Toward Scalable Indexing and Search on Distributed and Unstructured Data

Alexandru Iulian Orhean, Itua Ijagbone, Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616
aorhean@hwk.iit.edu, iijagbone@iit.edu, iraicu@cs.iit.edu

Kyle Chard
Computation Institute
University of Chicago
Chicago, IL, 60637
chard@uchicago.edu

Dongfang Zhao
CSE Department
University of Nevada
Reno, NV, 89557
dzhao@unr.edu

***Abstract*—The ubiquity of Big Data has greatly influenced the direction and the development of storage technologies. To meet the needs of storing and analyzing Big Data, researchers and administrators have turned to parallel and distributed storage and compute architectures in both industry and science. While the problems of securely and consitently storing and accessing data in large parallel and distributed file systems have been addressed in both the research and production systems, the indexing and search through large unstructured data and metadata has largely been overlooked. According to the International Data Corporation, more than 90% of data found in the digital universe is unstructured, emphasizing the importance of developing efficient solutions for querying distributed data. This paper proposes a novel indexing solution, called FusionDex, that significantly improves the performance of querying across distributed file systems. FusionDex leverages state-of-the-art, open-source indexing modules as its building blocks to deliver an integrated system for enabling efficient user-specified queries over distributed and unstructured data. FusionDex has been evaluated on a cluster of 64 nodes, and results show that it outperforms existing tools (in some cases by orders of magnitude), such as Hadoop Grep and Cloudera Search.**

***Keywords*-indexing methods; distributed file systems; unstructured data search**

I. Introduction

In the era of Big Data, the amount of information continuously produced, by scientific instruments, social media applications, smart devices and sensor networks, can not often be accommodated on a single storage node. Modern applications make use of parallel and distributed strategies to store and process data at scale. While extensive research has focused on the challenges associated with storing and processing large amounts of data efficiently, the area of information retrieval and data search in distributed systems is often overlooked. Although various indexing techniques have been studied in the database community, adapting them to meet the needs of those applications whose data are primarily distributed and unstructured is still in its infancy. To make matters worse, according to the International Data Company [1], most of the data found in distributed file systems is unstructured, posing a serious challenge to the development of efficient models for querying large, unstructured, and distributed data.

This paper proposes an indexing solution that takes into account the unstructured and distributed nature of Big Data. While many Big Data systems use a single coordinator to manage resources and data placement (e.g., Hadoop [2] and Myria [3]), the proposed indexing strategy envisions a fully distributed architecture, deploying indexing modules locally on each node of the underlying distributed file system. This model assumes a relatively common distributed file system model in which files are stored on the nodes as a whole. That is they are not segmented into blocks or chunks and spread across multiple nodes. Thus, at the cost of restricting the model to a file level storage paradigm, the indexing-related computations can be done locally, which reduces communication between nodes. This scheme removes the concept of a single point of failure and reduces the potential performance degradation from inter-node interference, while preserving scalability.

FusionDex represents the implementation of our proposed indexing solution. It is implemented in the FusionFS [4] distributed file system which leverages the ZHT [5] distributed key/value storage system, and uses the CLucene [6] framework as the indexing engine. CLucene provides the libraries for efficiently indexing and querying data. Inter-node communication is accomplished through FusionFS's data transfer service. Investigation of FusionDex's performance, on a cluster composed of 64 nodes on the Amazon EC2 cloud, showed that the distributed indexing approach has high performance gains in comparison with state-of-the-art approaches, such as Hadoop Grep [7] and Cloudera Search [8]. Some of the results in this paper were first reported in a master thesis [9] along with a poster [10].

The rest of the paper is organized as follows. Section II describes the principles of the proposed indexing scheme, before presenting the details of the underlying distributed file system (FusionFS) and the implementation (FusionDex) in section III and section IV, respectively. In section V the performance of FusionDex is investigated and compared with other state-of-the-art approaches. Finally, related work is discussed in Section VI and our contributions are summarized in Section VII.

