SCALABLE INDEXING AND SEARCH IN HIGH-END COMPUTING SYSTEMS

BY

ALEXANDRU IULIAN ORHEAN

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

Approved ______________________
Adviser

Chicago, Illinois
August 2023

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to my adviser, my professor and the chair of my committee Dr. Ioan Raicu for his invaluable patience and feedback and constant guidance and support, without which this endeavour would not have been possible. I would also like to express my deepest appreciation to Dr. Lavanya Ramakrishnan for her support and encouragement and Dr. Kyle Chard for his inspiration and enthusiasm that have been crucial in shaping my PhD, and I cannot thank them enough. Additionally, I would like to thank my defense committee (Dr. Ioan Raicu, Dr. Boris Glavic, Dr. Kyle Hale, Dr. Jia Wang, Dr. Lavanya Ramakrishnan and Dr. Kyle Chard) who generously provided their knowledge and expertise.

Furthermore I would like to thank all of the collaborators and colleagues for their honest advice and feedback, and for their symphatetic moral support and understanding. Lastly I would like to thank my mom (Violeta Orhean), my dad (Mircea Orhean) and my girlfriend (Kristen Efantis) for their persistent support, unyielding encouragement and enduring belief in me, and God for giving me hope and inspiration in my work and in my life.

AUTHORSHIP STATEMENT

I, Alexandru Iulian Orhean, attest that the work in this thesis is substantially my own.

In accordance with the disciplinary norm of Computer Science (see IIT Faculty Handbook, Appendix S), the following collaborations occurred in the thesis:

My adviser, Dr. Ioan Raicu, contributed to the design of all experiments and guided the interpretation of the data as is the norm for a PhD supervisor.

Dr. Ioan Raicu, Dr. Kyle Chard, Dr. Boris Glavic and Dr. Lavanya Ramakrishnan contributed to and guided the research and development of the core ideas behind this thesis. Dr. Ioan Raicu and Dr. Kyle Chard participated in polishing the research papers behind this work.

Dr. Ioan Raicu, Dr. Kyle Chard, Dr. Dongfang Zhao and Eng. Itua Ijgabone guided and contributed to the "Toward scalable indexing and search on distributed and unstructured data" research paper, that is included in chapter 2 of this thesis. Dr. Ioan Raicu, Dr. Kyle Chard, Dr. Lavanya Ramakrishnan and Dr. Anna Giannakou assisted with the development, implementation and evaluation of the ideas behind the "SCANNS: Towards Scalable and Concurrent Data Indexing and Searching in High-End Computing System" research paper, that is incorporated in chapter 3. Dr. Ioan Raicu, Dr. Lavanya Ramakrishnan, Dr. Anna Giannakou, Dr Katie Antypas and Eng. Matt Henderson contributed to and guided the "Evaluation of a Scientific Data Search Infrastructure" research paper, that is included in chapter 5. Lastly, Dr. Ioan Raicu, Dr. Boris Glavic, Dr. Kyle Chard, Dr. Lavanya Ramakrishnan and Dr. Anna Giannakou assisted with the development, implementation and evaluation of the ideas behind "SCIPIS: Scalable and Concurrent Persistent Indexing and Search in High-End Computing", that is included in chapter 4 of this thesis.

Dr. Poornima Nookala and PhD student Lan Nguyen contributed to this work through brainstorming sessions and discussions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Rapid advances in digital sensors, networks, storage, and computation coupled with decreasing costs is leading to the creation of huge collections of data. Increasing data volumes, particularly in science and engineering, has resulted in the widespread adoption of parallel and distributed file systems for storing and accessing data efficiently. However, as file system sizes and the amount of data "owned" by users grows, it is increasingly difficult to discover and locate data amongst the petabytes of data. While much research effort has focused on methods to efficiently store and process data, there has been relatively little focus on methods to efficiently explore, index, and search data using the same high-performance storage and compute systems. Users of large file systems either invest significant resources to implement specialized data catalogs for accessing and searching data, or resort to software tools that were not designed to exploit modern hardware. While it is now trivial to quickly discover websites from the billions of websites accessible on the Internet, it remains surprisingly difficult for users to search for data on large-scale storage systems.

We initially explored the prospect of using existing search engine building blocks (e.g. CLucene) to integrate search in a high-performance distributed file system (e.g. FusionFS), by proposing and building the FusionDex system, a distributed indexing and query model for unstructured data. We found indexing performance to be orders of magnitude slower than theoretical speeds we could achieve in raw storage input and output, and sought to investigate a new clean-slate design for high-performance indexing and search.

We proposed the SCANNS indexing framework to address the problem of efficiently indexing data in high-end systems, characterized by many-core architectures, with multiple NUMA nodes and multiple PCIe NVMe storage devices. We designed SCANNS as a single-node framework that can be used as a building block for implementing high-performance indexed search engines, where the software architecture of the framework is scalable by design. The indexing pipeline is exposed and allows easy modification and tuning, enabling SCANNS to saturate storage, memory and compute resources on different hardware. The proposed indexing framework uses a novel tokenizer and inverted index design to achieve high performance improvement both in terms of indexing and in terms of search latency.

Given the large amounts and the variety of data found in scientific large-scale file systems, it stands to reason to try to bridge the gap between various data representations and to build and provide a more uniform search space. ScienceSearch is a search infrastructure for scientific data that uses machine learning to automate the creation of metadata tags from different data sources, such as published papers, proposals, images and file system structure. ScienceSearch is a production system that is deployed on a container service platform at NERSC and provides search over data obtained from NCEM. We conducted a performance evaluation of the ScienceSearch infrastructure focusing on scalability trends in order to better understand the implications of performing search over an index built from the generated tags.

Drawing from the insights gained from SCANNS and the performance evaluation of ScienceSearch, we explored the problems of efficiently building and searching persistent indexes that do not fit into main memory. The SCIPIS framework builds on top of SCANNS and further optimizes the inverted index design and indexing pipeline, by exposing new tuning parameters that allows the user to further adapt the index to the characteristics of the input data. The proposed framework allows the user to quickly build a persistent index and to efficiently run TFIDF queries over the built index. We evaluated SCIPIS over three kinds of datasets (logs, scientific data, and file system metadata) and showed that it achieves high indexing and search performance and good scalability across all datasets.

CHAPTER 1
INTRODUCTION

Rapid advances in digital sensors, networks, storage, and computation coupled with decreasing costs is leading to the creation of huge collections of data—commonly referred to as “Big Data.” These data have the potential to enable new insights and discoveries that can change the way business, science, and government deliver services to their consumers and can impact society as a whole. Increasing data volumes, particularly in science and engineering, has led to the widespread deployment of parallel and distributed file systems for storing and accessing data efficiently. However, as file system sizes and the amount of data “owned” by users grow, it is increasingly difficult to discover and locate information amongst the petabytes of accessible data, with exabytes of storage capacity on the horizon. While much research effort has focused on the methods to efficiently store and process data, there has been relatively little focus on methods to efficiently explore, index, and search data using the same high-performance storage and compute systems.

One of the most significant burdens faced by the scientific community is the lack of efficient tools that enable targeted search and exploration of large file systems. While it is now trivial to quickly find websites from the approximately 2 billion websites in existence, it is remarkably difficult for researchers to search across their scientific data stored on large-scale storage systems. Google has pioneered much of the information retrieval and search engine research; however, its area of focus is large-scale distributed search over web data rather than searching over scientific data stored in high-performance file systems—two areas with significantly different data, storage, processing, and query models.

In the enterprise search domain there are several tools that are commonly used to enable search, such as Apache Lucene [1], Apache Solr [2], and ElasticSearch [3].

According to surveys from both academia [4] and industry [5], Apache Lucene is the most popular tool used to implement search engines. These surveys also show that the top three search tools are either Apache Lucene or services that build on Apache Lucene (Apache Solr and ElasticSearch), thus, Apache Lucene represents 69–73% of the enterprise search market. Apache Lucene was originally implemented in 1999 and was designed for commodity hardware that consisted primarily of single-core and single CPU systems, with a single hard disk, and for full-text indexing and search, and they are not designed to make use of the advanced features of HPC systems and modern hardware. Instead, they achieve scalability via distribution and index sharding and often rely on tight coupling with distributed file system, such as the Hadoop File System [6], which are not supported on HPC systems. Apache projects are often implemented in Java, which also has not garnered wide adoption in HPC systems. Other existing works from HPC have aimed to tackle this problem, however they typically have focused on indexing and search of metadata [7, 8] as opposed to the scientific data itself. Oftentimes, solutions are being built that are applications specific, leading to inefficiencies as the community continuously "reinvents the wheel."

In the absence of better options, scientists often fall back to the state of the art methods for finding data in single, centralized systems. That is, they use traditional Linux tools: `ls` and `grep`, or `find`. However, these tools are not designed for large file systems. For example, listing all files (a common operation when searching for a specific file name) in a production parallel file system commonly found on large computing clusters could take many weeks to complete (given that it contains billions of files and metadata performance is typically limited to thousands of operations per second). Further, this does not consider the time to read the data itself, a task that could compound the search time by several orders of magnitude. Searching through a 10 petabyte file system (the size of the persistent storage system on the Theta supercomputer at Argonne) by reading through the entire data could take over 3 years at a modest 100MB/sec read rate, a typical performance level if a user were to issue a search request from a login node.

Existing tools are not suitable in the context of large-scale storage systems. We believe that tools which allow data and metadata stored on today's HPC storage systems (e.g. Lustre [9], GPFS [10], Ceph [11]), many of which are accessible via Globus [12], should be index-able and search-able in a transparent effortless way while not impacting the performance of the storage system for I/O intensive workloads.

Scientific data comes in many flavors, from free-text data (e.g., logs in text files), to numerical data (e.g., matrices in HDF5, time-series data), to image data (e.g., medical images in DICOM format), to video data (e.g., videos from biology studying organisms behavior). Each data type might need specialized indexing and search methods, further complicating an already difficult problem at scale. Due to the sheer amount of data found in today's HPC systems, any solution must be distributed, be parallel in nature, and exploit recent advancements in non-volatile memory.

1.1 Indexing and Search Problem Space

This work aims at addressing the general problem of efficient and effective indexing and search in large-scale scientific file systems, and in this thesis we present and define the challenges and bottlenecks of indexing and searching large amounts of data on high-end computing systems, and include and describe valuable insights and novel techniques for designing salable index-based search engines. We first present and define, in the remainder of this chapter, the problem space and the core concepts behind search engines and information retrieval. We include motivating uses cases from science and engineering as well as cyberinfrastructure providers. In Chapter 2 we include the early work, in which we initially explored the prospects of using existing information retrieval libraries and search engine blocks to enable search in a high-performance distributed file systems and we discovered that indexing has the potential to become a significant bottleneck. We then propose the SCANNS indexing framework, in chapter 3, to address the problem of efficiently indexing data in high-end systems, characterized by many-core architectures, with multiple NUMA nodes and multiple PCIe NVMe storage devices. Drawing from the lessons learned and insights gained from SCANNS, we explored the problems of efficiently building and searching persistent indexes that do not fit into main memory, in Chapter 4. Finally, in Chapter 5, we conducted a performance evaluation of ScienceSearch, a production search infrastructure for scientific data that uses machine learning to automate the creation of metadata tags from different data sources, focusing on scalability trends in order to better understand the implications of searching over indexes built from generated metadata tags.

1.2 Motivation and Use Cases

Advances in search technologies over the last several decades have contributed to the evolution of computing. However, while internet, enterprise, and desktop search capabilities have changed drastically, relatively little focus has been given to search requirements in science and engineering domains as well as on large-scale cyberinfrastructure resources. We describe here motivating use cases from science and engineering as well as cyberinfrastructure providers.

1.2.1 Science.

Materials Science: The Materials Data Facility (MDF) [13] is a centralized hub for publishing, sharing, and discovering materials science data. MDF stores over 19 million files (61TB of data), uploaded by close to 1000 users from different research groups, spanning many disciplines of materials science. The repository contains a range of different data types from common formats (e.g., text and images) through to materials-specific formats. It includes both large (e.g., datasets gathered via x-ray scattering and tomography, high-energy diffraction microscopy, and neutron scattering datasets) as well as smaller datasets (e.g., datasets gathered via atomic force (AFM), scanning electron (SEM), and transmission electron (TEM) microscopy).

However, the expansive range of materials data held by MDF can make it difficult for users to find data relevant to their work, so utility is rooted in the quality of metadata elements to make data findable and accessible. MDF data are primarily stored at Argonne National Laboratory (ANL) and the National Center for Supercomputing Applications (NCSA), and are accessible via Globus (see Figure 1.1). MDF asks publishers to provide metadata describing uploaded data; however, in practice this metadata is limited to publication-style data, rather than metadata describing the specific data. The MDF team have spent considerable effort developing their own specialized extractors called MatIO; however, they are limited to a small slice of the data included in MDF. Currently, MDF uses Globus Search, an ElasticSearch-based service for indexing the available metadata. The index is approximately 1GB in size—a small fraction of the 61TB of data.

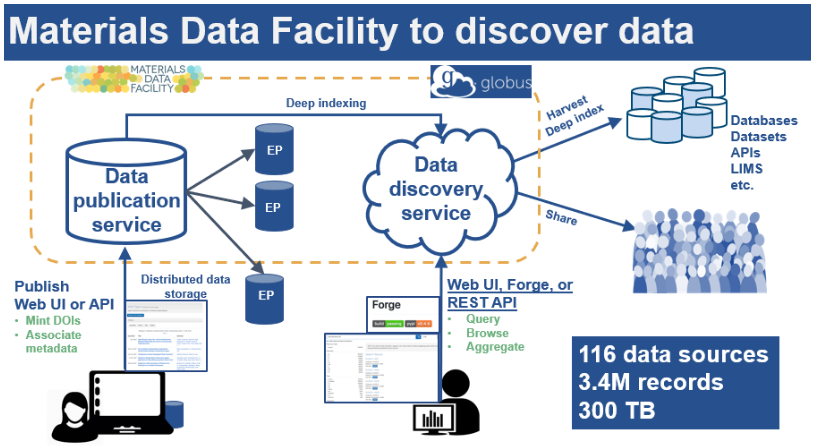


Figure 1.1. Materials Data Facility to discover data.

Earth and Environmental Data: Data Observation Network for Earth (DataONE) [14] is a community-based federated data repository that provides access to data from a set of 44 member repositories (see Figure 1.2). These repositories span a range of earth and environmental data sources including the arctic data center, biological and chemical oceanography data management office, Cornell lab of ornithology eBird, and National Ecological Observatory Network. DataONE currently manages more than 800K datasets exceeding 81 TB of data. Over the last decade, DataONE datasets have been downloaded more than 16M times. Each dataset in DataONE is associated with a metadata record, these records are primarily completed manually following community schema.

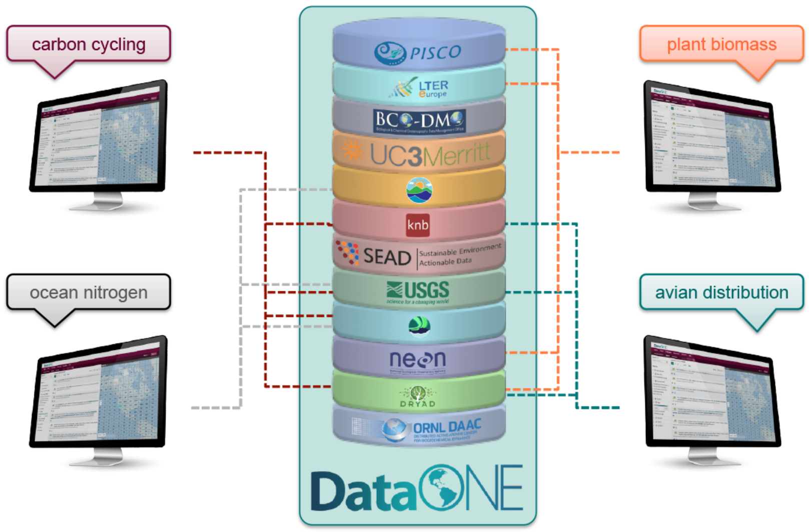


Figure 1.2. DataONE single unified search system that spans 44 repositories.

Cosmology: The Sloan Digital Sky Survey (SDSS), one of the most ambitious scientific projects of all time that aims to make a high-quality three-dimensional map of the universe. SDSS has been collecting data since 1998, and has created 17 data releases. The imaging data include preview images (JPGs) and raw data (FITS). The data also include catalogs of detected objects, with parameters measured from imaging, including positions and magnitudes. These catalogs are stored in a commercial relational database management system (DBMS), Microsoft's SQL Server, and are organized in several 2-dimensional tables. The SDSS dataset contains over a billion objects dispersed over 390 million files (5.5M directories) for a total 652TB of data [15, 16]. A visual representation of some of this data is captured in Figure 1.3.

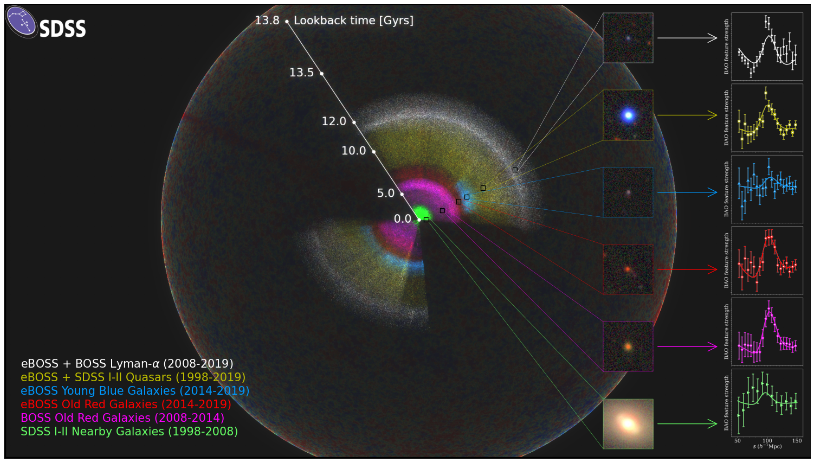


Figure 1.3. Two million galaxies and quasars covering 11 billion years of cosmic time from the eBOSS/SDSS surveys.

Much effort is spent in defining schemas, organizing data, storing metadata in databases, and building search interfaces for users. Many of the methods used are specific and tightly coupled to domain-specific data and/or software. The effort required to implement and maintain a catalog, combined with reduced generality, makes this an impractical solution for data exploration and search in many applications.

1.2.2 Cyberinfrastructure. Storage available on campuses, supercomputing centers, and even within individual research groups is growing rapidly. For example, campus storage routinely exceeds 1PB which is shared among students and faculty, research computing centers offer 10s of PB of storage to their users, while large national cyberinfrastructure providers offer 100 PB of usable storage. At these scales, the estimated resources needed to index data are astronomical. Table 1.1 summarizes storage sizes and expected costs for indexing. This data shows that even modest storage systems may have hundreds of millions to billions of files, cumulatively totaling petabytes to tens of petabytes of data. These file systems are in a constant state of change, with average data modification time measured in hours requiring constant updates to indexes. When analyzing file system traces, we observed deep directory hierarchies with an average of 20 to 212 files per directory, which may limit our ability to optimize the index based on paths. These file system traces offer us a glimpse into the complex world of distributed storage systems for scientific computing. It should be clear that Lucene indexing throughput of 383MB/sec or the grep utility's ability to find keywords deep in data files at 102MB/sec (the typical performance of these tools when running on login node accessing these file systems), would take months to years to completely index and search the storage systems in today's scientific computing systems.

Table 1.1. Characteristics of large file systems and search/indexing times.

File System	**Files**	**Size**	**ls+grep search time**	**Apache Lucene indexing time**
Institution	264M	1.2 PB	146 days	39 days
UChicago RCC	-	2.2 PB	267 days	71 days
NCSA Delta	-	7PB	850 days	227 days
NERSC	861M	6.3 PB	765 days	204 days
TACC Frontera Scratch	-	44PB	5340 days	1425 days
ALCF Eagle	-	100PB	12136 days	3238 days

Beyond simply needing to index these large file systems, there are frequent changes to files as they are added, modified, and removed. Figure 1.4 shows the number bytes created, modified, and deleted daily in 2018 for 35 consecutive days. The figure shows that peak days exhibit changes of nearly 80TB/day with over 2.5M files modified (not shown in the figure), and on average 16TB of data was either added or removed per day. It should be noted that this file system log was captured in 2018 and that file system sizes have increased by an order of magnitude since this time. We therefore expect that millions of file changes per day to be a common use case even on institution and resource computing center storage. While the number of changes for supercomputing facilities is likely much larger.

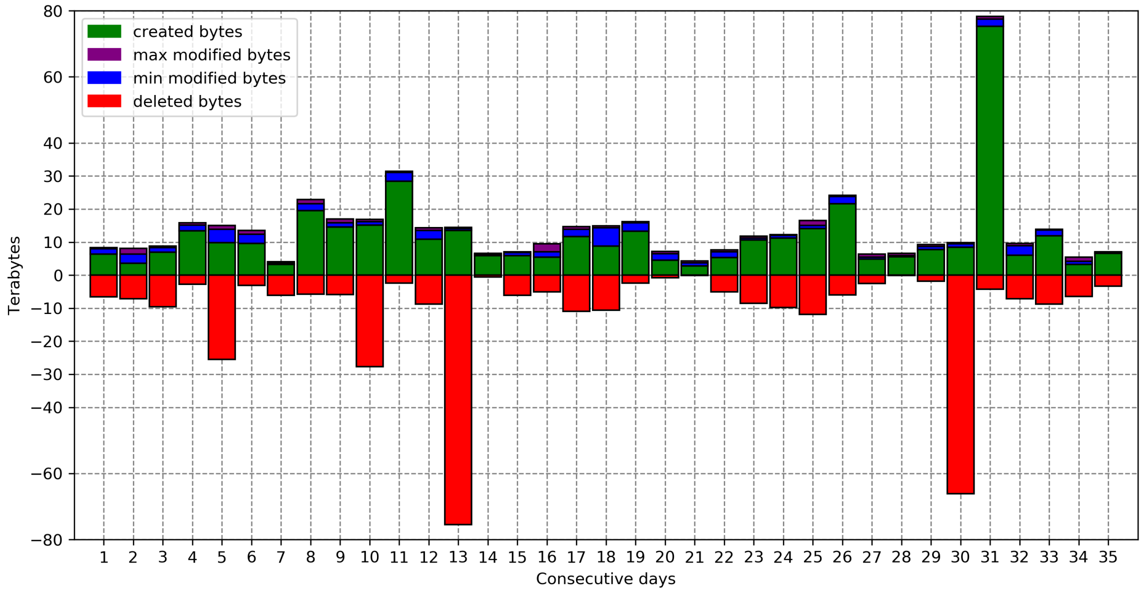


Figure 1.4. NERSC file system data created (green), modified (blue/purple) and deleted (red) measured in terabytes/day over a 35-day period.

1.3 Indexing and Search Engine Background

In this section we include a few general definitions and observations about information retrieval and search engine characteristics that are relevant to the problem of designing, building and maintaining indexed-based search engines. The field of information retrieval identifies and defines itself as the field that aims to "solve the problem of locating and retrieving materials from a collection of information resources, in order to satisfy the information need" [17]. Having this definition in mind, in practice an information retrieval solution, also called an "information retrieval (IR) engine" or colloquially a "search engine," has two main functionalities: indexing and search. Indexing refers to the problem of re-organizing the collection of information resources, in such a way that it makes it easy to locate and retrieve relevant materials from the collection according to some information need. Search is the second function, and it refers to the actual process of utilizing the re-organized collection of information resources, also knows as an "index" in practice, in order to locate and retrieve materials that are relevant to some information need. In the realm of computers, the definition of an IR engine, can be adjusted to: solving the problem of locating and retrieving relevant files from a file system in order to satisfy an information need.

From a structural point of view, a computer search engine can be decomposed into four main components, as seen in Figure 1.5. The *Index Engine* is responsible for extracting the contents of the files in order to re-organize it into an index. This component can perform myriad different types of operations to increase the quality of the index, including similarity analysis, stemming, and lemmatization.

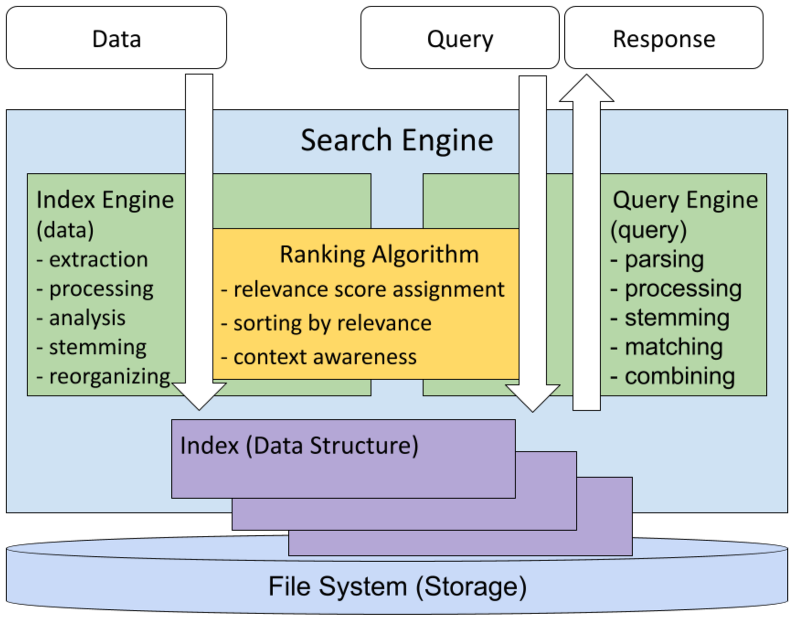

Data
Query
Response
Search Engine
Index Engine (data) - extraction - processing - analysis - stemming - reorganizing
Ranking Algorithm - relevance score assignment - sorting by relevance - context awareness
Query Engine (query) - parsing - processing - stemming - matching - combining
Index (Data Structure)
File System (Storage)

Figure 1.5. General architecture of an information retrieval engine.

The second component is the *Index* itself, which is typically implemented as an inverted index. The term “inverted index” comes from the inversion between content and source of the content that happens during indexing. The inverted index is typically implemented through the use of various search data structures in combination with container data structures, but it can also be implemented using mathematical constructs, such as vectors and matrices, and it can be stored persistently on disk or it can be kept in volatile memory or a combination of both.

If the list of files that are returned by the inverted index are not ordered in any particular way, then the search engine becomes a data retrieval engine, akin to a relational database that provides only the projection function. In order to be a truly information retrieval engine, the third, namely *Ranking Algorithm* component needs to be part of the overall search engine. The Ranking Algorithm, also sometimes used as a synonym to the information retrieval model, is responsible for providing a mechanism to order the returned files from an inverted index by relevance with respect to an information need. Term Frequency-Inverse Document Frequency (TFIDF) is a popular model that uses the frequency of words in files (Term Frequency) and the frequency of files that contain a word (Inverse Document Frequency) to build a mathematical formula that can be used in conjunction with the terms provided by the information need to sort the returned files by their relevance. TFIDF attempts to capture two observations: if a word exists in many files it is likely to be less relevant to the information need; and if a word occurs many times in a file it is likely to be relevant to the information need. TFIDF is not the only successful information retrieval model, but in this work we decide to use it due to its simplicity and effectiveness.

The final component is the *Query Engine*, that is responsible for processing the information need. This component typically reads a search query, applies some of the parsing and analysis techniques present in the Index Engine component, and filters and sorts the returned results according to the Ranking Algorithm.

