# XSearch: Distributed Information Retrieval in Large-Scale Storage Systems

Alexandru Iulian Orhean
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616
aorhean@hwk.iit.edu

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

Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616
iraicu@cs.iit.edu

***Abstract*—The pervasiveness and the continuous advancement of computer systems have determined a sudden upsurge of digital data, in terms of volume, velocity and variety, especially in the fields of science and engineering. This phenomena has resulted in a widespread adoption of parallel and distributed filesystems for efficiently storing and accessing data. As the filesystems and quantity of information increase in size, so it becomes more and more difficult to discover and locate particular relevant information amongst the already accessible data. While it is typical now for users to search for data on their personal computer or to discover information over the Internet at the click of a button, there is no such equivalent method for locating data on large parallel and distributed filesystems. This project argues the need for new methods to support information retrieval in the context of large-scale storage systems, and proposes the implementation of a scalable distributed indexing system. We, initially, investigate the increasing size and complexity of production parallel and distributed systems, in order to better outline the scale of the challenge at hand. We, then, explore the state-of-the-art in terms of frameworks, toolkits, libraries, data structures and algorithms that could be used as building blocks for an indexing and search system. Finally, we propose a scalable solution for achieving information retrieval in large-scale storage systems, aiming to integrate it with existing parallel and distributed filesystems.**

***Keywords* - parallel filesystems, distributed filesystems, information retrieval, concurrent search tree, concurrent hash map, distributed indexing;**

## I. Introduction

Information retrieval represents the process of finding and acquiring relevant materials, that usually contains unstructured information, from a large collection of information resources, satisfying the information need. In the context of computer systems, information is represented in digital form and is hierarchically organized in directories and files. Digital files represent the materials returned by an information retrieval engine, the content of the files being usually unstructured, with the exception of relational databases that store records under the form of tables and relation between the tables. The files representing the collection of information resources compose the filesystem. While the filesystem, that can either be part of the operating system or be implemented in user space, has its own internal data structures, that keep track of every bit of information in the computer and yield access interfaces, it does not facilitate direct search mechanisms. Accordingly, different constructs and applications have been proposed, that can deliver efficient and effective information retrieval, now being known as indexes and search engines.

Modern personal computers have operating systems that can install and run powerful search engines, allowing users to quickly retrieve files with a high degree of precision. But these are single node applications, that do not have the storage capabilities and the dynamics of multi-node architectures. Google has pioneered the research area of information retrieval and search engines [1], however its area of focus is web search over the Internet rather than filesystem search, with significantly different data and query models. The landscape of filesystems found in scientific computer systems is dominated by numerical data, high resolution images and specialized file formats, that are usually distributed across many storage nodes and accessed in parallel.

Supercomputers or High-Performance-Computing (HPC) systems represent a particular type of computer organization and design used in the fields of science and engineering, that have branched out from the fast and expensive mainframes, exploiting parallelism on many levels and lately growing in size in all dimensions. The storage arrangement found in HPC systems also benefit from parallelism, data units being split into smaller chunks that are spread throughout multiple storage devices, assuring high throughput access to the data. Example of filesystems that can be found in HPC systems include the IBM Spectrum Scale [2], Lustre [3] and the Parallel Virtual File System [4]. The quantity of information that these systems can store, process and generate, combined with the unstructured nature of the data, are inadvertently increasing the complexity of the information retrieval problem, ultimately relinquishing the awareness that the users had over scientific data. The problem is aggravated by the rush towards exascale systems [5], that are estimated to have exabytes of persistent storage and probably I/O throughput of PB/s, making real-time indexing and searching a problem that could undermine scientific progress and technological advancement.

While large-scale storage systems and the processing of large amounts of data have long been a focus of the HPC community, it is not uncommon now for large scale science to migrate to Cloud systems [6], transferring the information retrieval problem to Cloud infrastructures. In the area of

Cloud systems, the methods of organizing computers and the paradigm for storing and accessing data are different. Distributed filesystems scatter and replicate data across multiple nodes, offering parallel access and high availability. The distributed system middleware is in charge with task dissemination and the nodes play both the role of storage and compute units. The Hadoop Distributed File System [7] and Ceph [8] are well known distributed filesystems commonly used in Cloud systems, providing not only high performance but also fault tolerance. In enterprise Cloud systems, there are several projects and implementations that use the underlying middleware to achieve scalable and effective indexing and search, the most popular being: Apache Solr [9], Elasticsearch [10] and Cloudera Search [11]. While the tight integration with the middleware might hold certain advantages, the deficiencies are however inherited, thus, combined with the different architectural point of view of Cloud systems, it makes an interesting case for the information retrieval problem.

One of the most significant burdens faced by scientific communities is the lack of efficient tools that enable targeted search and exploration of large-scale filesystems. Some dedicated scientific communities have developed specialized catalogs and tools to aid discovery operations, while others have adopted standardized portable scientific data formats, such as HDF5 [12], but such approaches are limited in terms of generalizability and they are often cumbersome to use due to imposed schemes and the need for manual data wrangling. In the absence of better options, scientists and engineers fall back to the state-of-the-art methods for finding data in single, centralized systems. Linux traditional tools: *ls*, *cat* and *grep*, or *find*; are examples of methods of searching through single nodes systems, however in the context of a large-scale storage systems, the same tools can take days, months and ever years to return results. Preliminary work [13], [14] shows that building indexes and then searching them yields substantial performance gains, being able to search a 10GB file in mere milliseconds, while traditional tools have search latencies in the order of tens of seconds, for the same file. A filesystem that has a capacity of 10PB would imply search latencies in the order of years, clearly articulating need for newer indexing and search tools.

This project argues the need for a distributed search system, as a solution for the problem of efficiently and effectively retrieving information from parallel and distributed filesystems containing large mounts of data. The paper is organized as follows: In section II we analyze several production filesystems to determine typical requirements for indexing data in large filesystems. Section III continues with a general definition of the information retrieval problem in large-scale storage systems, taking into account the ascertained requirements and the potential implications of a distributed solution, all in the context of the scientific data space. The proposed solutions and research steps taken so far are presented in section IV. Experimental analysis and initial findings are discussed in section V. In section VI we review related work being done towards the same field, comparing it to this project's definition of the information retrieval problem. Lastly, we conclude in section VII, with a short description of future work and a synopsis of current discoveries and implications.

## II. Quantitative Analysis

Before embarking on this work, several large scale projects and institutional file systems, were analyzed. Specifically, file system dumps from several production file systems were obtained and the resulting information of their investigation characterizes usage across several dimensions. The resulting analysis aims to obtain both representative usage information as well as outer bounds from which distributed indexing and search system development can be guided. Table I highlights the size and complexity of a five very different storage systems. This table includes the data shared publicly on the Globus distributed storage cloud; Petrel: a data sharing service hosted at Argonne National Laboratory; a research computing scratch file system; a supercomputing center's file system; and an entire user file system from a large research institution.