II. Design Principle

In comparison with well structured data, found for example in relational database systems, unstructured data does not preserve the same structural characteristics, predefined data models, or well-described organization. As such, traditional models used for indexing data in relational database system are not applicable to unstructured data. While there are countless examples of indexing approaches for unstructured and text-based data, including Lucene and the many relational databases that now support free-text queries, these cannot be directly applied to large storage systems due to their distributed nature and extreme data scales. With these properties in mind, FusionDex aims to eliminate the performance bottleneck of distributed indexing of unstructured data, by removing a single point of failure and minimizing the penalties of such a strategy.

The first design principle of FusionDex is that it does not have global coordinators. This approach is in contrast with popular Big Data systems, like Hadoop [2], Myria [3] and SciDB [11], that are all designed with a single coordinator or master that is in charge of managing the entire system. FusionDex is completely distributed, each node playing the role of both an indexing unit and of a utility interface (e.g., query interface). A client may submit queries to any node, the operation is then distributed throughout the system (across nodes) automatically and efficiently. A response is then assembled by each node and sent back to the client via the same interface. Through caching mechanisms and the balancing of communication, the distributed query throughput can be significantly increased in contrast to the throughput of a single search server or coordinator.

Removing the coordinator has, admittedly, its own drawbacks. The reason why so many systems embrace the idea of single master is obvious: it is easy to maintain consistency and synchronizations between operations, while efficiently updating the control unit of the system. Proper motorization of the operations and the internal state of the components found in a peer-to-peer system, on the other hand, not only requires greater care from the perspective of design and development but also incurs N-to-N network overhead. For the general problem of querying distributed and unstructured data, there is no definitive solution that satisfies all constraints, but in practice, trade-offs are balanced so that the problem is solved for a reduced number of cases. In FusionDex's case, the benefits of efficiently and easily updating global system consistency and coherence, usually obtained through a master node or coordinator, are traded for the benefits of increased scalability, thus providing a means for managing and maintaining Big Data.

The second design principle is that FusionDex aims to minimize data movement, for example during the indexing of files in the distributed file system. This design decision is somewhat a consequence of the removal of global coordinators. Without a global coordinator the network traffic would follow a N-to-N pattern, which would cause a performance catastrophe if each node produces a considerable amount of data, that must then be transferred. To overcome this challenge, the proposed model precludes the inter-node interference at the file level, by only allowing message exchange across nodes. In other words, the system envisions one indexing module deployed on each node, being responsible only for the local files found on the respective node. Therefore, FusionDex expects many small-size messages to build up the global state. The rationale is that modern network hardware is usually throttled by bandwidth rather than latency, making FusionDex's small messages an ideal solution that does not pose much pressure on the systems.

III. FusionFS: a recap

As FusionDex is built upon the FusionFS [4] file system, we first outline FusionFS's architecture and unique features. FusionFS is a user-level file system that runs on the compute resource infrastructure. It enables every compute node to actively participate in both metadata and data management. A client (or application) is able to access the global namespace of the file system via a distributed metadata service. Metadata and data are completely decoupled: the metadata on a particular node does not necessarily describe the data residing on the same node. The decoupling of metadata and data allows different strategies to be applied to metadata and data management, respectively.

A. Distributed Metadata Management

While many distributed file systems have adopted a distributed model for metadata management, such as partition-based or hierarchical structure, FusoinFS's metadata takes a completely distributed topology directly on the flat file system namespace. More specifically, every entity (either file or directory) represents a key in a distributed hash table followed by a value, which could be the physical location of a file or a list of entities (files or directories) contained within a directory.

FusionFS has different data structures for managing regular files and directories. For a regular file, the field *addr* stores the node where the file resides. For a directory, the field *filelist* records all the entries contained within the directory. This *filelist* field is particularly useful for providing in-memory speed for directory listing, e.g. "ls /mnt/fusionfs". Both regular files and directories share some common fields, such as timestamps and permissions, which are commonly found in traditional i-nodes.

B. Data Movement Protocols

Given the distributed nature of FusionFS there is a need for nodes to communicate when accessing data and metadata. However, neither UDP nor TCP is ideal for supporting communication between HPC compute nodes: UDP is a

highly efficient protocol, but lacks reliability; TCP, on the other hand, supports reliable transfer of packets, but adds significant overhead.

To address this challenge FusionFS includes a new data transfer service called Fusion Data Transfer (FDT). FDT is built upon UDP-based Data Transfer (UDT)—a reliable UDP-based application level data transport protocol for distributed data-intensive applications [12]. UDT adds its own reliability and congestion control mechanisms on top of UDP and supports higher speed transfers than TCP.