Looking closer at the process of re-organizing the collection of information materials through indexing, certain observations can be made about the characteristics of this process, observations that can be used to aid the design of a more efficient index engine. One such observation pertains to the index itself and whether it already exists or not. Updating the index versus building the index from scratch imply different assumptions and requirements on how the index engine should work and what kind of structure will the inverted index have, which in turn influences the design and thus the performance. The dynamic versus static property of the collection of materials received as input can also dictate the behavior and performance of the index engine, where a constantly changing collection of materials will require an index that can support fast updates, while for a static collection of materials the index can be optimized for throughput. Another important observation related to the inverted index refers to the uniformity of the placement of the index. Maintaining for example a global index is harder, because it requires a scalable architecture and efficient synchronization and communication, but it can yield faster and more complete search results, while a set of local indexes are easier to build, because they require minimum synchronization and communication, while paying a higher cost for aggregation search results from different index sources. All of these characteristics and properties of the indexing process, index engine and inverted index, if understood and exploited properly can point to the design of scalable and high-performance search engine solutions.

CHAPTER 2

TOWARDS SCALABLE INDEXING AND SEARCH ON DISTRIBUTED AND UNSTRUCTURED DATA

In the era of Big Data, modern applications make use of parallel and distributed strategies to store and process data at scale [18, 19, 20]. While extensive research has focused on the challenges associated with storing and processing large amounts of data efficiently, little work has focused on efficient data search in distributed systems. 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 file-based is still in its infancy. And according to the International Data Company [21], 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.

We propose an indexing solution that takes into account the unstructured and distributed nature of Big Data [1]. While many Big Data systems use a single coordinator to manage resources and data placement (e.g. Hadoop [23] and Myria [24]), 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: one in which files are stored on the nodes as a whole. That is they are not segmented into blocks or chunks that are 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.

[1]The content of this chapter is gathered from published research [22]

`FusionDex` is the proposed indexing solution and is implemented in the FusionFS [25] distributed file system which leverages the ZHT [26] distributed key/value store, and uses the CLucene [27] 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 64 node cluster, showed that the distributed indexing approach has high performance gains in comparison with state-of-the-art approaches, such as Hadoop Grep [28] and Cloudera Search [29]. Comparison of the search latency between Linux grep (52000ms) and FusionDex (23ms), on a single node, shows significant speedup, thus highlighting the potential value of such an approach.

2.1 Design Principles

In comparison with well structured data, found for example in relational database systems, unstructured data does not preserve the same structural characteristics, pre-defined data models, or well-described organization. As such, traditional models used for indexing relational data cannot be applied 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 bottlenecks and single point of failure of distributed indexing of unstructured, file-based data.

The first design principle of FusionDex is that it does not have global coordinators. This approach differs from popular Big Data systems, like Hadoop [23], Myria [24] and SciDB [30], that rely on 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 user may submit queries to any node, the operation is then distributed throughout the system (across nodes) automatically and efficiently. The 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 search 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 coordinator is obvious: it is easy to maintain consistency and synchronization between operations. Proper management 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 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 a global coordinator. Without a global coordinator the network traffic would follow an N-to-N pattern, which would cause a performance catastrophe if each node exchanges a considerable amount of data. 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 is designed around a single indexing module deployed on each node, where each index module is 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.

2.2 Architecture

FusionDex is designed with a share-nothing policy in mind. Such systems do not require coordinators, instead they operate by relying on direct collaboration between the nodes. FusionDex is designed to work with a share-nothing distributed file system such as HDFS [23] or FusionFS [25], both of which store information in the form of files. Importantly, files are stored in their entirety on an individual node, that is, they are not split into blocks or chunks across nodes.

The general architecture of FusionDex is illustrated in Figure 2.1. 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*), and a client process (*Q Client*). The *Q Server* is responsible for the indexing of local data and performing local search. Where a search operation is triggered by any of the clients. The *Q Client* handles query execution, distributing it to other nodes and finally assembling a response. Query clients provide an interface through which users or applications can submit queries to FusionDex.

It should be noted that a query server does not necessarily receive requests from the local query client: any client can communicate with any server in the cluster in a N-to-N communication pattern. Figure 2.2 shows the communication pattern of a scenario in which a user sends a query to a *Q Client*. The *Q Client* then distributes that query operation to all *Q Servers*. FusionDex implements a custom communica-

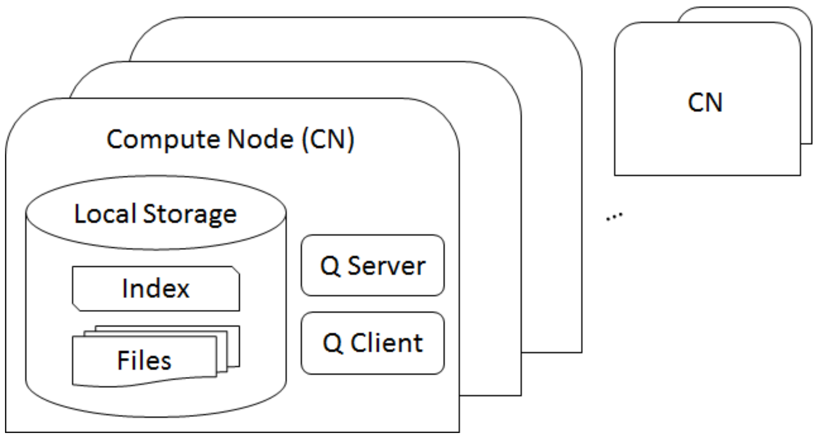


Figure 2.1. An architectural view of FusionDex deployed to a share-nothing cluster.

tion 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.

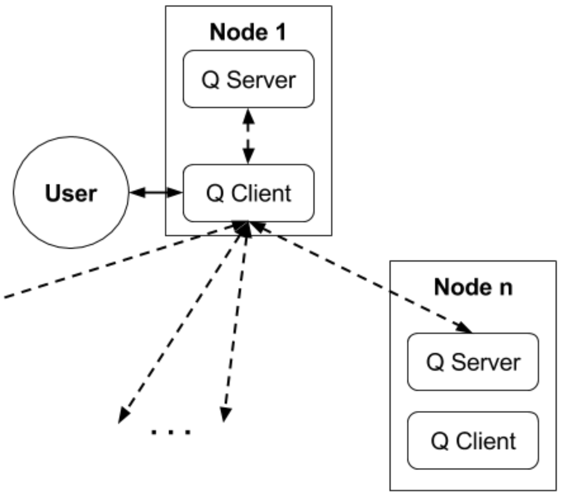


Figure 2.2. A scenario in which a user sends a query to a *Q Client*, that distributes the operation to all *Q Servers*.

2.2.1 Building Blocks. FusionDex leverages CLucene to enable efficient indexing and high performance search performance. It also leverages FusionFS's highly optimized communication model to share data between nodes.

2.2.1.1 CLucene: local indexing. As we can see in Figure 2.1, 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 [27]. CLucene is the C++ port of the popular and open source Apache Lucene [31]—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 written in C++.

2.2.1.2 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.

2.2.2 Index Creation, Removal, and Update. Since files are distributed among the FusionFS nodes, 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 file paths, 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 management 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.

2.2.3 Server Protocols. Figure 2.3 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. 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.

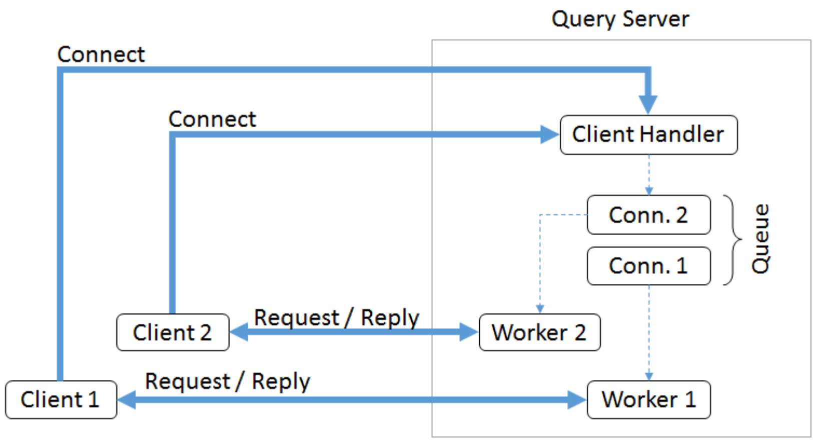


Figure 2.3. Protocols of Concurrent Requests on Query Server.

2.2.4 Client Protocols.

Figure 2.4 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 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 each query server in the file system, and sends the query to each 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.

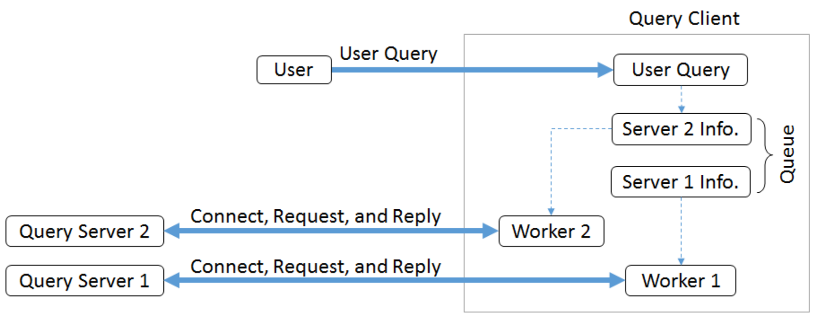


Figure 2.4. Protocols of Concurrent Requests on Query Client.

2.3 Evaluation

2.3.1 Experiment Setup. The test environment, on which we evaluated FusionDex, was deployed on Amazon Web Services Elastic Compute Cloud (EC2). In order to better investigate scalability and performance under realistic scenarios we configured two clusters with varying hardware. In each experiment we modified the number of nodes per cluster. The first cluster (C1) was 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, Linux *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 absence of data from a production distributed file system we developed a test dataset derived from the the Wikipedia dataset [32]. The test dataset was roughly 10 TB in size and it was split into 64 MB files, that were evenly distributed across the nodes provisioned for each experiment. The evaluation process encompassed the issuing of 1,000 queries over these files. The queries were expressed as simple searches of a single keyword picked from a pool of the most popular nouns and verbs found in the test dataset (i.e: *surprise, running, conduct, pale, allow, spent, plan, winter, middle, degree*. Experiments were carried out in an incremental manner with respect to the number of nodes, the upper limit was 64 nodes.

2.3.2 Baseline Performance.

Baseline performance evaluation of FusionDex is conducted on a single node.

2.3.2.1 Index and write throughput.

The raw indexing throughput and file write throughput are shown in Figure 2.5 for increasing data size. The write throughput is calculated as the size of the file, that can be pushed to FusionFS, per second, with indexing enabled. The index throughput is computed as the amount of data that can be indexed in FusionFS, per second. The figure illustrates that FusionDex can achieve a write throughput of approximately 100 MB/s and an index throughput of approximately 1 MB/s irrespective of data size.

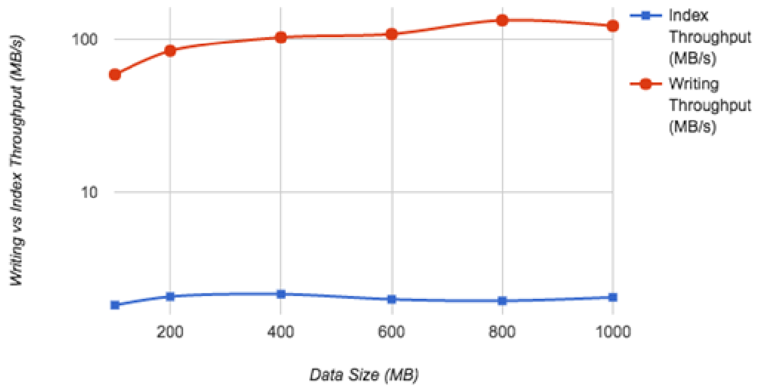


Figure 2.5. Indexing and write throughput on single node

2.3.2.2 Search latency.

The search latency is determined as the time it takes for the server to respond to a search request sent from a client. Figure 2.6 shows the search latency, with and without caching. Intuitively, the expectation is that as the file size increases the search latency also increases. However, these experiments showed that even with data sizes of 100 MB, 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 further improved by an order of magnitude.

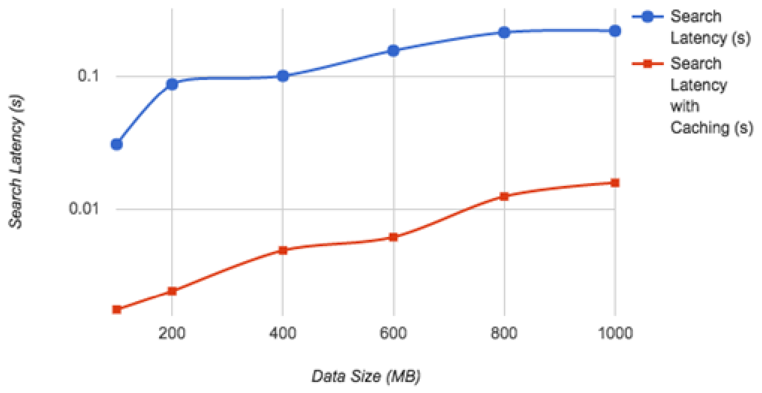


Figure 2.6. Search latency on single node

2.3.2.3 Search throughput. Search 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 to match the number of clients. Figure 2.7 shows that the search 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.

2.3.3 Comparison to State-of-the-art Systems. In this section we compare FusionDex to several systems that provide search capabilities on distributed file systems. More precisely, we explore the ability of these systems to scale to large distributed systems: from 4 to 64 nodes. We compared 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

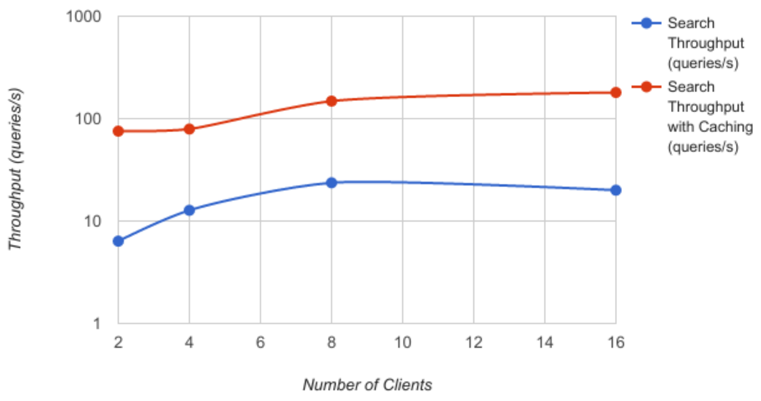


Figure 2.7. Search throughput on single node

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 [23] 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 MapReduce jobs to batch index documents. Cloudera Search uses the *MapReduceIndexerTool* [29], a MapReduce batch job driver that takes a configuration file and creates a set of Solr index shards from a set of input file. It then 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.

2.3.3.1 Indexing Throughput. Figure 2.8 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, as the distributed indexing process amongst multiple nodes amortizes FusionDex's overhead. That is not the case with Cloudera Search, and therefore its indexing throughput does not increase significantly with the number of nodes.

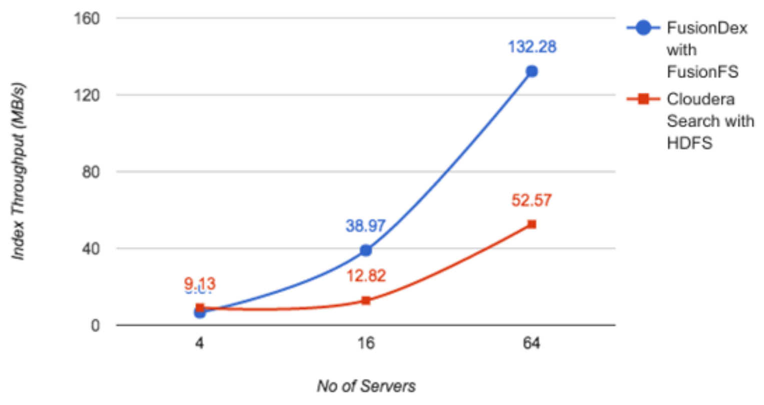


Figure 2.8. Indexing throughput on multiple nodes

2.3.3.2 Search Latency. Figure 2.9 shows the search 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.

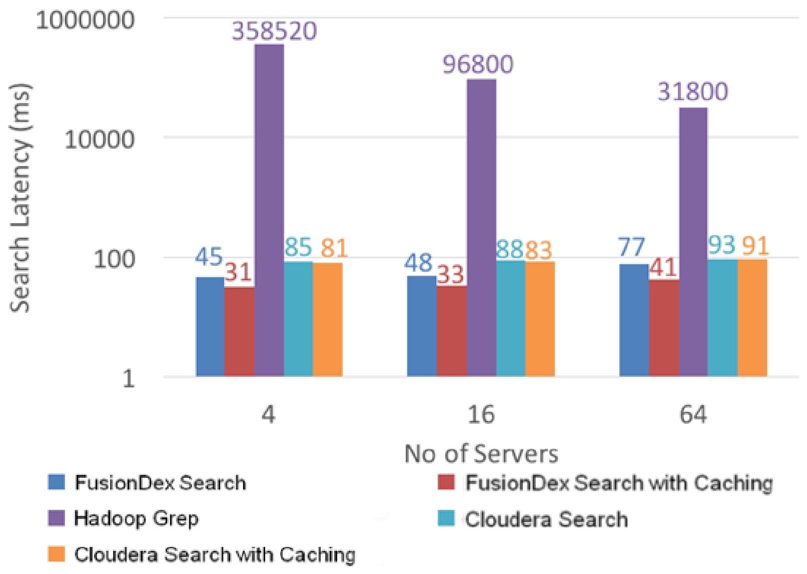


Figure 2.9. Search latency on multiple nodes

2.4 Conclusion

The advent of Big Data has resulted in a shift in paradigms and new ways of thinking about managing large quantities of data that are produced at high velocity. There are many solutions and research initiatives for optimizing 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 work, we proposed FusionDex [22], a distributed indexing and query scheme for unstructured data dispersed over distributed file system. FusionDex uses CLucene, which is a C++ port of the popular Apache Lucene, as the building block for the indexing engine, and FusionFS's data transfer services for inter-node communication. Investigation of FusionDex's performance showed high performance gains in comparison with state-of-the-art approaches, such as Hadoop grep and Cloudera Search. While FusionDex achieved significant performance gains in terms of latency, the proposed solution hinted towards indexing becoming a potential bottleneck for increasing data volumes. Using CLucene, FusionDex achieved an indexing throughput of only 2MB/sec per node, slower by about 2 orders of magnitude below the theoretical capabilities of the hardware used. Since we discovered indexing performance to be orders of magnitude slower than theoretical speeds we could achieve in raw storage input and output, we sought to investigate a new clean-slate design for high-performance indexing.

CHAPTER 3

SCALABLE AND CONCURRENT DATA INDEXING AND SEARCHING IN HIGH-END COMPUTING SYSTEM

In the previous chapter we explored the prospect of using CLucene, an existing information retrieval library and building blocks, to enable and integrate search in a high-performance distributed file system. We proposed, built and evaluated the FusionDex system, a distributed indexing and query model for unstructured data, and we found indexing performance to be orders of magnitude slower than theoretical speeds we could achieve in raw storage input and output.

In order to address the problem of slow indexing performance, we sought to investigate a new clean-slate design for high-performance indexing and search and proposed the SCANNS indexing framework [2]. SCANNS is an indexing library that is designed to be deployed on single-node high-end systems, characterized by many-cores architectures, multiple NUMA nodes and multiple PCIe NVMe devices. SCANNS is designed to be used as a building block for building high-performance index-based search engines. SCANNS redesigns and exposes the indexing pipeline, in such a way that it can exploit modern hardware capabilities and can allow users to tune certain aspects of the pipeline, in order to saturate available compute, memory, and/or storage resources. In this work we present the SCANNS framework and the many optimizations and techniques we applied to improve the performance of the overall framework, and of each pipeline component. We also present practical insights related to constructing indexes and tuning indexing performance that can be overlooked when building index-based search engines, such as the importance of the design of additional data structures required for the inverted index even when building on a fast search data structure. We perform an experimental evaluation of the framework

[2]The content of this chapter is gathered from published research [33]

and it's components, and we show that it can achieve magnitudes higher indexing and search performance when compared to Apache Lucene, a state-of-the-art information retrieval library.

The contributions of this work are as follows:

- Design and implementation of SCANNS, an tune-able indexing framework that can exploit the properties of modern high-performance computing systems;
- Tune-able modularized architecture that allows the saturation of storage, memory and compute resources;
- Evaluation on machines with up to 192-cores, 768GiB of RAM, 8 NUMA nodes, and up to 16 NVMe drives, and delivering 19x higher indexing throughput and 280X lower search latency;

3.1 Framework Architecture and Design

This section presents the SCANNS architecture, covering a general overview of the framework and its underlying components, and detailed description of the techniques and optimizations used to improve indexing performance.

3.1.1 SCANNS Goals. The primary goal of SCANNS is to support efficient indexing of data in high-end computing systems. With that in mind, SCANNS was designed to efficiently leverage systems that have many cores, multiple NUMA nodes, and multiple NVMe devices, by exploiting the inherent properties of such systems in order to saturate their compute, memory and/or storage resources. The secondary goal of SCANNS is to be versatile enough so that it can accommodate different data sources and formats, and various information retrieval models, thus the framework is designed as a search engine library, that can be used to implement specific search engine applications.

3.1.2 SCANNS Overview. In order to satisfy the goals of SCANNS, we studied the general process of performing indexing on high-end systems, and identified three key sub-processes. For each of sub-process we designed a component that focuses on a specific system resource and a precise part of the indexing process. When combined, these components form a complete indexing engine. A diagram of these components and how they are connected structurally and functionally can be seen in Figure 3.1. The three components are: the *ReaderDriver*, which is responsible for reading raw data from a storage system and is typically IO-intensive; the *Tokenizer*, which is responsible for parsing and tokenizing the raw data into units of data that are useful for a specific information retrieval model and is usually compute-intensive; and the *Indexer*, which is responsible for computing and storing the index from the units of data. All three components are designed as independent functions, that can be run by one or more threads, exclusively or shared, giving the the user option to fine tune the number of threads and the number of components according to the amount of compute, memory, and storage resources available.

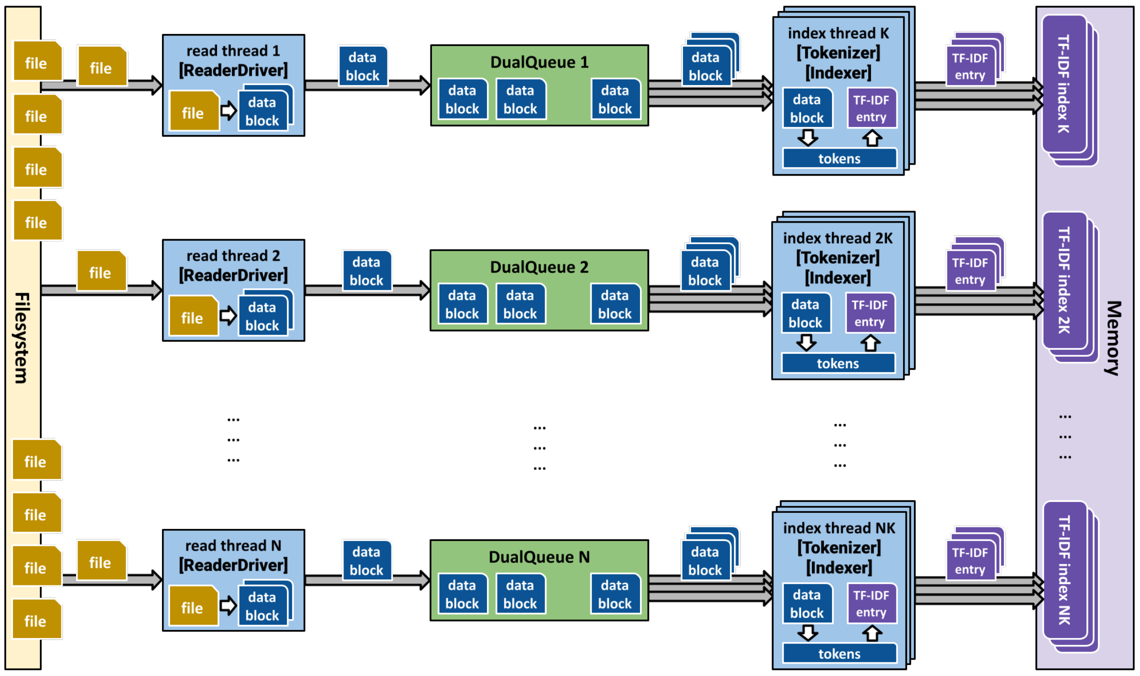


Figure 3.1. SCANNS framework indexing architecture and pipeline.

This framework implements a TFIDF search engine over a collection of files stored on multiple PCIe NVMe devices and is optimized to achieve high indexing speeds in the scenario where the index does not already exist and it is being built for the first time. In this work we assume that the input dataset will not change while the index is being built and the framework is designed to support fixed-term, extended boolean search.

3.1.3 Indexing Engine Execution. In terms of execution, SCANNS uses multiple threads to parallelize the execution of the indexing process by data but also by function. The framework uses two kinds of threads, as seen in Figure 3.1: *read threads* and *index threads*. Read threads are responsible for reading raw blocks of data from the file system(s) and for passing these blocks to the index threads, and they run local ReaderDriver instances. Index threads receive raw blocks of data from the read threads and process the blocks of data in order to build the local index, by running local pairs of the Tokenizer and Indexer components. Index threads implement the observer design pattern, where the Tokenzier is the subject and the Indexer is the observer, and makes use of the internals and interfaces of the Indexer component to store the local indexes in memory. The number of read and index threads are manually configured at the beginning of the execution of the indexing framework and remain static until the index is complete. The number of index threads needs to be a multiple of read threads and the read threads will communicate to the same specific group of index threads for the entire of the indexing process.

The read threads communicate and share blocks of data with the index threads through a set of specialized queues, that we called *DualQueues*. The DualQueue is a simple implementation of a thread-safe synchronized queue that follows the memory pool design pattern to recycle the blocks of data that are being pushed and popped to and from the queue. Figure 3.2 shows that, in terms of design, the DualQueue is implemented with two synchronized queues, one for the blocks that are empty and do not have any data, and one for the blocks that are full and have data read into them. The queues use mutexes and conditional variables to achieve synchronization and to relieve the system from unnecessary polling when either of the queues is full or empty. The read and index threads act as producers and consumers, respectively, and are responsible with popping, pushing and processing blocks of data.

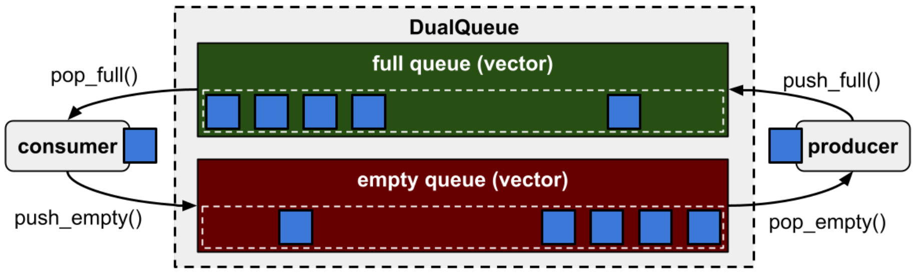


Figure 3.2. SCANNS DualQueue design.

3.1.4 ReaderDriver. The ReaderDriver is the SCANNS component responsible for ingesting raw data from the storage subsystem to main memory as fast as possible. In our case the ReaderDriver is designed to read blocks of data from a POSIX file systems as fast as possible and bring it to main memory so that it can be processed by the other components of the framework. This component is typically bound by the capabilities of the storage subsystem, but that is not always the case, especially in the case of many PCIe NVMe storage devices present in the system. We observed, in practice, that a standard approach to implementing this functionality, where each block of data read is allocated dynamically at runtime and deallocated when not needed, leads to suboptimal performance, in terms of how many blocks can be brought in main memory per second. Thus the first optimization that we propose avoids the overhead of allocating and deallocating each block of data through the use of the memory pool design pattern. Basically, since we know that the blocks will be discarded after they are processed by the framework, we allocate a certain number of blocks at the beginning of the program and we reuse them when they get discarded.

