Journal of Parallel and Distributed Computing 189 (2024) 104878

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

SCIPIS: Scalable and concurrent persistent indexing and search in high-end computing systems ☆

Alexandru Iulian Orhean[a,*], Anna Giannakou[c], Lavanya Ramakrishnan[c], Kyle Chard[d], Boris Glavic[e], Ioan Raicu[b]

[a] *Jarvis College of Computing and Digital Media, DePaul University, Chicago, IL, USA*
[b] *College of Computing, Illinois Institute of Technology, Chicago, IL, USA*
[c] *Scientific Data Division, Lawrence Berkeley National Lab, Berkeley, CA, USA*
[d] *Department of Computer Science, University of Chicago, Chicago, IL, USA*
[e] *Department of Computer Science, University of Illinois, Chicago, Chicago, IL, USA*

ARTICLE INFO

Keywords:
Search engine architecture
High-performance indexing
High-performance storage
Scientific data

ABSTRACT

While it is now routine to search for data on a personal computer or discover data online, there is no such equivalent method for discovering data on large parallel and distributed file systems commonly deployed on HPC systems. In contrast to web search, which has to deal with a larger number of relatively small files, in HPC applications there is a need to also support efficient indexing of large files. We propose SCIPIS, an indexing and search framework, that can exploit the properties of modern high-end computing systems, with many-core architectures, multiple NUMA nodes and multiple NVMe storage devices. SCIPIS supports building and searching TFIDF persistent indexes, and can deliver orders of magnitude better performance than state-of-the-art approaches. We achieve scalability and performance of indexing by decomposing the indexing process into separate components that can be optimized independently, by building disk-friendly data structures in-memory that can be persisted in long sequential writes, and by avoiding communication between indexing threads that collaboratively build an index over a collection of large files. We evaluated SCIPIS with three types of datasets (logs, scientific data, and metadata), on systems with configurations up to 192-cores, 768 GiB of RAM, 8 NUMA nodes, and up to 16 NVMe drives, and achieved up to 29x better indexing while maintaining similar search latency when compared to Apache Lucene.

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." 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 accessible data. While much research effort has focused on the methods to efficiently store and process data, there has been relatively little focus on methods that efficiently explore, index, and search data using the same high-performance storage and compute systems. Users of large file systems typically 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 with many-cores, multiple NUMA nodes, and multiple PCIe NVMe SSDs.

While it is now trivial to quickly discover websites from the billions of websites accessible on the Internet, it remains surprisingly difficult for researchers to search for data 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.

☆ This work is supported in part by National Science Foundation CNS-1730689 CRI and OAC-2107548 Core awards.

* Corresponding author.
E-mail addresses: aorhean@depaul.edu (A.I. Orhean), agiannakou@lbl.gov (A. Giannakou), lramakrishnan@lbl.gov (L. Ramakrishnan), chard@uchicago.edu (K. Chard), bglavic@uic.edu (B. Glavic), iraicu@cs.iit.edu (I. Raicu).

https://doi.org/10.1016/j.jpdc.2024.104878
Received 14 July 2023; Received in revised form 13 December 2023; Accepted 13 March 2024
Available online 25 March 2024

In the enterprise search domain there are several tools that are commonly used to enable search, such as Apache Lucene [2], Apache Solr [21], and ElasticSearch [6]. According to surveys from both academia [10] and industry [4], 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 [22], which are not supported on HPC systems. Thus, there is a need to revisit indexing and search methods and the building blocks of search engines as new hardware emerges [14].

In order to address the general problem of efficient data exploration and search in large file systems, we initially explored the problem of in-memory indexing and search on single-node high-end systems, characterized by many-cores architectures, multiple NUMA nodes and multiple PCIe NVMe devices. We proposed SCANNS (Scalable and Concurrent Indexing and Searching System) [15,16], a single-node indexing framework that exposes the indexing pipeline, allowing the user to tune certain aspects of the pipeline, in order to saturate available compute, memory, and/or storage resources. We also included 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 evaluated the performance of SCANNS and of its components and we showed that it can achieve orders of magnitude higher indexing throughput and search latency when compared to Apache Lucene, a state-of-the-art information retrieval library.

In this work we explore the problems of efficiently building persistent indexes that cannot fit in main memory and of efficiently processing TFIDF (Term-Frequency Inverse-Document-Frequency) queries over the built persistent index, where the TFIDF scores are computed during query processing time and where the results are sorted by relevance. We propose SCIPIS (Scalable and Concurrent Persistent Indexing and Search System), 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 similar TFIDF query latency when compared to Apache Lucene. We evaluate the proposed solution over three types of datasets, log files, scientific papers and data, and supercomputing center file system metadata, and show that, while the inherent nature of each dataset can affect indexing and search performance, SCIPIS still exhibits good scalability trends.

The contributions of this paper 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);

The rest of the paper is organized as follows. In Section 2 we review related work. In Section 3 we present an overview of SCANNS, introduce the general architecture of the SCIPIS framework and explain the new optimizations and techniques that we used in its design. In Section 4 we present an experimental evaluation of the SCANNS and SCIPIS frameworks and conclude in Section 5.

2. Related work

Some 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 [23,18,24], that implements a scalable data management service framework for scientific datasets, that is integrated with the underlying distributed file systems such as GlusterFS and CephFS that are used to store scientific datasets. The framework relies on a scalable and distributed metadata indexing framework, that can index file system related metadata. The system also supports custom metadata created by the users in the form of *tags* that can aid data discovery. Another framework that aims at enabling and supporting extensive metadata management for scientific data found in high-performance computing systems is SoMeta [25]. SoMeta can support parallel and concurrent locate and retrieve operations and it utilizes Bloom filter to further improve search performance over user-level tagged objects.

In the are of parallel and distributed file systems, the integration of distributed hash tables [28] and/or log-structured trees [19] with the metadata subsystem has shown that the challenge of centralized file system metadata can be overcome and that file system can scale to thousands of nodes both in terms of data and metadata operations. Other existing works from the HPC domain (e.g. GUFI [3,7,13]) have also aimed to tackle the indexing and search problem focusing on metadata as opposed to the scientific data itself, while other work provides indexing and search over persistent memory object storage [11,12]. We believe both the metadata and data are critical components to better accessibility of scientific data.

ScienceSearch [20] is a project that proposes a novel solution for the problem of performing effective search over scientific data, that builds an indexing framework that uses natural language processing and machine learning to generate metadata tags from collections of scientific data and to build relations between different data sources that cover the same topic. ScienceSearch uses traditional databases to store and manage the learned metadata tags and the indexes, which has its own advantages and disadvantages, but we argue that an efficient low level indexing framework would be a suitable replacement for the databases used in search applications.

There are researchers who actively look at how to design and implement the inverted index for a specific dataset or application. MIQS [27] 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. 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 [9].