When the application on machine A issues a POSIX fopen() call, it is caught by the FusionFS implementation in the FUSE user-level interface (i.e., *libfuse*) for file open. The metadata client then retrieves the file location from the metadata server via its distributed hash table. The location information might be stored in another machine B, so this procedure could involve a round trip of messages between A and B. Having resolved the location of the file to machine C FusionFS then needs to transfer the file from C to A. Finally the local operating system triggers the system call to open the transferred file and returns the file handle to the application.

Before writing to a file, FusionFS checks if the file is being accessed by another process. If so, an error is returned to the caller. Otherwise the process can do one of the following two things. If the file is originally stored on a remote node, the file is transferred to the local node in the *fopen()* procedure, after which the process writes to the local copy. If the file to be written is already on the local node, or it is a new file, then the process starts writing the file locally in the same way as any other system call. When the process finishes writing to a file that is originally stored in another node, FusionFS does not send the newly modified file back to its original node. Instead, it simply updates the metadata for that file. This saves the cost of transferring the file data over the network.

IV. FusionDex

FusionDex is designed with a share-nothing policy in mind. Such systems do not require coordinators, instead they operate by relying on dirrect collaboration between nodes. FusionDex is designed to work with a share-nothing distributed file system such as HDFS [13] or FusionFS [4], both of which store information in the form of files. Importantly, files are stored as a whole on individual nodes, that is, they are not split into blocks or chunks across nodes.

The general architecture of FusionDex is illustrated in Figure 1. Each compute node holds its own local storage comprised of local files and the associated index. In addition, each node is deployed with a daemon service, namely query server (*Q Server*), that listens to the request from query clients (*Q Client*). The query server is responsible for executing the query, distributing it to other nodes, and assembling a response to a client. Query clients may be implemented by users or applications to submit queries to FusionDex. It should be noted that a query server does not necessarily take the requests from the local query client. That is, any client can communicate with any server in the cluster in a N-to-N communication pattern. FusionDex implements a custom communication protocol that leverages the programming interfaces provided by FusionFS. This flexibility permits FusionDex to attain high performance through the seamless interconnection between the indexing service and the distributed file system, but also between the client and the server modules.

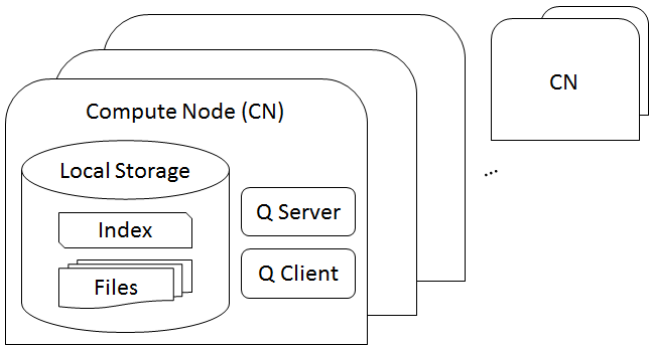

Figure 1. An architectural overview of FusionDex deployed to a share-nothing cluster.

A. Building Blocks

FusionDex leverages CLucene to enable efficient indexing and high performance, yet expressive query performance. It also leverages FusionFS's highly optimzed communication model to share data between nodes.

1) CLucene: local indexing: As we can see in Figure 1, from a single node's perspective all files are only indexed locally. While many popular indexing tools exist, we chose an open-source implementation as the local indexing module: CLucene [6]. CLucene is the C++ port of the popular and open source Apache Lucene [14]—a high-performance full-text search engine library, implemented in Java. CLucene is not as mature as Apache Lucene, but offers advantages including increased performance and implicit compatibility with the FusionFS distributed file system, that is also written in C++.

2) FDT: inter-node communication: In order to satisfy the design goals of the proposed solution, FusionFS has been extended to enable inter-node queries over the indexes and efficient communication between indexes. FusionFS has its own inter-node data transfer service, called Fusion Data Transfer (FDT), that migrates data chunks in the context of POSIX system calls. The flexibility of the FusionFS system allowed the easy integration of a new set of operation types to the FDT layer, such that an N-to-N index query is routed properly throughout the node topology and such that the

results are routed correctly back to the specific index node query server.

