Group Project
2024

INFO-H-415: Advanced Databases

Search engines with Sphinx, ElasticSearch and OpenSearch

Alfio Cardillo, Charlotte Garcia, Jule Grigat, Nicola Mambelli, Ignacio Del Río and Josu Bernal

Contents

1 Introduction

1.1 Overview

In today's digital age, the efficient retrieval of information from vast amounts of data has become paramount. Search engine databases play a crucial role in enabling users to quickly access the information they seek. Three popular solutions in this domain are Elasticsearch, OpenSearch and Sphinx. These technologies provide powerful tools for indexing, searching, and analyzing large volumes of data, offering valuable insights and enabling efficient information retrieval.

1.2 Aim and Objectives

This project aims to explore and compare the capabilities of Elasticsearch, OpenSearch and Sphinx as search engine databases. The objectives include evaluating their performance in indexing and querying data, assessing their scalability, and understanding their respective strengths and weaknesses. By conducting this study, we seek to gain a comprehensive understanding of these technologies and their applicability in real-world scenarios.

2 Search Engines

In this section, we will provide an explanation of search engine databases, including how they operate and the technology that powers them. This overview will be the initial step in comprehending the technology that we will be evaluating and incorporating into this project. During this article, we will assume that the reader is familiar with relational databases and the SQL language but has no prior knowledge of search engines.

2.1 Search Engines

A search engine is a practical application of information retrieval techniques for large-scale text collections. In simpler terms, it is a database designed to retrieve information from stored documents. For example, Google is a search engine that stores web page documents. These documents stored by search engines have some structure, such as title, author, and date, which are called attributes. However, the main difference between a document and a typical relational database is that most of the information in the document is in the document itself, in the form of unstructured text.

2.2 Conceptual and technical issues

To gain a better understanding of how a search engine works, it's important to acknowledge the challenges it faces. These challenges can be categorized into two main groups: inherent problems related to retrieving information and technical problems associated with implementing a search engine. Let's start by exploring the inherent problems of retrieval.

- **Relevance:** We refer to the significance of a document based on the query provided by the user as relevance. It is not just about matching words; the same concept can be expressed in various ways. It's about extracting the essence of each document. It is also crucial to differentiate between documents that are relevant to the user's query and documents on the same topic that may not be relevant. For instance, if a user is searching for tickets to Taylor Swift's upcoming concert, the search engine should not display a webpage about Taylor's 2002 tour, even though it is a Taylor Swift concert. As evident, determining relevance is a complex undertaking.

  In order to tackle the issue of relevance, researchers suggest retrieval models and assess their effectiveness. A retrieval model is an algorithm used in a search engine to generate a ranked list of documents. A good retrieval model will locate documents that are likely to be deemed relevant by the user who entered the query.

- **Evaluation:** Assessing the quality of the retrieval of the information can be hard, since the ranking of the documents completely depends on the user's query. It is important to have a method for determining whether the answer provided is good or not. Using measures such as precision and recall can be difficult because there may not be a definitive correct solution for every query. There are various approaches to address this issue, including testing the effectiveness of search engines using pre-labeled databases or using user interaction to evaluate the quality of the retrieval.

- **Information Needs:** It is important to emphasize that evaluations should be user-centered. This means that any retrieval process should seek to understand the underlying information need that is hidden within the user's query. For instance, a query like "Python" could have different meanings, referring to either the programming language or the animal. This type of specification problem in the queries is very usual since the person doing the query is the user, not a trained engineer. Techniques such as query suggestion, query expansion, and relevance feedback are used to address this issue.

In addition to conceptual issues, search engines themselves have to deal with many technical problems, just like any other system. The major challenges include performance, as we want queries to find solutions as quickly as possible; incorporating new data, since the index-based architecture of search engines makes it difficult to implement new data (this will be explained carefully in the next section); scalability and adaptability to the specific purpose of the database; and addressing specific problems such as avoiding spam documents.

2.3 Architecture of a Search Engine

A search engine performs six distinct processes, each of which is carried out by different components. In this section, we will delve into the architecture of search engines. However, we will only focus on the essential parts and provide details about the most important elements.

1. **Text Acquisition:** Even if text acquisition is an important part of search engines since in this project we will work only with defined databases, we will not explain in detail this process. It is just necessary for the reader to know that many search engines have crawling and feeding components in charge of finding documents and feeding them to the database in real-time.

2. **Text Transformation:** Now that we have our text storage, we don't want to store it as plain text. We actually want to structure it so we can search through it efficiently. To achieve this, we will create indexes. However, before creating the indexes, we need to transform and adapt our text to make it more meaningful. Many components are involved in this process.

The parser is responsible for processing the text as tokens. Tokenization is not trivial, as tokens may include information about the syntax and structure of each word. For example, the token should indicate if the word is part of the document's title or not.

We also have a stopping component, which removes common words before indexing, such as "the", "and" or "to". It is important to study the number of words that we can delete without losing essential information.

The streamer and classifier handle grouping similar words based on morphology or topic. Finally, we have other components responsible for retrieving special text, such as links or words in italics.

3. **Index Creation:** The most important part of search engines is the creation of indexes. This process starts by gathering statistical information about words, features, and documents, known as document statistics. This information is then used by the "weighting component" to calculate the importance of each word in the document. The weights of the words are used to create inverted indexes, which are essentially a list of each word and its weight for each document. These indexes need to be efficient and updatable to accommodate new words added to the document, and they are usually compressed to enhance their efficiency. This is not a easy task since the introduction of new words could affect the importance of the already indexed words. One of the key features of search engines is the distribution of these indexes across multiple computers, which allows for parallel querying, processing, and indexing. We will delve into this topic in more detail later.

