and searching mechanism works in Cloudera Search.

Grep [14] searches input files for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default), or produces whatever other sort of output you have requested with options. Hadoop Grep [14] works different from the default UNIX Grep in that it doesn't display the complete matching line, but only the matching string. Hadoop Grep runs two map/reduce jobs in sequence. The first job counts how many times a matching string occurred and the second job sorts matching strings by their frequency and stores the output in a single output file.

Cloudera Search provides the mechanism to batch index documents using MapReduce jobs. MapReduceIndexerTool [26] is a MapReduce batch job driver that takes a configuration file and creates a set of Solr index shards from a set of input files and writes the indexes into HDFS in a flexible, scalable, and fault-tolerant manner. The indexer creates an offline index on HDFS in the output directory. Solr merges the resulting offline index into the live running service. The MapReduceIndexerTool does not operate on HDFS blocks as input splits. It means that when indexing a smaller number of large files, fewer nodes may be involved. Searches in Cloudera Search are conducted using HTTP GET on the Apache Solr URL passing a query as part of the parameters. Since Solr wraps around Apache Lucene, the same way search works for us also applies to Solr.

4.3.1 Indexing Throughput. From Figure 11, on the 4 nodes, Cloudera's indexing mechanism does better than ours. This we believe is because of the way we currently index documents as compared to Cloudera's indexing batch tool. When one file is indexed in FusionFS, there are locks that happen underneath the cover. These locks have consequences especially when indexing lots of files under extremely short time frames as in our case. A lock is placed on the index when a file is indexed. There is index locking also in Cloudera Search, but it is different from ours because it is a batched process, locking the index happens once, during this time the batch files are indexed as a group unlike our design were a lock is needed everything time even though we are indexing a group of files at the same time. We address this issue in Chapter 5.

We see that as we increase the number of nodes, FusionFS does much better than Cloudera Search by at least 2.5x. This is because as we scale the number of nodes, the workload on each node is reduced. This however doesn't appear to be the case with Cloudera Search, increasing the number of nodes has little impact on the indexing throughput.

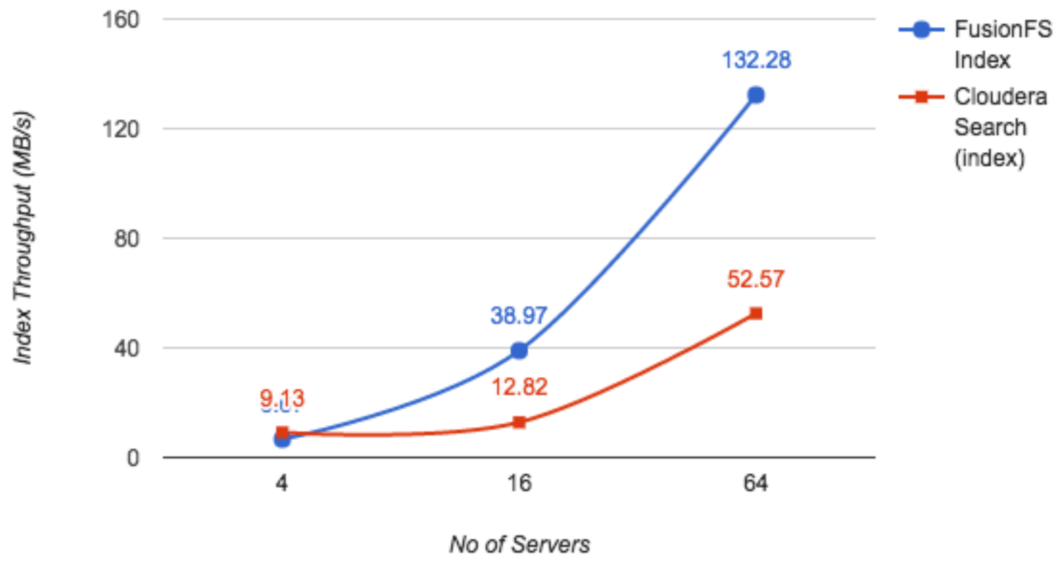

Figure 10. Indexing throughput in 10GB Cluster.

