Enabling Distributed Data Indexing and Search in the FusionFS Distributed File System

David Pisanski
University of Illinois at Chicago
dpisan2@uic.edu

Kevin Brandstatter, Dongfang Zhao, Calin Segarceanu, Ioan Raicu
Illinois Institute of Technology
{kbrandst, dzhau8, csegarce}@hawk.iit.edu, iraicu@cs.iit.edu

1. INTRODUCTION

Scientific applications and other High Performance applications tend to focus on the generation of large amounts of data for analytic purposes. This is the nature of data science. The big challenge in this field of science is being able to efficiently process results and to locate the items of interest in the results. Since much of the output data of some applications is primarily text based, being able to search the text for certain strings or patterns in order to locate the information relevant to the interest of the analyzer, it is necessary to build an index of the output. As data becomes large, the index also becomes large. Much the same as we must distribute the data across multiple systems because of space requirements, the index must also be able to be distributed and still efficiently maintained and queryable. Thus we present this project to enable transparent built in indexing capabilities to FusionFS[1]. This will allow FusionFS to maintain an up to date distributed index that can be queried through a standard API by user applications.

2. RELATED WORK

Indexing text based files for searching is a very common practice, and there are many utilities for the function of building and creating and index of singular files on a single machine. However, there are very few distributed indexing implementations. One common example of distributed indexing systems is search engines such as Google[2]. However, these search engines are primarily web based which means their index is build 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. Thus we looked for projects that implemented text and file based distributed indexing mechanisms. This lead us to the Apache Solr project[6]. Solr operates as a standalone distributed index, in which clients send it documents to index, and then allows queries to the system for text strings and returns a resulting list of documents. Being a standalone system, it is better deployable for enterprise operations, but not for big data operations that often accompany data science, and the other use cases of FusionFS. Finally, being written in Java, it would not be supported by most supercomputer and cluster environments, and thus would need to be external. As an external platform, it introduces significant overhead of additional storage and resources, as well as failing to take advantage of the low latency networks that the applications are running on.

3. DESIGN & IMPLEMENTATION

3.1 File Indexing

Since text based indexing and search is not a new concept, there is no need to create our own indexing library. Rather, we would like to make use of the Apache Lucene[4] project, which is the basis for Solr. Since the Lucene core is written in java, it isn't feasible to use it directly. Through some searching, we located a C++ implementation of the Lucene library called Clucene[5]. Being written in C++ has two main advantages. First, it can be easily integrated into FusionFS, since FusionFS is written in C++ as well, so the library routines can be used directly. Secondly, this makes it faster than a java implementation as it does not have the overhead of the JVM. The Clucene API provides all the necessary function for creating indexes of documents. The main functions we will be using are the functions to add and remove documents to the index, and the functions to search for keys in the index. One difficulty will be understanding the library, as the amount of documentation is lacking. We hope that we will be able to compliment it with the documentation of Apache's Lucene, as Clucene claims to be an implementation.

3.2 Library Abstraction

In order to make integration into the filesystem pieces very simple, we built a core library set of functions such as index_document and delete_document. This also keeps the method of indexing separate from the filesystem. 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 filesystem needs not be aware of the changes. This is necessary because the Clucene library is very flexible and require the program to determine what and how to index and organize data. Currently the index operation only indexes the raw contents of the file, but it could be modified in the future to also index metadata about file size, creation time, or any other data that may be useful to users for querying files.

3.3 FFSNET 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, ffsnet, to also handle requests for index and de-index operations. We were able to use much of the same logic as file operations, since the interface is very similar. Since 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 locations of all files that are stored on it. To build the index, we use FusionFS to translate the absolute path of each file in this 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 the remote nodes ffsnet daemon, we simply add another message to be sent prior to that nodes de-index ffsnet daemon. Thus the file is removed from the remote nodes 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 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 does not provide an update function, so the document must be deleted and readded. The other case to consider is that a file may be moved from one node to another. In this case, we can have triggers that wrap the file send and receive functions 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.

4. PERFORMANCE EVALUATION

4.1 Test bed

We use Amazon EC2 instances to create our testing cluster[3]. For this project, we use m3.large instances because they contain a 32GiB solid state disk local to the node. Each m3.large node mounts FusionFS onto their local SSD.

4.2 Indexing

We want to compare the writing throughput of FusionFS with and without indexing enabled. Each node in FusionFS stores five 100MiB files of English text. The number of nodes in the system vary between runs by successive powers of two. We begin the benchmark by using parallel-ssh to connect to each node. Each node then copies its dataset into the FusionFS mount point.

We have found that our index extension reduces the throughput of FusionFS by an average of 6%. Because indexing requests are asynchronous, writing to FusionFS with indexing enabled is fast.

4.3 Searching

We compare the run times of different grep strategies to the distributed query tool. We begin by storing five 100MiB files of randomly generated English text on each node, which is then copied into FusionFS; we conduct strong scaling experiments up to 16 nodes. The number of nodes in each run ranges from 1 to 16 in powers of 2. “Index” is the distributed query tool. “Network Grep” greps FusionFS from a single node. “Local Grep” greps a copy of FusionFS stored on a single node’s SSD. “HDFS Grep” uses MapReduce to run a parallel grep on a copy of FusionFS.

