# FusionFS: Enabling Distributed Indexing And Text Search

Kevin Brandstatter
Illinois Institute of Technology
kbrandst@hawk.iit.edu

**ABSTRACT**
This project will focus on extending the functionality of FusionFS[1] to enable file-system wide text indexing and searching capabilities. It will build on existing indexing libraries, and utilizes the distributed architecture in order to enable fast distributed text searching across a distributed file-system

## 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. This will allow FusionFS to maintain an up to date distributed index that can be queried through a standard apiby 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[3]. 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. 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 filesystem. 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 a index update whenever a file is closed. Clucene does not provide an update function, so the document must be deleted and re-added. 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.

### 3.7 Distributed Search

Currently we have functionality to query a local nodes index for a specific key word or phrase. In order to search the entire system we plan to add a query server. The query server will receive requests from a client, then run the query on the node's local index. It will then return a top subset of the results to the client, who will aggregate and present the results. This work is currently in progress.

## 4. EVALUATION

### 4.1 Test Bed

For our initial evaluation runs we deployed our solution on an amazon EC2 small instance. This deployment limited the scale of our tests, however we did not have sufficient time to deploy and evaluate on larger instances.

### 4.2 Experiment Setup

To evaluate the solution, we were primarily interested in understanding how much overhead adding this live indexing support to the file system incurred. As such, we sought to provide different workloads using differnet sized files and the number of them. As such, we gathered a text data set of 1 GB in sized for our initial testing. We then did tests of indexing the dataset with varying file sizes from 10MB to 1GB. Smaller file sizes proved difficult to test in this configuration because of limitations on directory size in FusionFS.

### 4.3 Results

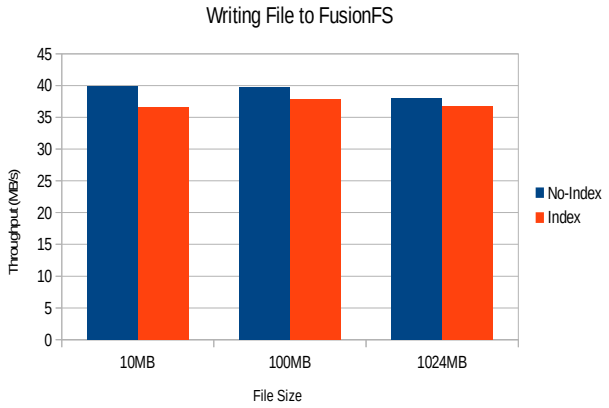


**Figure 1: Throughput of write operations**

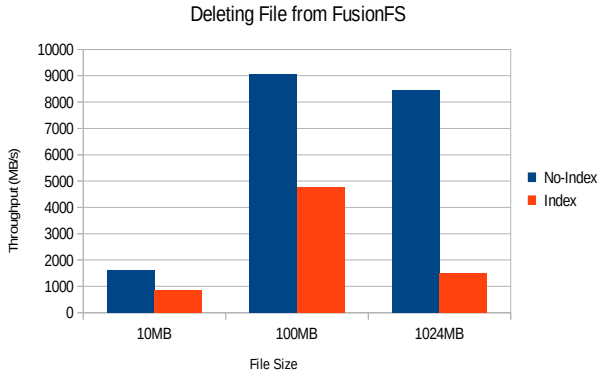


**Figure 2: Throughput of remove operations**

As is shown in figure one, the throughput of write operations is only minimally affected by the addition of the index operation. This minor performance penalty is due to the need to send an additional message to the server. Additionally, this is dependent on the number of files, more than the total amount of data. Thus indexing fewer but larger files sees less performance impact, up until the files create a bottleneck on memory usage. Similarly, this is the same of remove operations. Because the index and de-index requests are asynchronous, the only additional work needed is a single network communication.

## 5. FUTURE WORK

### 5.1 Distributed Search

The primary future development is to implement the program to search the system wide index and return results. The approach has been so far outlined, but due to time constraints, implementation has yet to be completed.

### 5.2 Further Benchmarking

We also plan to expand our benchmarking and evaluation. First, we would like to experiment with smaller files, and lots of files in order to fully understand the impact of the additional network communication. In addition to this, we still need to evaluate at scale. This requires us to redeploy over multiple nodes, and create a workload using multiple writers and removers in order to evaluate any issues with performance or reliability in a multi user, multi access system. Finally, after the distributed search is completed, we will need to evaluate its performance and accuracy, as well as it's improvement over other methods of searching.

### 5.3 Comparisons

In order to better evaluate our solution, we aim to compare it to existing ones. First, we want to show that an indexed approach to searching is a worthwhile improvement to existing methods. The primary existing method is to issue a recursive grep through the filesystem. We expect that this will be very slow, and seek to show that a distributed index based approach makes this search fast enough to be more practical and a worthwhile trade-off in performance. Finally, we want to evaluate it in comparison to existing distributed search platforms such as Solr, in order to effectively judge our ability to scale and search with reasonable performance.

## 6. CONCLUSION

We present a solution to adding functionality for a distributed up to date index of a distributed file-system. While it faces many challenges, we hope to show that it introduces little overhead in order to be a reasonable addition, while providing a very fast and scalable method to searching the contents of the file-system. This work will be useful for searching for specific result files from many output files for the use cases of finding the data needed without need for complex hierarchical storage patterns,, as well as providing the baseline for an extensible system that scientists could use to index their results at creation time for faster processing and querying.

## 7. REFERENCES


[1] FusionFS http://datasys.cs.iit.edu/projects/FusionFS/

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

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

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

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

[6] Zhao, Dongfang; Zhao Zhang, Xiaobing Zhou, Tonglin Li, Dries Kimpe, Phil Carns, Robert Ross, and Ioan Raicu; FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems

[7] Dean, Jeffery;Sanjay Ghemawat; MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004