Received: 23 December 2020 | Revised: 18 February 2022 | Accepted: 22 February 2022

DOI: 10.1002/cpe.7261

RESEARCH ARTICLE

Evaluation of a scientific data search infrastructure

Alexandru Iulian Orhean[1] | Anna Giannakou[2] | Katie Antypas[3] | Ioan Raicu[1] | Lavanya Ramakrishnan[2]

[1]College of Computing, Illinois Institute of Technology, Chicago, Illinois, USA

[2]Scientific Data Division, Lawrence Berkeley National Lab, Berkeley, California, USA

[3]NERSC, Lawrence Berkeley National Lab, Berkeley, California, USA

Correspondence
Alexandru Iulian Orhean, College of Computation, Illinois Institute of Technology, Chicago, IL, USA.
Email: aorhean@hawk.iit.edu

Funding information
DOE ASCR, Grant/Award Number: DE-AC02-05CH11231

Summary

The ability to search over large scientific datasets has become crucial to next-generation scientific discoveries as data generated from scientific facilities grow dramatically. In previous work, we developed and deployed ScienceSearch, a search infrastructure for scientific data which uses machine learning to automate metadata creation. Our current deployment is deployed atop a container based platform at a HPC center. In this article, we present an evaluation and discuss our experiences with the ScienceSearch infrastructure. Specifically, we present a performance evaluation of ScienceSearch's infrastructure focusing on scalability trends. The obtained results show that ScienceSearch is able to serve up to 130 queries/min with latency under 3 s. We discuss our infrastructure setup and evaluation results to provide our experiences and a perspective on opportunities and challenges of our search infrastructure.

KEYWORDS

scientific data, search engine

1 | INTRODUCTION

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

In previous work, we have developed and deployed ScienceSearch,[1] a scalable search engine that uses a wide range of machine learning techniques (natural language processing, deep learning, etc.) to automate metadata generation from different data sources, such as published papers, proposals, images and file system structure. The current implementation is deployed to provide search over data obtained from NCEM (National Center for Electron Microscopy) that includes around 5 TB of data (500k images). Users can interact with the ScienceSearch infrastructure through a dedicated web interface that accepts a text query and returns a list of relevant images, papers and proposals within seconds. Our system is deployed on a container service platform, called Spin, at the National Energy Research Computing (NERSC) Center, a HPC facility. Such container service platforms have more recently been deployed at HPC facilities to support science gateways, workflow managers, databases, and other supporting services. Deploying the ScienceSearch infrastructure on Spin allows us to leverage the high performance large filesystem at NERSC while allowing users to query the data through a web interface.

Our previous work showed that ScienceSearch is capable of generating relevant metadata and providing low-latency high quality query results for our initial use case from NCEM. The ScienceSearch infrastructure allows us to understand and address key questions related to the scalability of our infrastructure for increasing data sizes and number of users. The performance of dedicated search infrastructures greatly depends on the ability to simultaneously serve multiple types of queries while keeping search latency as low as possible. Previous work found in literature has explored scalable search in the context of Internet data.[2,3] However, these results cannot be directly applied to scientific data in HPC due to the unique characteristics of the scientific data (i.e., size, formats, volume) and the performance considerations of HPC environments. The focus of this article is an evaluation of ScienceSearch toward understanding design implications for current and future scientific search infrastructures. We perform a thorough evaluation of the current infrastructure and discuss our experiences and results.

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

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

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

The article is organized as follows. In Section 2, we present related work to put this article in context with existing work in the literature. In Section 3, we present the design of the search process of ScienceSearch. The evaluation setup and the selected performance metrics are detailed in Section 4. Experimental results are presented in Section 5, and in Section 6, we outline important observations from our experiences and experimental evaluation. Finally, in Section 7, we summarize our conclusions.

2 | RELATED WORK

In this section, we outline work related to building domain specific scientific search engines as well as evaluating the scalability of search infrastructures. In previous work,[1] we presented a detailed overview of ScienceSearch's architecture focusing on the machine learning models that were used to generate search-related metadata. A preliminary evaluation in terms of query latency and the effects of caching was conducted. In this article, we present an extensive infrastructure evaluation focusing on scalability trends as well as outlining important observations regarding search infrastructures on HPC.

Numerous research efforts have tried to address the issue of scientific search capabilities. The *Materials Project*[4] enables search over an individual material's characteristics, while *KBase*[5] provides search capabilities over systems biology data. *LODAtlas*[6] enables users to discover datasets of interest by facilitating content and metadata based search. Finally, *Thalia*[7] is a search infrastructure that enables semantic search in biomedical literature based on named entity recognition. However, none of the proposed solutions provide scalable search environments that can host vastly growing amounts of data and provide low latency search results and can't be used to compare our infrastructure.

Data Search[8] provides a scalable solution for metadata management but the system does not automatically infer or create metadata from the ingested datasets. The ability to reliably handle parallel job execution led to the adoption of MapReduce/Hadoop[2,3] for large parallel searches by different projects.[9,10]

Recent research efforts have tried to enable scalable search capabilities for scientific portals. *Varsome*[11] created a a search engine for human genomics variants that provides search over 500 million variant records. However, the deployed solution does not scale since *Varsome* stores every record in a massive database. In Reference 12, the authors present a scalable search engine for geospatial data that utilizes indexing shards in order to provide low-latency search results. *CellAtlasSearch*[13] enable search over thousands of cell profiles. Their approach relies on specific hardware (GPUs) to achieve low latency results for queries. The authors of Reference 14 built *Visibiome*, a scalable search architecture for microbiomes. Their solution is a distributed web application that only scales horizontally when dealing with increased number of parallel queries.

Numerous works evaluate the scalability of search infrastructures mainly focusing on the indexing and query processing parts. ElasticSearch[15] provides a set of dedicated metrics that measure query processing time, latency, and throughput while indexing performance is measured using indexing and flush latency. Solr[16] features built in timers for query latency and indexing performance. *Apache Lucene*[17] has been used as a core building block for scientific search engines and information retrieval tools. The authors in Reference 18 evaluate the scalability of *Anserini*, a Lucene-based information retrieval tool, by creating custom benchmarks for indexing and ranking. The presented solutions are limited in providing only horizontal scaling and do not comprehensively evaluate all scalability aspects. In our work, we provide a detailed scalability study by examining closely each step of ScienceSearch's search steps. We collect latency measurements for different types of queries while identifying scalability challenges in the infrastructure.

