SCANNS: Towards Scalable and Concurrent Data Indexing and Searching in High-End Computing System

Alexandru Iulian Orhean
College of Computing
Illinois Institute of Technology
aorhean@hawk.iit.edu

Anna Giannakou
Scientific Data Division
Lawrence Berkeley National Lab
agiannakou@lbl.gov

Lavanya Ramakrishnan
Scientific Data Division
Lawrence Berkeley National Lab
lramakrishnan@lbl.gov

Kyle Chard
Department of Computer Science
University of Chicago
chard@uchicago.edu

Ioan Raicu
College of Computing
Illinois Institute of Technology
iraicu@cs.iit.edu

***Abstract*—Increasing data volumes, particularly in science and engineering, has resulted in the widespread adoption of parallel and distributed file systems for data storage and access. However, as file system sizes and the amount of data "owned" by users has grown, it is increasingly difficult to discover and locate data amongst the terabytes or petabytes of accessible data. While it is now routine to search for data on a personal computer or discover data online at the click of a button, there is no such equivalent method for discovering data on large parallel and distributed file systems in high-performance computing systems. Popular search solutions, such as Apache Lucene, were designed and implemented to run on commodity hardware thus posing significant limitations in achieving good efficiency on large-scale storage systems with many-core architectures, multiple NUMA nodes, and multiple NVMe storage devices. In this work we revisit and propose methods and techniques to support efficient indexing of data in order to enable search. We propose SCANNS, an indexing framework that can exploit the properties of modern high-performance computing systems delivering an order of magnitude better performance. SCANNS supports out-of-the-box Term Frequency-Inverse Document Frequency information retrieval model. We evaluate SCANNS on the Mystic system with configurations up to 192-cores, 768GiB of RAM, 8 NUMA nodes, and up to 16 NVMe drives, and achieved performance improvements up to 19x better indexing while delivering up to 280X lower search latency when compared to Apache Lucene.**

***Index Terms*—search engine architecture, high-performance indexing, high-performance storage, scientific data**

I. 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 filesystems for storing and accessing data efficiently. However, as filesystem 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 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 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.

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. Thus, there is a need to revisit indexing and search methods and the building blocks of search engines as new hardware emerges.

In order to address the general problem of efficient data exploration and search in large file systems, we present the SCANNS indexing framework. 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 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 paper 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 up to 192-cores, 768GiB of RAM, 8 NUMA nodes, and up to 16 NVMe drives delivering 19x higher indexing throughput and 280X lower search latency;

The rest of the paper is organized as follows. Section II presents related work. In Section III we present the general architecture of the SCANNS framework and explain various optimizations and techniques that we used in its design. In Section IV we present an experimental evaluation of the SCANNS framework and conclude in Section V.

II. 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 [7]–[9], 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, such as GlusterFS and CephFS. 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. Other existing works from the HPC domain (e.g. GUFI [10], [11]) has also aimed to tackle the indexing and search problem focusing on metadata as opposed to the scientific data itself. We believe both the metadata and data are critical components to better accessibility of scientific data.

ScienceSearch [12] 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 [13] 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 [14] 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. 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 data structures.

III. 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.

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" [15]. Figure 1 shows a structural decomposition of the four main components of a search engine.

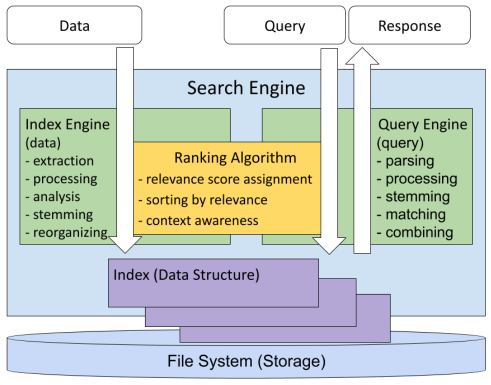


Fig. 1: General architecture of a search engine.

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.

A. 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 storage 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.

B. 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 2. 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.

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.

C. Indexing Engine Execution

In terms of execution, SCANNS uses multiple threads to parallelize the execution of the indexing process by data and also by function. The framework uses two kinds of threads, as seen in Figure 2: *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 data in order to build the local index, by running local pairs of the Tokenizer and Indexer components. Index threads follow the observer design pattern, where the Tokenzier is the subject and the Indexer is the observer is the Indexer, and use the internals of the Indexer component to store the local indexes in memory. The number of read and index threads are 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.