This optimization is built in tandem with the DualQueue, having the ReaderDriver generate, manage and push the blocks to the queue at the beginning of the program.

In a setup where a machine has many PCIe NVMe devices we observed that sometimes the memory subsystem of the OS that manages the file system caches and buffers can become a performance bottleneck. Since the data that is read from the input files by the indexing engine is being re-organized, it is not actually required to be stored in the index. Thus the second optimization that we proposed was to bypass the OS file system caches and buffers and tell the OS to bring the blocks of data from the disk directly into ReaderDriver buffer space. This optimization, in conjunction with enough multi-threading, allows the ReaderDriver to saturate available NVMe disks in terms of number of blocks read per second.

So far the described ReaderDriver was optimized to read fixed-size blocks of data from the file system as fast as possible, but in practice this approach can be problematic. The fixed-size approach can end up breaking tokens in halves, which need to be addressed and recombined in order to implement a correct indexing engine. To solve this issue, we proposed the *WaveReaderDriver*, which uses a small addon block to read additional data from disk and computes how long the blocks needs to be so that it does not break tokens in halves. The WaveReaderDriver exposes an idempotent method for reading blocks of data from a file, that returns a variable-size block and retains the memory pool design pattern and OS cache and buffer bypass optimizations. We solved this issue in the ReaderDriver, because we observed that it is the fastest component and had enough computing resources to spare.

3.1.5 Tokenizer. The second component in the SCANNS indexing pipeline is the Tokenizer. This component is responsible for reading the raw data passed from a ReaderDriver and transforming the raw data into smaller units of data that can be subsequently used by the Indexer. In the context of this work, the Tokenizer pops a variable-size block of data from a DualQueue and extracts tokens from the block, that are separated by some delimiter. Basically this component implements a *split* function, that splits a string into a list of *tokens* (i.e. substrings that are separated by a list of characters that act as delimiters). The process behind the Tokenizer is typically compute-intensive, reading the input string and extracting the tokens sequentially.

While this component can be implemented in a standard way in C through the use of the *strtok()* function, we observed that the performance of the standard approach is very low when compared to how fast the ReaderDriver can read data from disk. In order to improve the Tokenizer's performance, we proposed a re-implementation of the split function, where we replaced the call to *strtok()* with an approach that uses branchless programming. Figures 3.3 and 3.4, show the conceptual difference between the standard and the optimized Tokenizers.

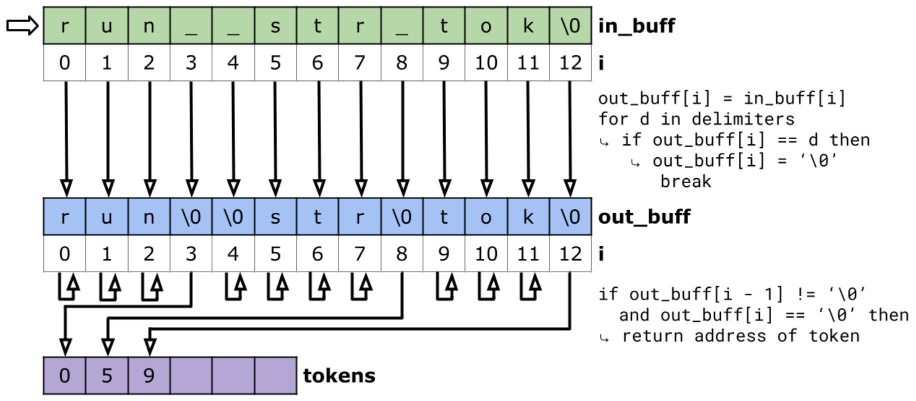
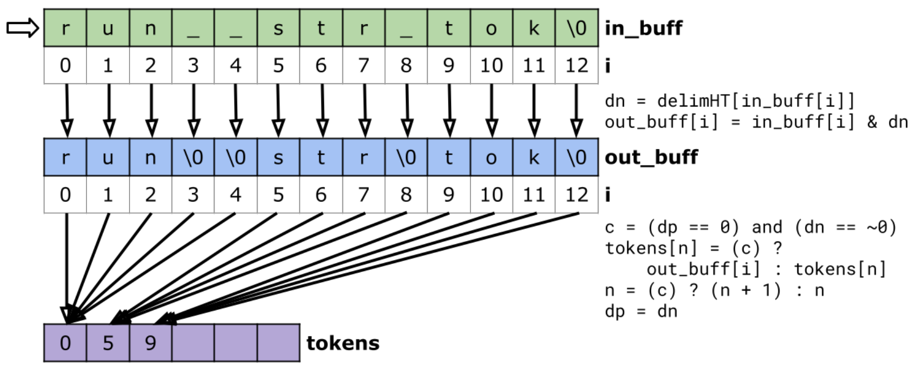

r u n _ _ s t r _ t o k \0
in_buff
0 1 2 3 4 5 6 7 8 9 10 11 12
i
out_buff[i] = in_buff[i]
for d in delimiters
↳ if out_buff[i] == d then
↳ out_buff[i] = '\0'
break
r u n \0 \0 s t r \0 t o k \0
out_buff
if out_buff[i - 1] != '\0'
and out_buff[i] == '\0' then
↳ return address of token
0 5 9
tokens

Figure 3.3. SCANNS Standard Tokenizer.

We replaced the for loop and the if-block, that *strtok()* used to iterate over the list of delimiters to find out if a byte in the input buffer is a delimiter or not, with an O(1) lookup in a hash table of delimiters. For each character the delimiter hash table returns zero if the character is a delimiter and zero negated otherwise. We then replaced the portion of the code where *strtok()* runs an if-block to check if



```
dn = delimHT[in_buff[i]]
out_buff[i] = in_buff[i] & dn
```

```
c = (dp == 0) and (dn == ~0)
tokens[n] = (c) ?
    out_buff[i] : tokens[n]
n = (c) ? (n + 1) : n
dp = dn
```

Figure 3.4. SCANNS Optimized Branchless Tokenizer.

it has reached the end of a token and returns the token address when true, with C ternary operations that implement the same functionality. The ternary operations get in turn generated into conditional assembly instructions that do not cause branches or jumps. This optimization removes the overhead of branch misses, that are caused by the CPU branch predictor and the unstructured nature of the input data, allowing the Tokenizer to catch up the ReaderDriver, in terms of performance.

3.1.6 Indexer. The Indexer is the third and last component of the SCANNS framework and is responsible with taking the tokens/terms, extracted by the Tokenzier, and with re-organizing them into a TFIDF inverted index, that is stored in main memory. Figure 3.5 shows the design of the inverted index, which is the core of the Indexer.

For this work we picked hash tables as the search data structure to be incorporated in the inverted index, due to their increased performance and their potential to be distributed across computers. The SCANNS inverted index does not depend on a specific implementation of a hash table and supports pluggable hash tables, in order to allow the user to use any hash table with any hash function that is appropriate for their dataset. In SCANNS we used two hash tables: the C++ unordered_map, for the standard hash table, and the Google Swiss Table [34], for the efficient hash table; and we show that while the search data structure is important, poor inverted index data structure design can lead to reduced performance.

The design in Figure 3.5 shows the additional data structures used to implement the TFIDF inverted index: *IDFIndexEntry* and *TFIndexEntry*. Both data structures are implemented as linked lists and each instance stores a pointer to the next element in the list. The hash table stores in each of its buckets a list of IDFIndexEntries, and each IDFIndexEntry keeps track of the token associated with the entry, the number of files that contain the term, a head and a tail to the TFIndexEntry linked list. The TFIndexEntry stores the index associated with a file, the frequency of a term in that file and a pointer to the next TFIndexEntry. During indexing, the Indexer will perform lookups in the hash table and create new IDFIndexEntries or TFIndexEntries if they don't exist and will update the frequency information for each term-file pair. Since SCANNS is aimed at building the index from scratch for the first time, we instructed the framework to pass the data blocks to the Indexers such that a block only belongs to the same file that is being processed or a new file, but never to a previously processed file. This high-level data flow optimization, allows the Indexer to avoid additional searches over the list of TFIndexEntries, performing either an update or an append operation on the tail TFIndexEntry of a IDFIndexEntry, and thus providing a boost in performance.

But even with this minimalist design and the proposed high-level optimization on how file blocks are passed to the Indexer, the inverted index yielded poor scalability with increasing number of cores. After further investigations we identified two main causes: (1) the standard memory allocator wasn't scaling to the number of small IDFIndexEntry and TFIndexEntry objects that were being created and (2) there were still too many CPU cache misses, caused by the hash table lookup and the indirection from the inverted index data structures.

To address the problem of memory allocation we proposed the implementation

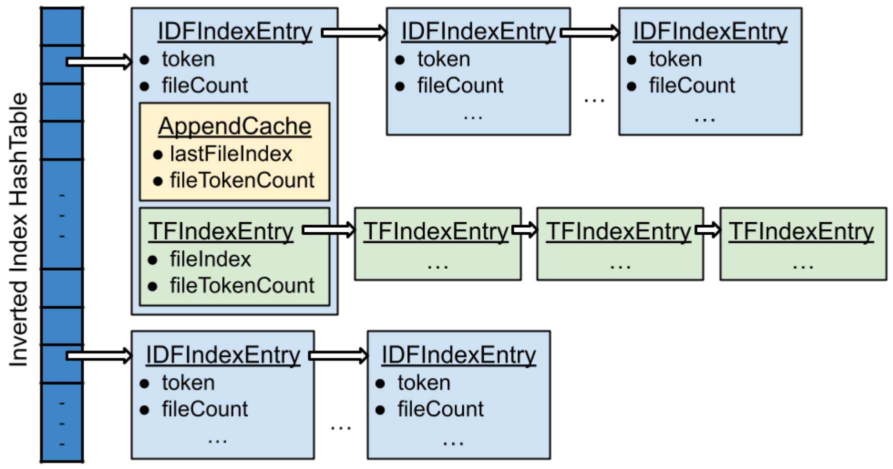


Figure 3.5. SCANNS Inverted Index Design.

of a monotonic paged sub-allocator for the index data structures. The sub-allocator allocates large pages of memory and then creates the required inverted index objects from those pages in user-space, at faster speeds than when allocating memory and calling a system call for each object individually.

To deal with the second issue, we introduce an *AppendCache* to the IDFIndexEntry that minimizes the number of memory indirections to the TFIndexEntry list tail during term frequency updates. The AppendCache is part of the IDFIndexEntry, thus whenever the IDFIndexEntry is being accessed the AppendCache is brought in the CPU cache as well, subsequently improving indexing performance. The cache is flushed when a block from a new file is processed. The last optimization scales well with datasets where terms appear frequently, and with the page sub-allocator and enough compute cores, the Indexer can achieve higher performance than state-of-the-art indexing solutions.

3.1.7 Global optimizations. The SCANNS framework also incorporates in its design optimizations that are global in nature and do not belong specifically to only one component. These optimizations deal with reducing the overheads of inter-NUMA communication, the page-fault subsystem of the OS and the tuning of the file block sizes and sub-allocator page sizes. The first optimization is applied over the ensemble of DualQueues, read and index threads, making sure that the threads are grouped by NUMA node and that the memory allocated and accessed by each component also resides in the same NUMA node. This is achieved through the use of the *libnuma* library, that allows users to set NUMA affinities and memory policies to programs.

The second global optimization is the use of huge pages for the monotonic sub-allocator and for any buffers. With huge pages, the application can relived the OS from having to handle many page faults, implicitly improving the performance of any memory-intensive application, including the Indexer component. And the last set of optimization relate to the tuning of ReaderDriver block sizes and Indexer sub-allocator page sizes, in order to further improve performance. The SCANNS framework exposes these parameters to the users, allowing them to better tune the indexing engine accordingly to the underlying hardware. All of the experimental results have a certain degree of manual tuning performed.

3.2 Performance Evaluation

In this section, we present the performance evaluation of the SCANNS framework and its constituting components. We include, in the discussion, details about the experimental setup, the used dataset and the SCANNS components variants.

3.2.1 Experimental Setup. The experimental setup is comprised of three single-node high-end systems deployed on Mystic, an NSF-funded testbed designed to study system re-configurability. The three systems differ in many aspects, but for this work the most important differences are the number of cores and the number of storage devices available on each machine. The number of cores are a reflection of computational power, while the storage devices showcase varied IO capabilities. Table 3.1 presents hardware details for each system. The three systems allow us to evaluate SCANNS under different environments: (a) a machine with many cores, 8 NUMA nodes, but few disks (*64cores-1disk*), (b) a machine with few cores, 2 NUMA nodes, but many disks (*32cores-16disks*), and (c) a machine with many cores, 8 NUMA nodes, and many disks (*192cores-16disks*).

Table 3.1. Mystic Cloud machines used for the experimental evaluation and their specifications.

machine name	**processors**	**cores**	**memory**	**nvme storage**
(a) 64cores-1disk	2 x AMD EPYC 7501	64	128GiB	1 x Intel Optane 900P
(b) 32cores-16disks	2 x Intel Xeon Gold 6130	32	192GiB	16 x Samsung 970 EVO
(c) 192cores-16disks	8 x Intel Xeon Platinum 8160	192	768GiB	16 x Intel Optane 900P

We configured the hardware and the OS to use performance governors and turbo-boost for all CPUs, and all of the storage devices used during experiments were PCIe NVMe SSDs, that were accessed exclusively, in order to eliminate any interference caused by other running applications. For systems that have only one disk we configured XFS directly on the device, while for systems that had more than one disk, we grouped the disks by NUMA nodes, and configured Linux software RAID0 arrays with XFS for each group.

In terms of software, 64cores-1disk and 32cores-16disks ran Ubuntu 18.04 LTS with Linux Kernel 4.15 and g++-8.4, while 192cores-16disks ran Ubuntu 20.04 LTS with Linux Kernel 5.4 and g++-10.3. For Google SwissTable we used version 20210324.2 from the abseil library. SCANNS is implemented in C++17 and we use openjdk-11 to run Apache Lucene.

The datasets used throughout the experimental evaluation were generated from a file system dump provided by NERSC. The file system dump is a snapshot of the file system metadata of the NERSC storage system, that was stored in one single 240GB file with each line of the file containing a full file path and all POSIX metadata information (size, timestamps, owners, permissions, inode etc) separated by space. We cleaned and split the 240GB file system dump file into smaller files of approximately and up to 32MiB in size. The ReaderDriver and Tokenizer evalution was done over a collection of small file system dump files of 6144 files (192GiB), while the TFIDF End-to-End indexing and search evaluation was conducted on a collection of 1536 files (48GiB). We picked the file system dump dataset because it represents a real dataset and it has interesting properties: most of the space or slash separated terms found in the file system dump are alphanumerical and numerical and only a few have only letters in their composition. This means that classical free-text stemming techniques cannot work with this dataset, which increases difficulty of building indexes by having many unique terms.

3.2.2 Component Variants. For each of the SCANNS framework components we implemented multiple variants to show performance improvements of each optimization and technique used. For the ReaderDriver we experimented with the following variants:

- *xs-rd-std* - (the baseline) reads fixed-size blocks of data without optimizations;
- *xs-rd-nonuma* - uses the memory pool design pattern and the OS cache and buffer bypass optimizations;
- *xs-rd-numa* - similar to xs-rd-nonuma, plus NUMA-aware thread scheduling and memory allocation;
- *xs-rd-wave* - similar to xs-rd-numa, but implements the WaveReaderDriver that reads variable-size blocks of data;

For the Tokenizer evaluation, the implementation used the WaveReaderDriver to read and pass blocks of data to the Tokenizer. Instead of index threads, we called the the threads that ran the Tokenizers tokenize threads. These are the Tokenizer variants that we experimented with:

- *xs-rdtokstd-nonuma* - implementation using *strtok()*;
- *xs-rdtokstd-numa* - similar to xs-rdtokstd-nonuma, plus NUMA-aware thread scheduling and memory allocation;
- *xs-rdtok-nonuma* - implementation that uses branchless programming and the delimiter hash table optimizations;
- *xs-rdtok-numa* - similar to xs-rdtok-nonuma, plus NUMA-aware thread scheduling and memory allocation;

The TFIDF End-to-End indexing and search evaluation is performed on variants that include both the WaveReaderDriver and the Tokenizer in their runtime. We compare the SCANNS variants between themselves but also to an indexing and search application implemented using the Apache Lucene information retrieval library. We used ClassicSimilarity and the WhiteSpaceAnalyzer to tell the Lucene variant to perform the same kind of indexing and search that SCANNS implements, namely TFIDF. We further tuned the Lucene variant by setting the JVM available and start memory to the maximum available on the system, we enabled server mode and parallel garbage collector, and we tuned Lucene itself to use 1GiB buffers and two merge threads per index thread. In the Lucene variant, similar to the SCANNS variant, each index thread builds a local index and there is no communication between the index threads. Here all of the variants that we experimented with during the TFIDF End-to-End indexing and search:

- *xs-rdtokidx-std* - implementation using C++ unordered_map and without any kind of optimizations;

- *xs-rdtokidx-swiss* - implementation using Google Swiss Table and without any kind of optimizations;
- *apache-lucene* - uses Apache Lucene;
- *xs-rdtokidx-std-pa* - similar to xs-rdtokidx-std, plus monotonic paged sub-allocator, append cache optimization, NUMA-aware affinity and huge pages;
- *xs-rdtokidx-swiss-pa* - similar to xs-rdtokidx-swiss, plus monotonic paged sub-allocator, append cache optimization, NUMA-aware affinity and huge pages;

3.2.3 ReaderDriver. Figure 3.6 shows the performance the ReaderDriver variants, measured in MiB/sec with increasing number of read threads, when running on a system that has only one NVMe device installed but many compute cores. Here we can see that all variants are able to saturate the single Samsung 970 EVO NVMe device (2.5 GiB/sec) with a sufficient number of read threads.

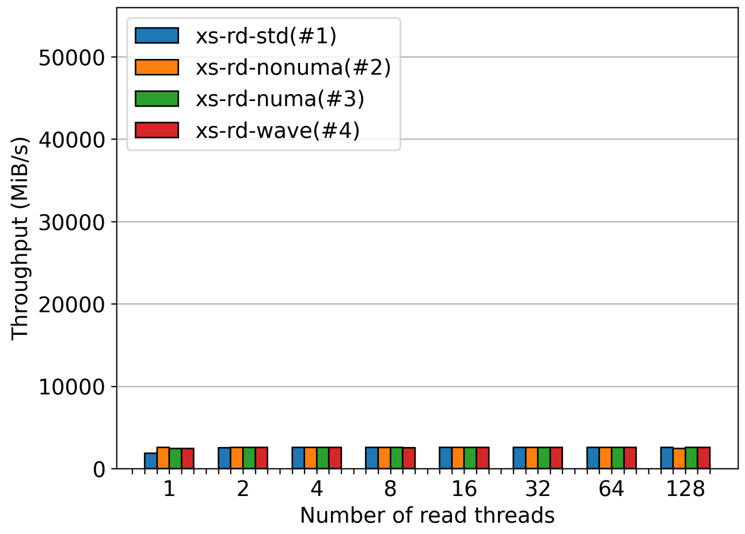


Figure 3.6. ReaderDriver throughput with increasing number of read threads on *64cores-1disk*.

In Figure 3.7 we see a different picture. The baseline ReaderDriver seems to cap at approximately 7.5 GiB/sec, while the optimized versions reach close to the theoretical limit, which is 56 GiB/sec for 16 Samsung 970 EVO NVMe SSDs (3.5 GiB/sec theoretical throughput per device), assuming linear scalability. The WaveReaderDriver's throughput caps at 40 GiB/sec, and after investigation we realized that this is caused by the fact that these SSDs have a 4GiB internal fast cache. The internal fast cache guarantees the advertised throughput as long as the data does not exceed the cache size, but in our case the data set size split across 16 devices does exceed the cache size, which causes the throughput to fluctuate. We consider this to be acceptable since the Tokenizer and the Indexer typically exhibit lower performance than the WaveReaderDriver.

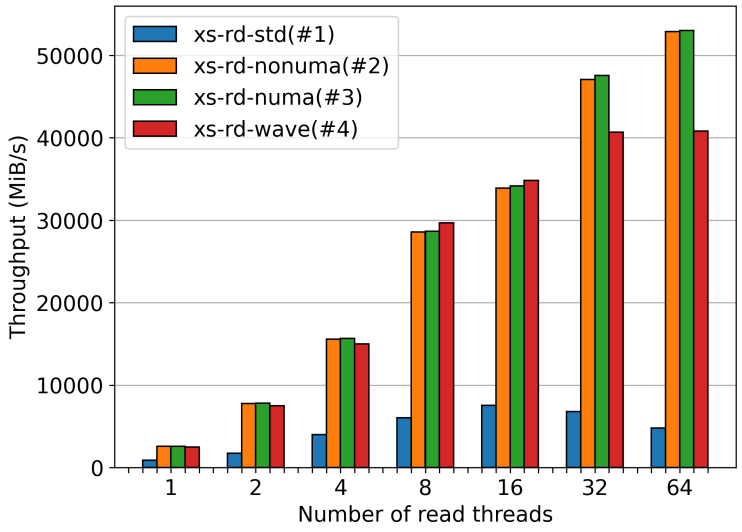


Figure 3.7. ReaderDriver throughput with increasing number of read threads on *32cores-16disks*.

Figure 3.8 shows the performance of the ReaderDriver variants on a system with many cores and multiple NVMe devices. The most interesting result in this configuration is the importance of NUMA-aware configurations. We can see an improvement of 20% between the variant that uses NUMA aware thread scheduling and memory allocation versus the one that does not. The WaveReaderDriver achieves approximately 35 GiB/sec which is close to the theoretical 40 GiB/sec that 16 Intel Optane 900P devices can achieve.

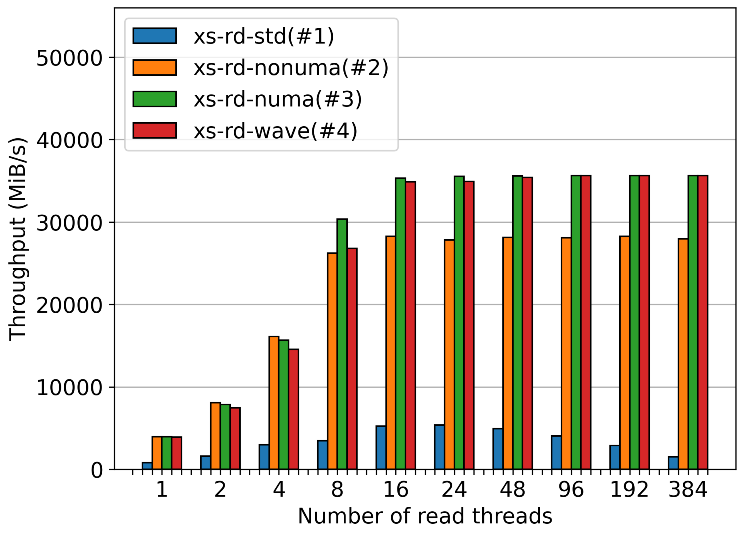


Figure 3.8. ReaderDriver throughput with increasing number of read threads on *192cores-16disks*.

3.2.4 Tokenizer. Figure 3.9 shows the performance, measured in MiB/sec with increasing number of read and tokenize threads, for all of the 4 variants, running on the system that has only one NVMe disk. We can see that all of the variants manage to reach the disk limit in terms of performance after 8 read threads plus 8 tokenize threads (for a total of 16 threads), but we can see that the optimized version is able to reach that limit faster than the standard versions, with or without NUMA-aware affinity. In this setup the NUMA-aware affinity have no affect as there is only one NVMe device.

Figure 3.10 shows performance on a system that has many NVMe devices but not many cores. Here we can see a significant difference in performance between the optimized and standard Tokenizer versions. Throughout all of the number of thread configurations, we can see that the optimized Tokenizer achieves performance that is roughly twice as fast as the standard version, reaching approximately 18.8 GiB/sec throughput with 32 read threads and 32 tokenize threads. In this setup the NUMA-aware configuration only makes a difference when we saturate the hardware threads of the machine, but the difference is slight, increasing the performance of the optimized

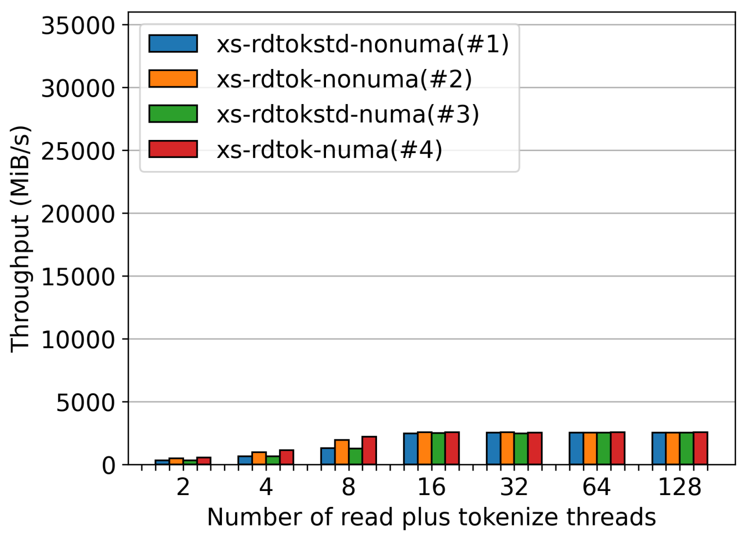


Figure 3.9. ReaderDriver and Tokenizer throughput with increasing number of read and tokenize threads on *64cores-1disk*.

version from 16.9 GiB/sec to 18.8 GiB/sec.

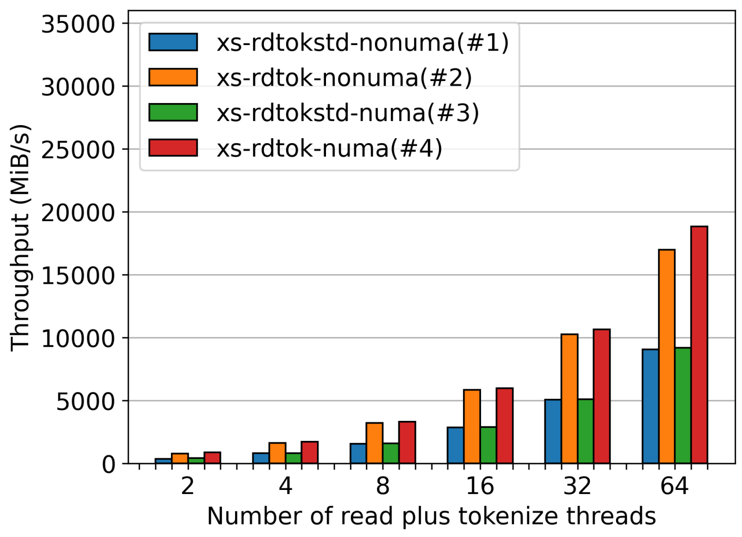


Figure 3.10. ReaderDriver and Tokenizer throughput with increasing number of read and tokenize threads on *32cores-16disks*.

Figure 3.11 shows performance with many cores and multiple NVMe devices and here we can clearly see the difference between all variants and thus between all optimizations. Between the versions that do not use any kind of NUMA-aware affinity, we can see that the optimized Tokenizer achieves better performance than the standard version capping up at around 20 GiB/sec, but both versions seem to start losing performance when the number of read plus tokenize threads exceeds 96. As for when the Tokenizer also uses NUMA-aware affinity, we can see that both optimized and standard Tokenizers reach the disk limit and flatten out at a throughput of approximately 34 GiB/sec. While both of these versions reach the disk cap, we can clearly see that the optimized version reaches the cap faster and if the disk wouldn't be a limit it would probably still maintain the 2x advantage over the standard variant. We consider these results satisfactory, since we observed that the slowest component is the Indexer, that cannot reach the Tokenizer or ReaderDriver in performance.