On one node, we wanted to see how CLucene handled increasing number of files. Pegging the total data size at 1.1 GB, we ran CLucene on varying number of files of: 1, 1025 and 1048577 respectively, with each set of files having a total size of 1.1GB. As shown in Figure 12, handling large number of files at once results to poor index throughput. Figure 12 shows a decrease in throughput as the number of files begins to increase from 1 to over 1 million. Opening and closing files adds to the performance overhead of indexing; this is therefore expected that as the number of files reduces the index throughput increases and vice versa. However, it should be noted that CLuence does very well handling individual small files as it takes about 889KB/s to index a 1KB file.

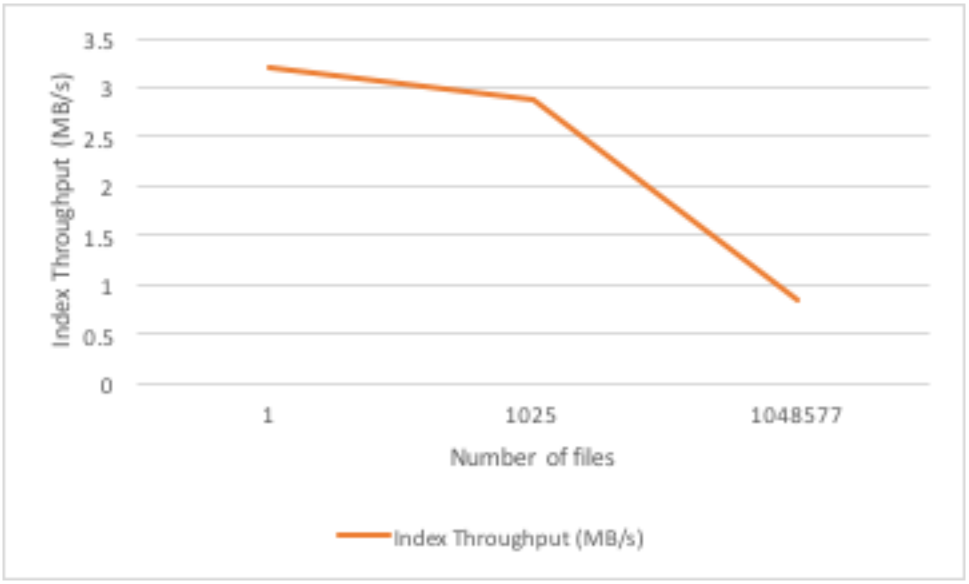

Figure 11. Index Throughput on 1 node with respect number of files

4.3.2 Search Latency. Since Grep is not a parallel tool, we played fair when comparing with the others as shown in Figure 13. We compared Grep and FusionFS Search on a 1 node cluster. Searching on FusionFS on 1 node proofs to be a lot faster than the GNU Grep as show in the figure 13.

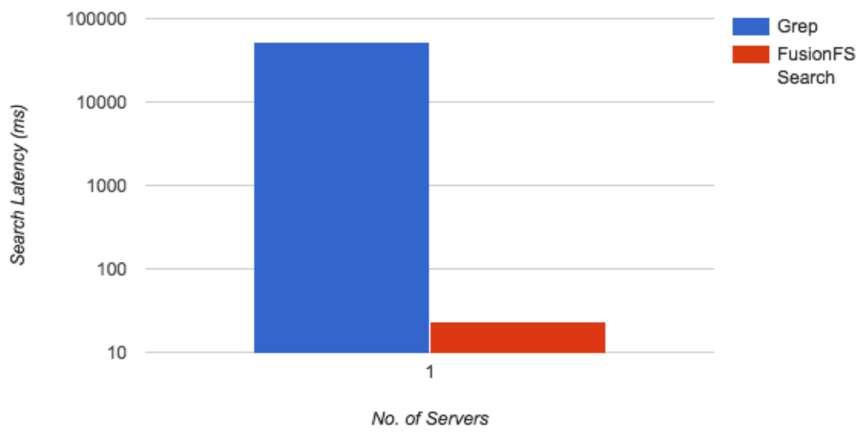

Figure 12. Search Latency of Grep and FusionFS in 1 node of 10GB

After the 1 node experiment, we compared Hadoop Grep, FusionFS Search and Cloudera Search on a 4, 16 and 64 10GB Cluster. Figure 14, shows Hadoop Grep performs the worse of all the implementations because as explained earlier on, Hadoop Grep counts how many times a matching string occurs and then sorts the matching strings. We knew we were going to do better than Grep and were excited to know we did better than Hadoop Grep because of its' method of implementation, we were more curious to know if we would do better than Cloudera Search since the underlying search mechanism is the same as ours (Apache Lucene). Fortunately, our search latency when not cached and when cached does better than Cloudera Search when not cached and when cached respectively. One reason could be because we are running vanilla Lucene. Cloudera Search runs Apache Solr, a child of Lucene, which we believe has extra syntactic sugar added on top of Lucene.