B. Index Creation, Removal, and Update

Since files are distributed among the nodes that comprise FusionFS, each node maintains the index of the files that reside on it. This is possible because FusionFS is designed to allow applications to carry out local reads and writes to files using a scratch location. FusionFS manages a global namespace for the path of the files, translating the absolute path of each file into a relative path on that node. FusionDex's Clucene index uses the translated paths to crawl local data and build the index for each file locally.

When a file is modified the index needs to be updated, in order to reflect the change. FusionDex relies on FusionFS's APIs to listen for notifications when files are modified. In FusionDex, an index update message is issued only when a file is closed, after the processing of the file has finished. Thus, opening or reading a file will not trigger an index update. This is possible as FusionFS tracks whether or not a file is modified. FusionDex uses the CLucene update function which automatically deletes the document from the index and re-adds a new version.

File de-indexing, or the process of index removal, occurs in two cases: when a file is removed from the distributed file system or when a file is moved from one node to another (usually in the case of remote writing). In either case the following de-indexing procedure is applied. FusionDex relies on FusionFS's managemnet of the removal process. When a file is to be removed, a FusionFS node sends a message to the node that "owns" that file. FusionDex extends this mechanism to add an additional message that instructs the node to de-index the file. This message is sent prior to the original removal message. Therefore, the file is removed from the remote node's index first. It is then followed by the removal of the actual file from the distributed file system. One additional step is needed in the case of file relocation: the file that would reside in the local node after the relocation process, will be added to the local node's index upon the completion of file migration and possibly write operation.

C. Server Protocols

Figure 2 illustrates the architecture of the query module on the server side. The query server is implemented with a thread pool, allowing concurrent requests to be satisfied in parallel without blocking. When a server receives a query request, it immediately queues the request. One of its worker threads then takes the request from the queue and performs the specific query task. Usually this task involves a search on the local index. Each worker thread keeps track of the index location, meaning that worker threads can perform queries in parallel without interfering with one another. One visible benefit of this approach is that the client waiting time for the first response is greatly reduced, since the query request is handled in an asynchronous fashion. The query server is responsible for distributing the query across the distributed system by directly contacting each of the other query servers. The initial query server is responsible for aggregating the results and returning a response to the requesting client.

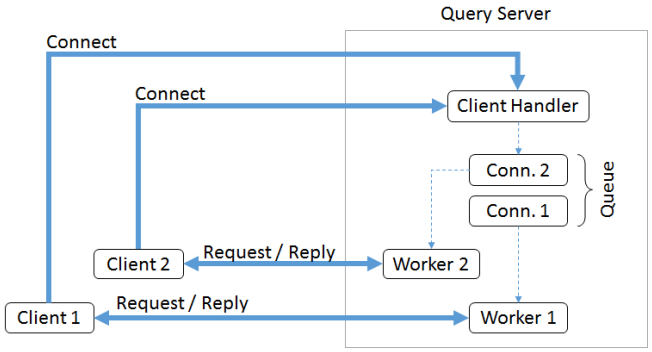

Figure 2. Protocols of Concurrent Requests on Query Server

D. Client Protocols

Figure 3 shows the internal organization of the query module from the client side. The query client provides an interface via which users (or applications) can submit queries to FusionDex. Each query client maintains a collection of worker threads. Each query client is aware of the topology and functional disposition of the members in the cluster, knowing the locations of the nodes where query servers modules are deployed. The membership information is initially read from a configuration file, the same configuration used to define the FusionFS distributed file system. When a user issues a query via a client, a particular worker thread picks a server node from the queue, establishes communication with the query server on that node, and sends the query to that server. Clients are independent in that they do not need to concern themselves with the locations of the indexes, thus the client modules can be deployed to any node as long as the membership information is available.