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 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.

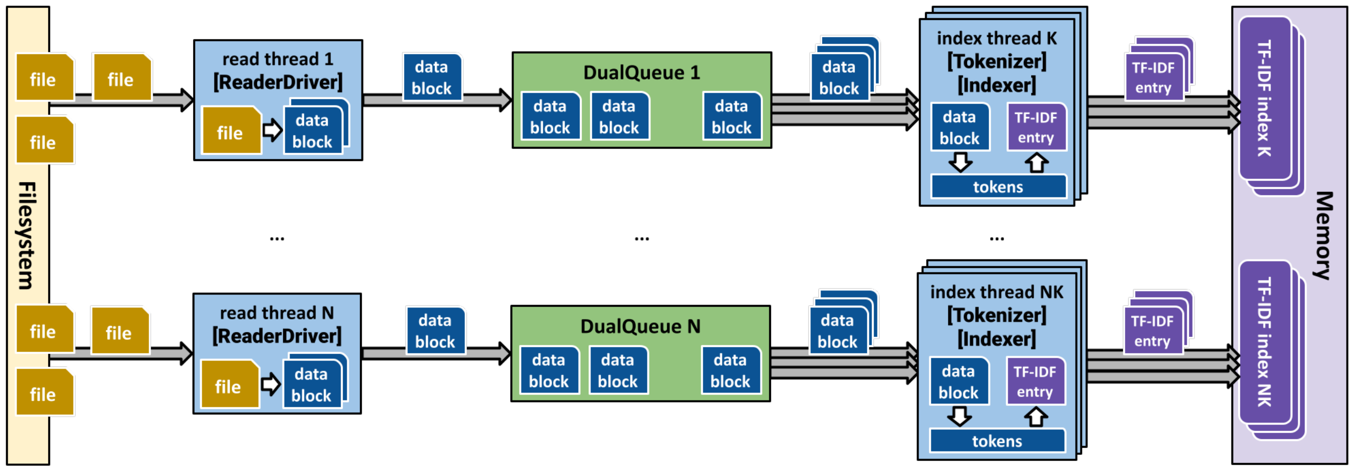


Fig. 2: SCANNS framework indexing architecture and pipeline.

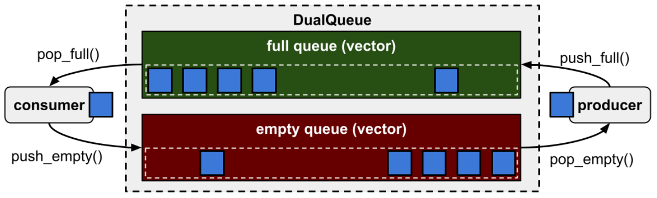


Fig. 3: SCANNS DualQueue design.

D. 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.

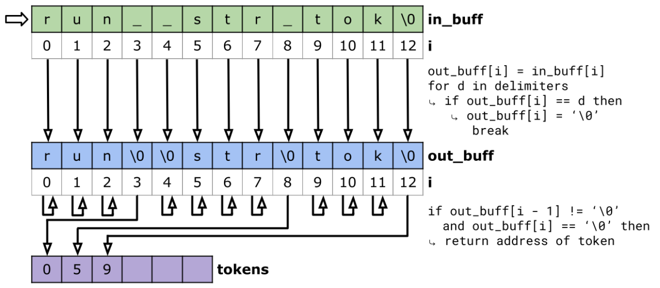
In a setup where a machine has many PCIe NVMe devices we observed that sometimes the memory subsystem of the OS that manages the file system caches and buffers can become a performance bottleneck. Since the data that is read from the input files by the indexing engine is being re-organized, it is not 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 the halves 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.

E. 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.

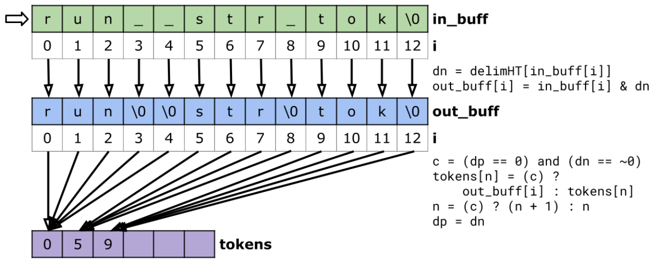
While this component can be implemented in a standard way in C through the use of the *strtok()* function, we observed that the performance of the standard approach is very low when compared to how fast the ReaderDriver can read data from disk. In order to improve the Tokenizer's performance, we proposed a re-implementation of the split function, where we replaced the call to *strtok()* with an approach that uses branchless programming. Figures 4 and 5, show the conceptual difference between the standard and the optimized Tokenizers.



```
out_buff[i] = in_buff[i]
for d in delimiters
↳ if out_buff[i] == d then
   ↳ out_buff[i] = '\0'
      break
```

```
if out_buff[i - 1] != '\0'
  and out_buff[i] == '\0' then
↳ return address of token
```