3 | SCIENCESEARCH

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

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

An overview of ScienceSearch's container-based architecture can be found in Figure 1. ScienceSearch groups all data-related components (i.e., the *data import*, *metadata extraction*, and *search engine*) in a single container instance while the *user interface* and *database* are deployed as separate instances. The *backend* and *database* instances can be deployed separately and independently of the *user interface* guaranteeing portability and execution at different locations adjacent to HPC environments (e.g., near the scientific instruments where the data is generated). Furthermore, all data-related components can be executed separately and can be triggered by events (e.g., creation of new data).

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

3.1 | Search lifecycle

Search is conducted in three steps: *user query processing*, *comparison with stored metadata*, and finally *result aggregation*. The three steps are depicted in Figure 2.

Once a user issues a search query, the user query processing step divides the query into words (i.e., for queries that contain multiple terms) and the extracted words are lemmatized (i.e., reduced to root stems). The lemmatized terms are then passed to the next stage where the comparison with the metadata occurs. During comparison with stored metadata, the query terms are compared against metadata that are stored in a database system and the results are ranked. The database lookup, comparison, and the intermediate result ranking is computationally intensive. In order to optimize performance and reduce latency inducing bottlenecks, we opt for a parallel environment with multiple workers that are managed by a master process. The master and workers actions are explained in detail in Section 3.3.

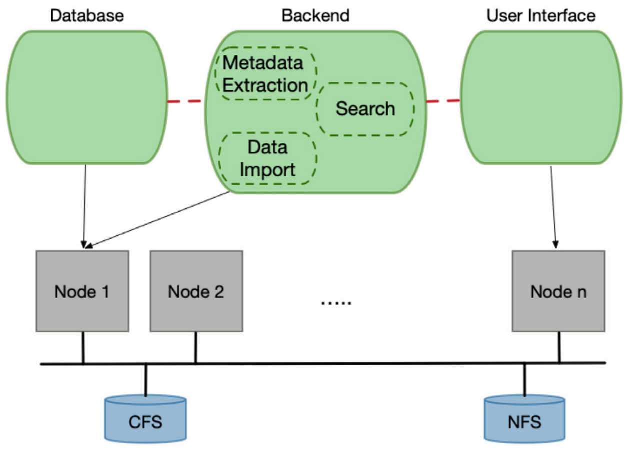

FIGURE 1 ScienceSearch container-based architecture and interaction with HPC resources through *Spin*. Container instances are denoted with light green and inter-container communication is represented with a dotted red line. HPC resources are in gray (physical nodes) and blue (remote and local storage). Arrows show where each instance is physically deployed.

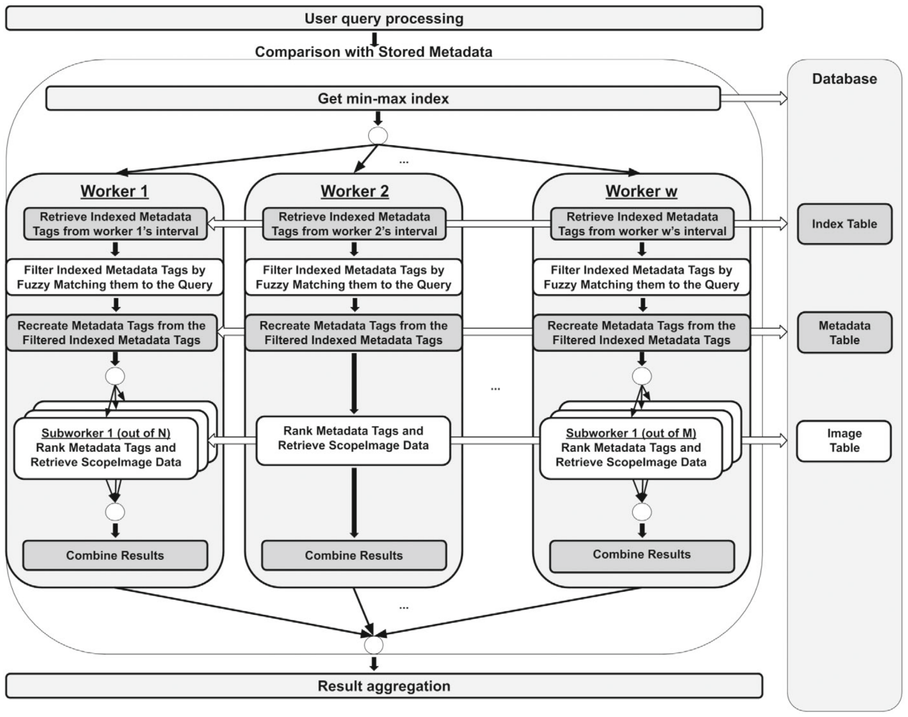
User query processing
Comparison with Stored Metadata
Get min-max index
Database
Worker 1
Worker 2
Worker w
Retrieve Indexed Metadata Tags from worker 1's interval
Retrieve Indexed Metadata Tags from worker 2's interval
Retrieve Indexed Metadata Tags from worker w's interval
Index Table
Filter Indexed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

FIGURE 2 Parallel architecture for *comparison with stored metadata* step. A master process slices the database index in *W* slices and spawns *W* workers, respectively. Each worker interacts with the database in order to fetch and rank intermediate results.

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

3.2 | Databases

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

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

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

3.3 | Comparison with stored metadata

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

Retrieve indexed metadata. Each worker queries the *index* table. Each worker retrieves unique from the slice assigned by the master process (see Figure 2). The retrieval is executed as a comparison between the *text_tag* field of each metadata entity and the lemmatized list's elements producing a *hit score* for each entity. The *hit score* is calculated based on the lexicographic distance of the two entities. Once the score is calculated, the worker keeps only the metadata entities that scored higher (if any) than an empirically set threshold, and discards the rest. The retained results are considered the *hits*.