Cavast [26] is a another project that aims to improve the performance of in-memory key-value stores, through a re-design of their 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. While we acknowledge the importance

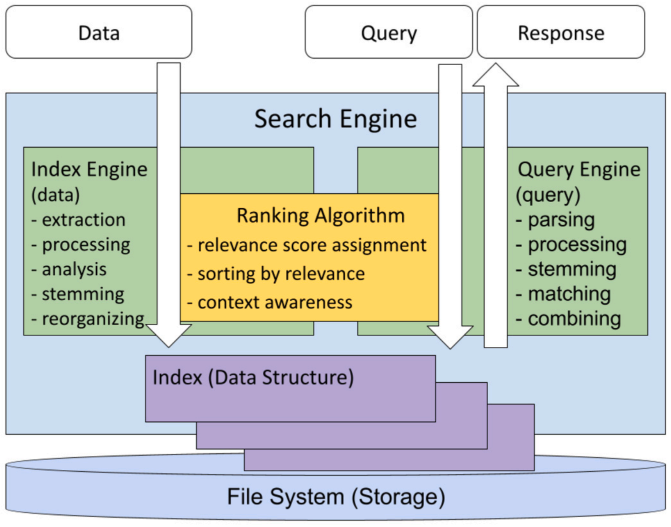


Fig. 1. General architecture of a search engine.

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 data structures.

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 high-performance computing 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 [17]. In the mentioned work, two distributed parallel hash table techniques are being proposed. These two techniques are optimized to use cache friendly algorithms for hashing, to overlap computation with communication 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.

3. Framework architecture and design

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

3.1. Search engine background

The problem that search engines solve in the realm of computers can be defined as the "problem of locating and retrieving relevant files from a file system in order to satisfy an information need" [1]. Fig. 1 shows a structural decomposition of the four main components of a search engine.

The *Index Engine* is responsible for extracting the contents of the files in order to re-organize it into an index. Similarity analysis and stemming are example of operations that this component can run to increase the quality of the index.

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 the 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 use the indexed frequencies 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 this model 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 present in the Index Engine component, and filters and sorts the returned results according to the Ranking Algorithm.

3.2. SCANNS framework overview

In order to efficiently leverage systems that have many cores, multiple NUMA nodes, and multiple NVMe storage devices and to saturate compute, memory and/or storage resources, 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 Fig. 2.

3.2.1. SCANNS components

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 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.

SCANNS is designed to implement 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. SCANNS assumes that the input dataset will not change while the index is being built and the framework is designed to support fixed-term and extended boolean search.

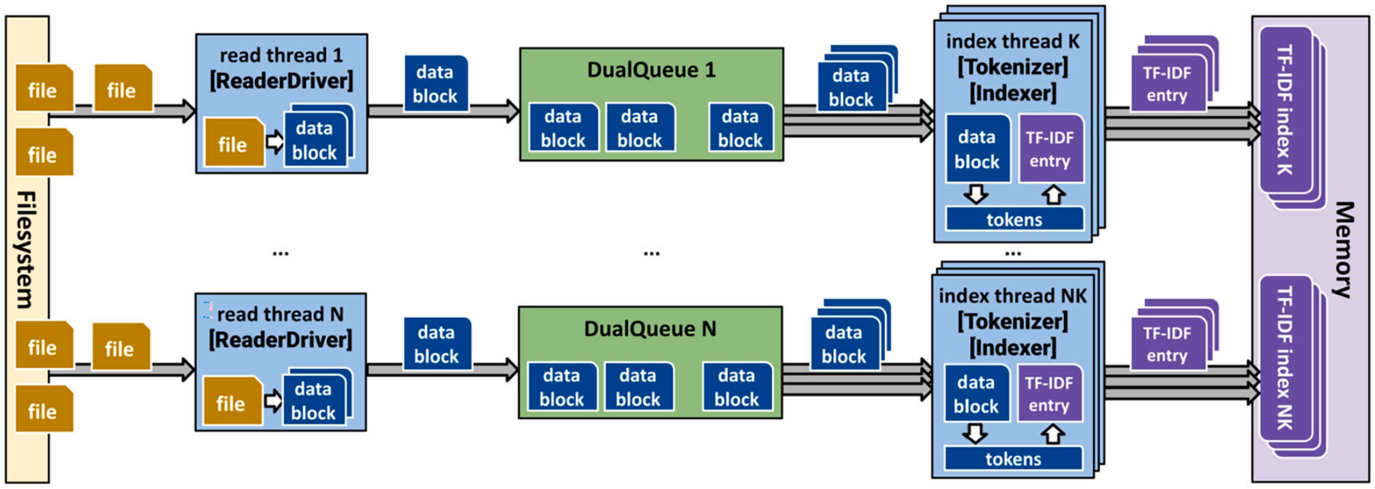

Filesystem
file
read thread 1 [ReaderDriver]
data block
DualQueue 1
index thread K [Tokenizer] [Indexer]
TF-IDF entry
tokens
TF-IDF index K
read thread N [ReaderDriver]
DualQueue N
index thread NK [Tokenizer] [Indexer]
TF-IDF index NK
Memory

Fig. 2. SCANNS framework indexing architecture and pipeline.

3.2.2. *SCANNS optimizations*

The *ReaderDriver* is the component that is responsible with reading data from the files stored in a file system as fast as possible to main memory so that the other components can further process it. This component reads blocks of data from disk and in order to increase the read performance of the blocks we adopted the memory pool design pattern and we used the DIRECT_IO flag. The memory pool design pattern is implemented as part of the *DualQueue* component, which allocates the blocks of data where the content of the files will be read at the beginning of the program and recycles them throughout the execution, eliminating the penalties incurred by constant memory deallocation and reallocation. We saw in practice that the operating system will try to cache the data read from disk, but since this data is read only once, that is not necessary. In order to avoid unnecessary memory allocation for caches and buffers, the *ReaderDriver* uses the DIRECT_IO flag to tell the IO subsystem to read data straight to the user space blocks of data. The SCANNS framework adds one more application-level detail to the implementation of the *ReaderDriver*, by implementing a mechanism that returns variable sized blocks of data that avoids splitting words or tokens in order to preserve correctness. This application-level detail is implemented as part of the *WaveReaderDriver* component, which is used throughout all of the SCANNS framework.

The *Tokenizer* is the next component in the indexing pipeline and is responsible with extracting the words or tokens sparated by one or more delimiters from the data block and passing the list of tokens as fast as possible to the next component. The *Tokenizer* is a CPU intensive component, because it processes the data by accessing it sequentially and thus benefits from the CPU cache system. In order to further optimize the process of tokenizing data, the SCANNS framework implements what we call a *BranchlessTokenizer*, that uses branchless programming and a delimiter hashtable. Through branchless programming we replace the part of the code, that decides if a new token was discovered through an *if-else* block, with ternary operations that will allow the compiler to generate conditional instructions. The conditional instructions bypass the CPU branch predictor allowing for a smoother scheduling of instructions on the CPU pipelines. The delimiter hashtable allows the *Tokenizer* to identify and replace a delimiter with a null byte in one single lookup operation, that is faster that iterating and comparing over the list of delimiters of each character.