Fig. 4: SCANNS Standard Tokenizer.



```
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
```

Fig. 5: SCANNS Optimized Branchless 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 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.

F. 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 an TFIDF inverted index, that is stored in main memory. At the core of the Indexer stands the design of the inverted index, which can be seen in Figure 6.

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 [16], 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.

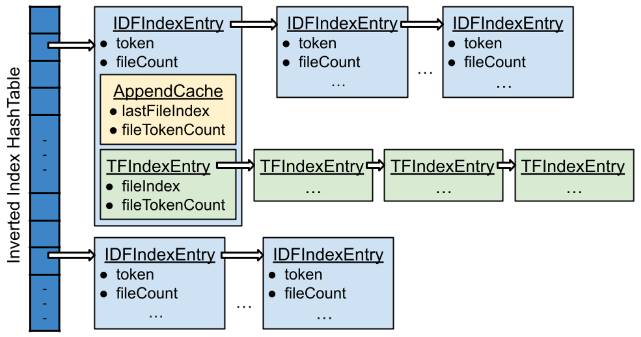


Fig. 6: SCANNS Inverted Index Design.

The design in Figure 6 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 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 in the list of TFIndexEntries, performing update or append 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 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 calling a system call for each object.

To deal with the second issue, we introduce an *Append-Cache* to the IDFIndexEntry that removes the 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.

G. *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.

IV. 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.

A. *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 I 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*).

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.

B. *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 any kind of optimizations;

machine name	processors	cores	main memory	nvme storage
(a) 64cores-1disk	2 x AMD EPYC 7501	64	128GiB DDR4 2666 MHz	1 x Intel Optane 900P SSD
(b) 32cores-16disks	2 x Intel Xeon Gold 6130	32	192GiB DDR4 2666 MHz	16 x Samsung 970 EVO SSD
(c) 192cores-16disks	8 x Intel Xeon Platinum 8160	192	768GiB DDR4 2666 MHz	16 x Intel Optane 900P SSD

TABLE I: Mystic Cloud machines used for the experimental evaluation and their specifications.

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

C. *ReaderDriver*

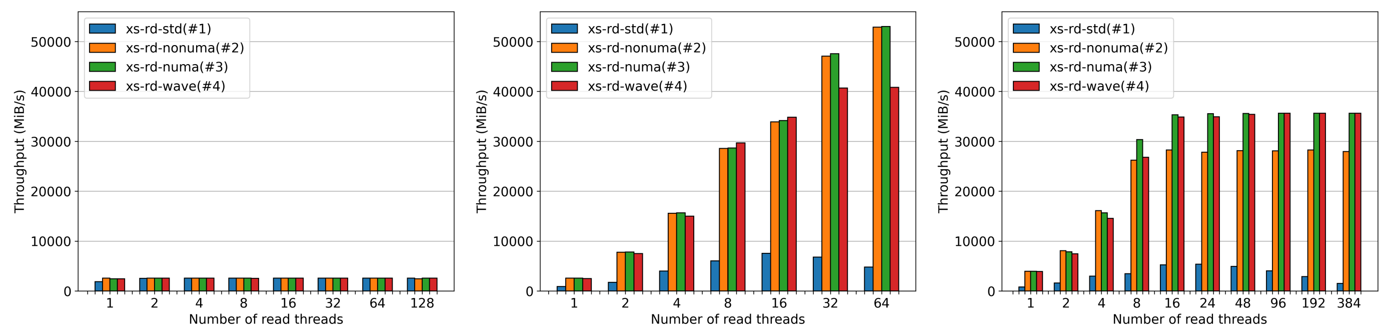
Figure 7a shows the performance the ReaderDriver variants, measured in MiB/sec with increasing number of read threads, when running on a system that has only one NVMe device installed. We can see that all variants are able to saturate the single NVMe device (2.5 GiB/sec) with sufficient threads.