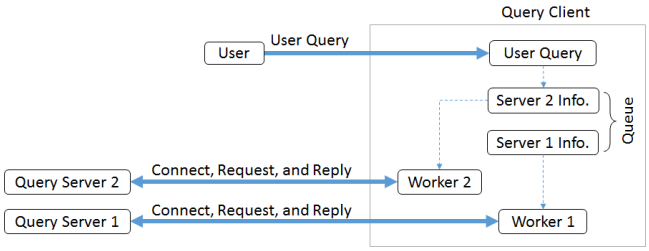

Figure 3. Protocols of Concurrent Requests on Query Client

V. Evaluation

A. Experiment Setup

The test environment, on which we eveluate FusionDex, is deployed on Amazon Web Services Elastic Compute

Cloud (EC2). In order to better investigate scalability and performance under realistic scenarios we conifgured two clusters with varying hardware and modified the number of nodes per cluster in each experiment. The first cluster (C1) is deployed on `m3.large` instances, each of which was equipped with 2 Intel Xeon E5 vCPUs, 7.5 GB of memory, and 32 GB SSDs. The second cluster (C2) was deployed on `m3.2xlarge` instances, each of which was equipped with 8 Intel Xeon E5 vCPUs, 30 GB of memory, and 160 GB SSDs. The evaluation process included: a performance comparison between FusionDex, GNU *grep* and Hadoop *grep* on the relatively lower-end cluster C1, and a more in depth analysis, comparing FusionDex and Cloudera Search, on the higher-end cluster C2.

In the abscene of data from a production distributed file system we developed a test dataset derived from the the Wikipedia dataset [15]. The test dataset was evenly distributed across the cluster's nodes. The evaluation process encompassed the issuing of 1,000 queries over 160 data chunks, each of which were 64 MB in size, all of these on each of the multiple systems mentioned. Therefore, the total amount of data was roughly 10 TB. Experiments were carried out in an incremental manner with respect to the number of nodes, the upper limit was 64 nodes.

B. Baseline Performance

To establish baseline performance of FusionDex we first evaluated its performance on a single node.

1) Index and write throughput:: First, the raw indexing throughput and rate of which files are written is shown in Figure 4 for increasing data size. The write throughput is calculated as the size of the file in MB, that can be pushed to FusionFS, per second, with indexing enabled. The index throughput is computed as the size of the file in MB, that can be indexed in FusionFS, per second. The figure illustrates that FusionDex can achive a write throughput of approximately 100 MB/s and an index throughput of approximately 1 MB/s irrespective of data size.

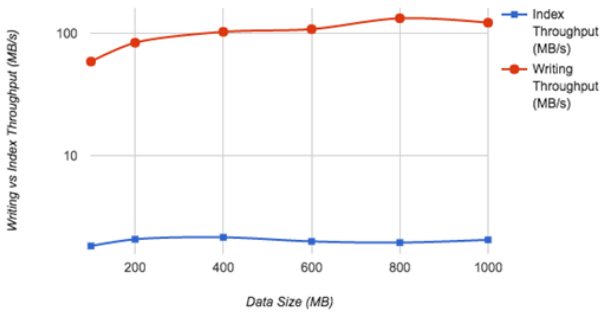

Figure 4. Indexing and write throughput on single node

2) Search latency:: The search latency is determined as the time it takes for the server to respond to a query request sent from a client. Figure 5 shows the search latency, with and without caching. Intuitively, the expectation is that as the file size increases the query latency also increases. However, these experiments also showed that even with relatively large data sizes of 100 MB that search latency barely exceeds 0.3 seconds. In order to further reduce latency, we enabled caching (this is useful in the cases when data is not frequently updated), and found that the latency could be significantly improved by an order of magnitude.

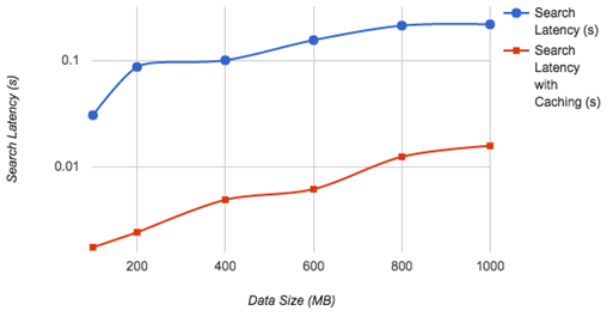

Figure 5. Query latency on single node

3) Query throughput:: Query throughput is calculated as the number of concurrent clients that the server can respond to per second. In carrying out this evaluation the data size was kept constant, at the arbitrary value of 1 GB. The number of clients that concurrently queried the server was increased by modifying the client configuration files. In this evaluation the number of worker threads on the server that handle incoming requests was also varied incrementally Figure 6 shows that the query throughput is poor when caching is disabled. This was expected since the search did not cache previous requests. On the other hand, when caching was enabled the throughput showed substantial improvement, as the number of clients increased.