The next and final component in the SCANNS indexing pipeline is the *Indexer*, which is responsible with reading the tokens received from the *Tokenizer* and writing them to a local TFIDF index stored in memory. The TFIDF index is built using a hashtable as a search data structure and a linked list to store the auxiliary inverted index information and keeps track, for each token, of the number of files the tokens appear in and for each of those files the number of times the tokens appear in the file. In order to further optimize the process of building the inverted index, which is the slowest component, due to the fact that it is memory intensive, exhibits random access patterns and does not benefit from the CPU cache subsystem, we adopted an append cache mechanism and a monotonic-page suballocator. The append cache, exploits the fact that SCANNS builds the index for the first time and that each file will only be read and indexed once, and adds an additional structure to the elements stored in the hashtable in order to minimize the number of memory indirections whenever the frequency of words in the same file is being incremented. The monotonic-page suballocator exploits the same principle and allocates the elements of the linked lists, or better said the auxiliary data structure of the inverted index, in pages that will never be deallocated, while also minimizing the penalties of memory allocation. The linked list elements will then be assigned from user space, thus further increasing the performance of this component.

Lastly, the SCANNS framework adopts two global optimizations that apply to all of the aforementioned components. The first optimization makes use of NUMA affinity/scheduling to group reader threads and indexer threads such that inter-NUMA communication is minimized, which leads to a significant increase in performance and good scalability for all components. The second optimization makes use of Linux huge pages to reduce the number of page faults. Because SCANNS was designed to index data in memory as fast as possible, the framework will try to use all of the available memory, which sometimes can be very large and which in turn will cause many page faults when allocating memory. Using huge pages, allows SCANNS to still utilize all of the memory with a reduced performance impact from the memory subsystem and thus increasing the overall performance of the framework.

3.3. *SCIPIS framework architecture*

The SCIPIS framework extends the SCANNS framework and follows a similar architecture. It reuses the *WaveReaderDriver* and *BranchlessTokenizer* components, as the default components responsible for 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. SCIPIS replaces the *Indexer* component with two components: *FilePathIndexer* and *TFIDFIndexer*, that are responsible for indexing the file path and the file content, respectively. This separation of components allows the user to finely tune the two aspects of processing indexes but also allows the framework to separately persist file path and file content indexes. Finally, SCIPIS adopts a new component, called the *IndexWriter*, that is responsible

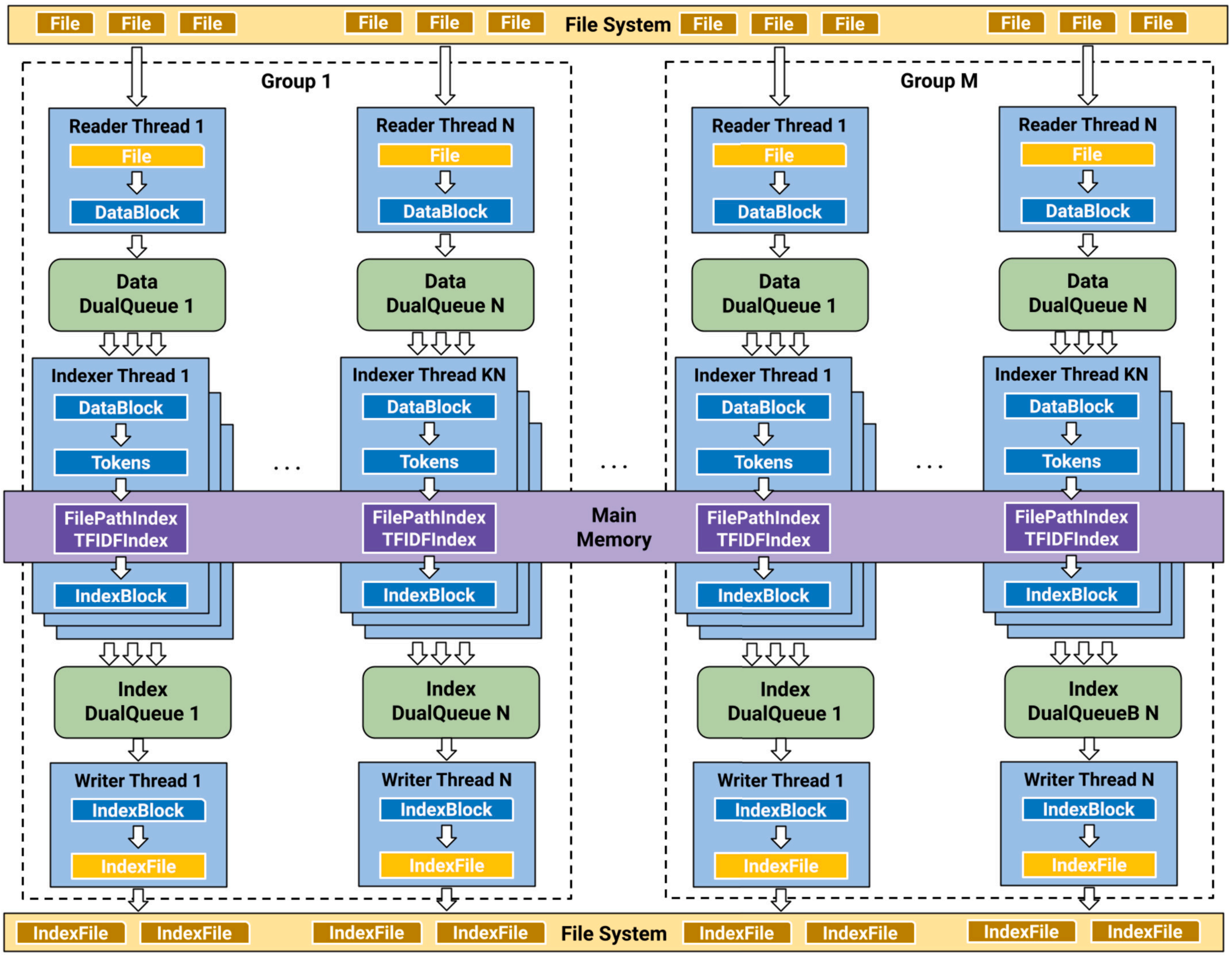


Fig. 3. SCIPIS framework indexing architecture and pipeline.

with reading blocks of index data and writing them as fast as possible to corresponding files in a file system. In terms of optimizations, the SCIPIS framework adopts all of the SCANNS component and system optimizations, including: direct IO, memory pool design pattern, branchless programming and the delimiter hashtable, the append cache mechanism and monotonic-page suballocator, NUMA affinity/scheduling and Linux huge pages; and incorporates two new optimizations that aim to capture the characteristics of the input dataset: tunable index depth and split factor.

The SCIPIS framework, like 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, removed 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 users could run offline on existing built indexes. Indexes could be optimized, merged and updated using common set operations, such as: union, intersection and difference. In terms of searching the persistent index, SCIPIS supports, unlike SCANNS, 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 scores.

3.3.1. *Indexing engine execution*

Fig. 3 shows the architecture of the SCIPIS indexing pipeline and the flow of data throughout the framework. One difference from the SCANNS framework, is that SCIPIS takes as an input a collection of input files, processes partial indexes in-memory, and returns as an output another collection of files that contain the re-organized input data, commonly known as the inverted index. The input data is consumed and transformed by the SCIPIS framework by various components that are responsible for 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.