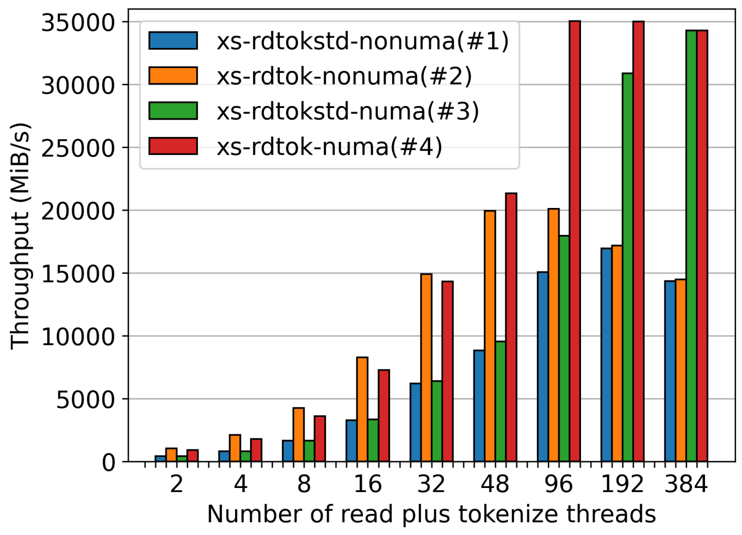


Figure 3.11. ReaderDriver and Tokenizer throughput with increasing number of read and tokenize threads on *192cores-16disks*.

3.2.5 End-to-end TF-IDF indexing and search.

Figure 3.12 shows the performance, measured in MiB/sec of End-to-End indexing with increasing number of read and index threads, for all variants. Each index thread is paired with a read thread, with the exception of the Lucene variant that two merge threads with each index thread instead. We can see that, for a system that has only one NVMe disk, solutions that do not use any kind of memory optimizations seem to reach a low performance threshold, at about 400 MiB/sec for the Lucene variant, 450 MiB/sec for the

Swiss Table implementation and 275 MiB/sec for the standard implementation. When using all of the memory optimizations, since the Indexer is more memory-intensive rather than compute-intensive, combined with the NUMA-aware tuning and huge-pages we can see that both the standard and the Swiss Table implementations can surpass the low performance threshold. The standard implementation reaches up to 815 MiB/sec with 32 index and 32 read threads, while the Swiss table reaches 2255 MiB/sec. These results show that in order to achieve high indexing performance, the inverted index needs a fast search data structure but also an efficient inverted index design.

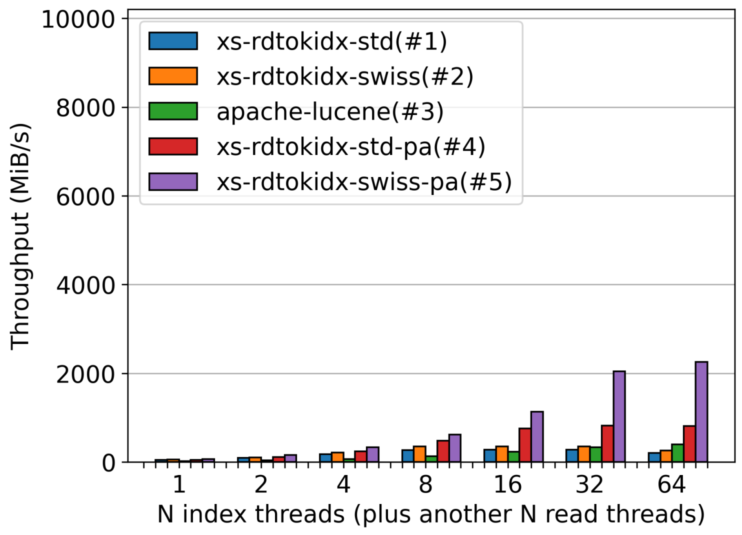


Figure 3.12. End-to-end TF-IDF indexing throughput with increasing number of read and index threads on *64cores-1disk*.

When looking at a system that has multiple NVMe devices but not that many cores, as depicted in Figure 3.13, we see a similar trend. The un-optimized solutions, including the Apache Lucene variant, due the fact that they do not exploit the memory hierarchy properties of these systems, cannot achieve very high performance and cap out at 366 MiB/sec for Apache Lucene, 628 MiB/sec for the Swiss Table implementation and 486 MiB/sec for the standard implementation. Only by incorporating the memory and NUMA-aware affinity can the standard implementation reach

1185 MiB/sec and the Swiss Table implementation reach 2431 MiB/sec, both with 32 index threads and 32 read threads. This system achieves better performance overall because there are more memory channels per NUMA node than on 64cores-1disk.

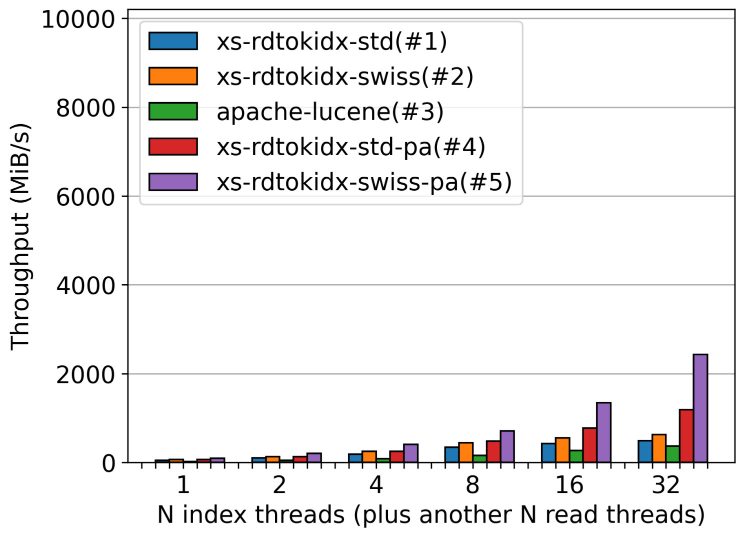


Figure 3.13. End-to-end TF-IDF indexing throughput with increasing number of read and index threads on *32cores-16disks*.

On the system that has many cores and multiple NVMe devices and the most memory channels per NUMA node, we can see that the SCANNS framework can reach very high throughput, when the proper optimizations are used. Figure 3.14 captures this performance, and shows that the un-optimized variants reach a similar performance limit to the previous setups, where the Apache Lucene implementation caps at 478 MiB/sec, the standard Indexer caps at 443 MiB/sec and the Swiss Table Indexer caps at 519 MiB/sec. The plot also shows that when using the memory optimizations to reduce the cache misses and to reduce the number of page faults while also using NUMA-aware scheduling of threads and allocation of memory, the standard Indexer can reach a throughput of 964 MiB/sec, with 24 index threads and 24 read threads, while the Swiss Table Indexer can reach a whopping 9425 MiB/sec, with 192 index threads and 192 read threads. This last result shows that actually in order to build a high-performance indexing engine on a single node computer, one needs a fast search data structure, such as the Swiss Table, but one also needs to design the TFIDF inverted index data structures in such a way that they can benefit from the memory hierarchy.

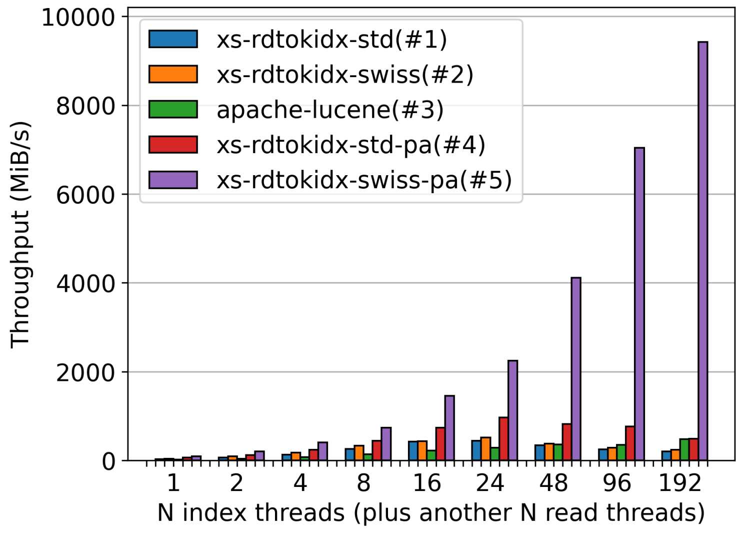


Figure 3.14. End-to-end TF-IDF indexing throughput with increasing number of read and index threads on *192cores-16disks*.

Table 3.2 presents the average search latency of the SCANNS TFIDF implementation that uses the Swiss Table as the search data structure and the efficient design and optimization of the inverted index and compares it against the Lucene variant, on the three different systems. The SCANNS variant exhibits magnitudes lower latency, overall under 500 microseconds, when compared to the Lucene variant that runs search queries on average with latency over 20,000 microseconds. One important observation to make is that even though both variants return the same results with the same TFIDF relevance scores, the lucene variant also sorts the results, while the SCANNS variant does not sort the results. The sorting of the results could add additional overhead to the SCANNS search operations, but optimizing the query engine is the subject of future work.

Table 3.2. TFIDF End-to-end search latency (microseconds).

cores	**64cores-1disk**		**32cores-16disks**		**192cores-16disks**	
	scanns	lucene	scanns	lucene	scanns	lucene
1	237	26143	134	23224	229	20056
2	210	27811	134	23327	233	21747
4	214	30866	142	27952	237	25160
8	180	47981	153	28831	238	29412
16	189	45232	160	36787	248	33601
24	-	-	-	-	269	39004
32	218	51520	173	39524	-	-
48	-	-	-	-	296	53666
64	264	65920	-	-	-	-
96	-	-	-	-	360	64651
192	-	-	-	-	476	134061

3.2.6 Random Access Memory Benchmark. The Indexer seems to be the only component that requires further exploration, as even with all our optimizations the throughput does not reach 10 GiB/sec, even with 192 cores, 8 NUMA nodes and 16 NVMe devices, when the IO-intensive ReaderDriver and the compute-intensive Tokenizer components with optimization can achieve throughput in the 30 to 50 GiB/sec. We argue that the reason for such relatively low performance, even in the presence of optimizations, is the memory-intensive nature of the component and the implied random access present when building an inverted index. We ran multiple random access memory benchmarks, where we copied the elements of an input buffer to an output buffer. Both buffers were pre-allocated in memory and were split into multiple blocks, and the benchmark distributed the blocks to multiple NUMA-aware threads that sequentially read the elements in from each input block and wrote them randomly in an output blocks (see Figure 3.15).

The results that we got for increasing block sizes and increasing number of threads, run on the 192cores-16disks machine, are depicted in Figure 3.16. It is interesting how much performance degrades when the block size exceeds a certain value,

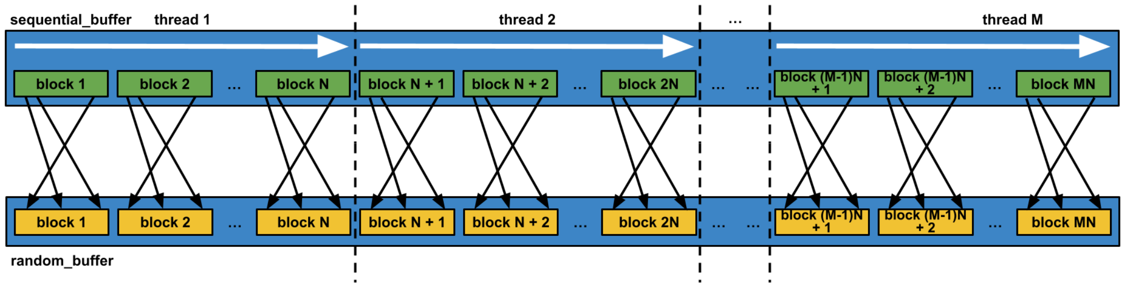


Figure 3.15. Random Access Memory Benchmark Design.

and in the context of re-organizing data when building and inverted index, we argue that it points to a practical upper bound in performance. An implementation of an inverted index does multiple random read and write accesses, and even if there were an implementation that would do a single random access it would not exceed the throughput measured in this experiment. We use this result to argue that the performance of SCANNS is good, when compared to the upper bound random memory access, and excellent when compared to existing or un-optimized solutions.

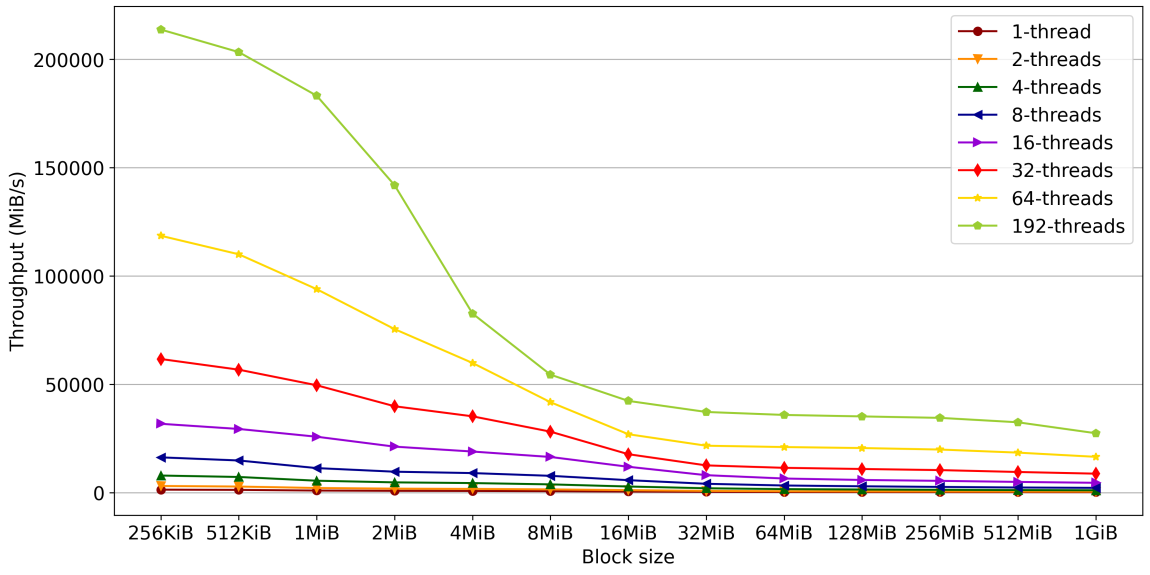


Figure 3.16. Random Access Memory Benchmark.

3.3 Conclusion

We introduced and presented the SCANNS indexing framework to address the problem of efficiently indexing data in high-end systems, characterized by many-core architectures, with multiple NUMA nodes and multiple PCIe NVMe storage devices. We designed SCANNS as a single-node framework that can be used as a building block for implementing high-performance indexed search engines, where the software architecture of the framework is scalable by design. The indexing pipeline is exposed and allows easy modification and tuning, enabling SCANNS to saturate storage, memory and compute resources on different hardware. SCANNS also provides a clear separation between platform or input specific components and platform independent components, achieving good versatility.

We showed that a naive approach to reading data from a modern filesystem, deployed on multi PCIe NVMe SSD storage devices, will lead to drastic performance degradation (up to 6x) and we presented several techniques (e.g., memory pool design pattern and direct IO) that can be used to avoid performance loss. We improved the speed at which the framework can tokenize blocks of data read from disk, by using a hashtable to replace delimiters in the block in O(1) and branchless programming to iterate over the bytes in the block without causing branches or jumps. Since the tokens from the blocks do not have a fixed length, the CPU branch predictor will not be able to identify a pattern and will cause branch mispredictions. The removal of branches from the tokenization process removes the associated cost of branch mispredictions and allows a better use of the CPU pipelines. The branchless tokenizer outperforms the standard C strtok() function while maintaining similar semantics.

We showed that the main bottleneck in inverted index solutions is not the process of reading from disk, or even the process of tokenizing blocks of data read from disk, but the process of re-organizing the data into the form of an inverted index.

Building the inverted index inherently exhibits random access read/write patterns which stress the memory subsystem and ultimately becomes the main bottleneck. However, we showed that with careful index data structure design, such as minimizing pointer indirection inside the inverted index data structure that subsequently reduces the number of cache misses, search engines can still obtain increased performance close to the upper bound supported by the memory subsystem. Finally, combining each of these components (ReaderDriver, Tokenizer and Indexer) with the proposed set of global optimizations (NUMA affinity and huge pages) we showed that SCANNS can achieve up to 19x better indexing while delivering up to 280x lower search latency when compared to Apache Lucene, on configurations with up to 192-cores, 768GiB of RAM, 8 NUMA nodes and up to 16 NVMe drives.

CHAPTER 4

SCALABLE AND CONCURRENT PERSISTENT INDEXING AND SEARCH IN HIGH-END COMPUTING SYSTEMS

In the previous chapter we explored the problem of building in-memory indexes and searching them on single-node high-end computing systems and proposed SCANNS, an indexing framework and a solution to the problem, that allows the user to tune various components of the indexing pipeline and thus saturate compute, memory and storage resources, achieving orders of magnitude higher indexing and search performance when compared to the state-of-the-art Apache Lucene. We also showed that the main bottleneck in building inverted indexes is the memory subsystem and that the design and implementation of the inverted index plays an important role in the performance of the indexing engine.

In this chapter we explore the problems of efficiently building persistent indexes that cannot fit in memory and of efficiently processing TFIDF queries over the built persistent indexes, where the TFIDF scores are computing during query processing time and where the results are sorted by relevance. We propose SCIPIS, a single node framework that can be used as a building block for building high-performance index-based search engines and that expands on the SCANNS framework by redesigning and further optimizing the indexing and search pipeline and by improving the inverted index design. In addition to the existing tuning parameters, that allows the user to adapt the framework to the characteristics of the computing system on which the framework runs, SCIPIS exposes new tuning parameters, that allows the user to adapt the structure of the inverted index and the persistent index to the properties of the input data, allowing SCIPIS to achieve higher indexing throughput when compared to SCANNS and lower TFIDF query latency when compared to Apache Lucene. We evaluated the proposed solution over three types of datasets, log files, scientific papers and data, and supercomputing center file system metadata, and showed that, while the inherent nature of each dataset can affect indexing and search performance, SCIPIS still exhibits good scalability trends. This work's contributions are as follows:

- Extension to the SCANNS framework in order to support efficient persistent indexing and TFIDF search queries;
- Redesign of the indexing and search pipeline, further improvement of the design of the inverted and the addition of new tuning parameters, that allows the user to further adapt the structure of the inverted index to the properties of the input data, in order to achieve high indexing and search performance;
- Evaluation over three kinds of datasets that are commonly found in science (logs, scientific papers and data, and file system metadata);

4.1 Framework Architecture and Design

This section describes the general architecture and the particularities of the SCIPIS framework, including details about the the redesigned inverted index structure but also about the structure of the persistent index as it is stored to disk.

4.1.1 SCIPIS Framework Overview. The SCIPIS framework extends the SCANNS framework and follows a similar architecture. It reuses the *WaveReader-Driver* and *BranchlessTokenizer* components, as the default components responsible with reading data from files and tokenizing them as fast as possible, respectively, and the *DualQueue* component for communication between components that reside on different execution threads. It also replaces the *Indexer* component with two components: *FilePathIndexer* and *TFIDFIndexer*, that are responsible with indexing the file path and the file content, respectively. This separation of components allows the user to finer tune the two aspects of processing indexes but also allows the framework to separately persiste file path and file content indexes. And lastly, SCIPIS adopts a new component, called the *IndexWriter*, that is responsible with reading block of index data and writing them as fast a possible to a corresponding files in a file system. In terms of optimizations, the SCIPIS framework adopts all of the SCANNS components and system optimizations, including: direct IO, memory pool design pattern, branchless programming and the delimiter hashtable, the append cache mechanism and monotonic page allocator, NUMA aware affinity and Linux huge pages; and incorporates two new optimizations that pertain to the characteristics of the input dataset: tunable index depth and split factor.

The SCIPIS framework, similarly to the SCANNS framework, can be used to implement fully functional TFIDF indexing engines, that are optimized for indexing data from a static dataset. By static dataset, we mean that the input dataset does not change while the index is being built (no files are added, remove or modified). The framework can be used to index a changing dataset, but it might end up building an incorrect or imprecise index. Since the output of the framework is a set of files that contain the persistent index, the intention is that, in the future, the SCIPIS framework will support various merge operations that the users could run offline on existing built indexes. Indexes could be optimzied, merged and updated using common set operations, such as: union, intersection and difference. In terms of searching the persistent index, SCIPIS supports now full TFIDF queries, where the query engine will access the persistent index files, will collect and compute the TFIDF scores and will merge and sort the results according to the computed TFIDF score, which will represent the relevance.

4.1.2 Indexing Engine Execution. Figure 4.1 shows the architecture of the SCIPIS indexing pipeline and the flow of data throughout the indexing framework. One distinction from the SCANNS framework, is that SCIPIS takes as an input a collection of input file, processes partial indexes in-memory, and returns as an output another collection of files that contain the re-organized input data, commonly known as an inverted index. The input data is consumed and transformed by the SCIPIS framework by various components that are responsible with a certain task and that target a specific compute resources, and these components are executed by worker threads that are spawned at the start of the program. In terms of execution the framework makes use of three types of worker threads: reader threads, indexer threads and writer threads.

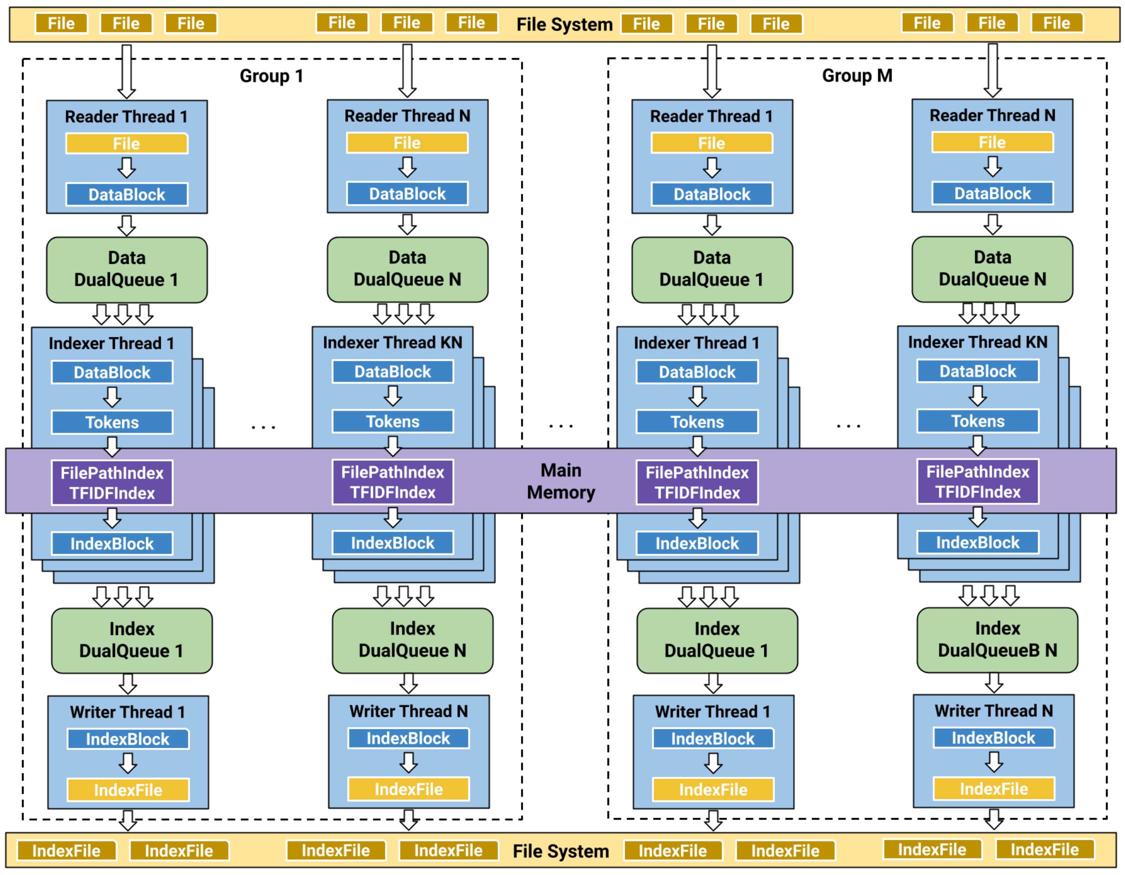


Figure 4.1. SCIPIS framework indexing architecture and pipeline.

Each reader thread manages a *WaveReaderDriver* component, which is responsible for reading variable sized blocks of data from the files stored in a file system as fast as possible and sending them to the indexer threads components through the *DualQueue*. The reader threads are IO-intensive and can be over-provisioned, and in addition to to the blocks of data coming for the content of the input files, the reader threads will also send an additional block containing the full file path to the indexer threads in order to index the file path alongside the contents of the files.

The indexer threads are responsible with receiving blocks of data from the reader threads, indexing the data from the blocks and with sending block of index data to the writer threads to be stored to disk. To accomplish this, each indexer thread manages a *BranchlessTokenizer*, a *FilePathIndex* and a *TFIDFIndex* component, each of them having different roles and using different system resources. The *BranchlessTokenizer* is responsible with breaking down a block of data into a list of tokens delimited by one or more delimiter characters and is a CPU-intensive component. The *FilePathIndex* component is a memory-intensive component and is responsible with computing a file index from the full file path and storing the index and the full file path into the inverted index in order to be retrieved during search operations. The file content, under the form of the list of extracted tokens, is then indexed by the *TFIDFIndex*, that is also a memory-intensive component, and that indexes the tokens and keeps track of the term frequencies and inverse document frequencies necessary for compting a relevance score. The *FilePathIndex* and the *TFIDFIndex* sizes are determined at program startup and usually reflect the amount of memory available in the system per thread, but also the ration between full file path size and file content size. For example, if the input data contains many small files that only contain a reduced number of tokens, the *FilePathIndex* should be allocated more memory than the *TFIDFIndex*, and vice versa for the converse. Once either the *FilePathIndex* or the *TFIDFIndex* reached or is over 90% of the index capacity, the index in question will go over a serialization step, in which the index will be organized into blocks of data that can then be sent to the writer threads to be written. The current implementation of the serialization of indexes actually only serializes the full file path and writes the remainder index information in binary format on disk. This means that the index cannot be simply copied from one architecture to another, since the endianess of the data might be different. But this problem can be simply overcome by using an efficient serialization library, such as the Google Protocol Buffer in order to standardize the index format on disk. It's also worth mentioning that during the serialization step, the indexer components become CPU-intensive, rather than memory-intensive, because serializing the index implies reading and writing data sequentially from the inverted index to the block buffer. Thus this actually allows the framework to also over-provision the indexer threads to the number of hardware-threads and not just at the number of physical cores, and still yield good performance.

The last type of thread is the writer thread, which manages an instance of *IndexWriter* component, that is responsible with receiving index blocks from the indexer threads through a *DualQueue* and writing them to the corresponding file on the file system. The *IndexWriter* is a IO-intensive component and can be over-provisioned without majorly impacting the overall performance of the framework.

In practice we observed that the reader and writer threads are orders of magnitude faster than the indexer threads, that is why there is a one to many mapping between reader/writer threads and indexer threads. This also allows the user to better load-balance indexing files with significant varying size at the cost of index size and, subsequently, of query latency. SCIPIS groups reader, writer and indexer threads into groups of threads that can be scheduled together on the same NUMA node, thus minimizing performance degradation due to inter-NUMA communication.

4.1.3 Inverted Index Design. Previously, in SCANNS, we showed that the inverted index design is crucial to the performance of the indexing process, noting that an efficient inverted index requires both an efficient search data structure but also an efficient auxiliary data structure. In that regard, SCANNS showed that when using the Google Swiss Table and the append cache mechanism, one can reduce the number of memory indirections when create a TFIDF index and significantly improve the performance of the indexing process. Since SCANNS was designed to index data as fast as possible in-memory, we permitted ourselves to use pointers to reference to file paths and other parts of the index and linked lists to store the elements of the auxiliary inverted index. But this solution would not work if we need to persist the index on disk, since the pointer values would not easily translate to offsets to disk and the linked list would yield low serialization performance due to traversing the memory using pointers to the next element.