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

Rank metadata objects. Each worker retrieve results by querying the *image* table. An image might have more than one matching tuple of metadata result matches. Each image's search score is calculated from the sum of the tuple's final scores. The ultimate search result is the list of images in descending order based on their score.

3.4 | Parallelism

ScienceSearch uses adaptive two-level parallelism to deal with open-ended search queries that return thousands or even million of results. At the first level, ScienceSearch uses parallel workers to handle hits in the *index* table. Based on our experiments the optimal number of first-level workers is 16. However, in the current version of ScienceSearch data is not shuffled, sometimes leading to an uneven distribution of *hits* between workers. In order to address this issue and enable elasticity for open-ended queries, a second-level of parallelism is enabled in the worker level (sub-workers 1 to *N* in Figure 2). If the number of metadata objects in the recreate metadata tags step exceeds a certain threshold, a new pool of sub-workers is created and the objects are distributed among them for ranking (sub-worker box in Figure 2). After each sub-worker has completed the ranking step the results are sent back to the worker that initiated the extra parallelism step. Currently, the threshold that triggers additional parallelism is empirically set to 150,000 metadata objects before ranking.

3.5 | Understanding the memory footprint

In order to identify potential latency inducing bottlenecks, we take a closer look at the memory consumption of each search worker due to intermediate object creation during the three search stages. We use the term *memory block* to capture the amount of memory consumed by the objects. At first, a block of memory is acquired by each worker during the retrieval of unique tags that correspond to each worker's index slice. The number of objects is equal to the number of tags that are stored in that particular database slice. However, during the recreate metadata tags phase, each worker is tasked with fetching all metadata objects that include a particular tag (one tag can have thousands of associated objects) consuming a significantly bigger memory block for the recreated metadata objects. Finally, after ranking each metadata object, a block is acquired for retrieving all images that correspond to the highest ranking metadata objects, in the retrieve image data phase. An overview of the memory blocks with the corresponding worker actions is shown in Figure 3.

It is evident that after recreate metadata tags phase, the number of objects in memory significantly increases, making individual object size a determining factor in ScienceSearch's memory footprint as well as its ability to process multiple objects in parallel. Table 7 shows the actual object numbers for a search worker after each search step. We present our analysis of memory footprint in Section 5.2.

4 | EVALUATION SETUP

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

4.1 | Infrastructure

The experimental setup consists of the deployment of the ScienceSearch components as dedicated containers in order to ensure portability and isolation. Four container instances are deployed: a Django backend, a PostgreSQL database, an Nginx web proxy which also acts as a load balancer and

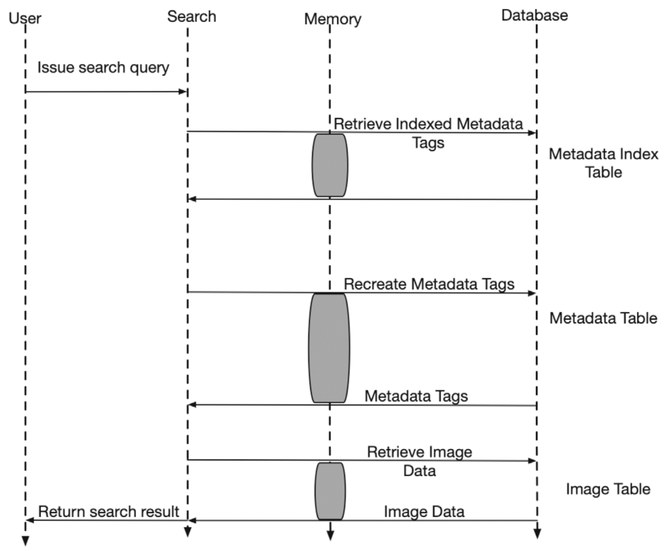

FIGURE 3 Memory consumption of each search worker for object creation during search stages. The number of objects significantly increases after *recreate metadata tags*.

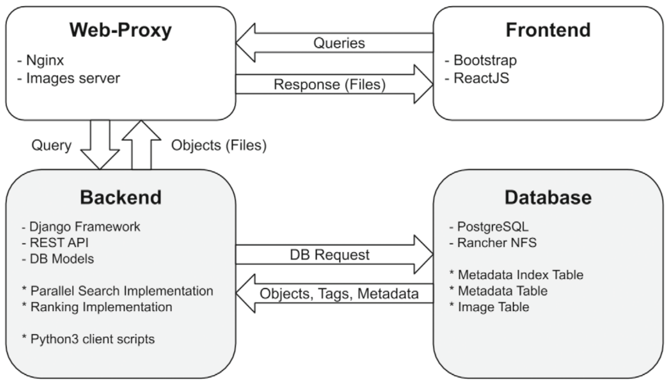

FIGURE 4 ScienceSearch deployment. Gray boxes are containers located on the same physical node to avoid network bottlenecks. White arrows represent requests between internal components.

the corresponding frontend that serves a web frontend. Our setup, together with the component's interaction during query execution, is depicted in Figure 4. The user is connected to ScienceSearch through a Web-proxy container which serves as a load-balancer. Once a user issues a search query, the query is forwarded to the backend container where the three search steps are conducted. The backend container, which is the only one with access to the database, fetches search results after issuing one or more database requests. Finally, the results are served back to the user through ScienceSearch's frontend.

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

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

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

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

4.2 | Software

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

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

4.3 | NCEM dataset

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

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

The Django framework transforms the database records to Python 3 objects. Table 1 shows the total number of objects and size (in the database) of the tables for each data type.

4.4 | Evaluation metrics

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

TABLE 1 Total number of objects/records found and amount of storage used by each table, for the evaluation search space

Database table	Number of records	Total size
Metadata index	1,184,851	115 MB
Metadata	1,1261,844	791 MB
Image	557,195	11 GB

TABLE 2 Example terms of targeted and open-ended queries along with the number of final results returned to the user

Query type	Term	Number of returned results
Targeted	Lattice	37
Targeted	50images.dm3	159
General	Vortices	3008
General	Frame	1001

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

In each experiment, we also measure the latency of each phase of the search algorithm (as presented in Section 3.3). For a detailed breakdown analysis, we combine the time to prepare the request, the time to receive the response and the time to aggregate the final results (as depicted in Figure 2) into *other* latency, which accumulated together with the execution time of each search phase constitutes the overall search time.