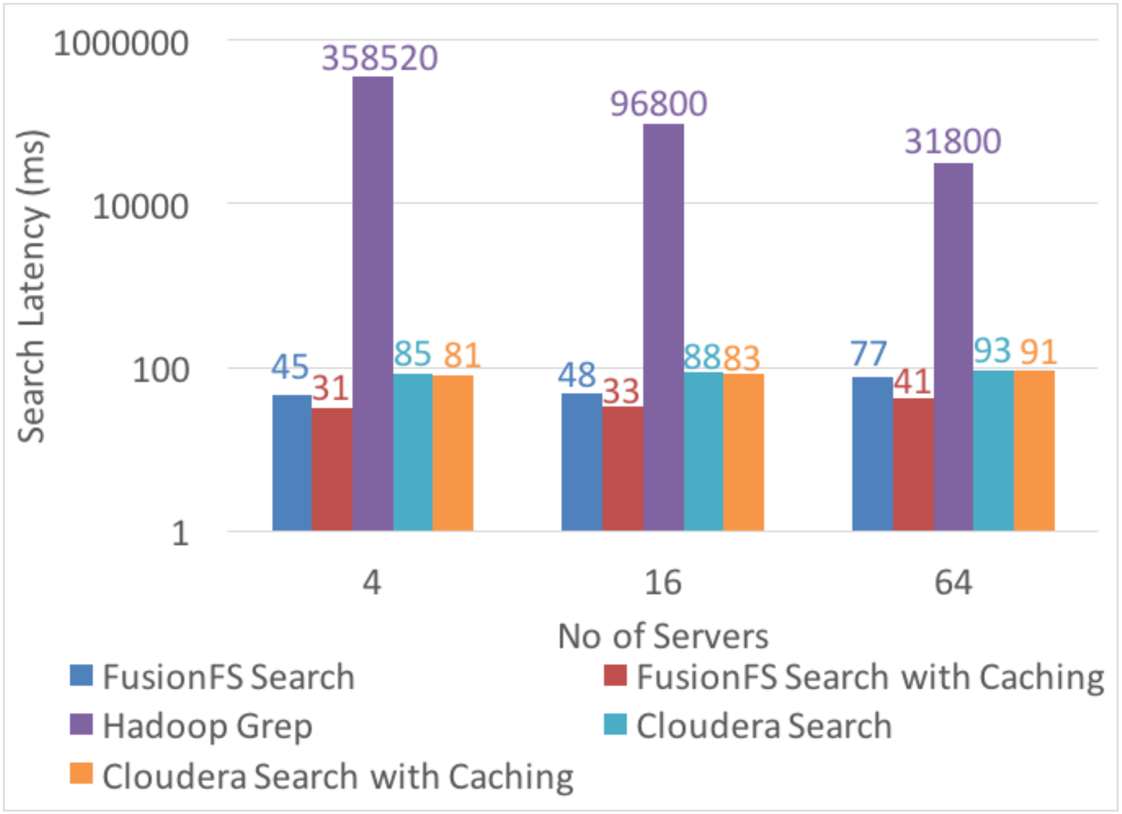

Figure 14. Search Latency in a 10GB Cluster

We also evaluated the search latency on FusionFS Search on a single node as we exponentially increased the data size. We see from Figure 15 that as the data size increases exponentially, the search latency grows linearly. This is in line with what we see in Figure 14. Showing that FusionFS Search scales as data size grows.