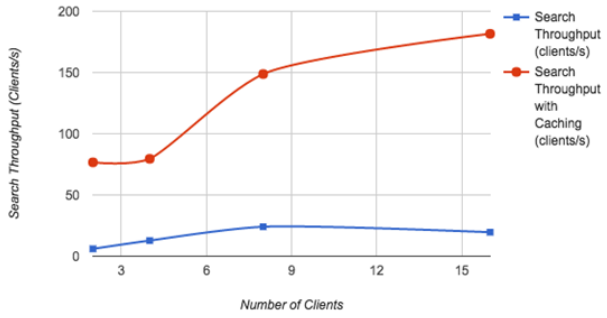

Figure 6. Query throughput on single node

C. Comparison to State-of-the-art Systems

In this section we compare FusionDex to several systems that aim to provide search capabilities on distributed file systems. More precisely, we aimed to explore the ability of these systems to scale to large distributed systems: from 4 to 64 nodes. We compare FusionDex with state-of-the-art solutions for querying distributed systems: *Linux grep*, *Hadoop grep*, and *Cloudera Search*.

Linux grep searches input files line by line, identifying matches to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default), or returns a user-specified format as described by the given parameters.

Hadoop grep [7] works differently from the default Linux grep, in that it does not display the complete matching line but only the matching string. Hadoop grep runs two MapReduce jobs in sequence. The first job counts how many times a matching string occurred in a given file and the second job sorts those matching strings by their frequency and stores the output in a single output file.

Cloudera Search relies on batch indexing documents using MapReduce jobs. Cloudera Search uses the *MapReduceIndexerTool* [8], 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 a live running service. The MapReduceIndexerTool does not operate on HDFS blocks as input splits, which means that when indexing a smaller number of large files, fewer nodes may be involved. Searches in Cloudera Search are conducted using the Apache Solr REST interface.

1) Indexing Throughput: Figure 7 shows the indexing throughput of FusionDex and CLoudera Search with an increasing number of nodes. The figure shows that FusionDex outperforms Cloudera Search except in a small cluster with 4 nodes. The reason for this behavior is due to FusionDex's indexing model, as compared to Cloudera's indexing batch tool. More precisely, when one file is indexed in FusionFS, the index is locked such that other files must wait. These locks have consequences especially when indexing a large number of files under extremely short time frames as in this case. Cloudera Search also implements index locking, however, rather than lock for individual files it instead locks once for the entire batch. Of course, this behavior also means that indexed documents are more quickly queryable in FusionDex than Cloudera Search. Nevertheless, as we increase the number of nodes FusionDex performs much better than Cloudera Search by a factor of at least 2.5. This is due to the decentralized approach employed by FusionDex, since the distributed indexing process amongst multiple nodes amortizes FusionDex's overhead. That is not the case with Cloudera Search, and therefore its indexing throughpt does not increase significanlty with the number of nodes.

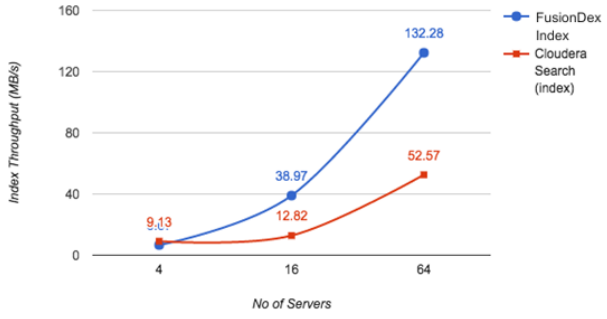

Figure 7. Indexing throughput on multiple nodes

2) Query Latency: Figure 8 shows the query latency of FusionDex and *Linux grep*. Since *Linux grep* does not exploit parallelism available in modern architectures, we focuse here on a single node deployment. It is interesting to note, that even under this single node deployment that FusionDex outperforms *Linux grep* by several orders of magnitudes. This is primarliy due to the multithreading strategy leveraged by the FusionDex implementation and its ability to satisfy the 1000 queries in parallel.