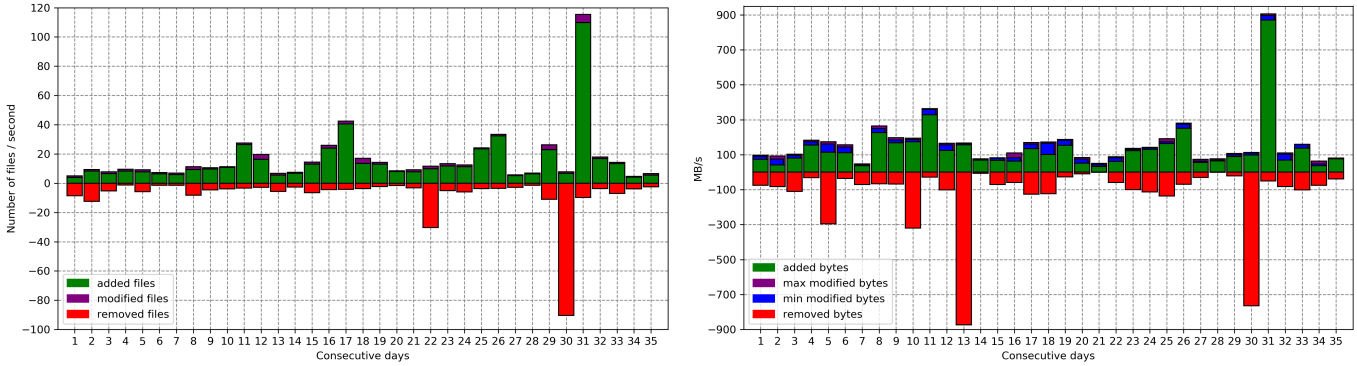
These results highlight the magnitude of the challenge when indexing data. Table I shows that even modest storage systems may have millions to tens of millions of files, cumulatively totaling hundreds of terabytes of data. The file systems surveyed range in size from several terabytes through to more than 6 petabytes and from 4.5 million files through to more than one billion. The number of users who own data on these file systems also ranges significantly. In several cases the number of users is small, however this means that the data likely to be searched upon by an individual user is large. For the supercomputing center and institution file system the number of users is much larger. Modification times on average are several hours, which means that crawling and re-indexing will need to occur regularly. Finally, directory sizes, on average are varied (between 20 and 212) which may limit our ability to optimize the index based on paths.

A more in depth analysis, depicted in figure 1, gives more insight in the daily dynamics of the large-scale filesystem found in the supercomputing center. Plot 1a shows the average I/O operations per second, or number of files per second, that were added, modified and removed throughout the period of a day, for a duration of 35 consecutive days. The IOPS are highly optimistic, since the filesystem dumps have a granularity of one day, meaning that the measured operations on any file accounts only for the last operation on the respective file. Nevertheless, the results show that at the end of one day there are more files added than modified, implying that index update due to file creation needs to be faster than the index update caused by file modification. With the exception of the occasional spikes, the average usage maintains at an average 20 IOPS, representing the metadata throughput that needs to be maintained in order to achieve real-time indexing.

Figure 1b captures the dynamics of the filesystem during the same 35 consecutive days, at the same one day granularity, but in terms of IO throughput. The analysis shows that there is more information added than modified, holding the claim that fast index update upon file creation is crucial, and it introduces

TABLE I
FILE SYSTEM CHARACTERISTICS.

| File system | Files | Size | Avg Depth | Users | Avg Mod Time | Avg Files per Dir |
|---|---|---|---|---|---|---|
| Globus Public | 4.5M | 30 TB | 8.9 | 39 | 67.7 hours | 77 |
| Research Computing Center | 55M | 80 TB | 6.0 | 543 | 5.8 hours | 26 |
| Petrel | 39M | 370 TB | 7.6 | 46 | 4.5 hours | 212 |
| Supercomputing Center | 861M | 6.3 PB | 12 | 16506 | - | 20 |
| Institution | 264M | 1.2 PB | - | 691 | - | - |




(a) Average added, modified and removed files per second.

(b) Average added, modified and removed megabytes per second.

Fig. 1. (a) Average IOPS throughout a day, consumed by a supercomputing center's filesystem, for a span of 35 consecutive days, in terms of average added, modified and removed files per second. (b) Average IO throughput throughout a day, consumed by a supercomputing center's filesystem, for a span of 35 consecutive days.

the idea of completely re-indexing a file when it is modified. Excluding the occasional spikes, the filesystem dumps show that the usage concentrates around a modest 200MB/s, which is far from the capabilities of a supercomputer, however implying that IO is done in bursts as is not uniform. Still, it sets the lower bounds for real-time indexing requirements. Given the number of files and the amount of data added, modified and removed, and the specific architecture of a supercomputer, it might be desirable to achieve full indexing of the filesystem at higher granularities, such as daily indexes, instead of real-time indexing. The same analysis results are going to aid in determining how efficient such a search engine should be.

## III. PROBLEM STATEMENT

This project resides at the intersection of two well established domains of computer science, namely: information retrieval and parallel and distributed systems. Information retrieval is the field that studies the processes and mechanisms involved with the discovery, location and retrieval of relevant materials from a collection of information resources, with the purpose of satisfying the information need. In the context of computer systems, the materials are represented by files, while the collection of information resources is comprised by the filesystem. Searching for some files by knowing some terms or some information contained in those files depicts the process of doing search in a computer, the application that implements this process being called a search engine. Research and engineering projects revolving around this area, focus on specific metrics for evaluating potential solutions and techniques, most of the metrics being concentrated on the quality of the search result. Precision represents one such metric, that captures how many of the returned documents are relevant to the entity that submitted the search query. Recall is another metric, that shows how many of the relevant documents are successfully retrieved after a search. F-score, R-precision and Discounted Cumulative Gain are examples of other performance metrics that construct evaluation tests for solutions to the information retrieval problem. Recent work in this area [15], [16], shows that probabilistic methods, natural language processing and inference techniques start becoming more popular, as they increase the quality and thus performance of search engines.

In the fields of compute clusters, supercomputers, grids and clouds, performance is appreciated through different lenses. Metrics that describe throughput, latency, scalability, availability, reliability and economics are significant in determining the efficiency and efficacy of such system. The inherent parallelism and distributed traits found in these architectures and configurations extend even to applications, single node implementations not having the theoretical computing power and storage capacity to deal with problems of large proportions that parallel and distributed systems specialize on solving. Thus, simple information retrieval is not enough and distributed strategies are needed. Due to the magnitude of the information resource space, shown in the previous section, metrics such as indexing latency, throughput and scalability, index size and distribution, search latency and throughput, become essential performance evaluation factors, contributing

the definition of the problem that this project undertakes: Distributed information retrieval deals with the problem of creating and coordinating processes and applications that discover, locate and retrieve relevant files, residing in parallel or distributed large-scale filesystems, either closely connected or distributed across the Internet, in a timely manner, while satisfying the information need.

In the context of the distributed information retrieval problem, the data resource collection space is different from the one found in single node personal computers or across the Internet. Numerical data, mathematical models and images form a higher proportion of the searchable collection of information, in supercomputer centers or research institutes, than text data, and while text data does still exist, it is usually organized in abstract formats, that make it easier for computer programs to interact with but does not extend the same convenience to human users. As result of this observation, the problem stated by this project inherits the challenges of achieving efficient and effective distributed indexing and search over large-scale storage systems, that are highly populated by non-uniformly structured numerical and/or image data.