In Figure 7b we see a different picture. The baseline ReaderDriver seems to be 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 7c 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.

D. *Tokenizer*

Figure 8a 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



(a) 64cores-1disk (b) 32cores-16disks (c) 192cores-16disks

Fig. 7: ReaderDriver throughput with increasing number of read threads.

configurations. In this setup the NUMA-aware configurations have no affect as there is only one NVMe device.

Figure 8b 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 version from 16.9 GiB/sec to 18.8 GiB/sec.

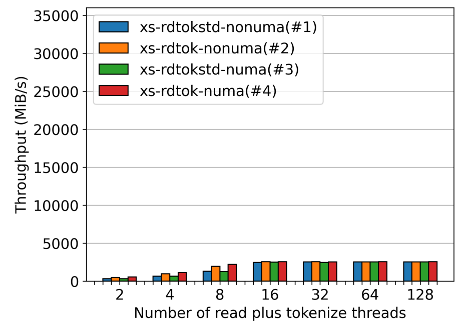
Figure 8c shows performance with many cores and multiple NVMe devices and here we can clearly see the difference between all variants and thus between all optimization options. Between the versions that do not use any kind of NUMA-aware optimizations, we can see that the optimized Tokenizer achieves 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 configurations, 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.

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

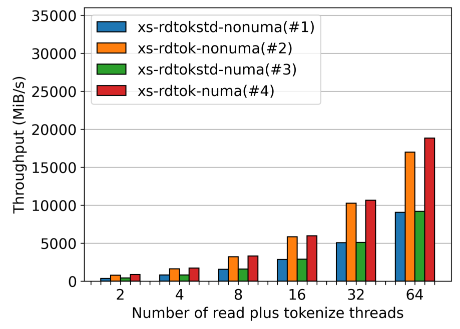
Figure 9a shows the performance, measured in MiB/sec of End-to-End indexing with increasing number of read and index threads, for all variants. Each index thread is paired with a read thread, with the exception of the Lucene variant that two merge threads with each index thread instead. We can see that, 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.

When looking at a system that has multiple NVMe devices but not that many cores, as depicted in Figure 9b, 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 optimizations 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 dues to the fact that there are more memory channels per NUMA node that on 64cores-1disk.

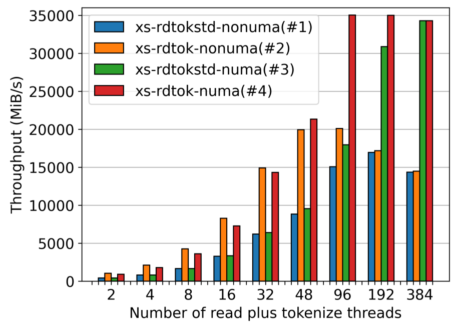
On the system that has many cores and multiple NVMe devices and the most memory channels per NUMA node, we can see that the SCANNS framework can reach very high throughput, when the proper optimizations are used. Figure 9c captures this performance, and shows that the un-optimized variants reach a similar 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



(a) 64cores-1disk

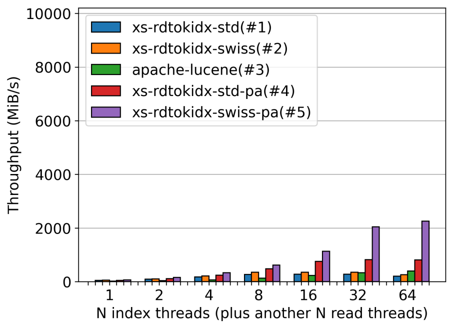


(b) 32cores-16disks

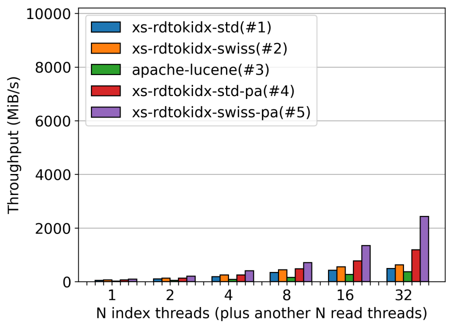


(c) 192cores-16disks

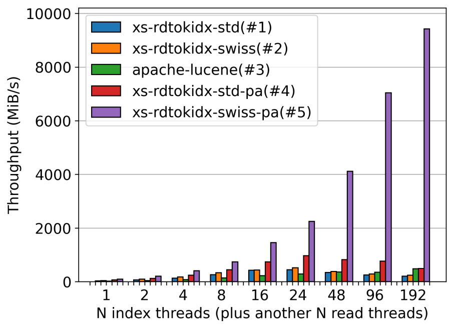
Fig. 8: ReaderDriver and Tokenizer throughput with increasing number of read and tokenize threads.