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 read the files that need to be indexed from a queue that is populated by a number of simple file systems crawlers, with one crawler per NUMA node. This allows the framework to achieve load balancing over the files read from one NUMA node, that have similar sizes. For files, that reside in the same NUMA node and that vary in size, the simple crawler could be expanded to support pushing into the queue the files that are larger first, in order to avoid any load balancing issues. In the scenario of load inbalance across NUMA nodes, the framework could use a work-stealing technique to achieve load balance, allowing reader threads to pop files from queues from different NUMA nodes. However, work-stealing could cause performance degradation because of inter-NUMA communication, and is the subject of future work. The reader threads are IO-intensive and can be over-provisioned. In addition to the blocks of data containing the content of the input files, the reader threads will send an additional block of data containing the full file path to the indexer threads in order to index the file path alongside the content of the files.

The indexer threads are responsible for receiving blocks of data from the reader threads, indexing the data from the blocks and sending blocks of index data to the writer threads to be stored to disk. To accomplish this, each indexer thread manages three components: a *BranchlessTokenizer*, a *FilePathIndex* and a *TFIDFIndex*; each of them having different roles and using different system resources. The *BranchlessTokenizer* is responsible for 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 for 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 a list of extracted tokens, is then indexed by the *TFIDFIndex*, which is also a memory-intensive component and that indexes the tokens and keeps track of the term frequencies and inverse document frequencies necessary for computing the 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 ratio 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 be subject to 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 to disk. The current implementation of the serialization of indexes 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 Buffers or FlatBuffers 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 over-provision the indexer threads to the number of hardware-threads 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 an 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 significantly varying sizes 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.

The SCIPIS framework, currently, does not implement any means for achieving fault tolerance when the compute node fails. However, fault tolerance can be implemented in a very easy way, by keeping track of high watermark records for files and positions inside large files. The framework can crawl the file system and generate a list of files to be indexed, in a deterministic and idempotent way, by, for example, sorting the directories and files per directory (in order to keep the sorting tasks small). Then, during indexing, the framework can keep track in a log of which files have been successfully indexed and, for large files, of how many blocks of data have been successfully indexed, by updating a high watermark per file, for small file, and per block, for large files, whenever SCIPIS flushes the index to disk. If the node crashes during indexing, the framework would then need to read the log file in order to figure out which files or blocks of data from large files have been successfully indexed and in order to determine from which point can the framework continue indexing.

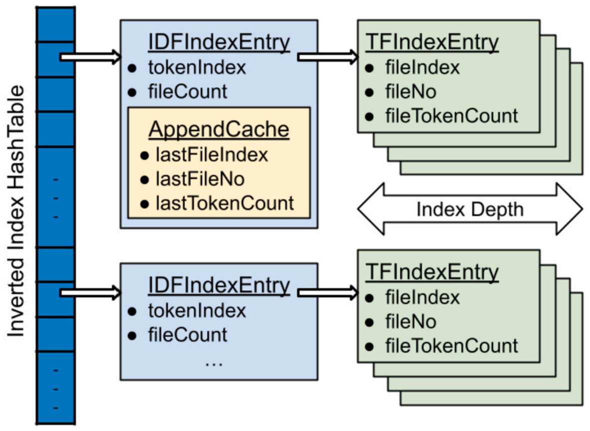


Fig. 4. SCIPIS inverted index design.

3.3.2. 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 creating 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 to use 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. Fig. 4 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. For example, unsigned integer values, calculated using a hash function, are used to provide file path and token identification. 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 into 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 many 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 maximum capacity, the inverted index signals the parent component that it is ready to serialize and flush the index 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.

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.

3.3.3. *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. Fig. 5 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 known, 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

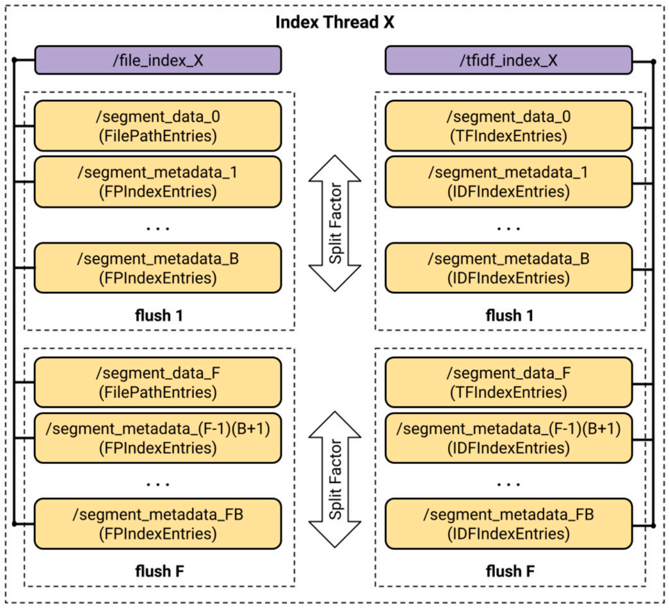

Index Thread X
/file_index_X
/tfidf_index_X
/segment_data_0 (FilePathEntries)
/segment_metadata_1 (FPIndexEntries)
...
/segment_metadata_B (FPIndexEntries)
flush 1
/segment_data_0 (TFIndexEntries)
/segment_metadata_1 (IDFIndexEntries)
...
/segment_metadata_B (IDFIndexEntries)
flush 1
Split Factor
/segment_data_F (FilePathEntries)
/segment_metadata_(F-1)(B+1) (FPIndexEntries)
...
/segment_metadata_FB (FPIndexEntries)
flush F
/segment_data_F (TFIndexEntries)
/segment_metadata_(F-1)(B+1) (IDFIndexEntries)
...
/segment_metadata_FB (IDFIndexEntries)
flush F
Split Factor

Fig. 5. SCIPIS persistent index design.

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 of the 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 access, for a given term, only 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 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 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.

The persistent index is produced by the SCIPIS framework in an immutable way and the framework itself does not yet provide mechanisms for updating the index in-place. However, additional programs can easily be built (as part of the framework) that can update the persistent index offline. One property of splitting the data and metadata components of the inverted index into multiple segments is that the index information for a file will reside in only one segment. The advantage of this property is that updating the index, when a new file is being added or when an old file is being modified, can be easily implemented by adopting similar techniques to log-structured merge trees. Since the

Table 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	192 GiB DDR4	16 x Samsung 970 EVO SSD
192cores-16disks	8 x Intel Xeon Platinum 8160	192	768 GiB DDR4	16 x Intel Optane 900P SSD

search operations have to access all of the metadata segments from a specific hash table key space, determined by the *split factor*, we can include at index creation time for each metadata segment a timestamp, and we can filter out at query time old metadata segments, thus always working on the most up to date index. Thus, the user, would only need to produce a new and smaller persistent index from the new and/or modified files, and then combine it with the existing persistent index, in order to update the index. An offline program could be additionally built to remove metadata segments that are old and merging metadata segments that are small, thus clearing up space and improving the performance of search operations when frequent updates happen. As for updating the persistent index after files have been removed from the file system, the framework could implement a tombstone mechanism similar to the one found in databases, where an index entry can be marked as removed and thus ignored during search time. The framework could further provide a mechanism (similar to database vacuum) for removing the tombstone marked entries from the persistent index in order to reclaim space.