The last part of the contextualization of the distributed information retrieval problem that this project undertakes, has to do with the environments where implementations of indexing and search solutions would run in. Parallel and distributed systems are composed of multiple inter-connected nodes, that appear as a single coherent system to the users, in which multiple users can deploy multiple, massive and scalable applications. In the case of supercomputers, physical computer resources are allocated to users and project for a specific amount of time, while in clouds the serviced resources are usually virtual and can reside on the same physical machines. An ideal solution for the distributed information retrieval problem, would take into account the possible application interference and network contention caused by the processes of data crawling, index update and search. The search engine needs to be aware of the existence of other applications running on the same system and must be able to accommodate to different resource access and movement patterns, while preserving the efficiency and efficacy made possible by the underlying architecture, over a landscape of data dominated by numerical and scientific formats, and maintaining scalability and fault tolerance qualities.

## IV. Approaches, Methods and Design

The complexity of the challenge at hand deems for a systematic approach to designing, implementing and evaluating solutions. Deep knowledge of how information retrieval system works, from the theoretical foundations to best practices, is vital to the successful outcome and operation of a scalable, fault-tolerant and efficient distributed information retrieval system, thus this project follows a bottom-up approach. Achieving high performance implies the necessity of looking into the building blocks of information retrieval systems, such as data structures, algorithms and existing software libraries. The magnitude of the problem was emphasized in the quantitative analysis, giving us an idea on the requirements of the final product. An early prototype of a distributed search engine called FusionDex [14], that was integrated in FusionFS [17], a user-level, completely decentralized distributed filesystems, showed the benefits of share-nothing distributed indexing and search, surpassing in performance industry-level solutions. FusionDex was built using an open-source library used for implementing local indexing and search, and was complemented with a custom communication mechanism, but was tightly integrated with the underlying filesystem. In order to squeeze all the performance of a distributed indexing system, the building blocks need to be investigated, in the hopes of discovering latent parallel or distributed characteristics that would eventually contribute to the communication mechanism. This investigation represents the first part of the project, with the detail that this time we are aiming for a solution that is independent from the filesystem.

The second part of the project takes a look at the properties of the search space and the formats that the information is presented. Natural language modeling techniques and inference engines work well with text data, as text data presented in documents usually reflects an ordering of ideas and meanings, making information retrieval more precise, but in the case of numerical data, different interpretation are needed. Mathematical representations, like matrices and vectors, and images, often hold intrinsic meaning and information, such as eigenvalues and distances, respectively, making the process of exploring numerical index representation a valuable and transformative research path. The higher understandings of such data is the subject of future work. In this project we focus in efficient numerical data indexing, efficient in term of throughput, latency and concurrency.

The third part of the project deals with concepts regarding parallelism and distributed architectures. There are multiple way to organize the steps of crawling, indexing and search, in order to achieve high throughput and low latencies, but the efficiency of the methods are dependent on the underlying hardware and multi-node organization. How indexes should be distributed, how nodes should communicate with each other and how updates can be achieved without degrading system-wide performance, are all challenges considered by this project. We define a general architecture of the distributed information retrieval problem and then emphasize the advantages and disadvantages of specialized version that work well on particular topologies.

### A. Building blocks of search engines

Figure 2 presents the anatomy of a basic search engine. The search engine has three main components in terms of functional units, and commonly multiple index data structures. The analyzer's job, also know as a 'stemmer' in literature, is to parse and deduce syntactic and semantic equivalence between the terms that compose the content of a document. Modern analyzers use language specific dictionaries and language modeling techniques in order to obtain a high degree of term comprehension. The simplest analyzer would just parse words out of a text document, or in the case of this project's

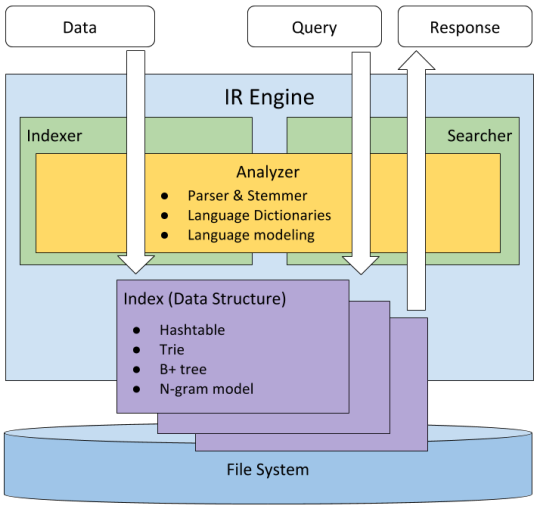



Fig. 2. Anatomy of an information retrieval engine.

search space, terms composed of alphanumeric characters. The analyzer provides constructs and methods to indexers and searchers. An indexer deals with the indexing process of the information retrieval engine. More precisely it opens a document for the analyzer, which in turn returns terms, that the indexer is able to process and add them to an index. The searcher also interacts with the parser, but it passes requests, that get transformed into sets of terms by the analyzer, and then used by the searcher to return index information. Another function of the searcher is to combine the result gathered from multiple indexes and return a cohesive answer to the user. Some search engines, who learn the relevance of returned files through the help of the user, might provide different interfaces through the searcher, that the user can use to interact with the system. The data structures known as indexes, transform the set of information organized in the form of documents and directories to a set of terms, defining the specification of access as a bijective function. Indexes are varied in design and purpose, and can be stored in main memory and/or on storage space, and were initially developed to speed up lookup in relational databases.

Anyone who wants to build an information retrieval application, doesn't actually have to implement all the components from scratch, because there exists several open-source libraries and toolkits that supply the building blocks for indexing and search. In our previous work, we used CLucene [18] to build the distributed information retrieval system, since it was written in C++ and represented a port of the popular Apache Lucene [19]. Since this project aims for high performance indexing and search, we searched for other information retrieval libraries that could be leveraged for building the distributed version of the solution. Benchmarking them incrementally, from single node to multi-node configurations, has the advantage of showing the performance of each alternative and also aids in understanding their handling, and thus we selected the following libraries for evaluation: Apache Lucene, CLucene, LucenePlusPlus [20], and Xapian. For the performance evaluation to be complete, we also provided two simple custom implementations in order to understand the overheads that the popular libraries add, when extended features are used. The custom implementation are based on variants of Tries and Hashmaps, and are discussed in detail in the following section.

Apache Lucene is a popular open source information retrieval library, written 100% in Java, that provides the building blocks for an indexing engine. It provides different analyzers or parser constructs that specialize in extracting terms from files that contain text, numbers or custom organized information (3D object descriptions). The analyzer usually gets plugged into a threaded index writer and reader, used for building the indexes and the latter for searching the indexes. The index writers can be tuned to be faster or more interactive, by specifying the merge policy and merge scheduler. During the benchmarks, the index writer has been tuned to build the index fast. The terms are grouped in segments, than can be build independently and merged, when a certain size is met, according to the set buffer limit of the index writer. Lucene contains fields for terms according to the information that they represent: text, number and custom formats. Overall the implementation of a search engine is fairly simple, but more importantly it removes the trouble of optimizing the process from the user through auto-thread management. This feature is pretty interesting, offering out-of-the-box high performance, in contrast to other solutions that require direct programmer involvement, and was discovered during single node single thread evaluations.