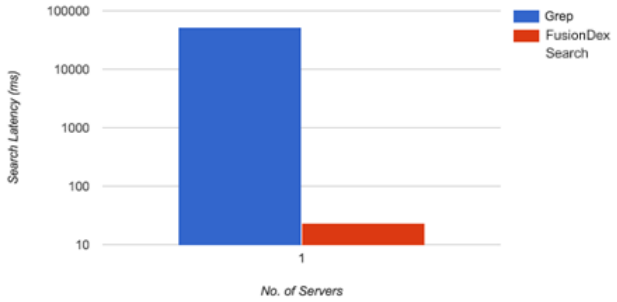

Figure 8. Query latency of Linux Grep and FusionDex

Figure 9 shows the query latency for Hadoop grep, FusionFS Search and Cloudera Search on cluster configurations of 4, 16 and 64 nodes. The figure shows that Hadoop grep has the worst performance of all search applications considered. This is because Hadoop grep counts how many times a matching string occurs and then sorts the matching strings. Cloudera Search and FusionDex outperform Hadoop Grep by several orders of magnitude. Cloudera Search with and without caching performs similarly for all cluster sizes with a difference of only 12 ms between all results. FusionDex performs significantly better than all other search applications, more than twice as fast as Cloudera Search for all configurations when using caching. Again, the improved

performance of FusionDex is due to its ability to distribute queries and perform operations in parallel.

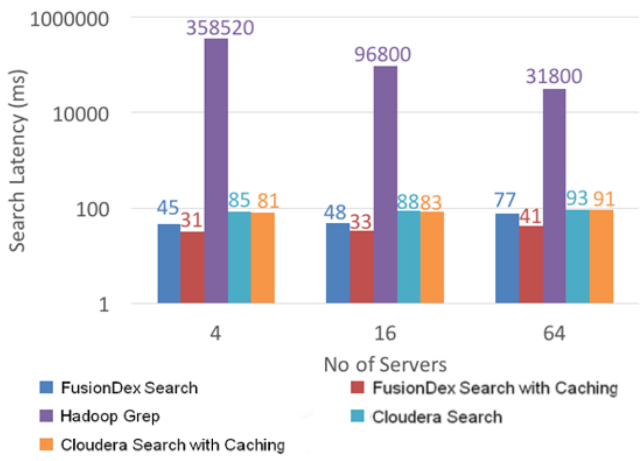

Figure 9. Query latency on multiple nodes

VI. Related Work

Indexing has been long studied in database systems [16], significant results being obtained and efficient solutions being developed. In such systems data is organized according to a pre-determined model. Adapting this approach to the context of unstructured data that is dispersed amongst multiple nodes, suddenly becomes more challenging. In the following paragraphs, we review different solutions and research projects, that tackle the problem of indexing in distributed systems.

One study follows the implementation of a B^+-tree-based indexing scheme [17], in which a structured overlay is constructed over the nodes. The overlay is kept up to date by local indexes in accordance with the data on each node. Clients are able to query the overlay using an adaptive selection algorithm. This solution is based on previous work on distributed b-trees [18], with modifications to address the needs of cloud computing environments.

Another study, uses the same model of building an overlay over the nodes found in a cluster, using R-trees and a custom routing protocol [19]. This approach leverages a query-conscious cost model, that selects beneficial local R-tree nodes for publishing to the overlay. This scheme was designed to work well in power-aware cloud computing environments (e.g., epiC [20]).

A different approach, named GLIMPSE [21], employs partial inverted indexes that consume smaller disk space than a full-text-inverted index. Geometric partitioning [22] also manipulates inverted indexes by splitting it according to updating time so to reduce the update overhead. Similarly, query-based partitioning [23] categorizes inverted indexes based on access and query frequency.

Recent prior work by Wu et. al. [24] have also looked at orthogonal issues in optimizing search performance, by reducing the network load in large-scale distributed systems in one-to-many and many-to-one communication patterns, commonly found in distributed search. We found that spanning trees are more efficient than direct one-to-many communication, allowing search queries to propagate to many distribtued indexes much faster with lower costs.