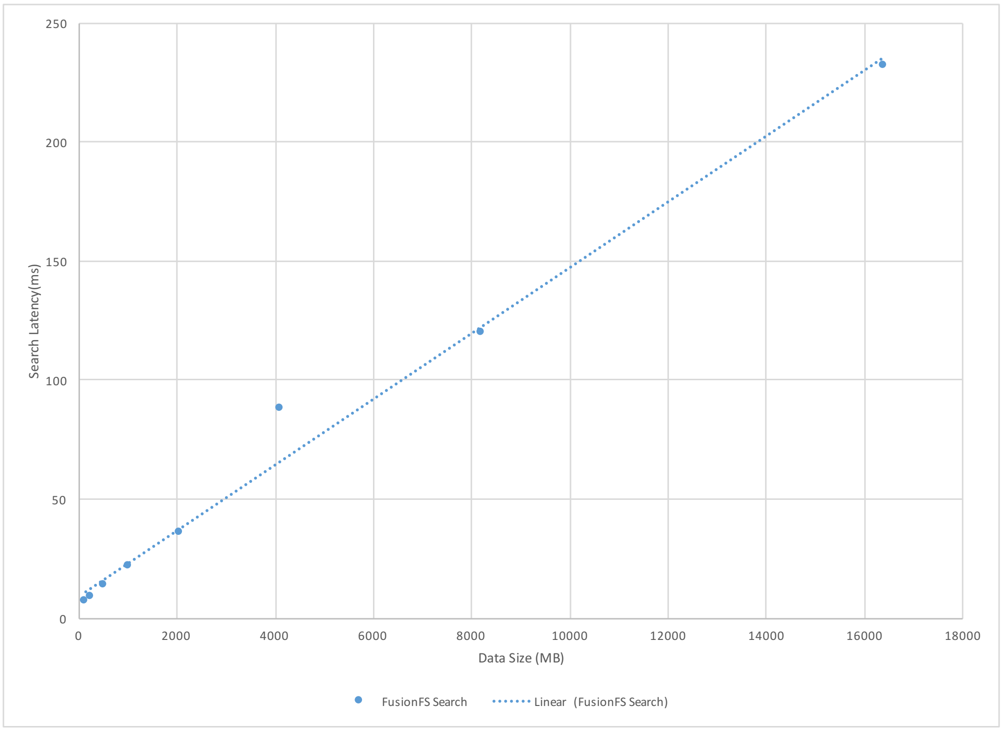

Figure 15. Search Latency of FusionFS on 1 node as data size increases exponentially.

4.3.3 Scalability of Search. We explained in section 4.2.3 how we measured our search throughput on a single node. We used the same method in measuring the search throughput in our cluster. Since we are measuring how much load one search server can handle, we made all the nodes acts as clients. Since client and server can co-exists on the same node without any interference, we also made one of the nodes a server. With all clients having the location of this search server, we made search requests and measured the search throughput. Figure 16 shows our results on a 2.5GB dataset. We compared the search throughput of Grep, FusionFS Search and Cloudera Search making 4 concurrent client request. Figure 14 shows that FusionFS Search responds to more queries per second than Cloudera Search and Grep. Scaling up, FusionFS Search will show to scale as the number of queries per second increases.