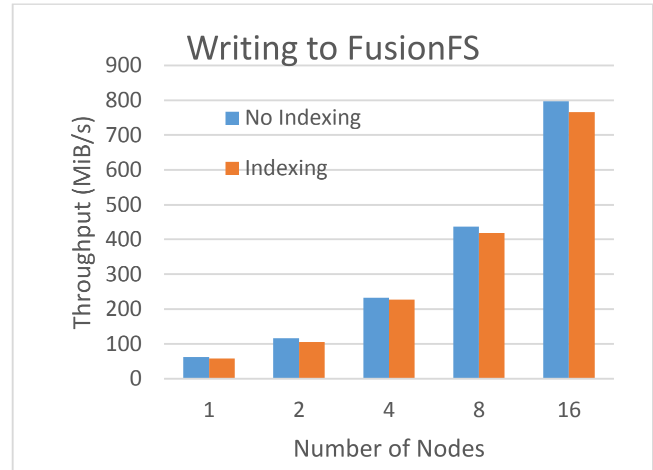

Figure 1. Writing Throughput

In the following graph, we plot the natural logarithm of the runtimes for each search strategy. Note that as the number of nodes double between runs, so does the total amount of data in FusionFS. The Network Grep is the slowest search strategy of all. Since it runs grep over FusionFS from a single node, it reads files from every other node over the network. Hence that single node bottlenecks the system. Both Network and Local Greps do not scale with more nodes.

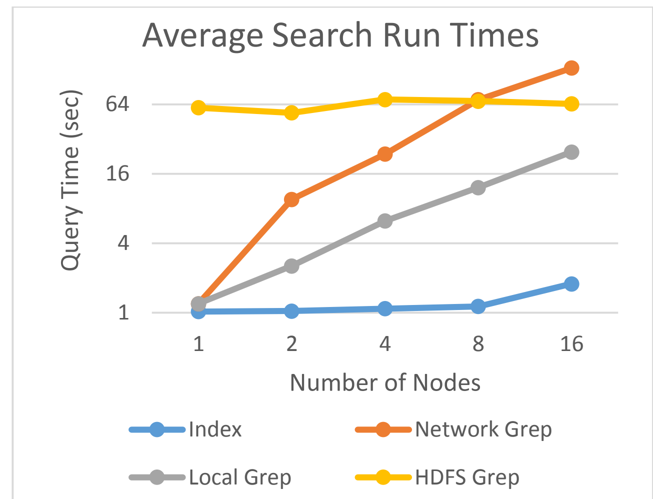

Figure 2. Average Search Run Times

HDFS Grep and the distributed query scale with more nodes. However, the distributed query tool is the fastest of all search strategies. The following graph compares speed of each search strategy to Network Grep. We treat Network Grep as a baseline because it is the most common search strategy.

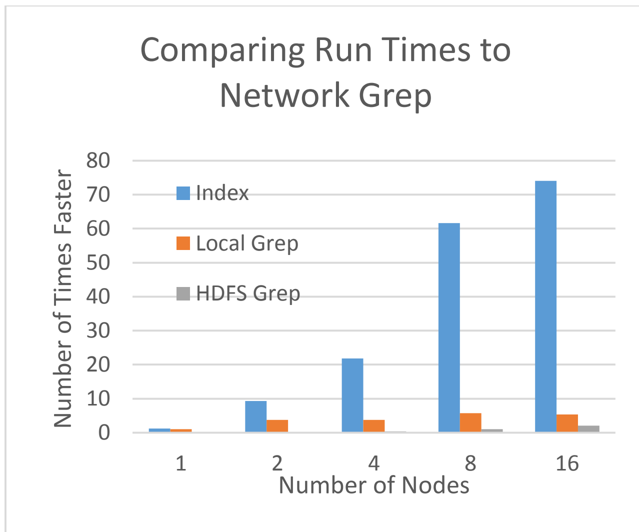

Figure 3. Comparison of Run Times

5. CONCLUSION

This project enables transparent, real time distributed indexing and searching capabilities while maintaining low overhead costs, all while providing over 70X query performance improvement at 16 node scales. Since each node in a FusionFS system indexes data asynchronously, applications are not slowed down and users reap the benefits of having their data indexed. We have also shown our indexing extension scales well to multiple nodes by taking advantage of data locality. Ultimately, with little overhead cost, this work presents a very fast way for users to search through their data without consciously structuring it to do so.

6. FUTURE WORK

6.1 Distributed Search

The primary future development is to implement a client server model for the distributed search tool. The current solution that uses parallel-ssh will not scale well with more nodes because of the overhead introduced by each ssh connection. Using a client server model, we can lower the connection costs. We can also implement fast techniques to send a query request to each node, and reduce the results back to the user.

7. REFERENCES

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

[2] “Crawling and Indexing” http://www.google.com/insidesearch/howsearchworks/crawling-indexing.html

[3] "AWS | Amazon EC2 | Instance Types." 2009. 6 Aug. 2015 <https://aws.amazon.com/ec2/instance-types/>

[4] "Parallel SSH - Google Code." 2009. 6 Aug. 2015 <http://code.google.com/p/parallel-ssh/>

[4] Apache Lucene http://lucene.apache.org/core/

[5] Clucene http://clucene.sourceforge.net/

[6] Apache Solr http://lucene.apache.org/solr/