(a) 64cores-1disk



(b) 32cores-16disks



(c) 192cores-16disks

Fig. 9: End-to-end TF-IDF indexing throughput with increasing number of read and index threads.

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 II 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.

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

TABLE II: TFIDF End-to-end search latency (microseconds).

F. 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 10).

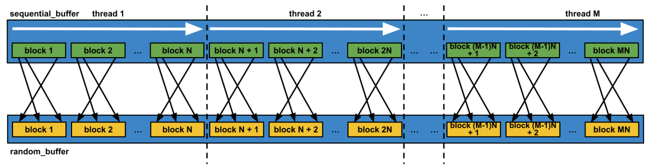


Fig. 10: Random Access Memory Benchmark Design.

The results that we got for increasing block sizes and increasing number of threads, run on the 192cores-16disks machine, are depicted in Figure 11. It is interesting how much performance degrades when the block size exceeds a certain value, 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.

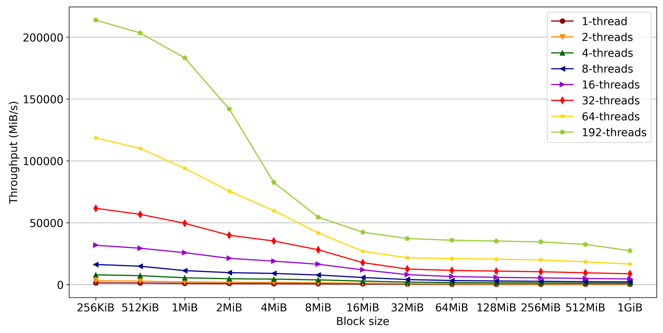


Fig. 11: Random Access Memory Benchmark.

G. Results Summary and Discussion

This evaluation showed that a naive approach to reading data from a modern filesystem, deployed on multi PCIe NVMe SSD storage devices, can 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 reduced the cost of tokenization of 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/jumps. Variable sized tokens can cause a significant number of branch mispredictions. The removal of branches from tokenization eliminates the cost of branch mispredictions and allows a better use of the CPU pipelines, leading to improved performance compared to standard C strtok() function.

We showed that the main bottleneck for the inverted index solution 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 stresses 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 reducing 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.

V. Conclusion

In this paper we 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 modularized, tunable and scalable by design. The indexing pipeline is exposed and allows easy modification and tuning, enabling SCANNS to saturate storage, memory and compute resources, and exploit the properties of modern high-end systems. Our evaluation showed that SCANNS can deliver, on machines with up to 192-cores, 768GiB of RAM, 8 NUMA nodes and up to 16 NVMe drives, up to 19x higher indexing throughput and 280x lower search latency, when compared to Apache Lucene.

In future work we will implement semi-automatic hyperparameter tuning as part of the SCANNS framework to allow easier selection of key parameters that affect performance on particular hardware. We will explore persistent indexes through the use of Intel Optane DC memory. We will explore methods for distributing indexing and search to scale to some of the largest HPC storage systems available. Specifically, we will investigate integration of the distributed SCANNS system into parallel and distributed storage systems [17]–[19] to enable automatic metadata and data indexing and search.

Acknowledgment

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

References

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

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

[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. [Online]. Available: 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).* Ieee, 2010, pp. 1–10.

[7] 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*, 2017, pp. 1–12.

[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).* IEEE, 2020, pp. 162–171.

[9] 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.

[10] D. J. Bonnie, "Gufi overview," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2018.

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

[12] 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).* IEEE, 2018, pp. 93–104.

[13] 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*, 2019, pp. 1–24.

[14] 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.

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

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

[17] D. Zhao, N. Liu, D. Kimpe, R. Ross, X.-H. Sun, and I. Raicu, "Towards exploring data-intensive scientific applications at extreme scales through systems and simulations," *IEEE Transaction on Parallel and Distributed Systems (TPDS) 2015*, 2015.

[18] 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*, vol. 28, no. 1, pp. 44–69, 2015.

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