Memory footprint. When evaluating the memory footprint of each search phase, we calculate the number and the size of all intermediate objects and data structures as well as the size of the final result objects. In the case of an image, the final result object consists of the image itself along with associated metadata (e.g., image location in the filesystem, instrument name on which the image was acquired, date, etc.; Section 3.2).

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

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

4.5 | Queries

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

Query Set 1: Our first query set is meant to capture queries that are commonly executed in practice over the NCEM dataset. They typically return on average under 200 results. We refer to these queries as *targeted* queries, they have a limited set of matching metadata tags, hence limited number of associated objects are generated during the recreate metadata tags search step (Figure 2). Targeted queries represent searches from users that are familiar with the underlying datasets and look for specific terms. The majority of queries executed on ScienceSearch are targeted queries.

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

For targeted queries, we randomly select 76 tags that have less than 200 matches on average in the entire search space of our dataset. For general queries we select 6 tags that have at least 1000 matches. Example of both query categories and the number of results returned are listed in Table 2.

5 | EXPERIMENTAL RESULTS

This section covers the results of the experimental evaluation performed on ScienceSearch. We exclude measurements from search queries that return zero results since they exhibit very low latency. The query sets and obtained results depend on the size and number of entries in the dataset and therefore can change if the evaluation process is conducted on a different dataset.

The experimental evaluation covers the performance of the proposed search platform from the perspective of: (a) the query processing rate metric (Section 5.1), (b) memory footprint (Section 5.2), (c) query latency metric (Section 5.3), (d) the breakdown analysis of the search phases (Section 5.4), (e) the query throughput metric (Section 5.5), and (f) a comparative evaluation of ScienceSearch on the two testbeds (Section 5.6).

5.1 | Query processing rate

Query processing rate, (i.e., how many objects per second or how much data gets processed per second with increasing number of parallel workers) is very useful when trying to understand the performance of a system such as ScienceSearch. Figure 5A depicts the overall query processing rate of ScienceSearch for both query sets. In the context of Query Set 1 (i.e., targeted queries), ScienceSearch shows a good scalability trend and exhibits a processing rate of approximately 87 and 518 kObjects/s, when increasing the number of parallel worker processes from 1 to 32, resulting in an almost 6x speedup. As for Query Set 2 (i.e., open-ended queries) the processing rate increases slightly from roughly 50 to 99 kObjects/s, when varying the number of workers from 1 to 32, achieving a speedup of 2x. It can be noted that query processing rate for Query Set 2 has a slower acceleration than Query Set 1.

Figure 5B,C shows that the metadata index retrieval and metadata index filter phases exhibit higher data processing rates than the recreate metadata tags and the image objects ranking phases. The ranking phase seems to cap at 1622 Objects/s for Query Set 1 and at 210 Objects/s for Query Set 2. The difference in the processing rate can be attributed to differing loads (more analyses in Section 5.2).

5.2 | Memory footprint

The memory footprint evaluation offers valuable insight on the nature of two query types, by looking at the total number of objects processed and generated by each search phase.

Table 3 shows the memory footprint at each search phase for Query Set 1. The index retrieval phase looks up approximately 1.2 million objects (number of unique tags in the index table) from the database, while only 11 objects on average are filtered after the index filter phase. The objects expand to 214 objects (~20× increase) on average at the metadata recreation stage. Finally, around 190 objects on average are returned in the end, after performing ranking on them during the ranking phase. In the case of Query Set 2 (Table 4), an average of 6 objects are transferred after index filtering. In the next stage (recreate metadata), 497k objects on average match of which 1586 of get returned as results, which is a significantly larger than what we see in Query Set 1.

Tables 5 and 6 contain the values at each search phase for targeted and general queries, respectively. The amount of data measured is proportional with the number of objects recorded at each phase, showing the size of the data with and without Python 3 overhead. The Python 3 overhead is a direct result of the data structures selected by Django for storing the database records requested during each search phase.

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

In addition, we perform an object-level optimization in order to reduce the amount of data moved from the database to the search workers during the final image retrieval stage. Specifically, we move each object's file metadata field (currently stored in JSON format) from the database to

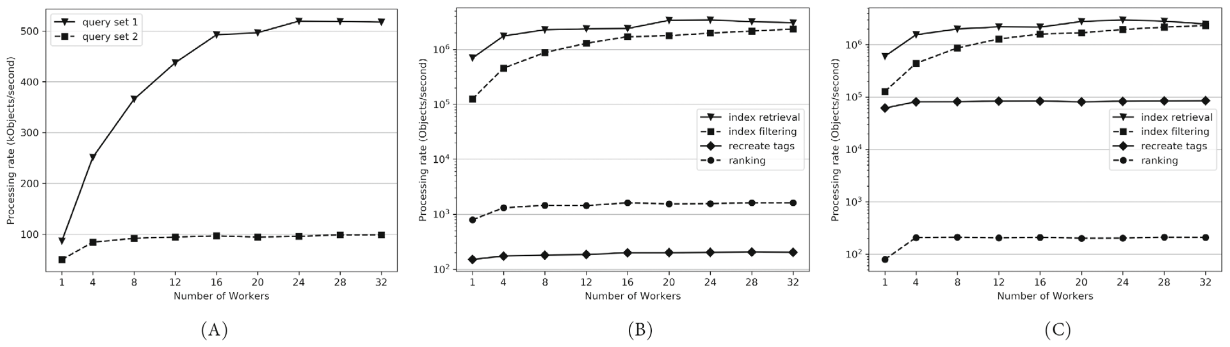

FIGURE 5 Average processing rate, measured in objects per second, with increasing number of parallel worker processes. (A) Comparison between Query Set 1 and Query Set 2, combining the average processing rate of all search phase. (B) Average processing rate of each search phase for Query Set 1. (C) Average processing rate of each search phase for Query Set 2

TABLE 3 Memory footprint is measured as the minimum, mean, and maximum number of objects processed when executing queries from Query Set 1, grouped by search phase.