CLucene is a C++ port of Apache Lucene, presenting the same interfaces and programming constructs. Though the library itself is outdated and there are no signs of potential updates to the software. LucenePlusPlus is the spiritual successor of CLucene, being an up-to-date port of Apache Lucene, written in C++. It follows the same structure, having threads built in from the boost library [21]. In contrast to its Java counterpart, Luceneplusplus does not provide thread auto-tuning, and leaves the option of fine tuning to the user. Initial development was cumbersome, since it required modifying some base classes in order to provide multithreaded optimizations. While the index build classes and methods are very well optimized, in accordance to the provided filed data structures for the terms, the parsing component was performing slowly and proved to be the main bottleneck of the entire indexing engine. Thus the reading of the contents of the files have been done in parallel, boosting the performance of the LucenePlusPlus implementation easily, since the provided constructs are thread-safe by default.

### *B. Indexing data structures and the nature of information*

The design of the index data structure is strongly linked to the nature of the information that needs to be indexed. In the space of text information, Tries are a popular data structure, since it has the advantage of naturally representing terms and it also compresses the stored information. A Trie is a search tree in which each node represents a symbol from the alphabet,

and a path in the Trie represents a term. This construct has the distinct advantage of allowing prefix and suffix searches to be performed on the index, the results in such cases being represented by all the terms included in a sub-tree. In the initial benchmarks the custom implementation was a naive C++ Trie implementation that did not take into account space utilization and had not thread-safety built in. In search for a better candidate for a custom index data structure we approached a special variant of Tries, namely a CTrie. A CTrie [22]–[24] is a minimal concurrent lock-free hash mapped Trie, that was build as a way to attain high performance Tries in functional programming languages. While there are efficient implementations in Scala, Haskell and Rust, the C or C++ implementation proved to be challenging. The CTrie abstract data structure makes use of an atomic compare and swap operation and assumes no memory control. For languages that have a garbage collector, there is no problem in implementing such a data structure. On C/C++ on the other hand, memory management becomes problematic, due to many possible memory leaks, and performance penalties, that has determined the non-existence of a universally accepted implementation.

The first iteration of the data structure contains per node locks, that would allow concurrent update of the Trie, at the tree level, and sequential update at the node level. The compare and swap was also achieved through the locking mechanism, provided by the POSIX Threads library, under the form of Mutex data structures. Also memory management was done by the implemented function that would interact with the Trie. The performance penalties were observed after the first iteration of experiments done on text data sets. Another solution was required.

The second attempt to implement a CTrie envisioned the usage of atomic intrinsics provided by the GCC compiler. The GCC provides an atomic compare and swap operation at the pointer level, allowing for concurrent access and update of nodes in the Trie. The problem of managing memory would be solved through the use of a conservative garbage collector for C/C++, namely the Boehm-Demers-Weiser conservative garbage collector [25]. The design of the data structure proved to be a challenge, especially from the point of view, of efficient update operations. Thus the CTrie implementation has been postponed in favor of a Hashmap-based solution.

The Trie offers the advantage of space economy and allows for prefix and suffix search queries to be run on the indexing engine. In this project, the main concern was to provide boolean search capabilities, making the Trie loose its main advantages. A Hashmap would be the perfect candidate, allowing relative constant lookup operations, and through the use of atomic compare and swap operations, together with a garbage collector, it would achieve great performance, which it did. The pitfall of this data structure represented the case of many unique terms that were added in the case of metadata indexing, in which it achieved extremely low performance, mainly caused by small key size. Due to hardware restrains, the hash key size was of 16 bits, this accounting for many collision, especially in the case of many unique values.

The mentioned implementations did not take into account fundamental aspects of the data that is to be indexed. In the case of many number, as found in metadata information, such as file size, UNIX timestamps and IDs, using Tries or Hashmaps do not offer real advantages, since the numbers are big and unique. The third iteration of implementation has envisioned the use of Search Trees, more specifically of self-balancing search trees, such B+trees, for storing numbers. Of course with the lessons learned from implementing lock-free compare-and-swap-based data structures, the same techniques could yield substantial performance benefits. Another addition would be represented by the implementation of a uniform search tree for the absolute paths of the documents. The compression of information is implicit and advanced accessing methods are practicable. But these last concepts are the subject of future work.

### C. Distributed information retrieval models

The project initially started with a top-down approach to the distributed information retrieval problem, trying to formulate the challenges of such an endeavour, before completely understanding the limitations of the building blocks. The challenges are expressed in parallel and distributed application terms, since the problem of distributed indexing and search gets transformed in such an application. Communication becomes more relevant, as it impacts performance directly, and in the context of shared access across computers, it also interferes with other applications. Fast placement of indexes on local nodes does not always guarantee long term search performance, since the search queries need to be dispersed always. In the following paragraphs, several architectures are presented, coning at the end the general view of the information retrieval system and the underlying storage system.

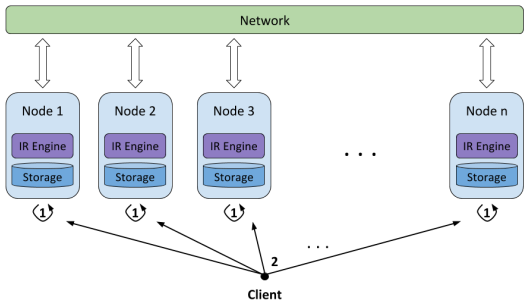



Fig. 3. Local indexing and distributed search approach.

Figure 3 shows an example of an architecture in which the indexes are stored on the same nodes with the actual data. In this scenario, the initial crawling and indexing step is done locally, in a system-wide parallel fashion. High indexing performance is expected, but the drawback come from the lack of a global ranking across the system. Thus any query needs to be forwarded to all the nodes, followed by a response from all the nodes, additional computation for building the global state and a final merges of results. There is a network component in

the search process, that has a one to all communication pattern, that might interfere with other applications for an increased query throughput.

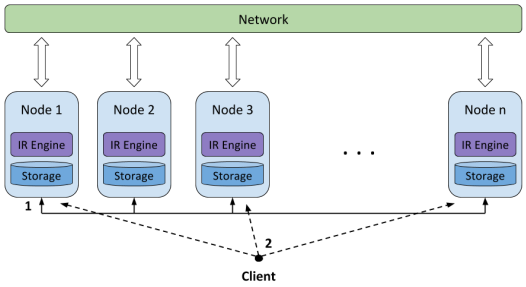



Fig. 4. Distributed indexing and confined search approach.

Figure 4 gives an alternative to the previous model. The indexes are build in a distributed fashion, using the network while building the indexes, but being able to achieve less network contention when searching. Global ranking and index information is kept in precise locations, being possibly implemented using a Zero-Hop Distributed Hash Table [26]. Search operations are being done in a confined area, knowing exactly where the indexes are and where to find the response.