In SCIPIS we proposed a new inverted index design that solves all of the issue stated previously and allows the framework to create an index that can easily be serialized to disk and that exposes a new dimension for adapting the inverted index to the input data. Figure 4.2 shows the design of the proposed inverted index. We eliminated all of the pointer references and replaced them with offsets to buffers and/or with unsigned integer values, as is the case for file and token identification or index. SCIPIS still uses the Google Swiss Table as the search data structure and the elements of the hash table are direct references to *IDFIndexEntry* data structures. By direct references, we meant that in order to perform a lookup over the hash table, the element is not a pointer to another region of memory that stores the actual element, but returns a reference to the element itself. This means that the memory for storing *IDFIndexEntry* elements is managed by the data structure itself. We still use the append cache mechanism to reduce the number of memory indirections, that relies on the assumption that the dataset is static in order to function properly. SCIPIS changes the way the *TFIndexEntry* is allocated and stored in memory. Since the inverted index is designed from the start to support persistent indexes that are larger than the main memory of the system, we can relax the requirement to try to fit as much data as possible inside the in-memory index, which allows us to better organize the *TFIndexEntry* elements, which represent the auxiliary inverted index data structure. Instead of creating a linked list of elements that are all chained together, we allocate small arrays of *TFIndexEntry* elements that then get assigned to each *IDFIndexEntry* element. When either the hash table or any array of *TFIndexEntry* element reached capacity, the inverted index signals the parent component that it is ready to serialize and flush to disk. By keeping the *TFIndexEntry* elements, which store the file index and the frequency information for the word that is found in that particular file, contiguously in memory, we enhance both the serialization and search performance at the cost of space utilization.

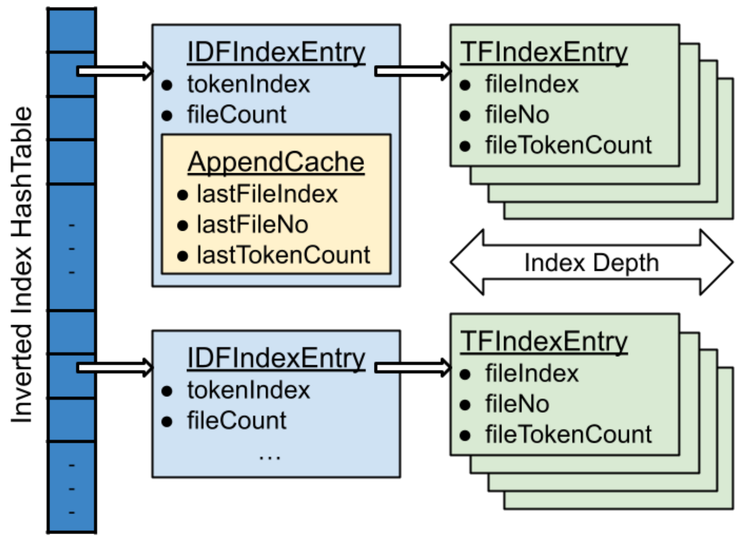


Figure 4.2. SCIPIS framework indexing architecture and pipeline.

In order to overcome the space utilization problem, we expose to the user a new tuning parameter that we call *index depth*, that allows the user to specify how big will the contiguous array of elements will be at program startup. The user only needs to know how big can the *TFIDFIndex* be, how big are the input files on average and what is the mapping between reader threads and indexer threads, to determine how many files would be able to fit inside the index and thus to how much to set the index depth. This additional tuning parameter allows the user to further tune SCIPIS and increase the performance of the indexing process by giving the framework some information regarding the system and the input dataset.

While the *index depth* parameter can help the user adapt the inverted index to the input data, the parameter alone does not capture the entire essence of the problem. If no assumptions regarding the input dataset are made or if no useful knowledge about the structure and patterns of the input dataset are known, it will be hard to create a general inverted index that will work well in all scenarios. However, if the user knows details regarding the number of file, the file sizes, the number of tokens per file and the total number of unique tokens from the entire dataset, then the inverted index structure can be additionally optimized, allowing the framework to use the in-memory index space more efficiently, while minimizing the number of index flushes to disk and the output index size. That information could be provided either offline, through an quick analysis of the input dataset, or online through space stealing or defragmentation techniques. SCIPIS does not implement these techniques and they are the subject of future work, but this work points towards an aspect of data indexing and search that is often underlooked and that could lead to a dynamic inverted index design that can yield even higher indexing and search performance.

4.1.4 Persistent Index Structure. When either the *FilePathIndex* or the *TFIDFIndex* reaches or goes over 90% of its allocated capacity, the index gets serialized and then sent into blocks to be written to disk. Figure 4.3 describes how the index is being store on disk for a particular index thread. Each index thread creates its own persistent index, that represents a local or partial index, implying that when a search query is being launched, the query engine will need to combine the results from each index thread in order to return a global view of the index. Each index thread will need to flush both an index for the file paths and an index for the term frequency and inverse document frequency information. The file paths index will be stored under the *file_index_<thread id>*directory, while the TFIDF index will be stored under the *tfidf_index_<thread id>*. For each flush of any index type, the framework creates one segment data file and one or more segment metadata files. The segment metadata files will contain the hash table entries for each index type. For example, for the TFIDF index, the segment metadata files will contain a list of *IDFIndexEntry* elements, written down in binary format. Since the size of *IDFIndexEntry* data structure is know, there is no need for there to be a delimiter between the elements, and the end of the list is marked with a special entry that has a file count of zero. The segment data files will contain the auxiliary data structure elements. In the case of the file path index, the auxiliary data structure is the actual full path of the files delimited by new line, and the corresponding element from the metadata file will store an offset to the position where the full file path can be found in the data file. For the TFIDF index, the segment data file contains a list of lists of *TFIndexEntry* elements of size *indexDepth*, with the entries from the metadata file containing and offset to the beginning corresponding list. In order to control the size of the segment metadata files, SCIPIS exposes the *split factor* parameter that allows the user to decide into how many segments to split the hash table key space, and thus have the ability to improve search or indexing performance at the cost of the other. Each key will be stored to the corresponding segment metadata file, minimizing the search space when computing search queries. Each index flush will create a new group of segment data and segment metadata files, that will be required to be queried when searching the persistent index.

One benefit for building the persistent index in this way, is that the index can be searched offline, while SCIPIS is building the persistent index or after the framework finished executing. Searching the index is trivial, as it only requires the query engine to search, for a given term, to access a subset of segment metadata and segment data files. For each query, the query engine will need to search the segment metadata files for offsets to the term-frequency information. After the offsets are retrieved the term frequency information is extracted from the segment data file using the offsets. While the term frequency information is being retrieved, the query

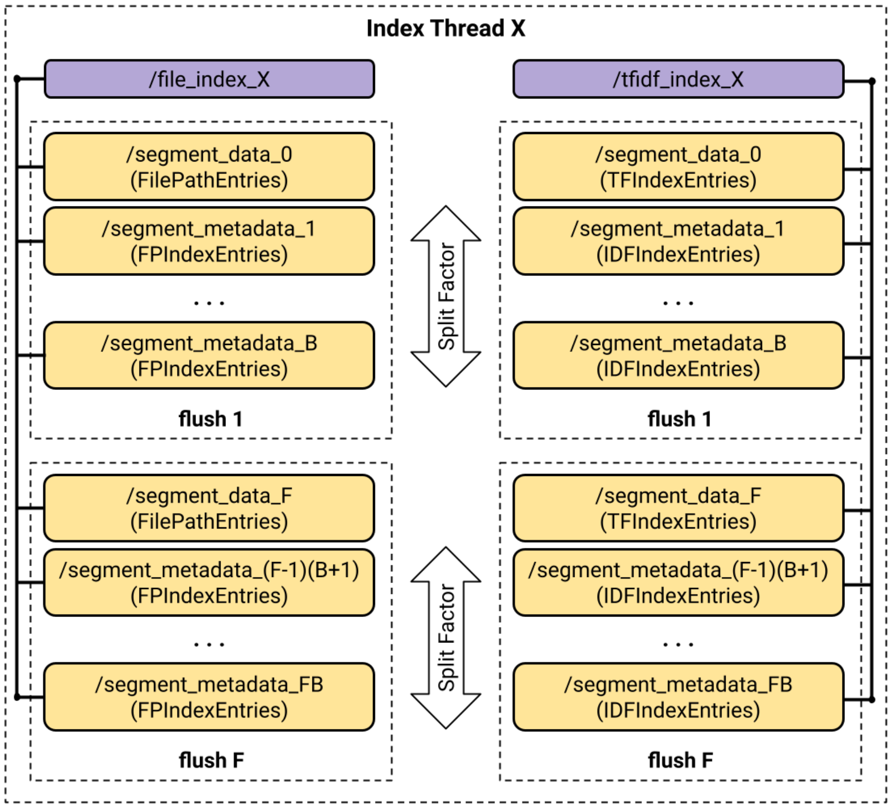


Figure 4.3. SCIPIS framework indexing architecture and pipeline.

engine can start sorting the results and filtering the ones that have a low score, thus reducing the amount of data that needs to be merged. Finally, once the all of the persistent indexes for each index thread have been searched, the query engine will combine the results from each index thread, using an n-way merge technique, into one final list of documents and their scores for a given query.

4.2 Performance Evaluation

This section contains the performance evaluation of the SCIPIS framework, which includes the end-to-end indexing performance of the proposed framework, the TFIDF search performance, the tuning experiments, but also descriptions of the experimental setup and the datasets used for the experiments.

4.2.1 Experimental Setup. For the purpose of evaluating SCIPIS experimentally we used two single-node high-end systems deployed on the Mystic Cloud, an NSF-funded testbed designed to study system re-configurability and to conduct computer systems research. Both of the machines have a high number of PCIe NVMe storage devices allowing for fast storage access in terms of both read and write, sequential and random. The difference between the two chose machines is the number of cores, memory channels and NUMA nodes present, and they were a selected in order to showcase the scalability trends of the SCIPIS framerwork both on a small machine, but also on a big machine. The small machine (*32cores-16disks*) represents an accessible machine that any computing facility could enlist, while the big machine (*192cores-16disks*) is more akin to what a supercomputer node would look like, in terms of CPU and memory capability. Table 4.1 shows the hardware details for each system used in the performance evaluations.

Table 4.1. Mystic Cloud machines used for the experimental evaluation and their specifications.

machine name	**processors**	**cores**	**memory**	**nvme storage**
32cores-16disks	2 x Intel Xeon Gold 6130	32	192GiB	16 x Samsung 970 EVO
192cores-16disks	8 x Intel Xeon Platinum 8160	192	768GiB	16 x Intel Optane 900P

In order to achieve high execution performance, we configured the scheduling governors to performance and enabled turbo-boost on all machines. When we ran experiments we also configured the listed storage devices to be accessed in exclusive mode, having the OS of these two machines run from different storage devices, all of this in order to eliminitate performance degradation caused by any interference. Since the PCIe NVMe drivers are spread evenly across the NUMA nodes, we grouped the devices by NUMA node and configured Linux software RAID0 with XFS on them. When running experiments, we distributed the data evenly accross the NUMA nodes and made sure that threads only accessed data from their own NUMA node.

In terms of software, both the machines ran Ubuntu 22.04 LTS with Linux Kernel 5.15 and g++-12. For the Google Swiss Table library we used version 20230125.3 lts from the abseil library. The SCIPIS framework is implemented in C++20.

4.2.2 Evaluation Datasets. For the evaluation of the performance of SCIPIS we used a total of 5 datasets, that represent 3 different scenarios or dataset types, and that have different properties. The used datasets, alongside the properties and characteristics of the datasets, can be seen in Table 4.2. We picked a diverse range of datasets in order to showcase the scalability and performance of SCIPIS, and to show that while the performance can change from dataset to dataset, the scalability trends remain similar.

Table 4.2. Evaluation datasets and characteristics.

dataset name	**size**	**number of files**	**file sizes**	**dataset type**
hdfs	18GB	40	200-800MB	logs
thunderbird	31GB	240	100-600MB	logs
windows	27GB	738	50-200MB	logs
thepile	1.1TB	5500	100-300MB	scientific
fsdumps	1.2TB	300000	4MB	metadata

The first type of dataset that we chose for the experimental evaluation is log data. Log data is a type of dataset that can easily be found in any computing systems and is especially more prevalent in supercomputing centers. A supercomputer can contain thousands, if not tens of thousands of nodes, that each can generate a considerable amount of log data, that when combined with the log data generated by the applications that run on supercomputers can easily reach large volumes, that becomes hard to search through without the help of an index-based search engine. Thus we decides to evaluate SCIPIS over log datasets, because it is a representative scenario. We used the *hdfs* cluster, *thunderbird* supercomputer, and *windows* operating system logs form the Loghub collection [35].

For the second type of dataset we decided to evaluate SCIPIS over a collection of scientific datasets. The Pile [36] collection contains publications, websites and books, from various fields of science, including: medicine, law, mathematics etc. The Pile collection is very representative to the kind of data that a scientific search engine would need to index and provide search over and we chose this dataset because of its high diversity in terms of topics and vocabulary.

Since many supercomputing centers use parallel and distributed file systems to organize data, we decided to use include a dataset that is representative to performing search over the metadata of the file system. Scientists and engineers often use various naming conventions for directories and files in order to make searching through browsing possible, but the same conventions that they use pose an interesting scenario when it comes to indexing and searching scientific data. Thus we decide to evaluate SCIPIS over the file system metadata provided from a supercomputing center, that we will refere to as the *fsdumps*.

4.2.3 Performance Tuning. Figure 4.4 shows the results of tuning the index depth parameter over the three log datasets on the *32cores-16disks* machine. In this experiment, SCIPIS was configured to run with 4 reader threads, 4 writer threads and 64 indexer threads, with a block size of 1MiB, a file path index maximum size of 128MiB and a TFIDF index maximum size of 1GiB.

For the *hdfs* dataset, it can be seen that the framework achieves the highest indexing throughput of 4.3 GiB/s, when configured with an index depth of 2, and that with increasing index depth the performance decreases. This behavior is attributed to the *hdfs* dataset having a reduced number of large files, that end up causing the TFIDF index to flush more often with increasing index depth. Thus, for the *hdfs* dataset we used an index depth of 2 for the rest of the experiments.

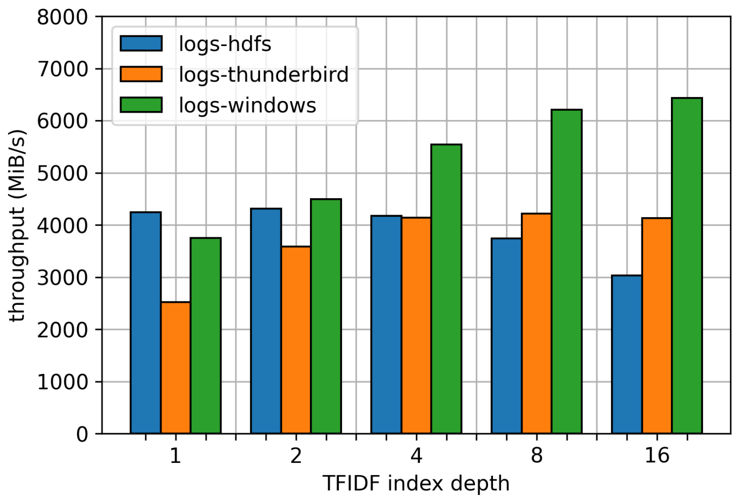


Figure 4.4. Tuning the index depth on *32cores-16disks*.

In the case of the *thunderbird* dataset, the framework achieves the highest indexing throughput of 4.2 GiB/s, when configured with an index depth of 8, and that with small index depths the performance degrades. The *thunderbird* dataset is composed of a balanced number of files of various sizes, smaller in size than the files from the *hdfs* dataset, which when indexed occupy a smaller space in the index. Thus with a higher value for the index depth, the framework is capable to achieve a better utilization of the memory space for the index and reduce the number of index flushes. For the rest of the experiments we used an index depth of 8 for the *thunderbird* dataset.

The difference in performance between different index depths can be seen for the *windows* dataset, which is comprised of many small files. If the index depth is small, then SCIPIS will cause many flushes to the disk and will yield degraded performance, but if the index depth is large, the index memory space will be better utilized, thus minimizing the number of index flushes. For an index depth of 1, the SCIPIS framework performs indexing at 3.7 GiB/s, while for an index depth of 16, the framework achieves an indexing throughput of 6.4 GiB/s, which is a 73% improvement in performance.

4.2.4 Indexing Throughput (SCIPIS vs SCANNS). We compare SCIPIS to SCANNS, in terms of indexing throughput, on the *fsdumps* dataset in order to showcase how the proposed inverted index design and indexing pipeline, along with the newly introduced optimizations, can yield better performance. Figures 4.5 and 4.6 show the the indexing throughout of both SCIPIS and SCANNS on the *32cores-16disks* machine and the *192cores-16disks* machine, respectively. In both plots we did not include the numbers for SCANNS running with 64 indexer threads, because SCANNS was exhibiting performance degradation due to CPU core over-provisioning. SCIPIS did not exhibit the same performance degradation, and actually showed improved performance when using all of the hardware threads available, because the indexer threads switch between memory-intensive computation to CPU-intensive computation when serializing the index.

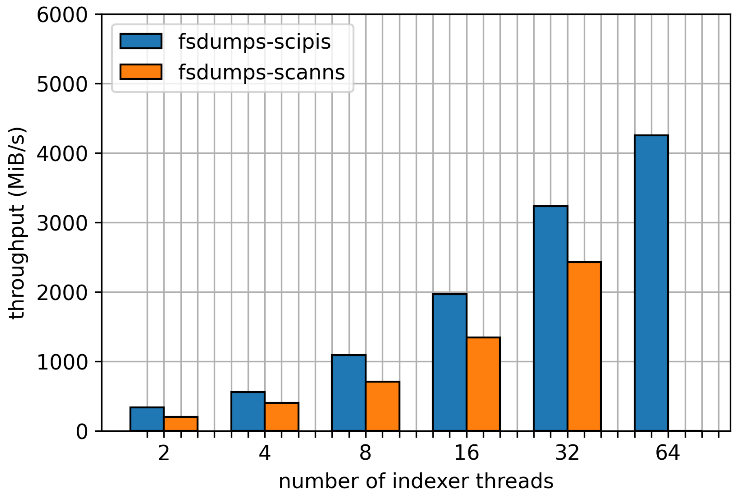


Figure 4.5. SCIPIS vs SCANNS indexing throughput the *32cores-16disks* machine.

From Figure 4.5 we can see that, on the *32cores-16disks* machine, the indexing throughput of SCANNS increased from 200 MiB/s with 2 indexer threads to 2.4 GiB/s with 32 indexer threads, while for SCIPIS the indexing throughput increased from 340 MiB/s with 2 indexer threads to 3.2 GiB/s with 32 indexer threads and to 4.2 GiB/s with 64 indexer threads. SCIPIS manages to achieve a performance boost of

75%, while at the same time building a persistent index and indexing approximately 12 time more data than SCANNS.

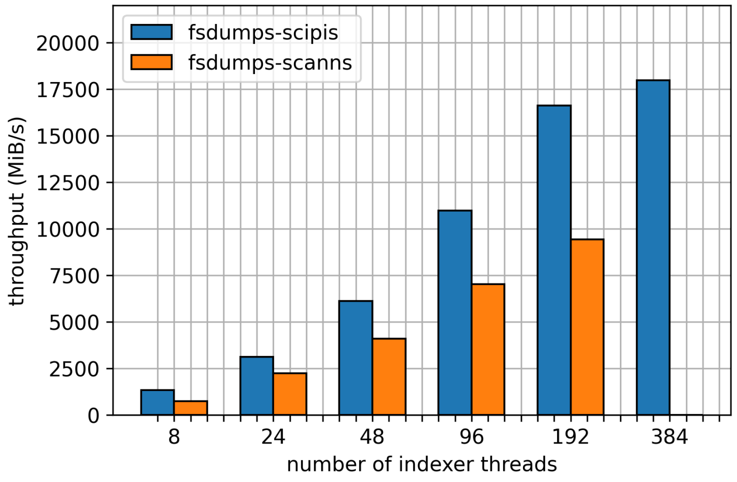


Figure 4.6. SCIPIS vs SCANNS indexing throughput the *192cores-16disks* machine.

On the *192cores-16disks* machine, SCANNS yielded an indexing throughput of 740 MiB/s with 8 indexer threads, that increased to 9.7 GiB/s with 192 indexer threads. The indexing throughput of SCIPIS increased from 1.3 GiB/s with 8 indexer threads to 16.6 GiB/s with 192 indexer threads, all the way to 17.9 GiB/s with 384 indexer threads. SCIPIS outperforms SCANNS by exhibiting a performance increase of 84%, while building a persistent index and indexing approximately 24 times more data that SCANNS.

4.2.5 Indexing Throughput (Various Datasets).

SCIPIS is not impervious to performance variation when it comes to indexing throughput, and the characteristics and structure of the input data does influence the overall performance of the proposed framework. Although indexing throughput is influenced in part by the input data, what is important is the scalability trend and how the system performs with increasing computing resources. Figures 4.7 and 4.8 show the results of the scalability experiment that we ran over three kinds of datasets and on both the *32cores-16disks* machine and the *192cores-16disks* machine, respectively. For each dataset we manually tuned the SCIPIS in order to yield the best indexing throughput, accordingly to the specifications of the computing systems but also to the properties of the datasets.

In Figure 4.7 we can see the indexing performance with increasing number of indexer threads for all five datasets. Across the log datasets, the *windows* dataset exhibits the best performance with an indexing throughput of 7.2 GiB/s with 64 indexer threads, while the *thunderbird* and *hdfs* datasets yield only 5.2 GiB/s and 4.6 GiB/s on the same configuration. This is explained by the properties of the three datasets, where the *windows* dataset has more and smaller files than the other datasets, that have a similar total size, which allows for a better utilization of the index memory space and a reduced number of index flushes, especially when also tuning the index depth parameter. When looking at the *fsdumps* dataset, SCIPIS yielded an indexing throughput of 4.2 GiB/s with 64 index threads. One would expect an even better performance that the *windows* dataset, because of how small the files are, but in this scenario, the difference in performance stems from the diverse vocabulary that the *fsdumps* dataset has in comparison to the *windows* dataset. The log datasets have many terms that repeat many times, such as time stamps and dates, but also repeating errors and error names and identifiers, while the *fsdumps* dataset has a uniform distribution of term frequencies. That meaning that the number of high frequency terms is similar to the number of medium and low frequency terms, because of the hierarchical nature of the file system metadata that is being indexed. And right now SCIPIS does not know how to adapt to the term frequency and uniqueness that characterizes a dataset. The *thepile* dataset is an extreme case of where the number of unique terms is high and their frequency low, when compared to the total number of terms. Also for this dataset, we only selected half of the total number of files, that ended up being the larges files in the dataset. So the *thepile* dataset exhibits a decreased indexing throughput of 3.2 GiB/s with 64 indexer threads both because the data set has a high term variety but also because the files are large.

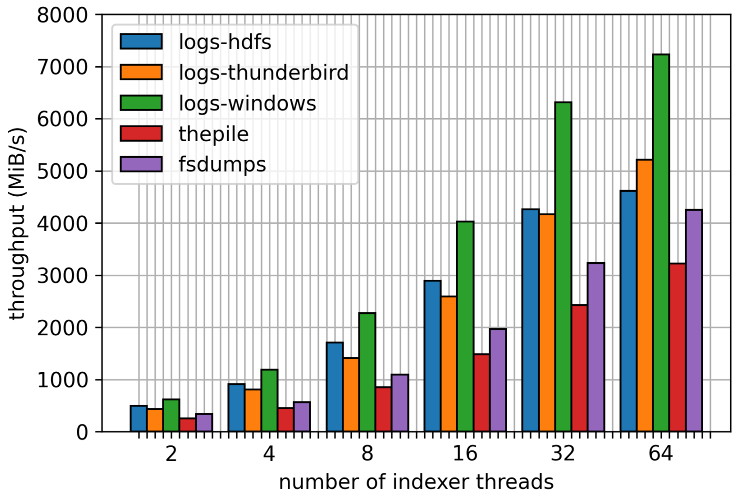


Figure 4.7. SCIPIS indexing throughput the *32cores-16disks* machine.

In the case of the *192cores-16disks*, Figure 4.8 contains the results of the indexing throughput performance evaluation with increasing number of indexer threads. The *hdfs* and *thunderbird* datasets were not included on this machine, because the number of files in these two datasets is less than 384 hardware threads, which would create load-balancing issues and would result in incorrect performance numbers. It can already be seen that in the case of the *windows* dataset, it looks like the indexing throughput of SCIPIS stagnates at 15.5 GiB/s when running with 96 indexer threads and drops to 13.4 GiB/s with 384 indexer threads, and this is caused by the fact that the files do not have a uniform size and that they cannot be distributed across the indexer threads uniformly. The *thepile* and the *fsdumps* datasets do show continuous increase in indexing throughut with increasing number of indexer threads, reaching 19.1 GiB/s and 17.9 GiB/s, respectively. In this scenario, SCIPIS computed the index over the entire *thepile* dataset and managed to achieve comparable performance to the *fsdumps* dataset. This is because, the second half of the *thepile* dataset contains a increased number of small files and a smaller vocabulary, when compared to the first half, allowing the proposed framework to catch up in terms of performance with the *fsdump* dataset.

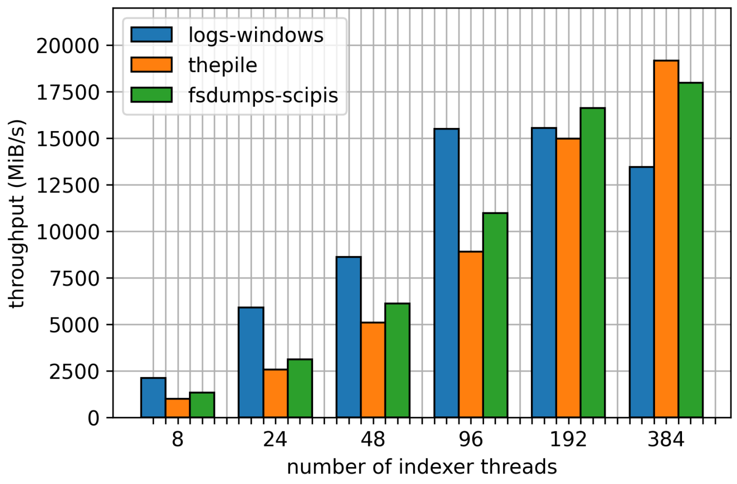


Figure 4.8. SCIPIS indexing throughput the *192cores-16disks* machine.

4.2.6 Search Latency. Since the SCIPIS framework supports TFIDF queries, that will compute the TFIDF scores for the returned documents given the query term and sort and select only the top relevant documents in order to satisfy the information need, we also conducted experiments where we measured the performance of the query engine. We compared the SCIPIS search results to the Apache Lucene search results collected from the SCANNS work. For each experiment we selected 1000 random terms from the respective dataset and performed 1000 individual queries 5 times.

Table 4.3 contains the average search latency measured in microseconds. We can see that on both machines, SCIPIS outperforms Apache Lucene in processing TFIDF queries, and on all datasets of similar size. We decide to run the search evaluation on datasets of similar size, because the persistent index size does influence the query processing latency of SCIPIS, and this way there is a more comparable comparison to Apache Lucene. From this evaluation it can be observed that while a greater number of indexer threads yields a higher indexing throughput, it also contributes to a lower search latency. For example, for the *windows* dataset on the *32cores-16disks* machine, with 2 indexer threads, SCIPIS can run a query in 2.5 milliseconds, while with 64 indexer threads, the query latency increases to approximately 43 milliseconds.