Number of objects	Min	Average	Max
Metadata indexes	1,184,850	1,184,850	1,184,850
Filtered indexes	1	11	64
Metadata tags	19	214	1589
Image objects	19	190	1414

TABLE 4 Memory footprint is measured as the minimum, mean, and maximum number of objects processed when executing queries from Query Set 2, grouped by search phase.

Number of objects	Min	Average	Max
Metadata indexes	1,184,850	1,184,850	1,184,850
Filtered indexes	1	6	25
Metadata tags	265,837	497,031	651,684
Image objects	505	1586	3008

TABLE 5 Memory footprint is measured as the minimum, mean, and maximum size of processed objects in bytes when executing queries from Query Set 1, grouped by search phase.

Total size (bytes)	Min	Average	Max
Metadata indexes (with Python)	661 MB	802 MB	813 MB
Filtered indexes (with Python)	2.8 kB	8.6 kB	36 kB
Metadata tags (with Python)	15.6 kB	168.5 kB	1.2 MB
Image objects (with Python)	204 kB	6.5 MB	48.7 MB
Metadata indexes	74 MB	74 MB	74 MB
Filtered indexes	26 B	356 B	2 kB
Metadata tags	608 B	7.6 kB	65.5 kB
Image objects	38.8 kB	2.3 MB	17.3 MB

Note: The table includes the size with and without Python 3 overhead.

disk. Reducing each object's size subsequently reduces the time spent to recreate final search results (retrieve image data step in interaction with the database; Section 3.2) and improves overall search latency. The optimization effect on search latency for both query sets for 16 parallel workers is shown in Figure 6. For general queries the object-level optimization reduces search latency by 2.2 s while for targeted queries the reduction is 0.2 s. The search latency reduction is between 7 and 12%.

5.3 | Query latency

Figure 7 contains the measured latency of ScienceSearch and the average maximum latency of each search phase for Query Set 1. Figure 7A shows that on average for targeted queries it takes ScienceSearch 13.72 s (1 worker) and 2.44 s (24 workers) to execute a query. For Query Set 2, Figure 7C shows a similar trend, but with significantly higher latency when compared to the Query Set 1. Without query processing parallelism, running a general query takes 34.19 s and drops to 18 s with 32 parallel workers.

Figure 7B shows that metadata index filtering is the most demanding search phase and that query processing parallelism increases the performance of each phase by reducing the latency significantly for metadata index retrieval and filtering. ScienceSearch scales reasonably well, optimizing the most costly phases of search and therefore improving the overall search latency. In the case of Query Set 2, the most demanding phase is not the index filtering anymore and further analysis showed that there is an emerging load balance issues caused by the way indexes are partitioned between the workers. This analysis is discussed in the following section.

TABLE 6 Memory footprint is measured as the minimum, mean, and maximum size of processed objects in bytes when executing queries from Query Set 2, grouped by search phase.

Total size	Min	Average	Max
Metadata indexes (with Python)	661 MB	762 MB	813 MB
Filtered indexes (with Python)	2.8 kB	5.8 kB	16 kB
Metadata tags (with Python)	209 MB	376 MB	524 MB
Image objects (with Python)	27.4 MB	85.2 MB	163.9 MB
Metadata indexes	74 MB	74 MB	74 MB
Filtered indexes	20 B	227 B	932 B
Metadata tags	9.8 MB	17.9 MB	31.9 MB
Image objects	10.4 MB	32.1 MB	62.1 MB

Note: The table includes the size with and without Python 3 overhead.

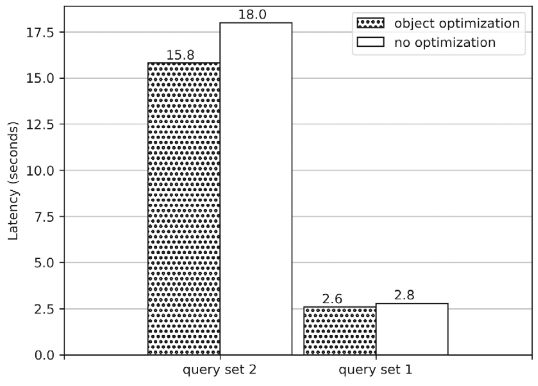

FIGURE 6 Reduced query latency for both targeted and open-ended queries after object size reduction

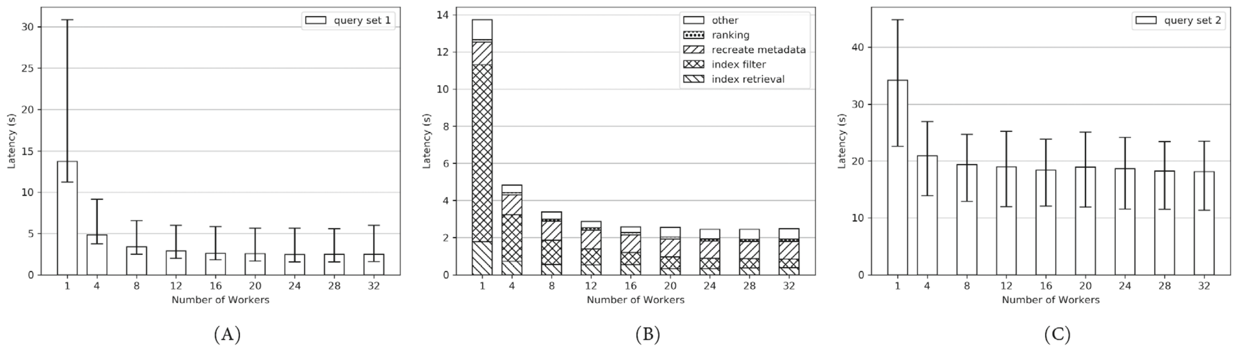

FIGURE 7 (A) Overall average query latency and min-max variation with increasing number of parallel workers (Query Set 1). (B) Average query latency grouped by search phase, with increasing number of parallel workers (Query Set 1). (C) Overall average query latency and min-max variation with increasing number of parallel workers (Query Set 2)

5.4 | Analysis of queries with large results

Figure 8 shows the latency of each search phase with 16 parallel worker processes. We can see that the metadata index retrieval phase takes up to 0.48 s and the metadata index filtering phase takes up to 0.56 s for all workers, and that each worker spends a similar amount of time in these two phases. When we look at the latency of the recreate metadata tags and the final image object ranking phases, we can see that workers do not spend the same amount of time. Some worker processes end up spending approximately 7.93 s, as is with worker 3, while other spend under 0.01 s. Worker 3 also takes 11.8 s to perform the ranking on the image objects, while other workers spend again under 0.01 s in the same phase.