4. Performance evaluation

In this section, we present the performance evaluation of the SCANNS and SCIPIS frameworks. We include details about the SCANNS variants, the end-to-end indexing performance of SCIPIS, the TFIDF search performance, the tuning experiments, but also descriptions of the experimental setup and the datasets used for the experiments.

4.1. 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 systems have a high number of PCIe NVMe storage devices allowing for fast storage access in terms of both read and write, and sequential and random access. The difference between the two systems 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 framework 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 deploy, while the big machine (*192cores-16disks*) is more akin to a supercomputer node, in terms of CPU and memory capabilities. Table 1 shows the hardware details for each system.

In order to achieve high execution performance, we configured the operating system to use the performance scheduling governor 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 eliminate 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 across the NUMA nodes and made sure that threads only accessed data from their own NUMA node.

For both SCANNS and SCIPIS, all the machines ran Ubuntu 22.04 LTS with Linux Kernel 5.15 and g++-12.1. For the Google Swiss Table library we used version 20230125.3 lts from the abseil library and we used openjdk-11 to run Apache Lucene. Both frameworks are implemented in C++20 and are compiled with the O3 optimization flag enabled.

Table 2
Evaluation datasets and characteristics.

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

For all of the experiments and all machines we configured the SCIPIS framework with a file path index size of 64 MiB, a TFIDF index size of 512 MiB and a split factor of 16 for each indexer thread. This implies that the TFIDF index will be split into 16 32 MiB non-overlapping hash table regions and that the maximum TFIDF metadata segment files size will be 32 MiB. The average size of the produced TFIDF metadata segment file is 22 MiB. We determined and chose the split factor experimentally and because it allows the framework to a achieve higher search performance at the cost of a slight decrease in indexing performance by about 5%. The size of the data segment file and the number of metadata segment files depend on the input dataset size and, given the mentioned configurations, the overall produced index is between 20% and 50% the size of the original input data.

4.2. Evaluation datasets

For the performance evaluation of SCIPIS we used five datasets, that represent three different scenarios or dataset types, and that have different properties. The datasets, with the properties and characteristics of each, can be seen in Table 2. We picked a diverse range of datasets in order to explore the scalability and performance of SCIPIS, and to show that while the performance can change from dataset to dataset, the scalability trends remain similar.

The first type of dataset we consider is log data. Log data is commonly found in all computing systems and is especially prevalent on supercomputers. 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. Logs are notoriously hard to search through without the help of an index-based search engine. Thus we evaluate SCIPIS using three log datasets, because it is a representative scenario. We used the *hdfs* cluster, *thunderbird* supercomputer, and *windows* operating system logs form the Loghub collection [8].

For the second type of dataset we consider a collection of scientific datasets. The Pile [5] collection contains publications, websites and books, from various fields of science, including medicine, law, and mathematics. The Pile is representative of 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 also include a dataset to explore search over the metadata of the file system. Scientists and engineers often use various naming conventions for directories and files for organization and provenance, but these conventions pose an interesting scenario when it comes to indexing and searching scientific data. Thus we eval-

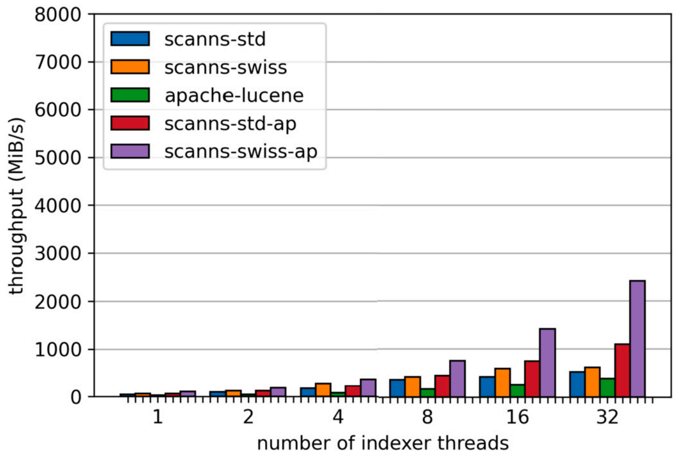


Fig. 6. End-to-end TFIDF indexing throughput with increasing number of read and index threads on *32cores-16disks*.

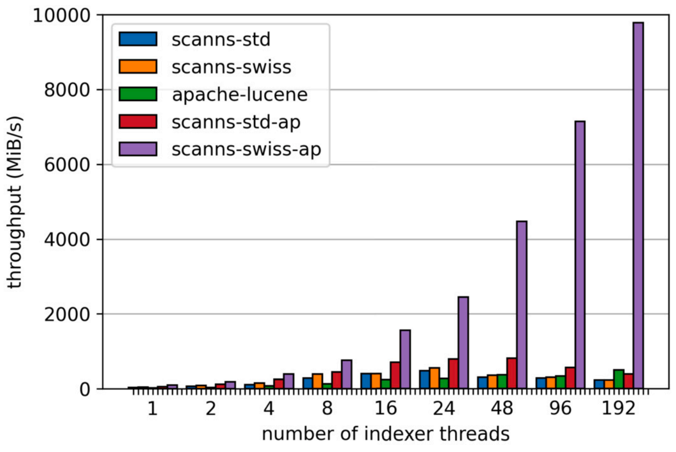


Fig. 7. End-to-end TFIDF indexing throughput with increasing number of read and index threads on *192cores-16disks*.

Table 3
End-to-end search latency (microseconds). SCANNS returns unsorted and unmerged results, while Lucene returns fully computed, sorted and merged TFIDF results.

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

uate SCIPIS over the file system metadata provided from a supercomputing center, that we will refer to as the *fsdumps*. SCANNS was also originally evaluated over the *fsdumps* dataset. For the SCANNS experiments we cleaned and split the file system dump dataset into smaller files of approximately and up to 32 MiB in size, and the evaluation was conducted over a collection of 1536 files (48 GiB).

4.3. SCANNS performance evaluation

The TFIDF End-to-End indexing and search evaluation of SCANNS is performed on variants that include both the WaveReaderDriver and the BranchlessTokenizer 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 1 GiB 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:

- *scanns-std* - implementation using C++ unordered_map and without any optimizations;
- *scanns-swiss* - implementation using Google Swiss Table and without any optimizations;
- *apache-lucene* - uses Apache Lucene;
- *scanns-std-ap* - similar to xs-rdtokidx-std, plus the monotonic paged sub-allocator, the append cache optimization, NUMA-aware configurations and huge pages;
- *scanns-swiss-ap* - similar to xs-rdtokidx-swiss, plus the monotonic paged sub-allocator, the append cache optimization, NUMA-aware configurations and huge pages;

4.3.1. SCANNS end-to-end TFIDF indexing and search

When looking at a system that has multiple NVMe devices but not that many cores, as depicted in Fig. 6, we see a similar trend. The unoptimized 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 369 MiB/sec for Apache Lucene, 607 MiB/sec for the Swiss Table implementation and 509 MiB/sec for the standard implementation. Only by incorporating the memory and NUMA-aware optimizations can the standard implementation reach 1093 MiB/sec and the Swiss Table implementation reach 2414 MiB/sec, both with 32 index threads and 32 read threads.