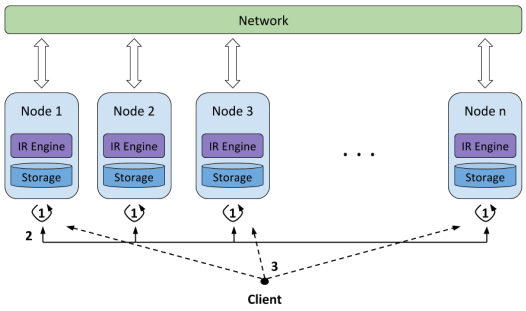



Fig. 5. Hybrid approach. Local indexing, index distribution and confined search.

Both of the previous cases present two opposite extremes to the design of an architecture for a distributed information retrieval system. Drawing from the pool of engineering wisdom, extremes are to be avoided, the balanced approach usually satisfying the common cases. Figure 5 shows a hybrid approach to indexing, distributing indexes and search. It combines the advantage of the first method, by quickly indexing files locally and the advantages of doing confined search. The intermediary step has to do with index rearrangement for cohesion purposes. On the short term, the advantages might not be clear, since the extra step is going to add its own performance penalties, but on the long term, the lack of network contention while doing search might prove more attractive. The difference between the distributed indexing and this approach is pointed out by contrast between organized dissemination of indexes and uncontrolled index dispersion.

All the three approaches assume that indexes are stored on the same nodes with storage nodes, and while this is the

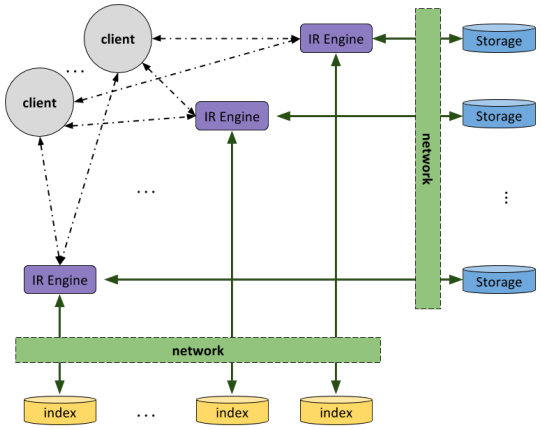



Fig. 6. General architecture of the distributed information retrieval system.

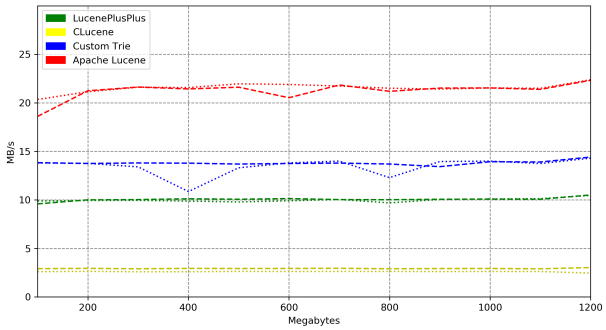
case, it does not illustrate the scenario in which this is not true. Computer clusters, such as Supercomputers, might have dedicated storage areas for indexes, thus all of the components from figure 6 depict the relation between information retrieval engines, storage area, index storage and network. If any two components reside on the same node, the network component becomes negligible. This representation gives a more general view over the architecture of a distributed information retrieval system, probably being able to formulate mathematical models for describing indexing and search performance.

## V. Experimental Analysis

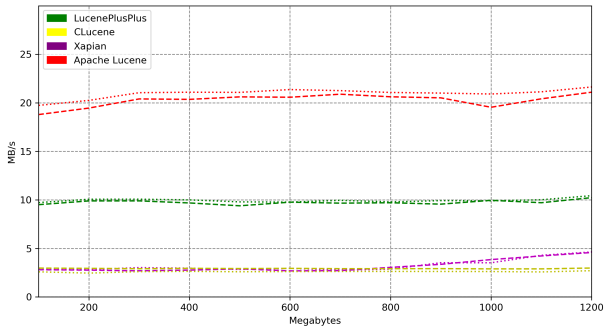
The experimental analysis was comprised of information retrieval applications built using the libraries and data structures discussed in the previous sections. In this paper we only have results for single node single-threaded and multi-threaded environments, multi-node experiments being scheduled for the future. All of the indexing and search applications were implemented in the same style. They would initially index a number of files at the beginning, doing batch indexing (in the case of metadata information retrieval the metadata was indexed). Afterwards they would read the terms and then execute search operations. These processes, along with the size of the indexes, are measured and printed to output. Several Bash scripts are used to iterate through the combinations of file sizes and storage sources.

The single node single-threaded experiments were run on a bare-metal storage node allocated in the Chameleon private cloud. The storage dedicated node had 2 Intel Xeon, each with 10 core (20 threads), 64GB of RAM, one 2TB HDD, that could perform 135 MB/s sequential read throughput, and one SATA 400GB SSD, with a sequential read throughput of 483 MB/s. Throughout the experiments the system page cache has been flushed and cleared before each run, simulating direct I/O access.

The single node multi-threaded experiments were run on a local setup with one computer, since Chameleon was being revamped and thus not completely functional. The computer had an AMD CPU with 8 cores, 16GB of RAM, one 1TB

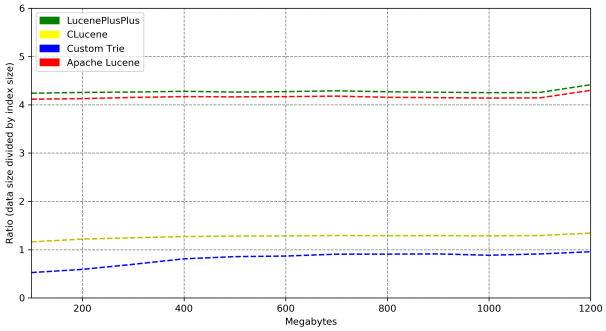



(a) Index stored in main memory.

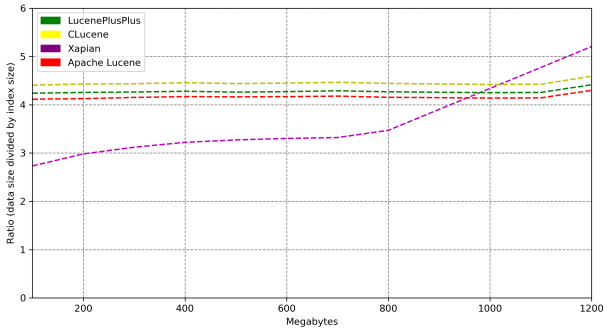



(b) Index stored on the persistent storage.

Fig. 7. Indexing throughput for text data.




(a) Index stored in main memory.




(b) Index stored on the persistent storage.

Fig. 8. Index size for text data.