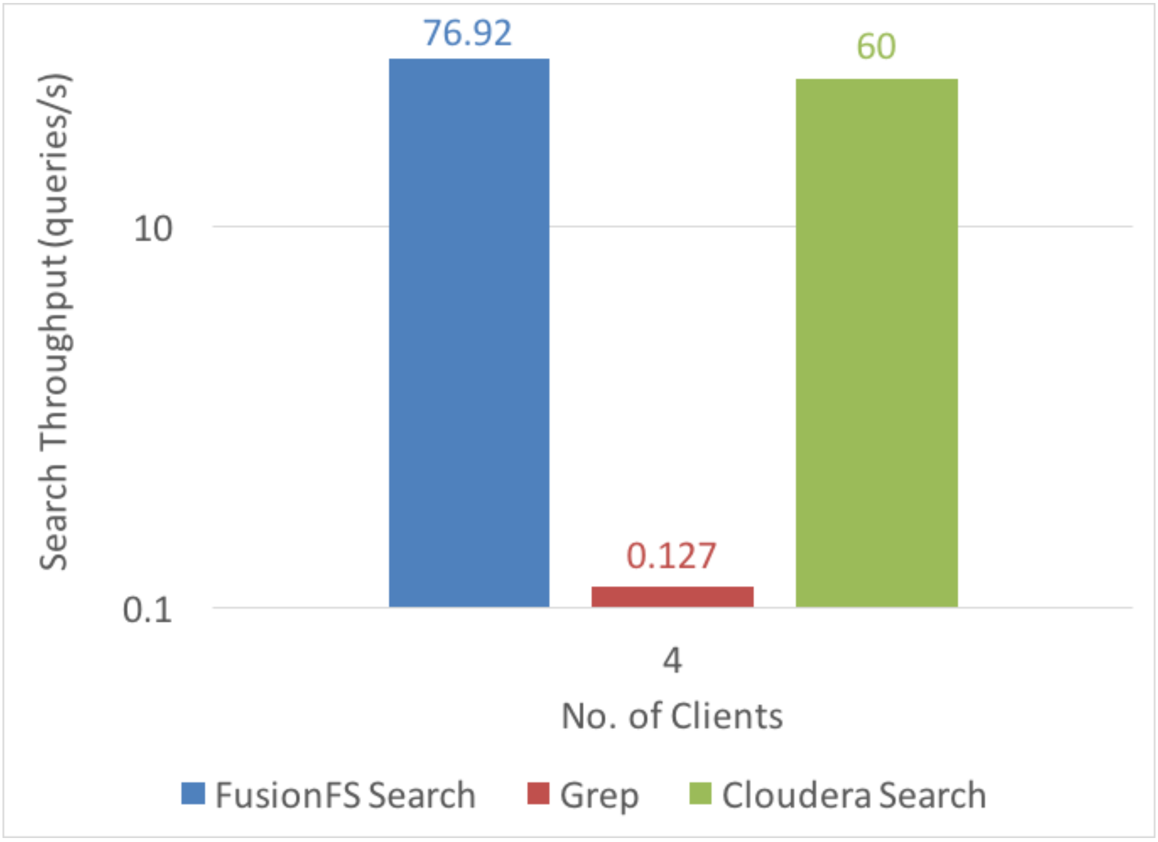

Figure 13. FusionFS Search Throughput in a 2.5GB with 4 clients concurrent request

4.3.4 Write Throughput. Finally, we wanted to know if adding indexing to FusionFS caused a drop in performance and if it did by how much. We compared the writing throughput of FusionFS when the indexing feature is enabled and vanilla FusionFS (without the indexing feature). We also wanted to know if there was a drop in performance with respect to writing to Cloudera HDFS as indexing was on going. This experiment was conducted on 10GB cluster made-up of 4 nodes. Figure 17 shows that our index feature reduces the throughput of FusionFS by an average of 6% while writing to Cloudera HDFS as indexing was happening showed a performance dropped of 50%. This drop in write performance in Cloudera can be attributed to the centralized metadata management of HDFS.

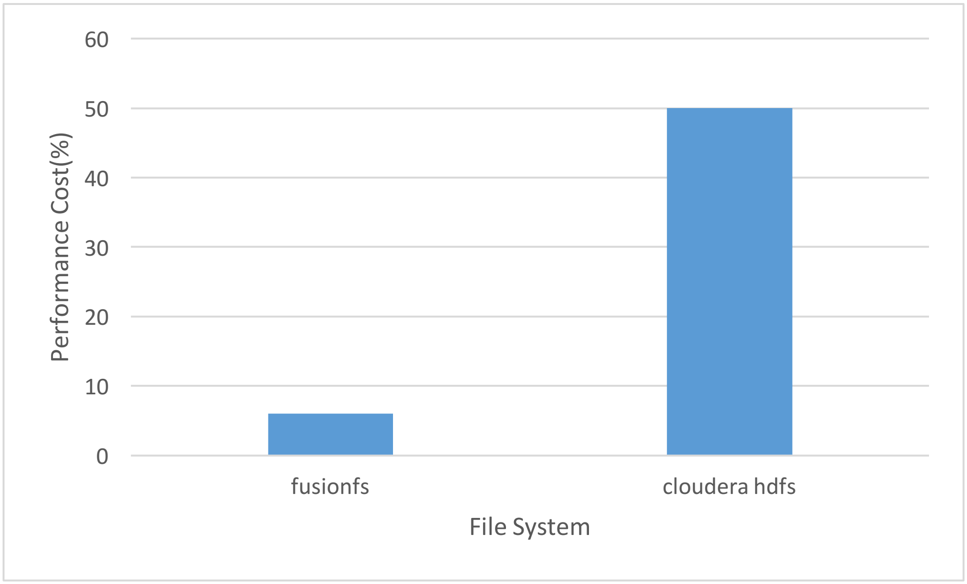

Figure 14. Writing Throughput of FusionFS and Cloudera HDFS in a 10GB Cluster

CHAPTER 5

CONCLUSION AND FUTURE WORK

We have talked about the growth of data and the challenges that arises when we need to find such data in HPC and MTC environments. We introduced a new system that extends the capabilities of FusionFS. Our system indexes and provides an interface for searching data stored in the cluster in a fast manner. We gave an overview of how search engines work especially Apache Lucene and how we used it in our work. We explained how we index files in FusionFS and how we make searching the file system possible. We showed with evaluations and experiments that our implementation is scalable, provides high throughput and has reduced latency. We also compared our implementation to Grep, Hadoop Grep and Cloudera Search, and showed through results that our system beats all three implementations. For the rest of this Chapter, we talk about our future work; how we plan to handle some of the bottlenecks we mentioned during the evaluation and experiment phase and other features we believe to be important to our work.