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. Fig. 7 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 501 MiB/sec, the standard Indexer caps at 475 MiB/sec and the Swiss Table Indexer caps at 551 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 810 MiB/sec, with 24 index threads and 24 read threads, while the Swiss Table Indexer can reach a whopping 9782 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.

Table 3 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 two different systems. The SCANNS variant exhibits magnitudes lower latency, overall under 500 microseconds, when compared to the Lucene variant that runs search

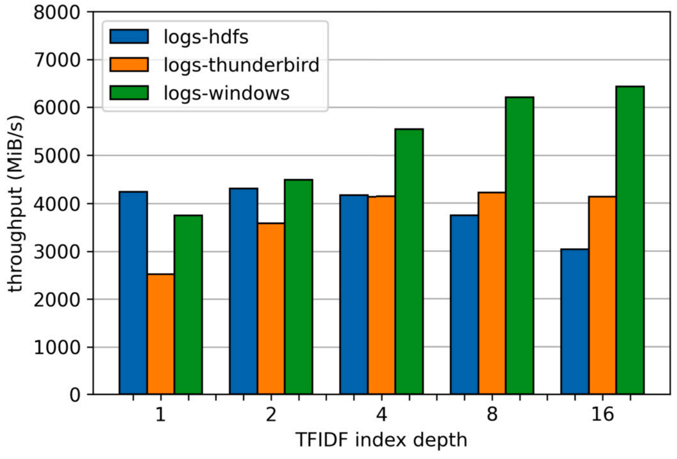


Fig. 8. Tuning the index depth on *32cores-16disks*.

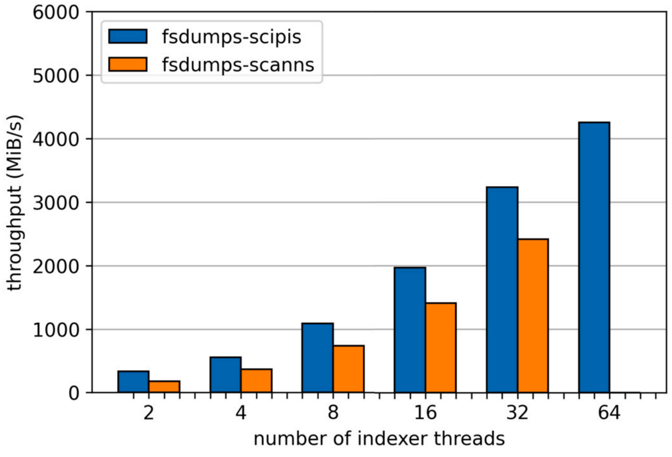


Fig. 9. SCIPIS vs SCANNS indexing throughput the *32cores-16disks* machine.

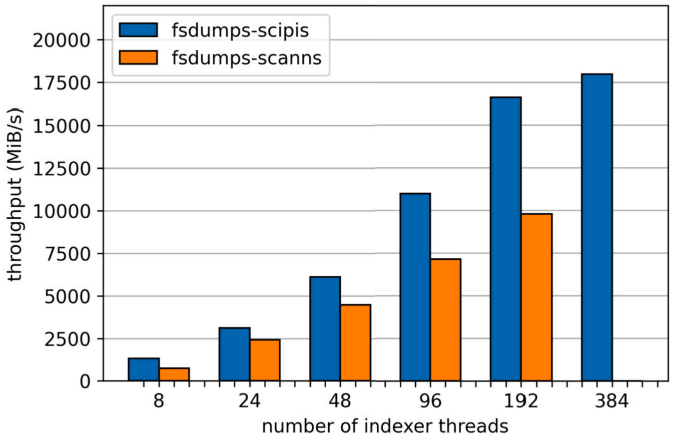


Fig. 10. SCIPIS vs SCANNS indexing throughput the *192cores-16disks* machine.

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. We explore the impact of sorting the results on the performance of queries in the performance evaluation of SCIPIS.

4.4. SCIPIS performance evaluation

We initially show the performance implications of tuning the *index depth*, after which we provide a comparison between SCIPIS and SCANNS in terms of indexing throughput. Afterwards we provide an analysis of the indexing performance of SCIPIS with various datasets, which is followed by a comparison between SCIPIS and Apache Lucene in terms of full TFIDF search latency.

4.4.1. SCIPIS index depth tuning

Fig. 8 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 1 MiB, a file path index maximum size of 128 MiB and a TFIDF index maximum size of 1 GiB.

For the *hdfs* dataset, it can be seen that SCIPIS 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.

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, SCIPIS is able 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. 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.

4.4.2. 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. Figs. 9 and 10 show the indexing throughput of both SCIPIS and SCANNS on the *32cores-16disks* machine and the *192cores-16disks* machine, respectively. In both plots we did not include results 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.

From Fig. 9 we can see that, on the *32cores-16disks* machine, the indexing throughput of SCANNS increased from 185 MiB/s with 2 indexer threads to 2.4 GiB/s with 32 indexer threads, while for SCIPIS the indexing throughput increased from 338 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 times more data than SCANNS. When we say 12 times more data we mean that since SCIPIS can store the index on disk and the disk is 12 times larger (including only for index size and not input data size) than main memory, we were able to index more data.

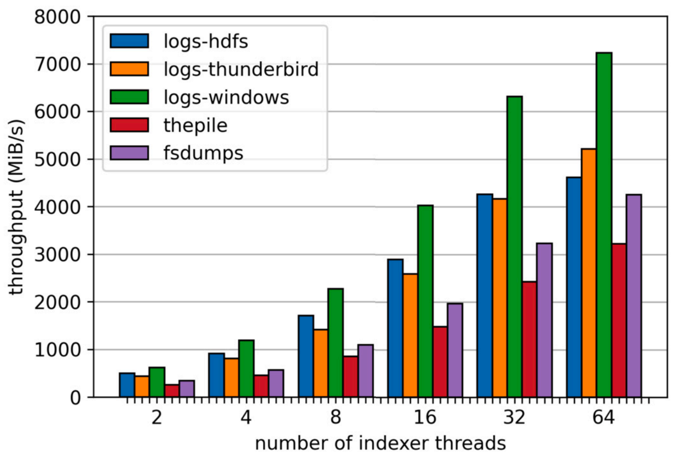


Fig. 11. SCIPIS indexing throughput the *32cores-16disks* machine.