Table 4.3. SCIPIS vs Apache Lucene TFIDF search latency (microseconds).

cores	**32cores-16disks**				**192cores-16disks**	
	hdfs	thunderbird	windows	fsdumps-lucene	windows	fsdumps-lucene
2	3958	5823	2557	23327	-	21746
4	3782	6707	5033	27952	-	25159
8	4138	10719	9372	28831	2568	29412
16	6309	10684	-	36786	-	33601
24	-	-		-	3219	39004
32	9168	12526	-	39524	-	-
48	-	-	-	-	7135	53666
64	18246	17835	-	42966	-	-
96	-	-	-	-	8936	64650
192	-	-	-	-	8980	134061
384	-	-	-	-	14083	117113

On the *192cores-16disks*, SCIPIS exhibits a query latency of 2.5 milliseconds with 2 indexer threads, that increases to a query latency of 14 milliseconds with 384 indexer threads.

4.3 Conclusion

In this work we proposed and presented SCIPIS, a single-node indexing and search framework that can efficienlty build persistent indexes that cannot fit into memory and efficiently search persistent indexes on high-end computing systems charactersized by many cores, multiple NUMA nodes and multiple PCIe NVMe devices. SCIPIS extends the SCANNS framework by adding support for writing partial indexes to disk and redesigns the inverted index and the indexing pipeline, obtaining higher indexing throughput than SCANNS and lower TFIDF search latency when compared to Apache Lucene. We also evaluated SCIPIS on three kinds of datasets, namely log files, scientific data and supercomputing file system metadata, and showed that SCIPIS scales well, achieving 1.8x better indexing performance than SCANNS and 8x better search performance than Apache Lucene.

We showed that the indexing pipeline can be further improved, by delegating the process of serializing the partial index to the indexer threads. By doing so, the indexer threads can change its processing pattern from memory-intensive to CPU-intensive, allowing the framework to safely over-provision the indexer threads and not exhibit performance degradation. We also showed that the inverted index can be further optimized by introducing the *index depth* parameter. The index depth allows the user to adapt the inverted index to the properties of the input dataset, which consequently enables the framework to more efficiently use the index memory space but to also reduce the number of index flushes to disk, and subsequently improves indexing performance. In this work we also introduced the *split factor* parameter that allows the user to control the space and size of the persistent index in order to improve search performance at the cost of indexing performance.

Lastly, designing SCIPIS and the new optimization, opened up the possibility of further exposing tuning parameters that are not only system related, but also pertain to the nature of the input data itself. As future work, it would be worth to explore capturing the uniqueness of terms and the size of the vocabulary extracted from the dataset and use that information manually or automatically to further improve indexing and search performance.

CHAPTER 5
EVALUATION OF A SCIENTIFIC DATA SEARCH INFRASTRUCTURE

Given the large amounts and the variety of data found in scientific large-scale file systems, it stands to reason to try to bridge the gap between various data representations and to build and provide a more uniform search space. In this chapter we present a performance evaluation of ScienceSearch, a production search infrastructure for scientific data that uses machine learning to automate the creation of metadata tags from different data sources, such as published papers, proposals, images and file system structure. The performance evaluation focuses on on scalability trends in order to better understand the implications of performing search over an index built from the generated tags.

Scientific instruments and facilities are generating data at a rapid pace. Future scientific discoveries will rely on insights derived from data, making search capabilities critical. However, many science users rely on manual browsing or tools such as find or grep which provide limited capabilities for large scale data and scientific file formats.

`ScienceSearch` [37, 38] is a scalable search engine that uses a wide range of machine learning techniques (natural language processing, deep learning etc) to automate metadata generation from different data sources, such as published papers, proposals, images and file system structure. The current implementation is deployed to provide search over data obtained from NCEM (National Center for Electron Microscopy) that includes around 5TB of data (500K images). Users can interact with the ScienceSearch infrastructure through a dedicated web interface that accepts a text query and returns a list of relevant images, papers and proposals within seconds. The system is deployed on a container service platform, called Spin, at the National Energy Research Computing (NERSC) Center, a HPC facility. Such container service platforms have more recently been deployed at HPC facilities to support science gateways, workflow managers, databases, and other services. Deploying the ScienceSearch infrastructure on Spin allows us to leverage the high performance large filesystem at NERSC while allowing users to query the data through a web interface.

Previous work showed that ScienceSearch is capable of generating relevant metadata and providing low-latency high quality query results for our initial use case from NCEM. The ScienceSearch infrastructure allows us to understand and address key questions related to the scalability of our infrastructure for increasing data sizes and number of users. The performance of dedicated search infrastructures greatly depends on the ability to simultaneously serve multiple types of queries while keeping search latency as low as possible. Previous work found in literature has explored scalable search in the context of Internet data [39, 40]. However, these results cannot be directly applied to scientific data in HPC due to the unique characteristics of the scientific data (i.e. size, formats, volume) and the performance considerations of HPC environments. The focus of this work is the evaluation of ScienceSearch towards understanding design implications for current and future scientific search infrastructures [3]. We perform a through evaluation of the current infrastructure and discuss our experiences and results.

In our evaluation, we consider scientific data search queries that can be roughly classified as two types: *targeted* and *open-ended*. A search query may be *targeted* where the query results in few hundreds results. While relatively rare in our infrastructure, scientists might also issue *open-ended* queries as part of data exploration where a search might return thousands or even millions of results. We evaluate ScienceSearch's underlying infrastructure performance under distinct search scenarios that emulate both *targeted* and *general/open-ended* queries.

[3]The content of this chapter is gathered from published research [41]

We present a thorough performance evaluation of ScienceSearch's infrastructure focusing on scalability trends under different query types. We conduct an in-depth analysis to identify the contribution of each search phase. For our experiments, we deploy ScienceSearch both on a shared supercomputer infrastructure and on a dedicated testbed. Our evaluation considers latency, processing rate, memory utilization and query throughput. Our evaluation also provides insights towards building generalized search infrastructure for future systems, including performance considerations for container platforms, need for load-balancing and parallelism, adaptive resource scaling, and data representation in memory. Our performance evaluation scenarios answer the following questions:

- How does the type of query (i.e. targeted or open-ended) affect search latency and underlying system requirements (such as memory footprint and CPU consumption etc)?
- Is ScienceSearch able to scale and serve parallel search queries independent of the underlying hardware infrastructure?
- What is the limit in terms of concurrent search queries that ScienceSearch infrastructure can serve?

5.1 ScienceSearch

ScienceSearch system's architecture features five components: *user interface*, *data import*, *metadata extraction*, *search engine*, and a *database*. Users can express their data needs in the form of a text query and receive back a list of relevant images, papers, and proposals through the *user interface*. The *data import* component is responsible for ingesting and storing scientific data and existing related metadata (e.g. an image and its location in the file system) in the *database*. Currently, the system supports images, papers, proposals, and calendar entries as data sources. The *metadata extraction* component uses machine learning to automatically generate the metadata tags. These tags are stored in the *database*. The *search engine* uses a model that connects metadata generated from the *metadata extraction* component with the imported images, papers and proposals.

The system utilizes *Spin*, a container deployment platform designed for HPC environments, in order to gain access to HPC storage, compute and network resources and provide users with low-latency search results. The data containing the scientific images resides on a shared file system hosted on the supercomputer infrastructure that is accessible from Spin. With *Spin* users can deploy their own container images in separate namespaces that can be located across different HPC nodes. The platform's orchestration layer facilitates access to local and remote (NFS) storage. Inter-container communication relies on underlying HPC network resources and the encapsulation mechanisms provided by *Spin*'s orchestration layer.

An overview of ScienceSearch's container-based architecture can be found in Figure 5.1. ScienceSearch groups all data-related components (i.e. the *data import*, *metadata extraction* and *search engine*) in a single container instance while the *user interface* and *database* are deployed as separate instances. The *backend* and *database* instances can be deployed separately and independently of the *user interface* guaranteeing portability and execution at different locations adjacent to HPC environments (e.g. near the scientific instruments where the data is generated). Furthermore, all data-related components can be executed separately and can be triggered by events (e.g. creation of new data).

In this section, we describe the search infrastructure in detail. We describe the search steps from when the user issues a search query until the list of matching results is returned.

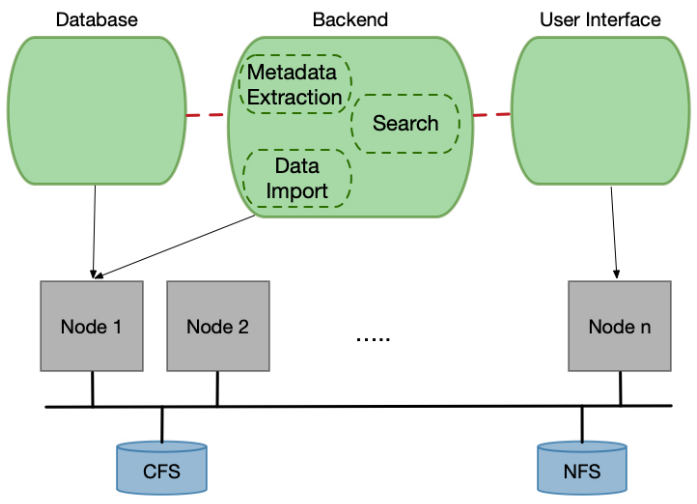


Figure 5.1. ScienceSearch container-based architecture and interaction with HPC resources through *Spin*. Container instances are denoted with light green and inter-container communication is represented with a dotted red line. HPC resources are in grey (physical nodes) and blue (remote and local storage). Arrows show where each instance is physically deployed.

5.1.1 Search Lifecycle. Search is conducted in three steps: *user query processing*, *comparison with stored metadata* and finally *result aggregation*. The steps are depicted in Figure 5.2.

Once a user issues a search query, the user query processing step divides the query into words (i.e. for queries that contain multiple terms) and the extracted words are lemmatized (i.e. reduced to root stems). The lemmatized terms are then passed to the next stage where the comparison with the metadata occurs. During comparison with stored metadata, the query terms are compared against metadata that are stored in a database system and the results are ranked. The database lookup, comparison and the intermediate result ranking is computationally intensive. In order to optimize performance and reduce latency inducing bottlenecks, we opt for a parallel environment with multiple workers that are managed by a master process.

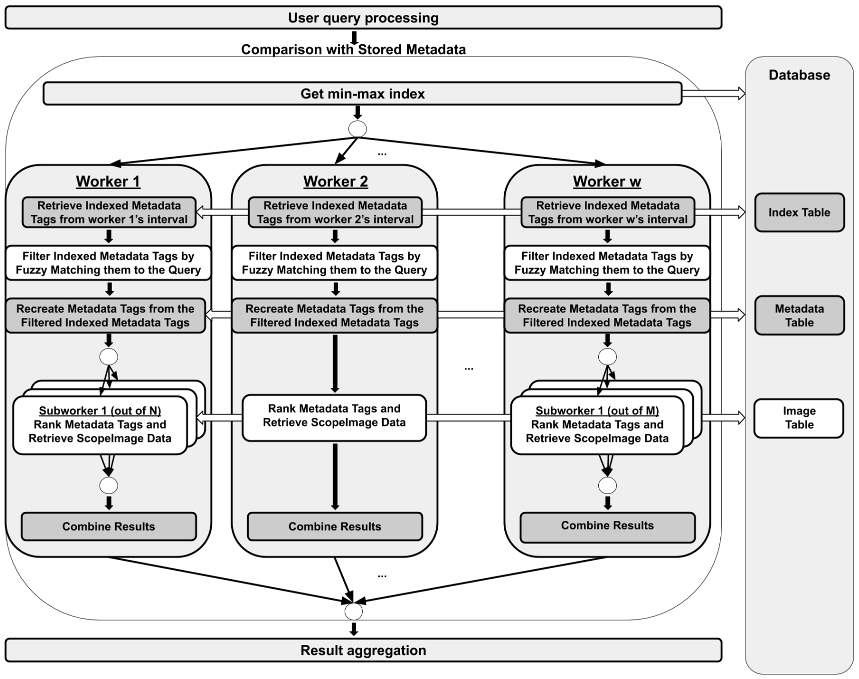

User query processing
Comparison with Stored Metadata
Get min-max index
Database
Worker 1
Worker 2
Worker w
Retrieve Indexed Metadata Tags from worker 1's interval
Retrieve Indexed Metadata Tags from worker 2's interval
Retrieve Indexed Metadata Tags from worker w's interval
Index Table
Filter Indexed Metadata Tags by Fuzzy Matching them to the Query
Recreate Metadata Tags from the Filtered Indexed Metadata Tags
Metadata Table
Subworker 1 (out of N) Rank Metadata Tags and Retrieve ScopeImage Data
Rank Metadata Tags and Retrieve ScopeImage Data
Subworker 1 (out of M) Rank Metadata Tags and Retrieve ScopeImage Data
Image Table
Combine Results
Result aggregation

Figure 5.2. Parallel Architecture for *Comparison with Stored Metadata* step. A master process slices the database index in W slices and spawns W workers respectively. Each worker interacts with the database in order to fetch and rank intermediate results.

The final step before the results are returned back to the user is result aggregation. In this step, the master process combines each individual worker's ranked results in a list and sends that list back to the user through the user interface. Finally, the master process terminates all worker processes.

5.1.2 Databases. One of the main components of ScienceSearch architecture is a database which stores information that is matched against the query string. The information stored in the database includes searchable data entities as well as automatically generated metadata that have been generated by the machine learning processes during the metadata extraction phase.

ScienceSearch uses three categories of tables to organize searchable data entities and associated metadata: data entries tables, metadata entries tables and index tables. The index tables function as indexes for one of the fields of each metadata entry and were created as an additional optimization to reduce the time spent for query matching against each metadata entry. Indexing is done based on *text_tag* field of each metadata entry containing a unique entry per tag. The *text_tag* is a string that semantically describes a metadata entry (e.g. *Graphene*).

Data entry tables are created by the *data import* component during data ingestion and contain a unique pointer to the data element itself (e.g. for an image that is its location on the file system) along with related information (e.g. timestamp, dimensions for images, author list for papers etc). There is one data entry table per searchable category (i.e. *Papers*, *Proposals* and *Images*). In our evaluation, we use the Data, Metadata and Index tables for *Images.* Metadata entries tables are created by the *metadata extraction* component and contain the metadata in the form of tuples which are later used by the search engine. ScienceSearch also features three Metadata entries tables – *Paper* Metadata, *Image* Metadata and *Proposal* Metadata respectively. These tables are used to store the metadata instances for each searchable data entity category. Currently, the *Images* table features around 500K entries and *Metadata* and *Index* tables contain 11M and 1M entries respectively.

5.1.3 Comparison with Stored Metadata. During the comparison with stored metadata phase, the lemmatized list from the user query processing stage is compared with the metadata entries that are stored in the database. We use a two-level parallelism since this phase is compute intensive. The first-level parallelism includes a master process that is responsible for slicing the index table in n pieces, spawning w workers and assigning one slice per worker. The slicing is performed in order to ensure adequate load balancing among the workers. We now describe each worker's actions as well as the individual interactions with the three database tables. Each worker performs three steps a) retrieving indexed metadata, b) recreating metadata objects, and c) ranking metadata objects.

Retrieve Indexed Metadata. Each worker queries the *Index* table. Each worker retrieves unique from the slice assigned by the master process (see Figure 5.2). The retrieval is executed as a comparison between the *text_tag* field of each metadata entity and the lemmatized list's elements producing a *hit score* for each entity. The *hit score* is calculated based on the lexicographic distance of the two entities. Once the score is calculated, the worker keeps only the metadata entities that scored higher (if any) than an empirically set threshold, and discards the rest. The retained results are considered the *hits*.

Recreate Metadata Objects. The goal is to recreate the metadata objects that match the query. Each worker queries the *Image Metadata* table. Once the objects are created they are grouped by *text_tag*. Each tag has many associated objects. The final product of this phase is a list of tuples containing the *hit score*, *metadata type*, *relevance score*, *text_tag* and *pointed image* for each metadata object.

Rank Metadata Objects. Each worker retrieves the results by querying the *Image* table. An image might have more than one matching tuple of metadata results. The search score of each image is calculated from the aggregation of the final score of each tuple. The final search result is the determined from the list of images that are sorted in descending order based on their computed score.

5.1.4 Parallelism. ScienceSearch uses adaptive two-level parallelism to deal with open-ended search queries that return thousands or even million of results. At the first level, ScienceSearch uses parallel workers to handle hits in the *Index* table. Based on our experiments the optimal number of first-level workers is 16. However, in the current version of ScienceSearch data is not shuffled, sometimes leading to an uneven distribution of *hits* between workers. In order to address this issue and enable elasticity for open-ended queries, a second-level of parallelism is enabled in the worker level (sub-workers 1 to N in Figure 5.2). If the number of metadata objects in the recreate metadata tags step exceeds a certain threshold, a new pool of sub-workers is created and the objects are distributed among them for ranking (sub-worker box in Figure 5.2). After each sub-worker has completed the ranking step the results are sent back to the worker that initiated the extra parallelism step. Currently, the threshold that triggers additional parallelism is empirically set to 150,000 metadata objects before ranking.

5.1.5 Understanding the Memory Footprint. In order to identify potential latency inducing bottlenecks, we take a closer look at the memory consumption of each search worker due to intermediate object creation during the three search stages. We use the term *memory block* to capture the amount of memory consumed by the objects. At first, a block of memory is acquired by each worker during the retrieval of unique tags that correspond to each worker's index slice. The number of objects is equal to the number of tags that are stored in that particular database slice. However, during the recreate metadata tags phase, each worker is tasked with fetching all metadata objects that include a particular tag (one tag can have thousands of associated objects) consuming a significantly bigger memory block for the recreated metadata objects. Finally, after ranking each metadata object, a block is acquired for retrieving all images that correspond to the highest ranking metadata objects, in the retrieve image data phase. An overview of the memory blocks with the corresponding worker actions is shown in Figure 5.3.

It is evident that after recreate metadata tags phase, the number of objects in memory significantly increases, making individual object size a determining factor in ScienceSearch's memory footprint as well as its ability to process multiple objects in parallel. Table 5.7 shows the actual object numbers for a search worker after each search step. We present our analysis of memory footprint in subsection 5.3.2.

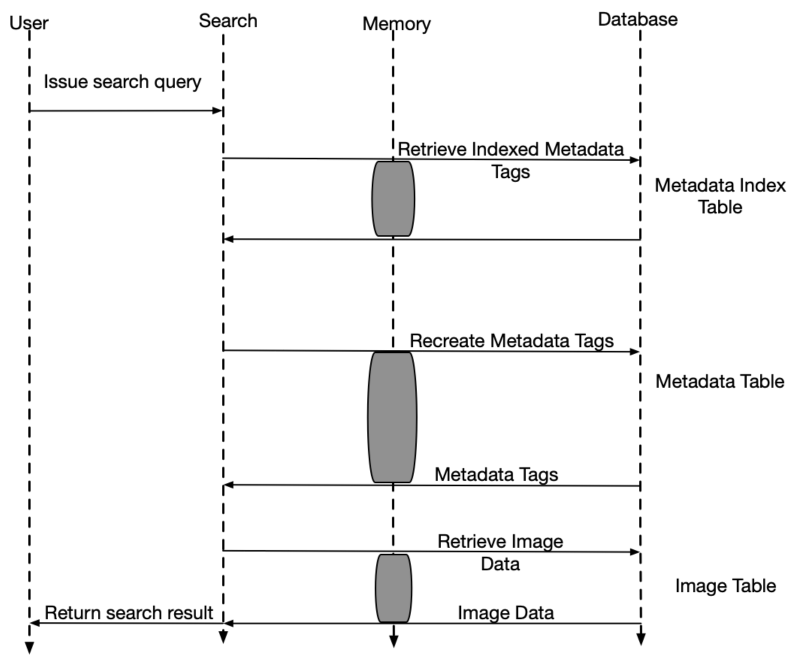


Figure 5.3. Memory consumption of each search worker for object creation during search stages. The number of objects significantly increases after Recreate Metadata Tags.

5.2 Evaluation Setup

This section contains a detailed explanation of the evaluation setup, that includes a description of the ScienceSearch deployment and the software used to automate the experiments. Furthermore, we outline our performance metrics, and the characteristics of the dataset that was used during the experiments, alongside the type of queries identified.

5.2.1 Infrastructure. The experimental setup consists of the deployment of the ScienceSearch components as dedicated containers in order to ensure portability and isolation. Four container instances are deployed: a Django backend, a PostgreSQL database, an Nginx web proxy which also acts as a load balancer and the corresponding frontend that serves a web frontend. Our setup, together with the component's interaction during query execution, is depicted in Figure 5.4. The user is connected to the ScienceSearch infrastructure through a Web-proxy container which serves as a load-balancer. Once a user issues a search query, the query is forwarded to the back-end container where the three search steps are conducted. The backend container, which is the only component with access to the database, fetches search results after issuing one or more database requests. Finally the results are served back to the user through ScienceSearch's frontend.

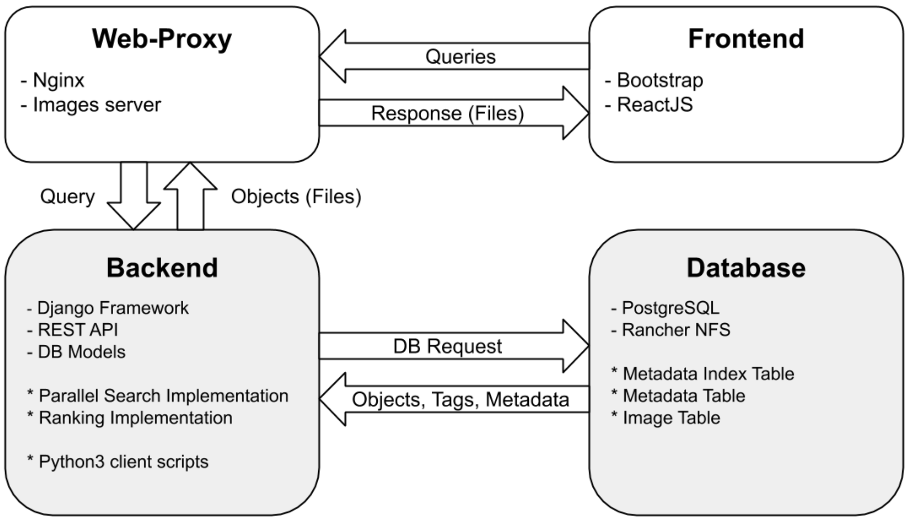


Figure 5.4. ScienceSearch deployment. Grey boxes are containers located on the same physical node to avoid network bottlenecks. White arrows represent requests between internal components.

ScienceSearch was deployed on two testbeds for the experimental evaluation. The first testbed is on Spin, similarly to our production deployment. Our second testbed is a dedicated single node environment. We use the first testbed to obtain a high level overview of ScienceSearch's performance in an context similar to our current deployment measuring latency and throughput. The dedicated system is used for an in-depth analysis of identified bottlenecks and for the measurement of performance without other application interference.

Spin based infrastructure. We deploy ScienceSearch on a container service platform at the National Energy Research Scientific Computing (NERSC) Center, called Spin, which provides a Docker container execution environment and automated resource management on top of supercomputer network and storage. In Spin, containers communicate over an overlay network implemented through IPsec over a 10GB Ethernet. Communication channels between containers that are part of the same deployment are encrypted. In order to avoid the encryption-imposed network overhead in inter-container communication we opt for placing the Backend and the database on the same physical node. ScienceSearch production deployment runs on a set of dedicated nodes at the HPC center that are reserved for the service. The node specifications are: 2x 24 core Intel(R) Xeon(R) CPUs E5-2680 v3 @ 2.50GHz and 256GB of RAM.

Perth. It is a dedicated single node computer system on which ScienceSearch was deployed exclusively. The single node computer has the following specifications: 2x 3.0 GHz 12-core Intel Xeon Gold 6136, 384 GiB 2666 MHz DDR4 RAM, 2x Intel SATA SSDs set up as a Linux software RAID1, running CentOS 7.7.1908 with Linux Kernel version 3.10.

The containers deployed on the dedicated testbed (Perth) communicate through a simple Docker bridge interface, different from the overlay network in Spin. In terms of storage, the single node testbed uses locally mounted directories in order to provide storage capabilities to the containers, while Spin makes use of storage volumes mounted over NFS provided by the HPC system as a highly available and fault tolerant storage service. ScienceSearch is currently deployed as a production service on Spin, and it is critical to understand the scalability of the presented search platform in a real HPC context. The Perth testbed was selected as a dedicated system on which we could run the evaluation without interference from other applications, without container overheads and because the node resembles a host from Spin.

5.2.2 Software. ScienceSearch is deployed on Docker 19.03.5, running PostgreSQL 10.10 as the database service and Django 2.0.4 with Python 3.6.9 as the backend. Throughout all of the experiments, the queries were launched from the backend using two client Python3 scripts that are setup to emulate user queries that would be submitted through a web interface. The scripts cover a) the query latency, query processing rate and memory footprint experiments and b) the query throughput experiments. The client scripts allow us to automate the process of submitting different queries with varying configuration parameters and to focus our analysis on the two search-critical services: the backend and the database. They use the Python *requests* standard package in order to submit requests, both for authentication token and for the actual queries, and the *multiprocessing* package for launching multiple processes during the query throughput experiments. The client scripts measured the overall latency of queries, while the backend logged the latency and memory footprint information of each search phase in a log file.

The backend is deployed with caching disabled for the entire evaluation. This ensures we showcase the actual performance of ScienceSearch, with a focus on the search phases and interaction with the database. The number of parallel worker processes was varied from 1 to 32 in multiples of 4 during the query latency, query processing rate and memory footprint experiments. The number of parallel worker processes was fixed to 4 for the query throughput experiments.

5.2.3 NCEM Dataset. The images produced by the microscopes found in the NCEM's electron microscopy user facility (i.e. micrographs) are the main type of data produced. ScienceSearch currently stores three types of inter-correlated data to the database that are made available through search: **images (e.g. micrographs)**, **proposals** and **calendar entries**.

In our evaluation, we conduct search over the images dataset, which occupies

5TB of storage space. During data ingest, ScienceSearch crawls the supercomputer file system hierarchy that contains the images, extracting file system metadata and experimental metadata added by scientists. The extracted metadata is stored in the Image table in the database. Only the file system metadata and the experimental metadata annotated by scientists are stored in the Image table and not the actual images. This is the search space over which ScienceSearch executes search queries.

The Django framework transforms the database records to Python 3 objects. Table 5.1 shows the total number of objects and size (in the database) of the tables.

Table 5.1. Total number of objects/records found and amount of storage used by each table, for the evaluation search space.

Database table	**number of records**	**total size**
MetadataIndex	1,184,851	115 MB
Metadata	1,1261,844	791 MB
Image	557,195	11 GB

5.2.4 Evaluation Metrics. In order to evaluate the scalability of the ScienceSearch and identify performance bottlenecks we focus on the following four metrics: query latency, query processing rate, memory footprint, and query throughput. The main focus of our evaluation is on ScienceSearch's infrastructure, hence we do not include *quality of search results* in our evaluation metrics set.

Query Latency. We measure overall query latency as the amount of time spent by the client program to prepare the request and to receive the response, as well as the amount of time of ScienceSearch to process the query. While, we include the time it took to perform token-based authentication, we exclude the time it took to acquire the token. We measure the minimum, average and maximum of average query latency for each query from a given workload.

In each experiment, we also measure the latency of each phase of the search algorithm (as presented in 5.1.3). For a detailed breakdown analysis, we combine the time to prepare the request, the time to receive the response and the time to aggregate the final results (as depicted in Figure 5.2) into *other* latency, which summed together with the execution time of each search phase, constitutes the overall search time.

Memory footprint. When evaluating the memory footprint of each search phase, we calculate the number and the size of all intermediate objects and data structures as well as the size of the final result objects. In the case of an image, the final result object consists of the image itself along with associated metadata (e.g. image location in the filesystem, instrument name and the date on which the image was acquired, etc 5.1.2).

Latency and memory footprint were measured in separate sets of experiments in order to separate the overhead of the memory footprint measurements from the latency of the actual query process.