We talked briefly about the lock issues we face when indexing a document in Section 4.3.1 of Chapter 4 and the drop in performance of FusionFS write throughput in Section 4.3.4 in Chapter 4, when the indexing feature is added. We are looking at different ways in which we can solve this problem. Cloudera Search provides two ways of indexing documents on HDFS, the first we already talked about in Section 4.3 of Chapter 4, which involves using a batch tool. Cloudera Search also gives you the option of streaming data to the index through Apache Flume [27] (a streaming data collection and aggregation system designed to transport massive volumes of data into systems such as Hadoop). This scenario comes in handy when you are interested in performing close to real-time search, were you want data searched as soon as it is index. This method cannot be used when the file size is large, however, it shows that Cloudera Search provides two kinds of policy for indexing documents. We are very much interested in taking this approach. We will have two policies, one to index documents in real time as we are currently doing and the other to index documents in batches. We use the first policy if we are indexing only one document or if the documents are of very small sizes. We use the second policy when we have a large file to index or as in our experiments are indexing many files in the MB range. Instead of FusionFS triggering the indexing as soon as files are written to the file system, we do nothing and let a daemon handle the process of indexing the files or make the administrator trigger this process. Batching, reduces the number of locks that are needed during indexing since we are indexing as a group. Batching also increases the writing throughput of FusionFS since indexing is not involved when files are written to the file system.

Another planned work is building a REST API. Since our target are researchers in the Scientific Community, we want to make our search process more user friendly by producing a simple to use web browser interface like Google Search. We are also looking into the possibility of adding SQL API so that researchers who are familiar with SQL can run SQL like queries on our system.

BIBLIOGRAPHY

[1] Z. Dongfang, et al., "FusionFS: Toward supporting data- intensive scientific applications on extreme-scale distributed systems," in *Proceedings of IEEE International Conference on Big Data*, 2014.

[2] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010.

[4] S. Conway and IDC., Scientific Computing (2014, Mar. 7). *High Performance Data Analysis: Big Data Meets HPC* [Blog]. Available: http://www.scientificcomputing.com/articles/2014/03/big-data-meets-hpc.

[5] S. Deutsch, Scientific Computing (2006, Dec. 12). *Tomorrow's Successful Research Organizations Face a Critical Challenge* [Blog]. Available: http://www.scientificcomputing.com/articles/2006/12/tomorrows-successful-research-organizations-face-critical-challenge.

[6] Kramer et al., "Deep scientific computing requires deep data," *IBM Research and Development*, vol. 48, no. 2, pp. 209–232, Feb. 2004.

[7] Knut et al., "Maguro, a system for indexing and searching over very large text collections," in *Proceedings of the sixth ACM international conference on Web search and data mining*, 2013, pp. 727-736

[8] GNU. (2016). *GNU Grep* [Online]. Available: http://www.gnu.org/software/grep/manual/grep.html.

[9] Clucene. (2013). Available: http://clucene.sourceforge.net.

[10] Apache. (2016). *Apache Lucene* [Online]. Available: http://lucene.apache.org/core/.

[11] Cloudera. (2016). Available: https://www.cloudera.com.

[12] Cloudera. (2014). *Cloudera Search* [Online]. https://www.cloudera.com/products/apache-hadoop/apache-solr.html.

[13] Apache. (2016). *Apache Solr* [Online]. Available: http://lucene.apache.org/solr/.

[14] Apache. (2009, Sept). *Hadoop Grep* [Online]. Available:

https://wiki.apache.org/hadoop/Grep.

[15] National Center for Atmospheric Research. (2016). *Community Climate System Model* [Online]. Available: http:// http://www2.cesm.ucar.edu/.