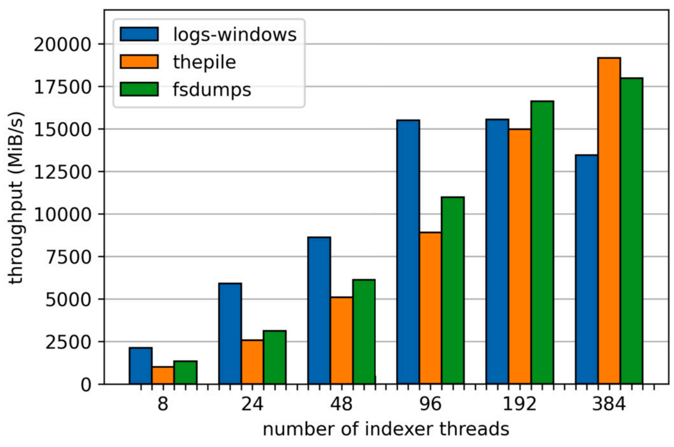


Fig. 12. SCIPIS indexing throughput the *192cores-16disks* machine.

On the *192cores-16disks* machine, SCANNS yielded an indexing throughput of 760 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. When we say 24 times more data we mean that since SCIPIS can store the index on disk and the disk is 24 times larger (including only for index size and not input data size) than main memory, we were able to index more data.

4.4.3. *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. 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. Figs. 11 and 12 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 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 Fig. 11 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 even better performance than 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 means 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 were the largest 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.

In the case of the *192cores-16disks*, Fig. 12 contains the results of the indexing throughput performance evaluation with increasing number of indexer threads. We did not consider the *hdfs* and *thunderbird* with this machine, because the number of files in these two datasets is less than 384 (the number of hardware threads), which would create load-balancing issues and would result in incorrect performance numbers. It can 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. 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 throughput 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 many more 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.

4.4.4. *TFIDF search latency*

The SCIPIS framework supports TFIDF queries—that compute the TFIDF scores for the returned documents given the query term, and that sort and select only the most relevant documents. We 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 contains the average search latency measured in microseconds. We can see that on both machines, SCIPIS performs relatively similarly to Apache Lucene in processing TFIDF queries, albeit slower over the *hdfs* and *thunderbird* datasets that have a small number of large files, but slightly faster on the *windows* dataset that contains many small files, especially with increasing number of indexer threads. We 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

Table 4
SCIPIS vs Apache Lucene TFIDF search latency (microseconds).

cores	32cores-16disks				192cores-16disks	
	hdfs-scipis	hdfs-lucene	thunderbird-scipis	thuderbird-lucene	windows-scipis	windows-lucene
2	3958	1686	5823	2465	-	-
4	3782	2261	6707	1903	-	-
8	4138	1200	10719	3783	2568	4279
16	6309	1744	10684	2574	-	-
24	-	-	-	-	3219	5756
32	9168	1625	12526	7129	-	-
48	-	-	-		7135	5268
64	18246	3159	17835	3790	-	-
96	-	-	-	-	8936	6069
192	-	-	-	-	8980	9485
384	-	-	-	-	14083	17934

a lower search latency. For example, for the *hdfs* dataset on the *32cores-16disks* machine, with 2 indexer threads, SCIPIS can run a query in 3.7 milliseconds, while with 64 indexer threads, the query latency increases to approximately 18 milliseconds. For the *windows* dataset 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.

5. Conclusion

In this work we proposed and presented SCIPIS, a single-node indexing and search framework that can efficiently build persistent indexes that cannot fit into memory and efficiently search persistent indexes on high-end computing systems characterized by many cores, multiple NUMA nodes and multiple PCIe NVMe devices. SCIPIS inherits the properties and architecture of the SCANNS framework, that exposes the inverted index pipeline interface, allowing easy modification and tuning, enabling this framework to saturate modern storage, memory and compute resources on a variety of hardware. 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 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, 29x better indexing performance than Apache Lucene and similar search performance to 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 their 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.

Some future work we plan on undertaking are to explore semi-automatic hyper-parameter tuning as part of the SCIPIS framework to allow easier selection of key parameters that govern the performance on a particular hardware or for a particular dataset. We plan to explore distributed indexing and search to allow scalability to some of the largest HPC storage systems available. Finally, we plan to integrate the distributed SCIPIS system into parallel and distributed storage systems to enable automatic metadata and data indexing and search.

CRediT authorship contribution statement

Alexandru Iulian Orhean: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, Writing – original draft, Writing – review & editing. **Anna Giannakou:** Investigation, Validation, Writing – original draft. **Lavanya Ramakrishnan:** Investigation, Validation, Visualization, Writing – original draft. **Kyle Chard:** Conceptualization, Data curation, Investigation, Validation, Visualization, Writing – original draft. **Boris Glavic:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Ioan Raicu:** Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Supervision, Validation, Visualization, Writing – original draft.

Declaration of competing interest

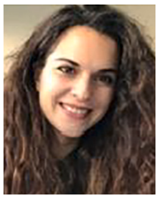
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] R. Baeza-Yates, B. Ribeiro-Neto, et al., Modern Information Retrieval, vol. 463, ACM Press, New York, 1999.
[2] A. Białecki, R. Muir, G. Ingersoll, L. Imagination, Apache lucene 4, in: SIGIR 2012 Workshop on Open Source Information Retrieval, 2012, p. 17.
[3] D.J. Bonnie, Gufi overview, Tech. Rep., Los Alamos National Lab. (LANL), Los Alamos, NM (United States), 2018.
[4] Datanyze, Enterprise search software market share, https://www.datanyze.com/market-share/enterprise-search--287, 2022.
[5] 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.
[6] C. Gormley, Z. Tong, Elasticsearch: the Definitive Guide: a Distributed Real-Time Search and Analytics Engine, O'Reilly Media, Inc., 2015.
[7] G.A. Grider, D.A. Manno, W.K. Poole, D.J. Bonnie, J.T. Inman, Grand unified file indexing, Tech. Rep., Los Alamos National Lab. (LANL), Los Alamos, NM (United States), 2021.
[8] S. He, J. Zhu, P. He, M.R. Lyu, Loghub: a large collection of system log datasets towards automated log analytics, arXiv preprint arXiv:2008.06448, 2020.
[9] S. Jamil, A. Salam, A. Khan, B. Burgstaller, S.-S. Park, Y. Kim, Scalable numa-aware persistent b+-tree for non-volatile memory devices, Clust. Comput. (2022) 1–17.
[10] S. Khalsa, P. Cotroneo, M. Wu, A survey of current practices in data search services, Research Data Alliance Data (RDA) Discovery Paradigms Interest Group, 2018.
[11] A. Khan, H. Sim, S.S. Vazhkudai, J. Ma, M.-H. Oh, Y. Kim, Persistent memory object storage and indexing for scientific computing, in: 2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), IEEE, 2020, pp. 1–9.
[12] A. Khan, H. Sim, S.S. Vazhkudai, Y. Kim, Mosiqs: persistent memory object storage with metadata indexing and querying for scientific computing, IEEE Access 9 (2021) 85217–85231.