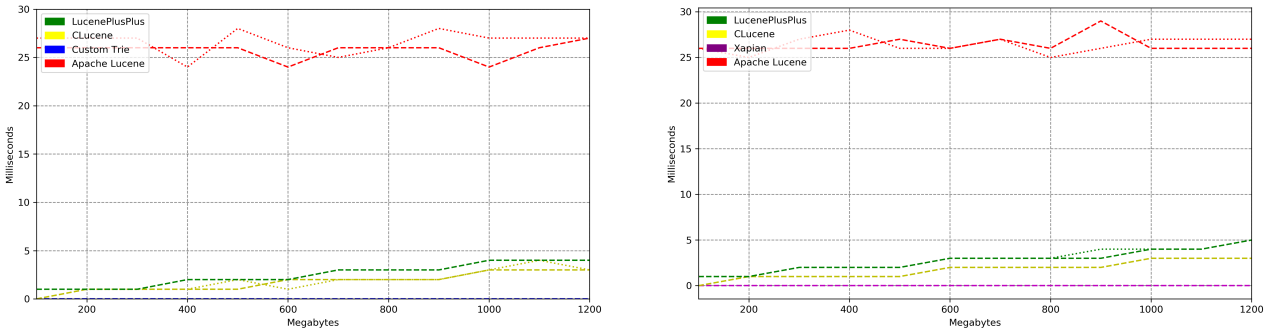
HDD, that could perform 112 MB/s sequential read throughput and 207 MB/s aggregated read throughput with 8 threads reading 8MB blocks, and one PCI 100GB SSD, that could perform 608 MB/s sequential read throughput and 876 MB/s aggregated read throughput with 8 threads reading 8MB blocks. Throughout the experiments the system page cache has been flushed and cleared before each run, simulating direct I/O access. The single node multi-threaded experiments were designed only for the case of indexes being stored in main memory and not to persistent storage area.

### *A. Single node single-threaded text data (the base line)*

Figure 7 shows the indexing throughput, measured as megabytes per second, of the simple single node single-threaded implementations, that indexed text data. As seen in the plots Apache Lucene holds the highest throughput, followed by the custom Trie. Later investigation revealed that Lucene is self-threading itself, being able to build and merge indexes in parallel, while the other libraries did not have these features. If the custom Trie had a throughput close to Lucene, without any improvements, it means that Lucene's overhead is truly affecting performance. Looking at the read throughput supported by the underlying storage, the best implementation is achieving between 20% and 25% of capable read performance, combining this with the finding from the quantitative analysis, further optimizations are needed to reach the required throughput.

In terms of index size, compared to the amount of text data indexed, all the libraries perform very good, compressing the information over 4 times. The custom Trie was naively implemented, creating the entire list of pointer to children for each non-leaf node, even if the path was not used. This determined in a close to exponential growth in size of the search tree, especially in the first five to six layers, causing the lack of data compression. This can be observed in figure 8. The interesting case of Xapian, which creates an artificial database, shows that with large enough data sets, it can compress information very efficiently.

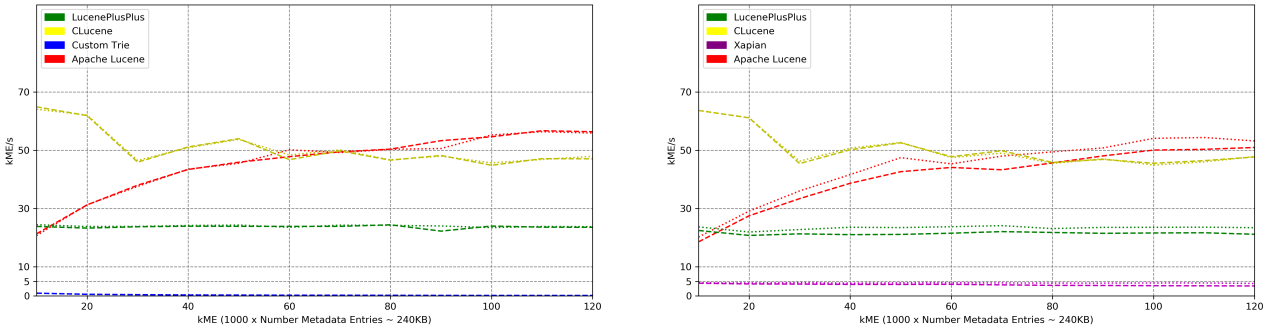
Evaluation of search latency over the build indexes of the selected implementation in the same scenarios as the above are shown in figure 9. The search tests were consisting of 100 search queries of randomly predefined words from the text data. The more lightweight the library the faster the search operations are. Apache Lucene has a complex query builder object that can create and work with widely complex search models, including numerical ranges. The experiments consisted only of boolean type search queries, in which the full term was given to the program, no other prefix or suffix queries being made.




(a) Index stored in main memory.

(b) Index stored on the persistent storage.

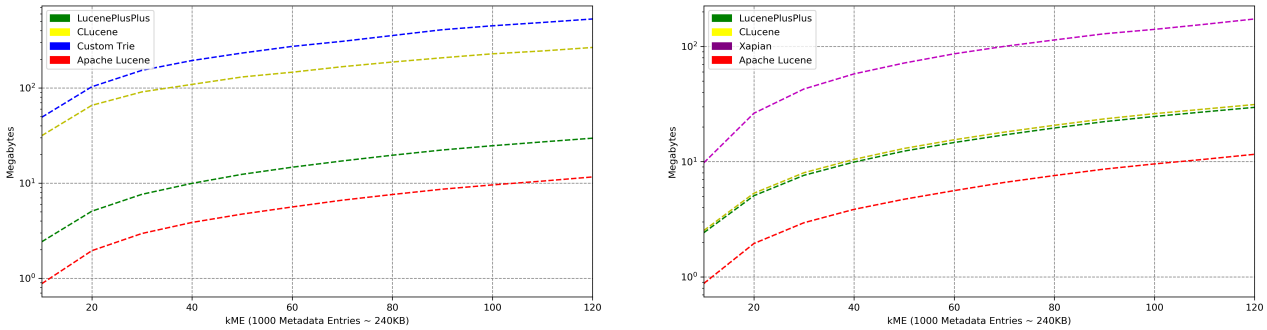
Fig. 9. Search latency for text data.




(a) Index stored in main memory.

(b) Index stored on the persistent storage.

Fig. 10. Indexing throughput for metadata.




(a) Index stored in main memory.

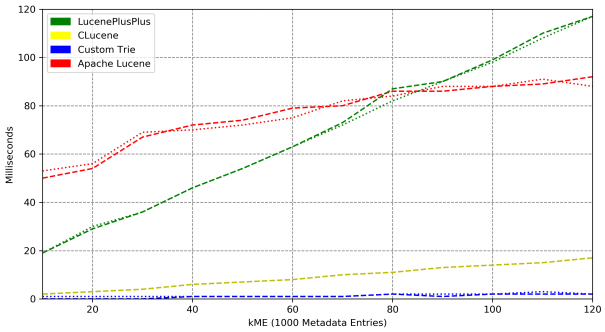
(b) Index stored on the persistent storage.

Fig. 11. Index size for metadata.