[16] Berkeley Lab. (2012, Sept.). *NERSC Helps Climate Scientists Complete First Ever 1,000-Year Run of Nation's Leading Climate Change Modeling Application* [Online]. Available: http://www2.lbl.gov/Science-Articles/Archive/NERSC-1000-Year-climate-model2.html.

[17] J. Gantz and D. Reinsel, "Extracting Value from Chaos" International Data Corporation, Framingham, MA, 2011. Available: https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

[18] U.S. DOE. *Coupled Climate Model Data Archive* [Online]. Available: http:// www.nersc.gov/projects/gcm_data/.

[19] Tech Target. *Distributed Search* [Online]. Available: http://whatis.techtarget.com/definition/distributed-search.

[20] F. Schmuck, R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[21] C. Manning, P. Raghavan, H. Schutze, *Introduction to Information Retrieval*. Cambridgeshire, UK: Cambridge University Press, 2008.

[22] L. Bing, *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data.* Berlin, Germany: Springer Berlin Heidelberg, 2011.

[23] L. A. Barroso, J. Dean, U. Holzle, "Web search for a planet: The Google cluster architecture," in *IEEE Micro*, 2003, pp. 22–28.

[24] M. McCandless, E. Hatcher and O. Gospodnetic, *Lucene in Action*, 2nd. ed. Shelter Island, NY: Manning Publications Co, 2010.

[25] Wikipedia. *Wikipedia Dataset* [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Database_download,

[26] Cloudera. (2014). *MapReduceIndexerTool* [Online]. Available: http://www.cloudera.com/documentation/archive/search/1-3-0/Cloudera-Search-User-Guide/csug_mapreduceindexertool.html.

[27] Apache Flume. (2015). [Online]. Available: https://flume.apache.org.

[28] I. Foster, Y. Zhao, I. Raicu and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," in *IEEE Grid Computing Environments,* 2008.

[29] I. Raicu, I. Foster and Y. Zhao, "Many-Task Computing for Grids and Supercomputers," presented at the IEEE Workshop on Many-Task Computing on Grids and Supercomputers co-located with IEEE/ACM Supercomputing, 2008.

[30] I. Raicu, *Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing*. Saarbrücken, Germany: Verlag Dr. Muller Publisher, 2009.

[31] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, K. Iskra, P. Beckman and I. Foster, "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers," presented at the Scientific Discovery through Advanced Computing Conference, 2009.

[32] A. Rajendran and I. Raicu, "MATRIX: Many-Task Computing Execution Fabric for Extreme Scales," M.S. thesis, Dept. of Comput. Sci., Illinois Inst. of Tech., Chicago, IL, 2013.

[33] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang and I. Raicu, “Paving the Road to Exascale with Many-Task Computing,” presented at the IEEE/ACM Supercomputing, 2012.

[34] I. Raicu, P. Beckman and I. Foster, “Making a Case for Distributed File Systems at Exascale,” presented at the ACM Workshop on Large-scale System and Application Performance, 2011.

[35] D. Zhao and I. Raicu, “Distributed File Systems for Exascale Computing,” presented at the IEEE/ACM Supercomputing, 2012.

[36] D. Zhao, C. Shou, T. Malik and I. Raicu, “Distributed Data Provenance for Large-Scale Data-Intensive Computing” in *IEEE Cluster*, 2013.

[37] D. Zhao, K. Qiao and I. Raicu, “HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems,” in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[38] D. Zhao and I. Raicu, “HyCache: A User-Level Caching Middleware for Distributed File Systems,” in *IEEE International Workshop on High Performance Data Intensive Computing*, 2013.

[39] D. Zhao, J. Yin, K. Qiao and I. Raicu, "Virtual Chunks: On Supporting Random Accesses to Scientific Data in Compressible Storage Systems," in *IEEE International Conference on Big Data*, 2014.

[40] D. Zhao, N. Liu, D. Kimpe, R. Ross, X. H. Sun and I. Raicu, "Towards Exploring Data-Intensive Scientific Applications at Extreme Scales through Systems and Simulations," in IEEE Transactions on Parallel and Distributed System, vol.PP, no.99, pp.1-1.

[41] T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang and I. Raicu, "A Convergence of Key-Value Storage Systems from Clouds to Supercomputers," in Concurrency and Computation: Practice and Experience, vol. 28, pp. 44-69, 2015.

[42] Kodiak. (2012). [Online]. Available: “https://www.nmc-probe.org/wiki/Machines:Kodiak,”