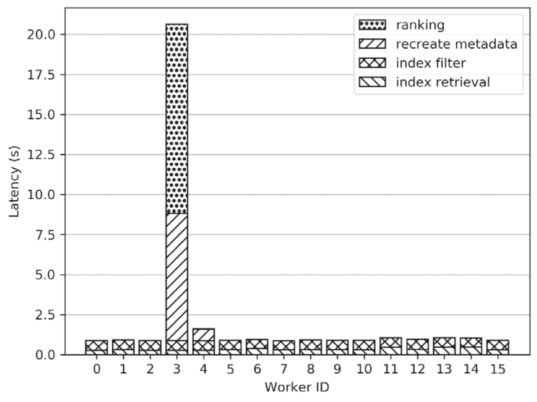

FIGURE 8 Average latency of each search phase for each worker for a query (Frame) from Query Set 2 with 16 parallel workers

TABLE 7 Number of objects processed, filtered, or generated at each search phase for each worker for an query (Frame) from Query Set 2 with 16 parallel workers

Worker ID	Metadata indexes	Filtered indexes	Metadata tags	Image objects
0–2	74,053	0	0	0
3	74,053	2	549,226	1000
4	74,053	1	1	0
5–15	74,053	0	0	0

The imbalance in the worker processing time is caused by data that is not distributed evenly across worker processes (Table 7). This causes the remaining phases in the query processing pipeline to suffer from a similar imbalance, and for some phases it can be much worse. For example, the recreate metadata phase does not convert the metadata index objects to metadata tag objects on a one-to-one basis, but follows a one-to-many structure, generating in the best case scenario 410x more objects while in the worst case scenario 14,000× objects (Tables 3 and 4).

5.5 | Query throughput

Figure 9A shows the average query throughput while running concurrent targeted queries. Query throughput increases significantly with increasing number of concurrent queries being issued during the same experiment. When only one query is issued at one moment of time ScienceSearch achieves a throughput of roughly 12 queries/min. When running 8 queries at the same time, each query using 4 parallel worker processes, there are a total of 32 concurrent processes that access the database and process the queries, but the resulting throughput is approximately 65 queries/min, which is even faster than using 32 parallel worker processes while executing only one query at a time, which achieves a throughput of roughly 25 queries/min. When we get to the point of executing 32 queries concurrently, that means 128 concurrent processes in total, the throughput increases to 88 queries/min, even though the system was over-provisioned. In the case of Query Set 2, as depicted in Figure 9B, ScienceSearch shows similar trends in terms of scalability, experiences reduced performance degradation when running concurrent queries, even when over-provisioned, but achieves a lower query throughput when compared to the Query Set 1 performance, which is in concordance with the difference in latency between the two datasets.

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

5.6 | Spin infrastructure versus dedicated testbed

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

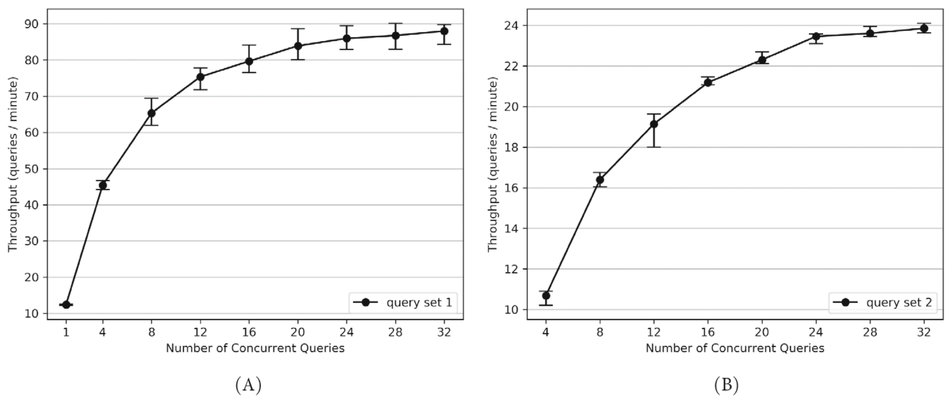

FIGURE 9 (A) Average query throughput and min-max variation for Query Set 1 with increasing number of concurrent queries. (B) Average query throughput and min-max variation for Query Set 2 with increasing number of concurrent queries

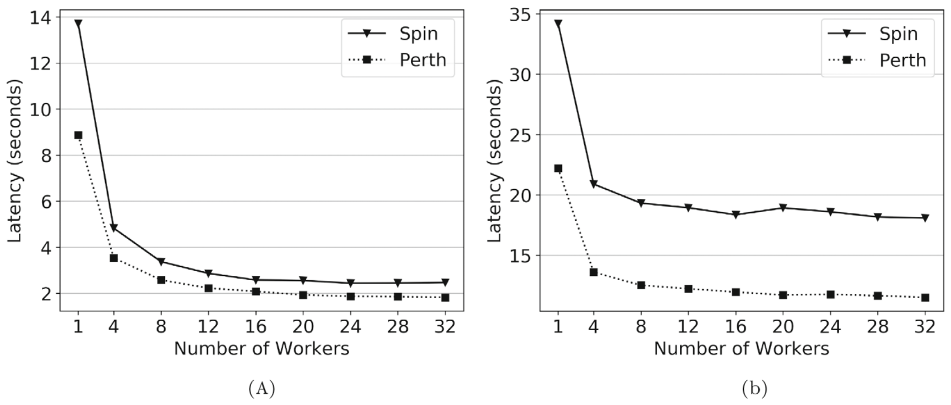

FIGURE 10 (A) Average query latency for Query Set 1 on *Spin* and Perth. (B) Average query latency for Query Set 2 on *Spin* and Perth