Query Processing Rate & Throughput. Overall query processing rate is calculated as the total number of objects divided by the overall latency. For the query processing breakdown of each search phase we divide the total number of objects corresponding to a search phase with the maximum latency registered at that particular phase between all worker processes used. Since each worker process runs in parallel and the entire search program needs to wait for the last worker to finish in order to complete the query execution, we select the latency of the slowest worker process when calculating the query processing rate. Finally, we measure ScienceSearch's throughput in terms of concurrent queries served per minute.

5.2.5 Queries. We have identified two types of terms that when included in a query exhibit different behavior. For the evaluation of ScienceSearch, we consider two different query sets that highlight that difference in performance, and suitable terms for each query set were selected as a result of a short examination of the metadata tags found in the database.

Query Set 1: Our first query set is meant to capture queries that are commonly executed in practice over the NCEM dataset. They typically return on average under 200 results. We refer to these queries as *targeted* queries, they have a limited set of matching metadata tags, hence limited number of associated objects are generated during the recreate metadata tags search step (Figure 5.2). Targeted queries represent searches from users that have domain specific knowledge and that are familiar with the underlying datasets and look for specific terms. The majority of queries executed on ScienceSearch are targeted queries.

Query Set 2: For the current data set in consideration, we pick queries that are more general or *open-ended* and would return a large number of results. Specifically, we pick queries that return over one thousand results. General queries match almost every stored metadata tag and allow us to thoroughly test the limits of ScienceSearch's infrastructure by maximizing both the computational load and memory footprint of each search step. Furthermore, the high number of associated objects allows us to identify performance bottlenecks.

For targeted queries, we randomly select 76 tags that have less than 200 matches on average in the entire search space of our dataset. For general queries we select 6 tags that have at least 1,000 matches. Example of both query categories and the number of results returned are listed in Table 5.2.

5.3 Experimental Results

This section covers the results of the experimental evaluation performed on ScienceSearch. We exclude measurements from search queries that return zero results since they exhibit very low latency. The query sets and obtained results depend on the

Table 5.2. Example terms of targeted and open-ended queries along with the number of final results returned to the user.

Query Type	**Term**	**Number of Returned Results**
Targeted	lattice	37
Targeted	50images.dm3	159
General	vortices	3,008
General	Frame	1001

size and number of entries in the dataset and therefore can change if the evaluation process is conducted on a different dataset.

The experimental evaluation covers the performance of the proposed search platform from the perspective of: a) the query processing rate metric (Subsection 5.3.1), b) memory footprint (Subsection 5.3.2), c) query latency metric (Subsection 5.3.3), d) the breakdown analysis of the search phases (Subsection 5.3.4), e) the query throughput metric (Subsection 5.3.5), and f) a comparative evaluation of ScienceSearch on the two testbeds (Subsection 5.3.6).

5.3.1 Query Processing Rate. Query processing rate, (i.e. how many objects per second or how much data gets processed per second with increasing number of parallel workers) is very useful when trying to understand the performance of a system such as ScienceSearch. Figure 5.5 depicts the overall query processing rate of ScienceSearch for both query sets. In the context of Query Set 1 (i.e. targeted queries), ScienceSearch shows a good scalability trend and exhibits a processing rate of approximately 87 kObjects/second and 518 kObjects/second, when increasing the number of parallel worker processes from 1 to 32, resulting in an almost 6x speedup. As for Query Set 2 (i.e. open-ended queries) the processing rate increases slightly from roughly 50 kObjects/second to 99 kObjects/second, when varying the number of workers from 1 to 32, achieving a speedup of 2x. It can be noted that query processing rate for Query Set 2 has a slower acceleration than Query Set 1.

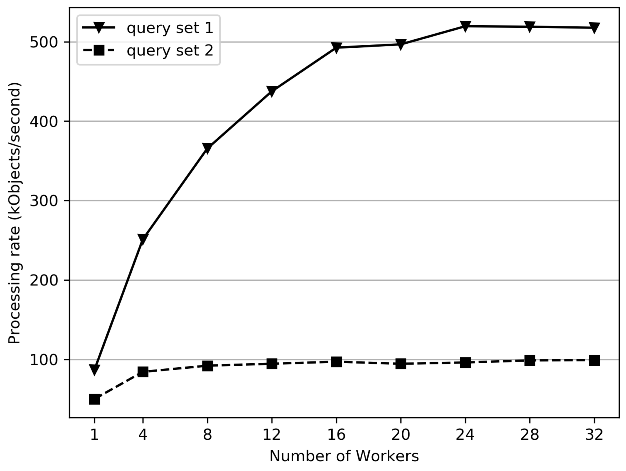
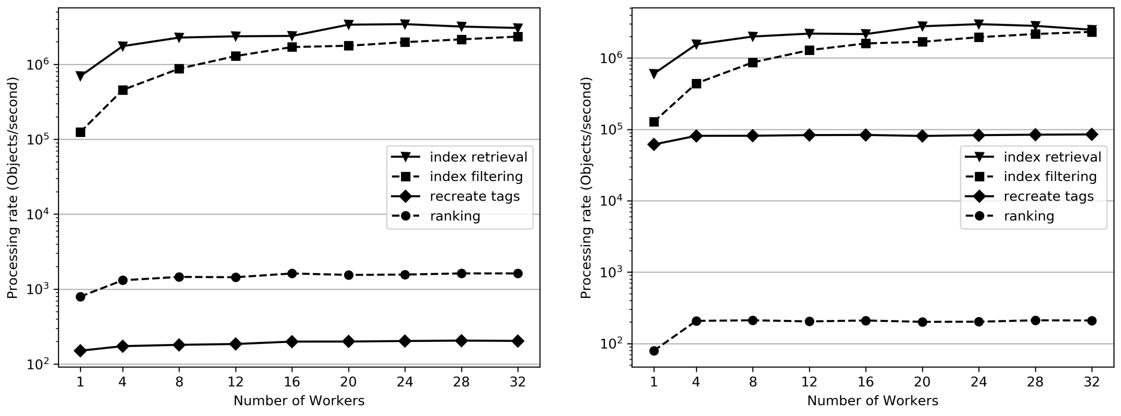


Figure 5.5. Average processing rate, measured in objects per second, with increasing number of parallel worker processes. Comparison between Query Set 1 and Query Set 2, combining the average processing rate of all search phase.

Figures 5.6a and 5.6b show that the metadata index retrieval and metadata index filter phases exhibit higher data processing rates than the recreate metadata tags and the image objects ranking phases. The ranking phase seems to cap at 1622 Objects/second for Query Set 1 and at 210 Objects/second for Query Set 2. The difference in the processing rate can be attributed to differing loads (more analyses in Section 5.3.2).



(a) Processing rate Query Set 1.

(b) Processing rate Query Set 2.

Figure 5.6. Average processing rate, measured in objects per second, with increasing number of parallel workers. (a) Average processing rate of each search phase for Query Set 1. (b) Average processing rate of each search phase for Query Set 2.

5.3.2 Memory Footprint. This evaluation offers valuable insight on the nature of two query types, by looking at the total number of objects processed and generated by each search phase.

Table 5.3. Memory footprint measured as the minimum, mean and maximum number of objects processed when executing Query Set 1 queries, grouped by search phase.

number of objects	**min**	**average**	**max**
metadata indexes	1,184,850	1,184,850	1,184,850
filtered indexes	1	11	64
metadata tags	19	214	1,589
Image objects	19	190	1,414

Table 5.4. Memory footprint measured as the minimum, mean and maximum number of objects processed when executing Query Set 2 queries, grouped by search phase.

number of objects	**min**	**average**	**max**
metadata indexes	1,184,850	1,184,850	1,184,850
filtered indexes	1	6	25
metadata tags	265,837	497,031	651,684
Image objects	505	1,586	3,008

Table 5.3 shows the memory footprint at each search phase for Query Set 1. The index retrieval phase looks up approximately 1.2 million objects (number of unique tags in the index table) from the database, while only 11 objects on average are filtered after the index filter phase. The objects expand to 214 objects (20x increase) on average at the metadata recreation stage. Finally, around 190 objects on average are returned in the end, after performing ranking on them during the ranking phase. In the case of Query Set 2 (Table 5.4), an average of 6 objects are transferred after index filtering. In the next stage (recreate metadata), 497K objects on average match of which 1,586 of get returned as results, which is a significantly larger than what we see in Query Set 1.

Tables 5.5 and 5.6 contain the amount of data, measured in bytes, at each

Table 5.5. Memory footprint measured as the minimum, mean and maximum size of processed objects when executing Query Set 1 queries, grouped by search phase.

total size (bytes)	**min**	**average**	**max**
metadata indexes (with Python)	661 MB	802 MB	813 MB
filtered indexes (with Python)	2.8 KB	8.6 KB	36 KB
metadata tags (with Python)	15.6 KB	168.5 KB	1.2 MB
Image objects (with Python)	204 KB	6.5 MB	48.7 MB
metadata indexes	74 MB	74 MB	74 MB
filtered indexes	26 B	356 B	2 KB
metadata tags	608 B	7.6 KB	65.5 KB
Image objects	38.8 KB	2.3 MB	17.3 MB

Table 5.6. Memory footprint measured as the minimum, mean and maximum size of processed objects when executing Query Set 2 queries, grouped by search phase.

total size	**min**	**average**	**max**
metadata indexes (with Python)	661 MB	762 MB	813 MB
filtered indexes (with Python)	2.8 KB	5.8 KB	16 KB
metadata tags (with Python)	209 MB	376 MB	524 MB
Image objects (with Python)	27.4 MB	85.2 MB	163.9 MB
metadata indexes	74 MB	74 MB	74 MB
filtered indexes	20 B	227 B	932 B
metadata tags	9.8 MB	17.9 MB	31.9 MB
Image objects	10.4 MB	32.1 MB	62.1 MB

search phase for targeted and general queries, respectively. The amount of data measured is proportional with the number of objects recorded at each phase, showing the size of the data with and without Python 3 overhead. The overhead is a direct result of the data structures selected by Django for storing the database records requested during each phase.

Optimization. We have implemented two optimizations that are the result of the memory footprint analyses. First, the second-level of parallelization of workers was introduced to alleviate the challenges encountered if a user were to issue an open-ended query (rare in our current use case, but possible).

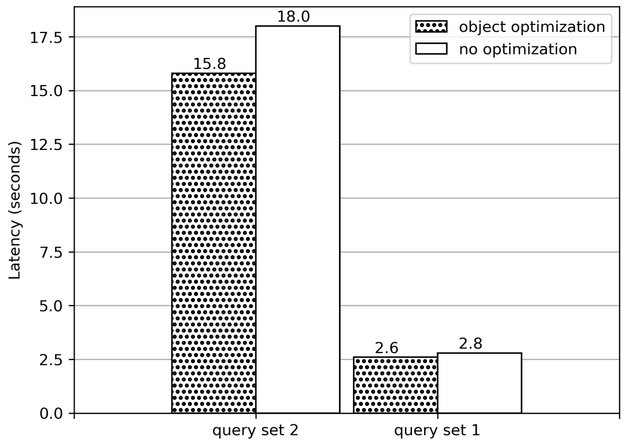
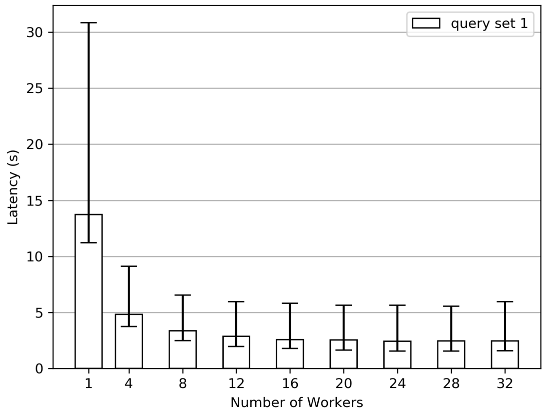


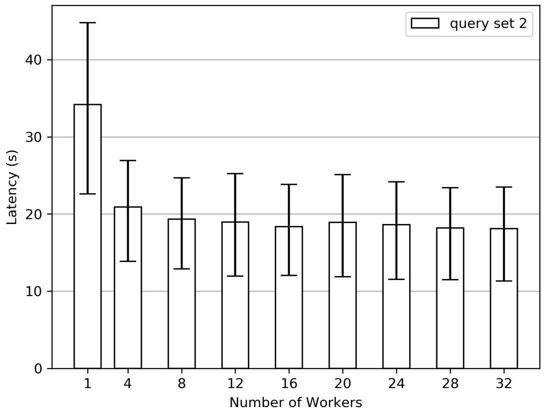
Figure 5.7. Reduced query latency for both targeted and open-ended queries after object size reduction.

In addition, we perform an object-level optimization in order to reduce the amount of data moved from the database to the search workers during the final image retrieval stage. Specifically, we move each object's file metadata field (currently stored in JSON format) from the database to disk. Reducing each object's size subsequently reduces the time spent to recreate final search results (retrieve image data step in interaction with the database 5.1.2) and improves overall search latency. The optimization effect on search latency for both query sets for 16 parallel workers is shown in Figure 5.7. For general queries the object-level optimization reduces search latency by 2.2 seconds while for targeted queries the reduction is 0.2 seconds. The search latency reduction is between 7 and 12%.

5.3.3 Query Latency. Figures 5.8 and 5.9 contain the measured latency of ScienceSearch and the average maximum latency of each search phase for Query Set 1. Figure 5.8a shows that on average for targeted queries it takes ScienceSearch 13.72 seconds (1 worker) and 2.44 seconds (24 workers) to execute a query. For Query Set 2, Figure 5.8b shows a similar trend, but with significantly higher latency when compared to the Query Set 1. Without query processing parallelism, running a general query takes 34.19 seconds and drops to 18 seconds with 32 parallel workers.



(a) Lantecy Query Set 1.



(b) Latency Query Set 2.

Figure 5.8. (a) Overall average query latency and min-max variation with increasing number of parallel workers (Query Set 1). (b) Overall average query latency and min-max variation with increasing number of parallel workers (Query Set 2).

Figure 5.9 shows that metadata index filtering is the most demanding search phase and that query processing parallelism increases the performance of each phase by reducing the latency significantly for metadata index retrieval and filtering. ScienceSearch scales reasonably well, optimizing the most costly phases of search and therefore improving the overall search latency. In the case of Query Set 2, the most demanding phase is not the index filtering anymore and further analysis showed that there is an emerging load balance issues caused by the way indexes are partitioned between the workers. This analysis is discussed in the following section.

5.3.4 Analysis of Queries with Large Results. Figure 5.10 shows the latency of each search phase with 16 parallel worker processes. We can see that the metadata index retrieval phase takes up to 0.48 seconds and the metadata index filtering phase takes up to 0.56 seconds for all workers, and that each worker spends a similar amount of time in these two phases. When we look at the latency of the recreate metadata tags and the final image object ranking phases, we can see that workers do not spend the same amount of time. Some worker processes end up spending approximately 7.93

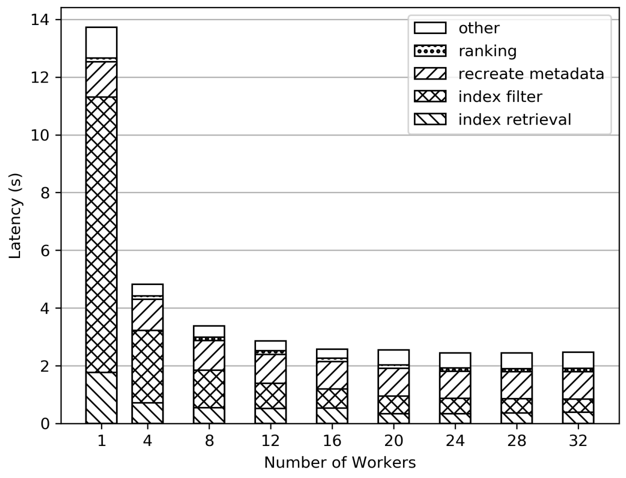


Figure 5.9. Average query latency grouped by search phase, with increasing number of parallel workers (Query Set 1).

seconds, as is with worker 3, while other spend under 0.01 seconds. Worker 3 also takes 11.8 seconds to perform the ranking on the image objects, while other workers spend again under 0.01 seconds in the same phase.

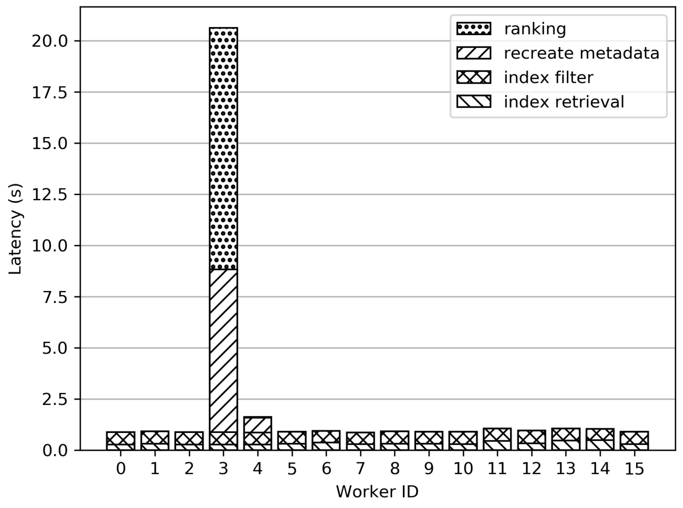


Figure 5.10. Average latency of each search phase for each worker for a query (Frame) from Query Set 2 with 16 parallel workers.

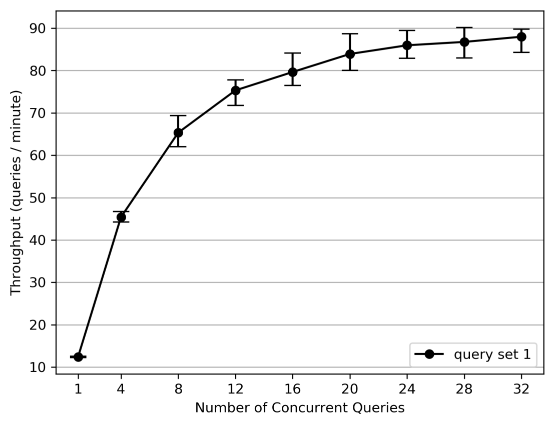
The imbalance in the worker processing time is caused by data that is not distributed evenly across worker processes. (Table 5.7) This causes the remaining phases in the query processing pipeline to suffer from a similar imbalance, and for some phases it can be much worse. For example, the recreate metadata phase does not convert the metadata index objects to metadata tag objects on a one-to-one basis, but follows a one-to-many structure, generating in the best case scenario 410x more objects while in the worst case scenario up to 14,000x more objects, as observed in Tables 5.3 and 5.4).

Table 5.7. Number of objects processed, filtered or generated at each search phase for each worker for an query (Frame) from Query Set 2 with 16 parallel workers.

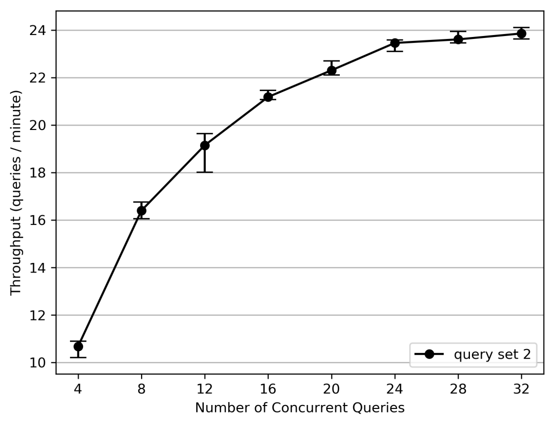
worker ID	metadata indexes	filtered indexes	metadata tags	Image objects
0-2	74,053	0	0	0
3	74,053	2	549,226	1,000
4	74,053	1	1	0
5-15	74,053	0	0	0

5.3.5 Query Throughput. Figure 5.11a shows the average query throughput while running concurrent targeted queries. Query throughput increases significantly with increasing number of concurrent queries being issued during the same experiment. When only one query is issued at one moment of time ScienceSearch achieves a throughput of roughly 12 queries/minute. When running 8 queries at the same time, each query using 4 parallel worker processes, there are a total of 32 concurrent processes that access the database and process the queries, but the resulting throughput is approximately 65 queries/minute, which is even faster than using 32 parallel worker processes while executing only one query at a time, which achieves a throughput of roughly 25 queries/minute. When we executed 32 queries concurrently, that means 128 concurrent processes in total, the throughput increases to 88 queries/minute, even though the system was over-provisioned.

For Query Set 2, as depicted in Figure 5.11b, ScienceSearch shows similar trends in terms of scalability, experiences reduced performance degradation when running concurrent queries, even when over-provisioned, but achieves a lower query



(a) Throughput for Query Set 1.



(b) Throughput for Query Set 2.

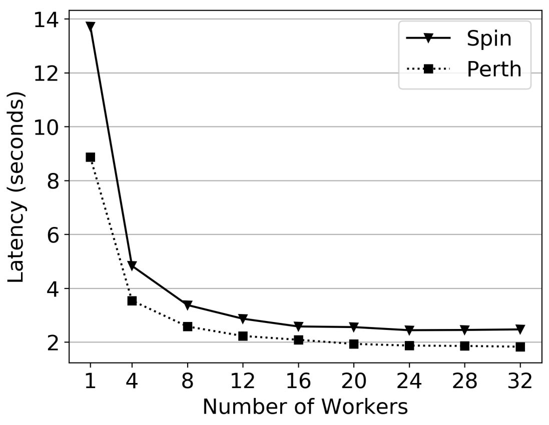
Figure 5.11. (a) Average query throughput and min-max variation for Query Set 1 with increasing number of concurrent queries. (b) Average query throughput and min-max variation for Query Set 2 with increasing number of concurrent queries.

throughput when compared to the Query Set 1 performance, which is in concordance with the difference in latency between the two datasets.

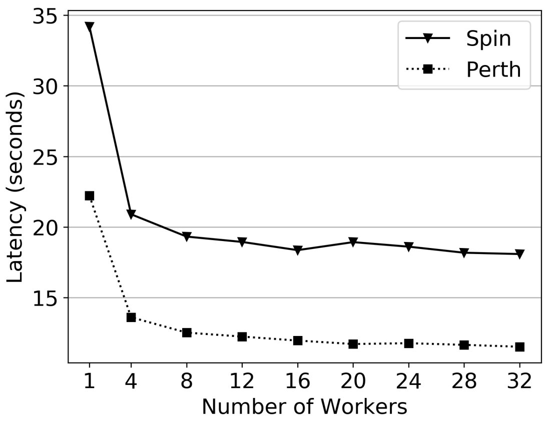
The key takeaway from these results is that ScienceSearch can serve multiple clients at the same time minimizing the side effects of workload imbalance.

5.3.6 Spin Infrastructure vs Dedicated Testbed. This subsection covers the latency and throughput experiments that were run on the single node system called Perth, on which ScienceSearch was deployed exclusively. ScienceSearch performs better by a small degree on Perth than on *Spin*, and that could be attributed to different factors, including hardware specification, infrastructure particularities and the lack of interference caused by other applications.

Figure 5.12a shows that for Query Set 1, the system can end up achieving a latency of under 2 seconds, when configured with at least 20 parallel worker processes. For Query Set 2 (Figure 5.12b), the latency can go as low as approximately 11.5 seconds when using 32 parallel worker processes.



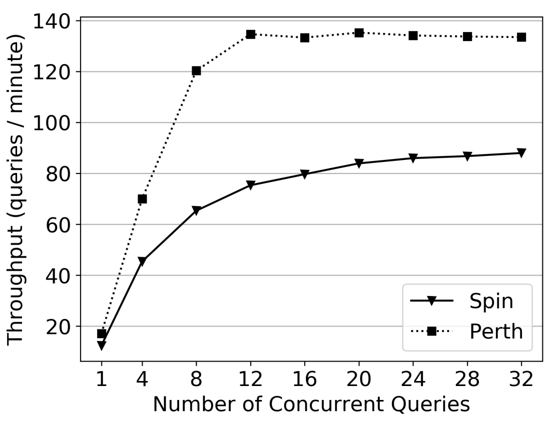
(a) Average latency for Query Set 1.



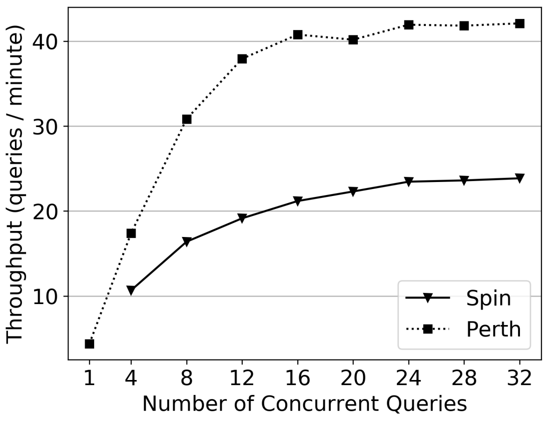
(b) Average latency for Query Set 2.

Figure 5.12. (a) Average query latency for Query Set 1 on *Spin* and Perth. (b) Average query latency for Query Set 2 on *Spin* and Perth.

In terms of throughput, the lack of overhead from the storage system and the lack of interference allows ScienceSearch to reach up to 134 queries/minute while running 32 concurrent targeted queries and roughly 42 queries/minute while running 32 concurrent Query Set 2 queries. As seen in Figure 5.13, the dip in throughput while running 20 concurrent Query Set 2 queries is caused by the fact that we ran 96 queries which does not divide exactly to 20.



(a) Query throughput for Query Set 1.



(b) Query throughput for Query Set 2.

Figure 5.13. (a) Average query throughput for Query Set 1 on *Spin* and Perth. (b) Average query throughput for Query Set 2 on *Spin* and Perth

It can be observed from the latency and throughput experiments, that the performance of ScienceSearch plateaus halfway through the number of available cores on all testbeds. From Figures 5.12, we can see that latency does not seem to decrease at the same speed after 12 cores when running on Spin and Perth. This can be attributed to the fact that in each experiment, the number of processes created is in practice double the amount of configured parallel workers processes. The above results in reaching the limit of available cores for both testbeds, while in fact only half of them belong to ScienceSearch's backend. The other half belong to the database, that spawns a process for each parallel work process. The same effect can be observed in Figures 5.13 that encompasses the throughput experiments. After 12 cores for Spin and Perth, throughput does not increase as fast as before the number of cores get over-provisioned with processes.

5.4 Discussion

In this section, we discuss key insights from our experiences and results. We focus our analysis on six key elements of ScienceSearch: a) *Spin* and it's underlying mechanisms that export HPC resources to ScienceSearch's deployment, b) internal load balancing and c) adaptive resource scaling that are necessary for dealing with open-ended queries, d) the effect of structures used in data representation and size in Python3, e) the strengths of ScienceSearch as a solution for performing search over scientific data and f) finally a discussion about how future hardware could further improve the performance of ScienceSearch.

5.4.1 Spin. HPC facilities ingest data at increasingly rapid rates (in some cases exceeding petabytes) increasing the demand for solutions that will enable scientific search. Supercomputing facilities are providing platforms such as Spin that enable atypical software stack deployment on HPC resources while benefiting from the resources at the center. Deploying our infrastructure on Spin allows us to access HPC network, storage, and compute resources.

Spin has been critical to enable the ScienceSearch infrastructure at the HPC facility. ScienceSearch is designed to be deployed as a service that is available all the time, without the need to re-compute indexes and re-learn metadata tags every time a user issues a search. This is different from existing search solutions that are implemented as a library or a program that runs ephemerally either on the logins nodes or on the actual supercomputer, and that usually requires at least the index to be reloaded in memory.

We observe from Figures 5.12a and 5.12b that between *Spin*, a shared multi-user supercomputer infrastructure, and the single node testbeds, ScienceSearch exhibits similar performance trends, albeit at slightly different latency. The latency discrepancy can be explained by the underlying hardware, (faster CPU, memory) but also by the inherent latency of the two different infrastructures. We know that ScienceSearch is latency-sensitive, thus the remote storage volume that *Spin* provides and which the search platform uses, will induce a certain, latency penalty when contrasted to local storage (remote storage has the added latency of both network and storage). On the other hand, ScienceSearch was designed to run on Spin and can exploit the inherent benefits that come with it: mobility, fault tolerance and scalability. The database has its data stored over the network, and can easily be moved to another compute host, enabling ScienceSearch to achieve mobility.