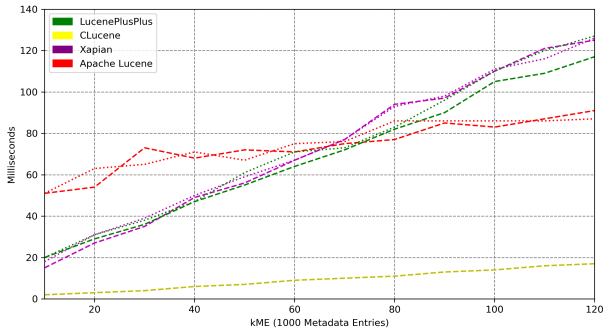
### *B. Single node single-threaded metadata (the base line)*

The experiments that reflect indexing and search performance on file metadata data sets, show distinctively different results for the same implementation strategies. Figure 10 depicts the indexing throughput, measured as thousands of metadata entries processed per second, which would be equivalent to IOPS, and the drawbacks of a naive Trie. Taking into account the results from the quantitative analysis, the selected popular libraries, with the exception of Xapian, can sustain the required IOPS. This implies that metadata search and indexing can easily be achieved in a single node, which is the case with most parallel and distributed file systems.

Index size shows even more interesting results. While the popular indexing libraries can handle well the quantity of metadata indexed and can store it efficiently, Xapian and the custom Trie, are having a hard time compressing the information. The indexed are actually bigger than the data that was indexed. In the case of the custom Trie, on of the reasons was the naive implementation that determined a close to exponential growth in size of the search tree, especially




(a) Index stored in main memory.




(b) Index stored on the persistent storage.

Fig. 12. Search latency for metadata.

since now the terms were composed of large unique numbers and paths. This can be seen in figure 11.

Finally, search performance of the implementations shows that current libraries encounter losses in efficiency when searching metadata information-based indexes. Figure 12 shows the almost linear increase in search time, that is executed on the stored indexes. The only implementation that showed the desired performance when doing search, was, despite the performance losses regarding indexing and index size, the custom Trie implementation.

### C. Single node multi-threaded text data

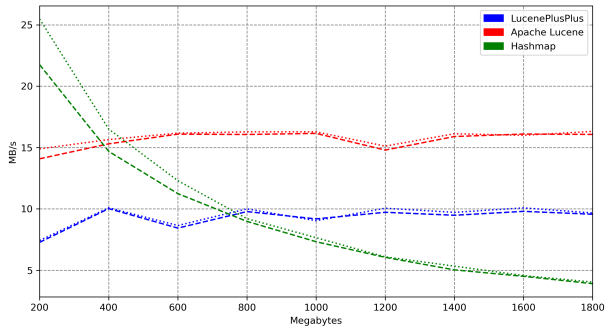



Fig. 13. Indexing throughput for text data.

Figure 13 shows the evolution of the indexing throughput of Lucene, LucenePlusPlus and the Hashmap implementation for the text data set. While the Hashmap has a great advantage at the beginning, but due to many collisions and the fact that the data structure is being updated concurrently its throughput decreases significantly. Lucene achieves its performance through auto-tuning of the thread capabilities, while LucenePlusPlus does manage to get a stable throughput but still lower than Lucene.

Figure 14 shows the data versus index sizes ratio, values larger than one meaning high compression of information, while values lower than one denote poor information index capabilities. The Hashmap behaves the poorest, while a Trie would have inherently compressed the information indexed.

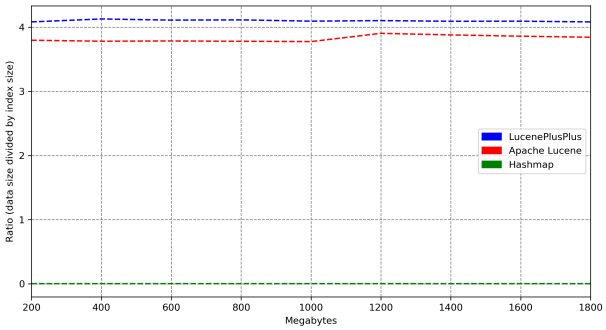



Fig. 14. Index size for text data.

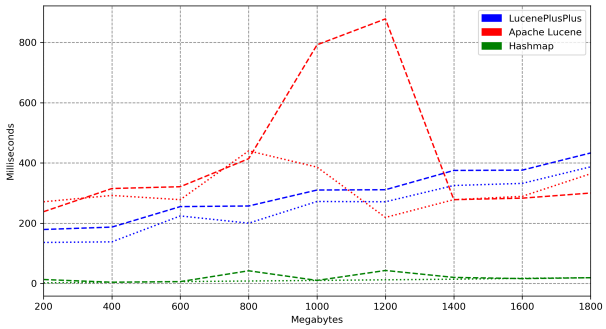



Fig. 15. Search latency for text data.

Figure 15 shows that search on a Hashmap is faster, due to the fact that it is more lightweight than a Trie or segment. Lucene and LucenePlusPlus perform relatively better. Search in this scenario was performed as queries of exact terms picked randomly from the data set. The number of queries per iteration was 1000, and the terms existed in the searched space.

### D. Single node multi-threaded metadata

On the metadata sets LucenePlusPlus outperforms Lucene significantly, as shown in Figure 16. Here the Hashmap obtained really poor execution times, from 24 minutes with

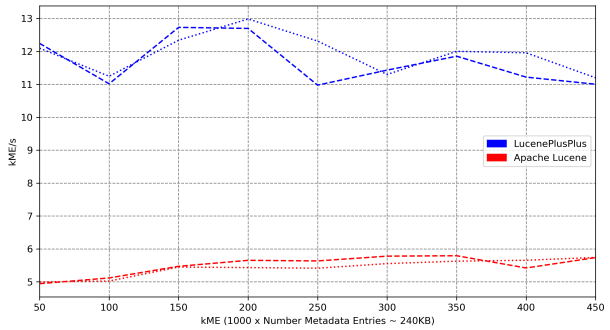



Fig. 16. Indexing throughput for metadata.

50 kME, to 4.5 hours with 100 kME, to 12 hours with 150 kME. This is caused by the short hash key and the fact that the Hashmap encounters many unique terms, as found in the meta information of files, that generate many collisions, that get stored in a dynamic array, making the lookup operation more costly.

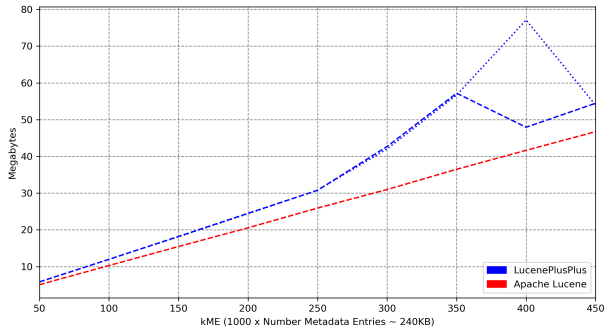



Fig. 17. Index size for metadata.

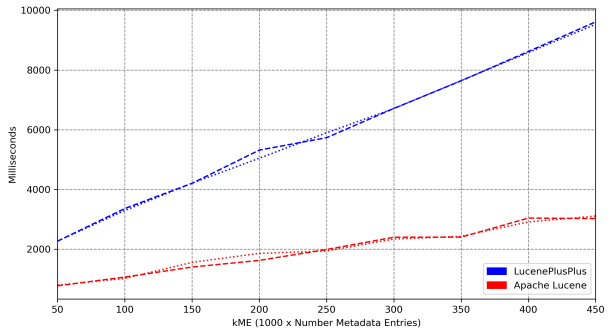



Fig. 18. Search latency for metadata.