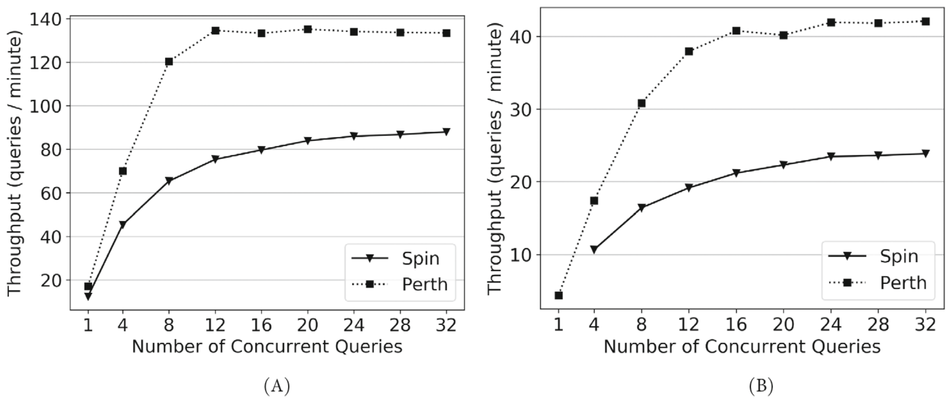

FIGURE 11 (A) Average query throughput for Query Set 1 on *Spin* and Perth. (B) Average query throughput for Query Set 2 on *Spin* and Perth

Figure 11A shows that for Query Set 1, the system can end up achieving a latency of under 2 s, when configured with at least 20 parallel worker processes. For Query Set 2 (Figure 10B), the latency can go as low as approximately 11.5 s when using 32 parallel worker processes. In terms of throughput, the lack of overhead from the storage system and the lack of interference allows ScienceSearch to reach up to 134 queries/min while running 32 concurrent targeted queries and roughly 42 queries/min while running 32 concurrent Query Set 2 queries. As seen in Figure 11, the dip in throughput while running 20 concurrent Query Set 2 queries is caused by the fact that we ran the experiment on 96 queries which does not divide exactly to 20.

It can be observed from the latency and throughput experiments, that the performance of ScienceSearch plateaus halfway through the number of available cores on all testbeds. From Figure 10, we can see that latency does not seem to decrease at the same speed after 12 cores when running on Spin and Perth. This can be attributed to the fact that in each experiment, the number of processes created is in practice double the amount of configured parallel workers processes. The above results in reaching the limit of available cores for both testbeds, while in fact only half of them belong to ScienceSearch's backend. The other half belong to the database, that spawns a process for each parallel work process. The same effect can be observed in Figure 11 that encompasses the throughput experiments. After 12 cores for Spin and Perth, throughput does not increase as fast as before the number of cores get over-provisioned with processes.

6 | DISCUSSION

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

6.1 | Spin

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

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

We observe from Figure 11A,B that between *Spin*, a shared multi-user supercomputer infrastructure, and the single node testbeds, ScienceSearch exhibits similar performance trends, albeit at slightly different latency. The latency discrepancy can be explained by the underlying hardware, (faster CPU, memory) but also by the inherent latency of the two different infrastructures. We know that ScienceSearch is latency-sensitive, thus the remote storage volume that *Spin* provides and which the search platform uses, will induce a certain, latency penalty when contrasted to local storage (remote storage has the added latency of both network and storage). On the other hand, ScienceSearch was designed to run on Spin and can exploit the inherent benefits that come with it: mobility, fault tolerance, and scalability. The database has its data stored over the network, and can easily be moved to another compute host, enabling ScienceSearch to achieve mobility.

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

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

6.2 | Search engine internal load balancing

Our evaluation results have demonstrated that the root cause of increased latency for open-ended queries, is the uneven distribution of results between workers before ranking (see Figure 8). In order to ensure low query latency for large scientific datasets, a scalable search infrastructure needs to achieve adequate load balancing of query matches between search workers. In the context of a master-worker search model, load balancing can be achieved by placing intermediate matches in a common queue and coordinating redistribution between idle workers. The common queue operations might introduce some overheads that will need to be considered.

Currently, in-node parallelism reduces query latency (see Figure 7) by a factor of 3. However, enabling search over a significantly larger dataset would require across-node parallelism. To address this issue a master-worker deployment can be utilized where multiple search and database instances are spawned by a master search process.

6.3 | Adaptive resource scaling

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

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

6.4 | Data representation and sizes

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

One of the key findings in our evaluation process is that the total size of each metadata instance (and subsequent memory footprint of the parallel search worker) is greatly increased (sometimes by 9×) due to the object representation in Python3 by Django (Tables 5 and 6). As we use these frameworks in HPC environments, we will need to investigate appropriate optimizations. One solution that partially mitigates the memory overhead issue (especially for Python dictionaries) is the use of alternate data structures such as *slots* or *namedtuples*.

6.5 | ScienceSearch performance

ScienceSearch provides specific advantages to the problem of implementing search engines over domain specific scientific data found in HPC systems. ScienceSearch's novelty is its mechanism for combining and correlating information from data with other data sources, such as research papers, images, proposals, calendar entries. This is accomplished through a set of deep learning and natural language processing algorithms employed by ScienceSearch, that are used to generate metadata tags. The majority of classical search engines designs and architectures[19] assume the input data to be a collection of data sources that area flat domain of input data where the collection of data sources are homogeneous in structure and semantics, and ScienceSearch innovates in this area by providing a mechanism for combining structurally and semantically different data sources.