In comparison with these systems, FusionDex implements a completely distributed model across all participating nodes. It leverages properties of the distributed file system to optimize indexing, synchronization, and querying with the aim to remove performance bottlenecks and removing the single-point-of-failure.

VII. Summary

The advent of Big Data has resulted in a shift in paradigms and way of thinking about managing large quantities of data that are produced at a high velocity. There are many solutions and research initiatives for optomizing distributed storage and data processing. However, in the context of data indexing and querying in distributed systems there remain significant challenges with respect to efficiency, especially when unstructured data is increasingly common. In this paper we proposed a distributed indexing scheme for unstructured data dispersed across a distributed file system. The proposed model, trade-offs and design decisions were explored, focusing in particular on the importance of eliminating a single-point-of-failure and its impact on performance, scalability and utility. We presented, FusionDex, an implementation of our approach within the FusionFS distributed file system. Comparative analysis showed that FusionDex performs better than state-of-the-art tools and applications that tackle the same problem. It offers the ability to realize high query throughput while also being able to scale both horizontally and vertically.

Acknowledgment

This work was supported in part by the National Science Foundation under award OCI-1054974 (CAREER) and NSF-1461260 (REU).

References

[1] (2016, Nov.) International data corporation. [Online]. Available: https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf

[2] (2014, Sep.) Apache hadoop. [Online]. Available: http://hadoop.apache.org/

[3] (2016, Jul.) Myria. [Online]. Available: http://myria.cs.washington.edu

[4] 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," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1824–1837, 2016.

[5] ——, "A convergence of key-value storage systems from clouds to supercomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1824–1837, 2016.

[6] (2016, Oct.) Clucene. [Online]. Available: http://clucene.sourceforge.net

[7] (2016, Nov.) Hadoop grep. [Online]. Available: https://wiki.apache.org/hadoop/Grep

[8] (2016, Nov.) Mapreduce index tool. [Online]. Available: http://www.cloudera.com/documentation/archive/search/1-3-0/Cloudera-Search-User-Guide/csug_mapreduceindexertool.html

[9] I. Ijagbone and I. R. (advisor), "Scalable indexing and searching on distributed file systems," *Department of Computer Science, Illinois Institute of Technology, MS Thesis*, 2016.

[10] I. Ijagbone, S. Vinayagam, D. Pisanski, K. Brandstatter, D. Zhao, and I. Raicu, "Towards scalable searching of distributed file systems," *GCASR*, 2016.

[11] (2016, Jul.) Scidb. [Online]. Available: https://paradigm4.atlassian.net/wiki/display/ESD/SciDB+Documentation

[12] Y. Gu and R. L. Grossman, "Supporting configurable congestion control in data transport services," *ACM/IEEE Conference on Supercomputing*, 2005.

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

[14] (2016, Oct.) Apache lucene. [Online]. Available: https://lucene.apache.org

[15] (2016, Oct.) Wikipedia data download. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Database_download

[16] E. Bertino, B. C. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and D. Andronico, *Indexing techniques for advanced database systems*. Kluwer Academic Publishers, 2012.

[17] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient b-tree based indexing for cloud data processing," *Proc. VLDB Endow.*, vol. 3, no. 1-2, Sep. 2010.

[18] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, Aug 2008.

[19] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2010.

[20] Elastic Power-aware Data-intensive Cloud, "http://www.comp.nus.edu.sg/ epic," Accessed January 15, 2015.

[21] U. Manber and S. Wu, "Glimpse: A tool to search through entire file systems," in *USENIX Winter Technical Conference*, 1994.

[22] N. Lester, A. Moffat, and J. Zobel, "Fast on-line index construction by geometric partitioning," in *Proceedings of ACM International Conference on Information and Knowledge Management*, 2005.

[23] S. Mitra, M. Winslett, and W. W. Hsu, "Query-based partitioning of documents and indexes for information lifecycle management," in *Proceedings of ACM International Conference on Management of Data*, 2008.

[24] J. Wu, S. Chafle, and I. Raicu, "Optimizing search in unsharded large-scale distributed systems," *IEEE/ACM SuperComputing/SC*, 2016.