Fault tolerance and scalability are accomplished through the use of multiple instances, either for the front-end or the back-end, but as well as for the database, that can easily be deployed on multiple hosts, while data can be protected against faults and scaled up independently. Of course upgrading the hardware to faster counterparts and increasing the number of compute units, while fixing the load balance issue, could easily lead to better performance. However, the benefits of running ScienceSearch in the current infrastructure far outweigh the cost.

Our extensive performance evaluation has unveiled some key challenges that will make scalability difficult as data sizes increase. Achieving network performance across the containers is still hard and impedes our ability to deploy database instances across multiple nodes and scale. Currently, Spin encrypts communication between containers that are located in different physical nodes by default. The encryption significantly reduces available throughput and the ability to transfer data from the database instance to the back-end container executing search. Meeting security requirements while achieving performance will be critical for future scaling.

5.4.2 Search Engine Internal Load Balancing. Our evaluation results have demonstrated that the root cause of increased latency for open-ended queries, is the uneven distribution of results between workers before ranking (see Figure 5.10). In order to ensure low query latency for large scientific datasets, a scalable search infrastructure needs to achieve adequate load balancing of query matches between search workers. In the context of a master-worker search model, load balancing can be achieved by placing intermediate matches in a common queue and coordinating redistribution between idle workers. The common queue operations might introduce some overheads that will need to be considered.

Currently, in-node parallelism reduces query latency (Figure 5.8) by a factor of 3. However, enabling search over a significantly larger dataset would require across-node parallelism. To address this issue a master-worker deployment can be utilized where multiple search and database instances are spawned by a master search process.

5.4.3 Adaptive Resource Scaling. During the initial deployment phase of ScienceSearch open-ended queries were unable to complete, causing system wide exceptions and memory leaks. The underlying exceptions were attributed to the high number of metadata instances that workers had to manage after the initial load distribution phase, i.e., the *rank metadata tags* step. We addressed this issue by introducing a second level of adaptive parallelism. Adaptive parallelism is necessary for serving open-ended queries hence enabling scalable exploration of the scientific search space.

In order to be scale dynamically depending on the query, adaptive resource scaling is necessary. Search architectures need to be able to elastically provision HPC resources for serving computationally demanding searches and release those resources when they are no longer needed. While this is a common resource usage model for cloud computing, is difficult or impossible to do in most current HPC systems. It is critical for HPC facilities to provide abstractions to enable adaptive resources scaling while keeping utilization high.

5.4.4 Data Representation & Sizes. During the *recreate metadata tags* step 5.1.3 the metadata instances are generated from the filtered indexed metadata tags in a one-to many fashion (one filtered indexed metadata tag can have many associated metadata instances), which results in a large number of objects that need to be processed by each worker. Introducing an intermediate ranking step would reduce the objects number and thus the overall latency for all queries, especially the open-ended queries.

One of the key findings in our evaluation process is that the total size of each metadata instance (and subsequent memory footprint of the parallel search worker) is greatly increased (sometimes by 9x) due to the object representation in Python3 by Django (Tables 5.5, 5.6). As we use these frameworks in HPC environments, we will need to investigate appropriate optimizations. One solution that partially mitigates the memory overhead issue (especially for Python dictionaries) is the use of alternate data structures such as *slots* or *namedtuples*.

5.4.5 ScienceSearch Performance. ScienceSearch provides specific advantages to the the problem of implementing search engines over domain specific scientific data found in HPC systems. ScienceSearch's novelty is its mechanism for combining and correlating information from data with other data sources, such as research papers, images, proposals, calendar entries. This is accomplished through a set of deep learning and natural language processing algorithms employed by ScienceSearch, that are used to generate metadata tags. The majority of classical search engines designs and architectures [17] assume the input data to be a collection of data sources that area flat domain of input data where the collection of data sources are homogeneous in structure and semantics, and ScienceSearch innovates in this area by providing a mechanism for combining structurally and semantically different data sources.

Vertical scalability is an inherent property of the ScienceSearch design and architecture, while horizontal scalability is achieved through container platforms that can make use of HPC hardware, such as *Spin*. In this work, we emphasize on the vertical scalability of ScienceSearch and pinpoint techniques used for achieving good scalability and performance, such as the adaptive two-level parallelism technique. ScienceSearch can also achieve horizontal scalability, due to its decomposition of compute, storage and interface components into containers. ScienceSearch can decide how many containers of each type can run in a deployment.

For targeted queries ScienceSearch can achieve query latency as low as 2.5 seconds, which is satisfactory, given that the current users had no search engine solution available and thus no means to search over their data. Existing solutions, such as Apache Lucene advertise to achieve sub-second query latency, but when comparing ScienceSearch to existing solutions, we have to keep in mind the different indexing and query pipelines and the differences in the inverted index structure that ScienceSearch employs that adds additional overhead but provides a richer and more meaningful user experience for scientific data.

5.4.6 Impact of Future Hardware. While the performance of the ScienceSearch platform does not solely depend on a set of hardware properties, as shown by the experiments across different testbeds, ScienceSearch can still benefit from the performance improvements of computer hardware (CPU, memory, storage and network). Some immediate improvements in performance stem from, as mentioned in subsection 5.4.1, from better inter-container communication through a network that maintains a certain degree of security and isolation without significantly sacrificing performance. Using hardware solutions such as VLANs could alleviate many performance bottlenecks either from the network devices, the operating systems or the container hypervisor, while still retaining the similar security properties to IPsec without the need for encryption.

Other more system-wide improvements could come from the development and adoption of exotic hardware. For example NVIDIA's emerging [42] (DPU) could be used to improve the performance and scalability of HPC applications by providing the means to overlap communication with computation [43]. UPMEM have been working on Processing-in-Memory PIM [44] devices that could be used to accelerate database query processing [45] by moving computation to where data resides (i.e. memory DIMMS) and by avoiding bringing the data to where computation is performed (i.e. the CPU). These are just two examples of future technologies that ScienceSearch could exploit in order to achieve even higher performance and scalability.

5.5 Conclusion

We present a detailed evaluation of ScienceSearch's underlying infrastructure. Our results have shown that ScienceSearch can serve up to 130 queries per minute while keeping latency around 2.5 seconds for typical user queries (where results are in the hundreds). In order to deal with the increased number of results from open-ended queries, we have introduced an additional level of parallelism that load balances both object recreation and ranking.

Our work also provides considerations and insights in the design and support of search systems on HPC systems. While a container-based infrastructure at an HPC infrastructure lets us leverage the high-performance filesystem, it provides other challenges with performance that need to be considered by applications. We also highlight the need and opportunity for adaptive resource scaling, considerations of data representation in memory.

CHAPTER 6
RELATED WORK

In this chapter, we talk about existing work in the areas of data retrieval and date indexing. We cover exiting work in information retrieval, inverted index design and search engine design, including a few domain specific search solutions from various scientific research projects.

6.1 Data Indexing and Retrieval

Indexing has been long studied in database systems [46]. 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 [47], 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 [48], 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 [49]. 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 [50]).

A different approach, named GLIMPSE [51], employs partial inverted indexes that consume smaller disk space than a full-text-inverted index. Geometric partitioning [52] also manipulates inverted indexes by splitting it according to updating time so to reduce the update overhead. Similarly, query-based partitioning [53] categorizes inverted indexes based on access and query frequency.

Recent prior work [54] 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 distributed indexes much faster with lower costs.

6.2 Search Engine Design

Other research focuses on the high-level indexing pipeline and the integration of indexing and search in existing parallel and distributed file systems. TagIt is one such project [55, 8, 56], that implements a scalable data management service framework for scientific datasets, that is integrated with the underlying distributed file systems that house the scientific datasets. The framework relies on a scalable and distributed metadata indexing framework, that can index file system related metadata as well as custom metadata created by the users, under the form of tags, that can aid data discovery. The authors aim at making the indexing framework not reside on external hardware, the same way catalog solutions do, but tightly integrate it with the distributed components of the file system and making use of file extended attributes. But ultimately the proposed indexing framework is implemented as a collection of distributed databases, making this solution appropriate for structured and semi-structured data, but more difficult to use for unstructured free-text data, where inverted indexes are a better choice. TagIt was integrated with and evaluated on GlusterFS and CephFS, and while the overall service was able to achieve good performance with minimal overhead, due to optimizations such as data and index co-location and asynchronous processing and communication, we suggest that there is still room for improvement at the low level design of the indexing framework at that single-node level, which could accelerate high-level solutions even more beyond what they can currently achieve. Other existing works from the HPC domain (e.g. GUFI [7, 57, 58]) has also aimed to tackle the indexing and search problem focusing on metadata as opposed to the scientific data itself, while other look at providing indexing and search over persistent memory object storage [59, 60]. We believe both the metadata and data are both critical components to better accessibility of scientific data.

One of the more important parts of an information retrieval solution or search engine, that can directly influence the performance of the index and search processes, is the design and implementation of the inverted index. The inverted index uses one or more search data structures as its constituent components in addition to the other data structures used to store any kind of information related to the entries in the inverted index. There are researchers who actively look at how to design and implement the inverted index for a specific dataset or application. MIQS [61] is a solution that aims to efficiently index self-describing data formats, such as HDF5 and netCDF, through the use of a custom in-memory index implementation. MIQS provides a portable and schema-free solution that is aligned with the paradigm of self-describing data, and it uses a combination of search trees to build the index.

Cavast [62] is a another project that aims to improve the performance of in-memory key-value stores, through a re-design of hash table implementation, in order to better exploit the CPU caches and memory subsystem. Cavast achieves this through a combination of methods and techniques: the separation of key and value placement in memory, laying out the hash table elements in memory so that they can better benefit from cache locality and exposing the kernel cache coloring scheme, to name a few. Other existing works from HPC look at redesigning search tree data structures stored on persistent memory in order to make them NUMA aware, and thus avoid the performance overhead of inter-NUMA communication [63]. While we acknowledge the importance of the search data structure, we emphasize that the search data structure alone cannot guarantee high indexing performance and that the inverted index needs to be designed and implemented as a scalable and tightly coupled combination of search data structure and inverted index.

Numerous works evaluate the scalability of search infrastructures mainly focusing on the indexing and query processing parts. ElasticSearch [64] provides a set of dedicated metrics that measure query processing time, latency and throughput while indexing performance is measured using indexing and flush latency. Solr [65] features built in timers for query latency and indexing performance. *Apache Lucene* [66] has been used as a core building block for scientific search engines and information retrieval tools. The authors in [67] evaluate the scalability of *Anserini*, a Lucene-based information retrieval tool, by creating custom benchmarks for indexing and ranking. The presented solutions are limited in providing only horizontal scaling and do not comprehensively evaluate all scalability aspects.

6.3 Search in Science

Indexing and search in large high-performance file systems is not a problem that is solely specific to search engine applications, but other domain specific applications could also benefit from having an efficient indexing and search service that runs well on HPC systems. Genomics research is a field that could benefit from efficient indexing methods, and there is work that looks, for example, at ways to improve the performance of DNA k-mer sequence counting using indexing techniques [68]. In the mentioned work, the authors propose two distributed parallel hash table techniques. These two techniques are optimized to use cache friendly algorithms for hashing, to overlap computation with communication in order to hide latency and to use a vector-based computation technique to compute the hashes of many k-mer indices. Their solutions can process 1TB over 4096 cores in 11.8 and 5.8 seconds, demonstrating high improvements over the state-of-the-art. We argue that an efficient indexing framework, with an exposed indexing pipeline, should be able to achieve, after tuning of course, similar if not better performance to the two proposed solutions, while still maintaining enough generality to be easily used in other scenarios.

Numerous research efforts have tried to address the issue of scientific search capabilities. The *Materials Project* [69] enables search over an individual material's characteristics, while *KBase* [70] provides search capabilities over systems biology data. *LODAtlas* [71] enables users to discover datasets of interest by facilitating content and metadata based search. Finally, *Thalia* [72] is a search infrastructure that enables semantic search in biomedical literature based on named entity recognition. However, none of the proposed solutions provide scalable search environments that can host vastly growing amounts of data and provide low latency search results and can't be used to compare our infrastructure.

Data Search [73] provides a scalable solution for metadata management but the system does not automatically infer or create metadata from the ingested datasets. The ability to reliably handle parallel job execution led to the adoption of MapReduce/Hadoop [39, 40] for large parallel searches by different projects [74, 75].

Recent research efforts have tried to enable scalable search capabilities for scientific portals. *Varsome* [76] created a a search engine for human genomics variants that provides search over 500 million variant records. However, the deployed solution does not scale since *Varsome* stores every record in a massive database. In [77] the authors present a scalable search engine for geospatial data that utilizes indexing shards in order to provide low-latency search results. *CellAtlasSearch* [78] enable search over thousands of cell profiles. Their approach relies on specific hardware (GPUs) to achieve low latency results for queries. The authors of [79] built *Visibiome*, a scalable search architecture for microbiomes. Their solution is a distributed web application that only scales horizontally when dealing with increased number of parallel queries.

CHAPTER 7
CONCLUSION AND FUTURE WORK

The ubiquitous nature of big data has resulted in the development of highly scalable parallel and distributed file systems. As such, researchers and engineers can now efficiently store and manage petabytes of data. However, while much research effort has focused on the methods to efficiently store and process data, there has been little focus on data indexing and discovery. Thus, users of large file systems are increasingly frustrated at their inability to easily locate data. In order to address the general problem of efficient and effective data exploration and search in large-scale storage systems, we initially defined the problem space, through an analysis and characterization of several scientific production file systems. The analysis highlighted the magnitude of the challenge when indexing large quantities of data. We then explored the prospect of using existing search engine building blocks, such as CLucene, to build and integrate search capabilities into a distributed file system over unstructured data. The resulting solution, called FusionDex, achieved better search latency than existing cloud solutions, but hinted towards data indexing becoming a potential bottleneck for increasing data volumes.

In order to solve the problem of data indexing, we proposed SCANNS, a indexing and search framework that uses a novel tokenization method and a novel inverted index design, and that exposes the indexing pipeline allowing the user to saturate compute, memory and storage resources of single-node high-end systems, characterized by many cores, multiple NUMA nodes and multiple NVMe storage devices. When then switched focus to ScienceSearch, a search platform used in production at NERSC, that uses machine learning and natural language processing to generate metadata tags for data provided from various sources, such as published papers, proposals, images and file system structure. We conducted a performance evaluation of the ScienceSearch infrastructure in order to better understand the implications and performance considerations of searching over an index that is build from learned metadata tags.

Drawing from the insights gained from SCANNS, we explored the problem of building persistent indexes and of efficiently searching persistent indexes. We proposed SCIPIS, a single-node indexing and search framework, that builds on top of SCANNS and extends it by adding support for efficiently building persistent indexes and processing TFIDF queries over them on high-end computing systems. SCIPIS also introduced new tuning parameters that open up a new dimension for adapting the framework and the inverted index to the characteristics of the input data, allowing SCIPIS to better utilize the index memory space and to yield higher indexing throughput than SCANNS.

In terms of future work we plan to explore methods for distributing indexing and search to scale to some of the largest HPC storage systems available. We will be using SCIPIS as the building block and we will study the problems of adding the network resources to indexing and search, and the problem of organizing the index across multiple machines. We will also look into the problems of running and maintaining an indexing and search system in the context of scientific data and storage systems, but also the problem of how do we interface indexing and search with existing systems and users. Specifically, we will investigate integration of the distributed SCIPIS system into parallel and distributed storage systems to enable automatic metadata and data indexing and search.

BIBLIOGRAPHY

[1] A. Białecki, R. Muir, G. Ingersoll, and L. Imagination, "Apache lucene 4," in *SIGIR 2012 workshop on open source information retrieval*, p. 17, 2012.

[2] D. Shahi, "Apache solr: an introduction," in *Apache Solr*, pp. 1–9, Springer, 2015.

[3] C. Gormley and Z. Tong, *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[4] S. Khalsa, P. Cotroneo, and M. Wu, "A survey of current practices in data search services," *Research Data Alliance Data (RDA) Discovery Paradigms Interest Group*, 2018.

[5] Datanyze, "Enterprise search software market share." https://www.datanyze.com/market-share/enterprise-search–287.

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

[7] D. J. Bonnie, "Gufi overview," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2018.

[8] A. K. Paul, B. Wang, N. Rutman, C. Spitz, and A. R. Butt, "Efficient metadata indexing for hpc storage systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 162–171, IEEE, 2020.

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

[10] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, (USA), p. 19–es, USENIX Association, 2002.

[11] 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*, pp. 307–320, 2006.

[12] K. Chard, I. Foster, and S. Tuecke, "Globus: Research data management as service and platform," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, pp. 1–5, 2017.

[13] B. Blaiszik, K. Chard, R. Chard, I. Foster, and L. Ward, "Data automation at light sources," in *AIP Conference Proceedings*, vol. 2054, p. 020003, AIP Publishing, 2019.

[14] R. Cook, W. K. Michener, D. A. Vieglais, A. E. Budden, and R. J. Koskela, "Dataone: A distributed environmental and earth science data network supporting the full data life cycle," in *EGU General Assembly Conference Abstracts* (A. Abbasi and N. Giesen, eds.), EGU General Assembly Conference Abstracts, 2012.

[15] "Sdss data release 17: Data volume table"," 2021. https://www.sdss.org/dr17/data_access/volume/.

[16] R. Ahumada, C. A. Prieto, A. Almeida, F. Anders, S. F. Anderson, B. H. Andrews, B. Anguiano, R. Arcodia, E. Armengaud, M. Aubert, *et al.*, "The 16th data release of the sloan digital sky surveys: First release from the apogee-2 southern survey and full release of eboss spectra," *The Astrophysical Journal Supplement Series*, vol. 249, no. 1, p. 3, 2020.

[17] R. Baeza-Yates, B. Ribeiro-Neto, *et al.*, *Modern information retrieval*, vol. 463. ACM press New York, 1999.

[18] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, S. Kenny, K. Iskra, P. Beckman, and I. Foster, "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers," *Journal of Physics: Conference Series*, vol. 180, no. 1, 2009.

[19] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang, and I. Raicu, "Paving the road to exascale with many-task computing," 2012.

[20] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, "Towards data intensive many-task computing," in *Data Intensive Distributed Computing: Challenges and Solutions for Large-scale Information Management*, pp. 28–73, 2012.

[21] "International data corporation," 2017. https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

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

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," *MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010.

[24] "Myria," 2017. http://myria.cs.washington.edu.

[25] 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.

[26] T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang, and I. Raicu, "A convergence of key-value storage systems from clouds to supercomputers," *Concurrency and Computation: Practice and Experience (CCPE) Journal*, vol. 28, no. 1, pp. 44–69, 2015.

[27] "Clucene," 2017. http://clucene.sourceforge.net.

[28] "Hadoop grep," 2017. https://wiki.apache.org/hadoop/Grep.

[29] "Mapreduce index tool," 2017. http://www.cloudera.com/documentation/archive/search/1-3-0/Cloudera-Search-User-Guide/fcsug_mapreduceindexertool.html.

[30] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. Dewitt, B. Heath, D. Maier, S. Madden, M. Stonebraker, and S. Zdonik, "A demonstration of scidb: A science-oriented dbms," in *VLDB'09: Proceedings of the 2009 VLDB Endowment*, VLDB Endowment, 2009.

[31] "Apache lucene," 2017. https://lucene.apache.org.

[32] "Wikipedia data download," 2017. https://en.wikipedia.org/wiki/Wikipedia: Database_download.

[33] A. I. Orhean, A. Giannakou, L. Ramakrishnan, K. Chard, and I. Raicu, "Scanns: Towards scalable and concurrent data indexing and searching in high-end computing system," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 51–60, IEEE, 2022.

[34] S. Benzaquen, A. Evlogimenos, M. Kulukundis, and R. Perepelitsa, "Abseil," 2021. https://abseil.io/about/design/swisstables.

[35] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: a large collection of system log datasets towards automated log analytics," *arXiv preprint arXiv:2008.06448*, 2020.

[36] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[37] G. P. Rodrigo, M. Henderson, G. H. Weber, C. Ophus, K. Antypas, and L. Ramakrishnan, "Sciencesearch: Enabling search through automatic metadata generation," in *2018 IEEE 14th International Conference on e-Science (e-Science)*, pp. 93–104, IEEE, 2018.

[38] L. Ramakrishnan, A. I. Orhean, M. Henderson, G. Rodgrigo Alvarez, A. Giannakou, G. Weber, and K. Antypas, "Sciencesearch metadata infrastructure (sciencesearch-mi) v1. 2," tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2020.

[39] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, p. 107–113, Jan. 2008. https://doi.org/10.1145/1327452.1327492.

[40] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th ed., 2015.

[41] A. I. Orhean, A. Giannakou, K. Antypas, I. Raicu, and L. Ramakrishnan, "Evaluation of a scientific data search infrastructure," in *Concurrency and Computation: Practice and Experience*, 2022.

[42] I. Burstein, "Nvidia data center processing unit (dpu) architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp. 1–20, IEEE, 2021.

[43] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda, "Bluesmpi: Efficient mpi non-blocking alltoall offloading designs on modern bluefield smart nics," in *International Conference on High Performance Computing*, pp. 18–37, Springer, 2021.

[44] V. Zois, D. Gupta, V. J. Tsotras, W. A. Najjar, and J.-F. Roy, "Massively parallel skyline computation for processing-in-memory architectures," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, (New York, NY, USA), Association for Computing Machinery, 2018. https://doi.org/10.1145/3243176.3243187.

[45] T. R. Kepe, E. C. de Almeida, and M. A. Z. Alves, "Database processing-in-memory: An experimental study," *Proc. VLDB Endow.*, vol. 13, no. 3, p. 334–347, 2019. https://doi.org/10.14778/3368289.3368298.

[46] 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.

[47] 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, Sept. 2010.

[48] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 598–609, 2008.

[49] 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*, pp. 591–602, 2010.

[50] "Elastic power-aware data-intensive cloud," 2017. http://www.comp.nus.edu.sg/ epic.

[51] U. Manber and S. Wu, "Glimpse: A tool to search through entire file systems," in *USENIX Winter Technical Conference*, pp. 4–4, 1994.

[52] 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*, pp. 776–783, 2005.

[53] 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*, pp. 623–636, 2008.

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

[55] H. Sim, Y. Kim, S. S. Vazhkudai, G. R. Vallée, S.-H. Lim, and A. R. Butt, "Tagit: an integrated indexing and search service for file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[56] H. Sim, A. Khan, S. S. Vazhkudai, S.-H. Lim, A. R. Butt, and Y. Kim, "An integrated indexing and search service for distributed file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2375–2391, 2020.

[57] G. A. Grider, D. A. Manno, W. K. Poole, D. J. Bonnie, and J. T. Inman, "Grand unified file indexing," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2021.

[58] D. Manno, J. Lee, P. Challa, Q. Zheng, D. Bonnie, G. Grider, and B. Settlemyer, "Gufi: fast, secure file system metadata search for both privileged and unprivileged users," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, IEEE, 2022.

[59] A. Khan, H. Sim, S. S. Vazhkudai, J. Ma, M.-H. Oh, and Y. Kim, "Persistent memory object storage and indexing for scientific computing," in *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 1–9, IEEE, 2020.

[60] A. Khan, H. Sim, S. S. Vazhkudai, and Y. Kim, "Mosiqs: Persistent memory object storage with metadata indexing and querying for scientific computing," *IEEE Access*, vol. 9, pp. 85217–85231, 2021.

[61] W. Zhang, S. Byna, H. Tang, B. Williams, and Y. Chen, "Miqs: Metadata indexing and querying service for self-describing file formats," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–24, 2019.

[62] K. Wang, J. Liu, and F. Chen, "Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, 2020.

[63] S. Jamil, A. Salam, A. Khan, B. Burgstaller, S.-S. Park, and Y. Kim, "Scalable numa-aware persistent b+-tree for non-volatile memory devices," *Cluster Computing*, pp. 1–17, 2022.

[64] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2015.

[65] *Apache Solr*, 2020 (accessed January 22, 2020). https://lucene.apache.org/solr/.

[66] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Greenwich, CT, USA: Manning Publications Co., 2010.

[67] P. Yang, H. Fang, and J. Lin, "Anserini: Enabling the use of lucene for information retrieval research," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, (New York, NY, USA), pp. 1253–1256, ACM, 2017. http://doi.acm.org/10.1145/3077136.3080721.

[68] T. C. Pan, S. Misra, and S. Aluru, "Optimizing high performance distributed memory parallel hash tables for dna k-mer counting," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 135–147, IEEE, 2018.

[69] A. Jain, S. P. Ong, G. Hautier, W. Chen, W. D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder, and K. A. Persson, "Commentary: The materials project: A materials genome approach to accelerating materials innovation," *APL Materials*, vol. 1, no. 1, p. 011002, 2013. https://doi.org/10.1063/1.4812323.

[70] R. W. Cottingham, "The doe systems biology knowledgebase (kbase): Progress towards a system for collaborative and reproducible inference and modeling of biological function," in *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '15, (New York, NY, USA), pp. 510–510, ACM, 2015. http://doi.acm.org/10.1145/2808719.2811433.

[71] E. Pietriga, H. Gözükan, C. Appert, M. Destandau, Š. Čebirić, F. Goasdoué, and I. Manolescu, "Browsing linked data catalogs with lodatlas," in *The Semantic Web – ISWC 2018* (D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee, and E. Simperl, eds.), (Cham), pp. 137–153, Springer International Publishing, 2018.

[72] P. Przybyla, A. J. Soto, and S. Ananiadou, "Identifying personalised treatments and clinical trials for precision medicine using semantic search with thalia," in *TREC*, 2018.

[73] D. Brickley, M. Burgess, and N. Noy, "Google dataset search: Building a search engine for datasets in an open web ecosystem," in *The World Wide Web Conference*, WWW '19, (New York, NY, USA), pp. 1365–1375, ACM, 2019. http://doi.acm.org/10.1145/3308558.3313685.

[74] B. Pratt, J. J. Howbert, N. I. Tasman, and E. J. Nilsson, "MR-Tandem: parallel X!Tandem using Hadoop MapReduce on Amazon Web Services," *Bioinformatics*, vol. 28, no. 1, pp. 136–137, 2011. https://doi.org/10.1093/bioinformatics/btr615.

[75] S. Lewis, A. Csordas, S. Killcoyne, H. Hermjakob, M. R. Hoopmann, R. L. Moritz, E. W. Deutsch, and J. Boyle, "Hydra: a scalable proteomic search engine which utilizes the hadoop distributed computing framework," *BMC bioinformatics*, vol. 13, no. 1, pp. 1–6, 2012.

[76] C. Kopanos, V. Tsiolkas, A. Kouris, C. E. Chapple, M. Albarca Aguilera, R. Meyer, and A. Massouras, "VarSome: the human genomic variant search engine," *Bioinformatics*, vol. 35, no. 11, pp. 1978–1980, 2018. https://doi.org/10.1093/bioinformatics/bty897.

[77] P. Corti, A. T. Kralidis, and B. Lewis, "Enhancing discovery in spatial data infrastructures using a search engine," *PeerJ Computer Science*, vol. 4, p. e152, 2018.

[78] D. Srivastava, A. Iyer, V. Kumar, and D. Sengupta, "CellAtlasSearch: a scalable search engine for single cells," *Nucleic Acids Research*, vol. 46, no. W1, pp. W141–W147, 2018. https://doi.org/10.1093/nar/gky421.

[79] S. K. Azman, M. Z. Anwar, and A. Henschel, "Visibiome: an efficient microbiome search engine based on a scalable, distributed architecture," *BMC Bioinformatics*, vol. 18, p. 353, July 2017. https://doi.org/10.1186/s12859-017-1763-0.