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

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

6.6 | Impact of future hardware

While the performance of the ScienceSearch platform does not solely depend on a set of hardware properties, as shown by the experiments across different testbeds, ScienceSearch can still benefit from the performance improvements of computer hardware (CPU, memory, storage and network). Some immediate improvements in performance stem from, as mentioned in Section 6.1, from better inter-container communication through a network that maintains a certain degree of security and isolation without significantly sacrificing performance. Using hardware solutions such as VLANs could alleviate many performance bottlenecks either from the network devices, the operating systems or the container hypervisor, while still retaining the similar security properties to IPsec without the need for encryption.

Other more system-wide improvements could come from the development and adoption of exotic hardware. For example NVIDIA's emerging[20] (DPU) could be used to improve the performance and scalability of HPC applications by providing the means to overlap communication with computation.[21] UPMEM have been working on processing-in-memory (PIM)[22] devices that could be used to accelerate database query processing[23] by moving computation to where data resides (i.e., memory DIMMS) and by avoiding bringing the data to where computation is performed (i.e., the CPU). These are just two examples of future technologies that ScienceSearch could exploit in order to achieve even higher performance and scalability.

7 | CONCLUSION

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

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

ACKNOWLEDGMENTS

This material is based on work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) and resources used at the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility, supported by the Office of Science of the U.S. Department of Energy, both under Contract No. DE-AC02-05CH11231. We thank NCEM's staff for their help in providing data and insights that made this work possible. We also thank NERSC's Spin team for supporting the deployment of this work.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available at https://bitbucket.org/sciencesearch/sciencesearch-mi/src/master/.

ORCID

Alexandru Iulian Orhean https://orcid.org/0000-0002-9633-5790

REFERENCES

1. Rodrigo GP, Henderson M, Weber GH, Ophus C, Antypas K, Ramakrishnan L. ScienceSearch: Enabling search through automatic metadata generation. Proceedings of the 2018 IEEE 14th International Conference on e-Science; 2018:93-104; IEEE.
2. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun ACM*. 2008;51(1):107-113. doi:10.1145/1327452.1327492
3. White T. *Hadoop: The Definitive Guide*. 4th ed. O'Reilly Media, Inc; 2015.
4. Jain A, Ong SP, Hautier G, et al. Commentary: the materials project: a materials genome approach to accelerating materials innovation. *APL Mater*. 2013;1(1):011002. doi:10.1063/1.4812323
5. Cottingham RW. The DOE systems biology knowledgebase (kbase): progress towards a system for collaborative and reproducible inference and modeling of biological function. *BCB'15*. ACM; 2015:510.
6. Pietriga E, Gözükan H, Appert C, et al. Browsing linked data catalogs with LODAtlas. In: Vrandečić D, Bontcheva K, Suárez-Figueroa MC, et al., eds. *The Semantic Web – ISWC 2018*. Springer International Publishing; 2018:137-153.
7. Przybyla P, Soto AJ, Ananiadou S. Identifying personalised treatments and clinical trials for precision medicine using semantic search with Thalia; 2018.
8. Brickley D, Burgess M, Noy N. Google dataset search: building a search engine for datasets in an open web ecosystem. *WWW'19*. ACM; 2019:1365-1375.
9. Pratt B, Howbert JJ, Tasman NI, Nilsson EJ. MR-tandem: parallel X!Tandem using Hadoop MapReduce on Amazon web services. *Bioinformatics*. 2011;28(1):136-137. doi:10.1093/bioinformatics/btr615
10. Lewis S, Csordas A, Killcoyne S, et al. Hydra: a scalable proteomic search engine which utilizes the Hadoop distributed computing framework. *BMC Bioinform*. 2012;13:324. doi:10.1186/1471-2105-13-324
11. Kopanos C, Tsiolkas V, Kouris A, et al. VarSome: the human genomic variant search engine. *Bioinformatics*. 2018;35(11):1978-1980. doi:10.1093/bioinformatics/bty897
12. Corti P, Kralidis AT, Lewis B. Enhancing discovery in spatial data infrastructures using a search engine. *PeerJ Comput Sci*. 2018;4:e152.
13. Srivastava D, Iyer A, Kumar V, Sengupta D. CellAtlasSearch: a scalable search engine for single cells. *Nucl Acids Res*. 2018;46(W1):W141-W147. doi:10.1093/nar/gky421
14. Azman SK, Anwar MZ, Henschel A. Visibiome: an efficient microbiome search engine based on a scalable, distributed architecture. *BMC Bioinform*. 2017;18(1):353. doi:10.1186/s12859-017-1763-0
15. Gormley C, Tong Z. *Elasticsearch: The Definitive Guide*. 1st ed. O'Reilly Media, Inc; 2015.

16. Apache Solr; 2020. Accessed January 22, 2020. https://lucene.apache.org/solr/.
17. McCandless M, Hatcher E, Gospodnetic O. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Company; 2010.
18. Yang P, Fang H, Lin J. Anserini: enabling the use of lucene for information retrieval research. *SIGIR'17*. ACM; 2017:1253-1256.
19. Baeza-Yates R, Ribeiro-Neto B. *Modern Information Retrieval*. Vol 463. ACM Press; 1999.
20. Burstein I. Nvidia data center processing unit (DPU) architecture; 2021:1-20; IEEE.
21. Bayatpour M, Sarkauskas N, Subramoni H, Maqbool Hashmi J, Panda DK. Bluesmpi: efficient MPI non-blocking alltoall offloading designs on modern bluefield smart nics; 2021:18-37; Springer.
22. Zois V, Gupta D, Tsotras VJ, Najjar WA, Roy JF. Massively parallel skyline computation for processing-in-memory architectures. *PACT '18*. Association for Computing Machinery; 2018.
23. Kepe TR, de Almeida EC, Alves MAZ. Database processing-in-memory: an experimental study. *Proc VLDB Endow*. 2019;13(3):334-347. doi:10.14778/3368289.3368298

How to cite this article: Orhean AI, Giannakou A, Antypas K, Raicu I, Ramakrishnan L. Evaluation of a scientific data search infrastructure. *Concurrency Computat Pract Exper*. 2022;34(27):e7261. doi: 10.1002/cpe.7261