Figure 17 shows that metadata has more unique values, thus the size of the index increases almost linearly. It can be observed that with metadata search time increases with more metadata entries significantly, as seen in Figure 18. The increase in latency is staggering, since the number of metadata entries is just a fraction of the 1 billion files storage system, showing the lack in capability to satisfy the requirement of large-scale file systems.

## VI. Related Work

Elasticsearch [10] is the most commonly deployed system for indexing text-based documents in a distributed system. The search engine is built on Apache's Lucene, and therefore provides flexible free-text search capabilities. It is designed to be scalable via a sharded model in which the search index can be split across many nodes for performance and availablity. As data sizes increase, additional nodes can be added to the cluster and the management layer will adjust the shards accordingly. This system is also very versatile, offering out of the box monitoring features, a pluggable architecture for integration with different environments and an API for custom front-end applications. While Elasticsearch has been well adopted, its primarily aimed at indexing arbitrary documents (e.g., products on an e-commerce site) it is not optimized for the types of data or queries commonly seen on a parallel file system. Furthermore, it is designed to be operated as a stand alone cluster, rather than integrated with the nodes of a file system.

Another solution that integrates a database in the architecture of an existing parallel file system is Lustre's Robinhood Policy Engine [27]. Its initial purpose was to simplify common administrative tasks on a large Lustre [3] deployment, and also increase performance, from the perspective of managing file lifetime, scheduling data copies and generating overall file system statistics. But the success of this application has allowed it to be extended to support other POSIX file systems and to offer many search possibilities through a mirrored database of collected system information. Different criteria can be used to explore the entries of the database and the solution allows custom statistics to be extracted at fast speed and high effectiveness.

Integrated databases offer a flexible platform for storing and querying metadata and data from distributed file systems. However, they require user-management to define schemas and map metadata to these schemas. Furthermore, they are designed for structured (rather than free-text) queries and they cannot be easily integrated into a file system model. However, there is much we can learn from decades of database research for example using search trees to distribute and parallelize the queries [28]. Evaluated in a large scale unsharded distributed system, the performance of this approach to do indexes search has surpassed classical methods and modern centralized models, obtaining even better scalability.

## VII. Conclusion

Modern large-scale parallel and distributed file systems, specifically in scientific communities and engineering projects, do not present any means to accomplish efficient and effective information retrieval. Users can explicitly read and write specific data by using file names, but when file systems contain millions to billions of files in a complex and deep directory hierarchy, finding data is analogous to looking for a needle in

a hay stack. Our work has shown the extent of the dynamics of such systems and deficiencies of the state-of-the-art search engine building blocks. Indexing on large amounts of text data has been shown not to achieve the required throughput in order to sustain real-time indexing and even daily indexing. While text data indexes are efficient and compress the information stored, indexes built with the same common index data structures show deficiencies when dealing with file metadata. Search latency over metadata indexes also show fast decreasing performance for increasing amount of data. Lastly, it is intriguing that modern tools, even when multi-threaded can achieve between 20% - 25% of the persistent storage throughput, indicating that indexing becomes more a computing problem than storage related problem. Future work consists of further development of the single node multi-threaded solutions, the aim being the optimization of the custom data structures so that close to optimal performance can be achieved. Subsequently, we will mode the proposed solution to a multi-node architecture, continuing research on the communication aspect and scalability of the distributed information retrieval system. The potential impacts of our work are likely to be transformative as it will make vast storage systems implicitly searchable, departing from traditional brute force data searching or the explicit (a priori) creation of specialized catalogs.

## ACKNOWLEDGMENT

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

## REFERENCES

[1] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[2] D. Quintero, L. Bolinches, P. Chaudhary, W. Davis, S. Duersch, C. H. Fachim, A. Socoliuc, O. Weiser *et al.*, *IBM Spectrum Scale (formerly GPFS)*. IBM Redbooks, 2017.

[3] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.

[4] R. B. Ross, R. Thakur *et al.*, "Pvfs: A parallel file system for linux clusters," in *Proceedings of the 4th annual Linux showcase and conference*, 2000, pp. 391–430.

[5] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill *et al.*, "Exascale software study: Software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[6] D. Lifka, I. Foster, S. Mehringer, M. Parashar, P. Redfern, C. Stewart, and S. Tuecke, "Xsede cloud survey report," *Technical report, National Science Foundation, USA, Tech. Rep.*, 2013.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 2010, pp. 1–10.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[9] A. Solr, "Apache solr," *Apache Lucene*, 2011.

[10] B. Elasticsearch, "Elastic search," 2015.

[11] *Cloudera Search*. 1001 Page Mill Road, Bldg3, PaloAlto, CA94304: Cloudera, Inc., 2017.

[12] M. Folk, A. Cheng, and K. Yates, "Hdf5: A file format and i/o library for high performance computing applications," in *Proceedings of supercomputing*, vol. 99, 1999, pp. 5–33.

[13] I. Ijagbone, *Scalable indexing and searching on distributed file systems*. Illinois Institute of Technology, 2016.

[14] A. I. Orhean, I. Ijagbone, I. Raicu, K. Chard, and D. Zhao, "Toward scalable indexing and search on distributed and unstructured data," in *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE, 2017, pp. 31–38.

[15] "The xapian project," 2018, https://xapian.org/.

[16] W. B. Croft, J. Callan, J. Allan, C. Zhai, D. Fisher, T. Avrahami, T. Strohman, D. Metzler, P. Ogilvie, M. Hoy *et al.*, "The lemur project," *Center for Intelligent Information Retrieval, Computer Science Department, University of Massachusetts Amherst*, vol. 140, 2012.

[17] 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.

[18] (2018) Clucene. [Online]. Available: http://clucene.sourceforge.net

[19] (2018) Apache lucene. [Online]. Available: https://lucene.apache.org

[20] "Luceneplusplus," 2018, https://github.com/luceneplusplus/LucenePlusPlus.

[21] "Boost," 2017, http://www.boost.org/.

[22] A. Prokopec, P. Bagwell, and M. Odersky, "Cache-aware lock-free concurrent hash tries," *arXiv preprint arXiv:1709.06056*, 2017.

[23] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Acm Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 151–160.

[24] A. Prokopec, P. Bagwell, and M. Odersky, "Lock-free resizeable concurrent tries." in *LCPC*. Springer, 2011, pp. 156–170.

[25] "Boehm-demers-weiser conservative garbage collector," 2017, http://www.hboehm.info/gc/.

[26] 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 *Parallel & distributed processing (IPDPS), 2013 IEEE 27th international symposium on*. IEEE, 2013, pp. 775–787.

[27] T. Leibovici, "Taking back control of hpc file systems with robinhood policy engine," *arXiv preprint arXiv:1505.01448*, 2015.

[28] S. Chafle, J. Wu, I. Raicu, and K. Chard, "Optimizing search in unsharded largescale distributed systems."