[13] D. Manno, J. Lee, P. Challa, Q. Zheng, D. Bonnie, G. Grider, 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, IEEE, 2022, pp. 1–14.
[14] A.I. Orhean, I. Ijagbone, I. Raicu, K. Chard, D. Zhao, Toward scalable indexing and search on distributed and unstructured data, in: 2017 IEEE International Congress on Big Data (BigData Congress), IEEE, 2017, pp. 31–38.
[15] A.I. Orhean, A. Giannakou, L. Ramakrishnan, K. Chard, 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), IEEE, 2022, pp. 51–60.
[16] A.I. Orhean, K. Chard, I. Raicu, Scalable indexing and search in high-end computing systems, Ph.D. thesis, Illinois Institute of Technology, Department of Computer Science, 2023.
[17] T.C. Pan, S. Misra, 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, IEEE, 2018, pp. 135–147.
[18] A.K. Paul, B. Wang, N. Rutman, C. Spitz, A.R. Butt, Efficient metadata indexing for hpc storage systems, in: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), IEEE, 2020, pp. 162–171.
[19] K. Ren, Q. Zheng, S. Patil, G. Gibson, Indexfs: scaling file system metadata performance with stateless caching and bulk insertion, in: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2014, pp. 237–248.
[20] G.P. Rodrigo, M. Henderson, G.H. Weber, C. Ophus, K. Antypas, L. Ramakrishnan, Sciencesearch: enabling search through automatic metadata generation, in: 2018 IEEE 14th International Conference on e-Science (e-Science), IEEE, 2018, pp. 93–104.
[21] D. Shahi, Apache solr: an introduction, in: Apache Solr, Springer, 2015, pp. 1–9.
[22] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Ieee, 2010, pp. 1–10.
[23] H. Sim, Y. Kim, S.S. Vazhkudai, G.R. Vallée, S.-H. Lim, 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, 2017, pp. 1–12.
[24] H. Sim, A. Khan, S.S. Vazhkudai, S.-H. Lim, A.R. Butt, Y. Kim, An integrated indexing and search service for distributed file systems, IEEE Trans. Parallel Distrib. Syst. 31 (10) (2020) 2375–2391.
[25] H. Tang, S. Byna, B. Dong, J. Liu, Q. Koziol, Someta: scalable object-centric metadata management for high performance computing, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2017, pp. 359–369.
[26] K. Wang, J. Liu, F. Chen, Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores, Proc. VLDB Endow. 13 (9) (2020).
[27] W. Zhang, S. Byna, H. Tang, B. Williams, 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, 2019, pp. 1–24.
[28] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, I. Raicu, Fusionfs: toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems, in: 2014 IEEE International Conference on Big Data (Big Data), IEEE, 2014, pp. 61–70.

Alexandru Iulian Orhean is an Assistant Professor in Computer Science in the School of Computing at the Jarvis College of Computing and Digital Media at DePaul University. He received his PhD in Computer Science in 2023 from Illinois Institute of Technology under Professor Ioan Raicu. He received his BS in Computer Science in 2016 from University Politehnica of Bucharest. His research expertise resides at the intersection of Parallel/Distributed Systems and Information Retrieval, currently working on the problem of efficient and effective indexing and search in large-scale scientific storage systems, and on the problem of efficient data organization, visualization and search in data science and machine learning. His broad research interests revolve around the areas of data organization, storage and exploration, information discovery, retrieval and search, distributed indexing and search engine design, high-performance computing systems, cloud systems and databases.



Dr. Anna Giannakou is a research scientist at Lawrence Berkeley National Lab. Her research interests include machine learning for intelligent network management and end-to-end infrastructure adaptation. She is currently working on methods and tools to verify AI-supported system adaptation decisions. Anna has served as a program committee member for various conferences. Anna Giannakou received her Ph.D. from Institut National des Sciences Appliquées at Inria Rennes under the supervision of Dr. Christine Morin and Prof. Jean-Louis Pazat. In Dr. Christine Morin's research group, Anna's research focused on self-adaptable security monitoring for cloud environments. Anna holds a master's in Information Security from the University of Luxembourg and a Bachelor's in Computer Science from the University of Athens.

Boris Glavic is an Associate Professor of Computer Science at University of Illinois at Chicago. He received his PhD in Computer Science from the University of Zurich in Switzerland being advised by Gustavo Alonso and Michael Böhlen. Afterwards, he did spend two years as a PostDoc in the Department of Computer Science at the University of Toronto working with Renée J. Miller. His research spans several areas of database systems and data science including data provenance and explanations, data integration, query execution and optimization, uncertain data, and data curation. Boris strives to build systems that are based on solid theoretical foundations.

Ioan Raicu is an associate professor in Computer Science at Illinois Institute of Technology, as well as a guest research faculty in the Math and Computer Science Division at Argonne National Laboratory. His research work and interests are in the general area of distributed systems. He obtained his MS and PhD degree in Computer Science from University of Chicago under Prof. Ian Foster in 2005 and 2009 respectively. He obtained his BS and MS in Computer Science from Wayne State University in 2000 and 2002 respectively. His research work has focused on resource management in large scale distributed systems with a focus on many-task computing, data intensive computing, cloud computing, grid computing, and many-core computing. Over the past decade, he has co-authored over 140 peer reviewed articles, which received over 12K citations, with a H-index of 46. He has been a TED speaker advocating for a renewed interest in computer science and has been a strong supporter of engaging students early in their educational careers (high school and undergraduate students) to prepare them for graduate school and careers in research.

Kyle Chard is a Research Associate Professor in the Department of Computer Science at the University of Chicago. He also holds a joint appointment at Argonne National Laboratory. He received his Ph.D. in Computer Science from Victoria University of Wellington, New Zealand in 2011. He is a member of the ACM and IEEE, received the IEEE TCHPC Award for Excellence for Early Career Researchers in HPC, was part of the Globus team that won an R&D100 award, and received the New Zealand Top Achiever Doctoral Scholarship. He co-leads the Globus Labs research group, which focuses on a broad range of research problems in data-intensive computing and research data management.

Dr. Lavanya Ramakrishnan is Senior Scientist and Division Deputy in the Scientific Data Division at Lawrence Berkeley National Lab and Deputy Project Director for the High Performance Data Facility (HPDF). Her research interests are in building software tools for computational and data-intensive science with a focus on workflow, resource, and data management. More recently, her work explores the methods and infrastructure needed to support automation and self-driving labs. In addition, Ramakrishnan established and leads a scientific user research program focusing on studying and enumerating the way that scientists and communities use data and workflows to build usable tools for science. She currently leads several project teams that consist of a mix of social scientists, software engineers, and computer scientists.

Ramakrishnan serves on the High Performance Distributed Computing Steering Committee, iHARP NSF HDR InstituterQOs Advisory board and has previously served as the Associate Editor for Journal of Parallel and Distributed Computing and as program committee chair for various conferences. She has masters and doctoral degrees in computer science from Indiana University and a bachelor degree in computer engineering from VJTI, University of Mumbai. She joined Berkeley Lab as an Alvarez Fellow. Previously she has worked as a research software engineer at Renaissance Computing Institute and MCNC in North Carolina