4. **User Interaction:** The components responsible for this process take the query as input, analyze, correct, and expand it to enhance the results.

5. **Ranking:** The ranking process is based on the retrieval model, which is used to generate the ranking displayed to the user. The scoring process is highly optimizable because there are many ways to calculate scores. Additionally, ranking can be distributed horizontally in two ways. A query broker allocates each query to different processors horizontally to improve performance. Caching distribution can also be implemented, leaving popular indexes in memory for time-saving purposes.

6. **Evaluation:** This process is in charge of evaluating the performance and relevance of the engine in an attempt of improving. Evaluation is performed in many different ways: user interaction, prelabeled document ranking analysis and component monitoring are involved in this process.

3 Elasticsearch

Elasticsearch is an open source distributed, RESTful search and analytics engine, scalable data store, and vector database capable of addressing a growing number of use cases. As the heart of the Elastic Stack, it centrally stores your data for lightning-fast search, fine-tuned relevancy, and powerful analytics that scale with ease.[8]

3.1 Apache Lucene

A default search index, as described above, is an inversed index that has for keys the tokenized words. The most popular open source, search index is Apache Lucene. On top of which technologies like Elasticsearch are built.

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Figure 1: Levenshtein distance matrix

Lucene, expands the concept of a search index on full text, by allowing for more complex operations, examples are Levenshtein distance1, finding words that are close to the word typed, coordinate and time searching and so on.

3.2 Why Elasticsearch

Elasticsearch was born on top of Lucene and as stated by its creator, Shay Banon, in this Stack Overflow post [4], he created it because of two main reasons:

- Lucene is not that usable, especially when you want it to perform well.
- Lucene is just a library and does not support a distributed system.

To address that, he first created Compass, which was a simplification of Lucene to make everyday operations simpler.

Later he needed a system able to be distributed, and that was able to talk easily with other platforms. That is the reason for Elasticsearch, a distributed search engine that speaks JSON.

3.3 The architecture

The main components of the Elasticsearch architecture are: [5]

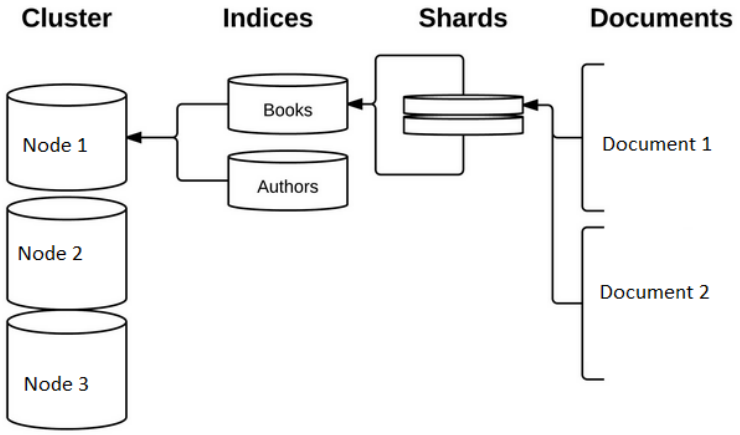


Figure 2: Elasticsearch architecture[12]

- **Clusters**: A cluster is a collection of nodes that together hold all the data and provide search and indexing capabilities across all nodes. A cluster ensures that the data and the load is distributed across the nodes. That means that they are able to scale horizontally, by allocating to that same cluster more nodes.
- **Node**: The single instances of Elasticsearch that belongs to a specific cluster and that is identified with an unique id. It is responsible for storing data and participate in the cluster's indexing and search capabilities. Every node has a role, this creates different categories of nodes: [9]
  - **Master-eligible node**: The master node is responsible for cluster-wide actions like creating and deleting an index, tracking which node are part of the cluster and allocating shards to nodes.
  - **Data node**: Nodes that hold the shards and handle data with search and aggregation operations. It is also possible to create different nodes depending on how often the data they contain is queried or changed. The scale of such nodes takes the name from temperature and indeed Elastic defines Hot, Warm, Cool and Frozen nodes, underlining that the data contained in these type of nodes is decreasingly used. This means that nodes of high temperature requires more and faster resources like SSDs, and they might have more replicas assigned.
  - **Ingest node**: Responsible for pre-processing pipelines. They are needed when the type of operations performed when ingesting requires such resources, that having specialized nodes for them actually makes sense.

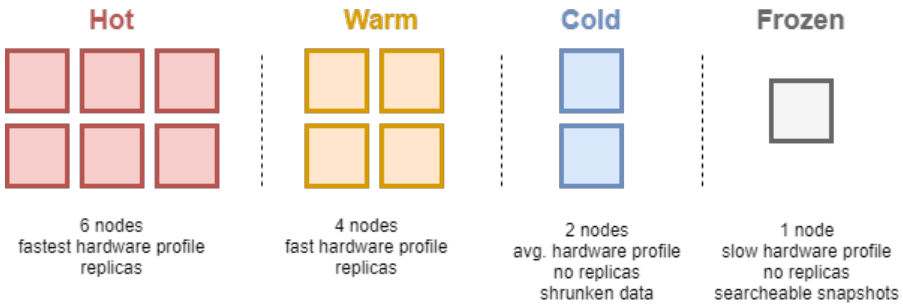


Figure 3: Different data node roles

  - **ML node**: Run jobs and handle machine learning API requests
  - **Transform node**: Like the ML, they are specialized in Transformation requests.

  Every cluster must have one master and at least a data role, and because a node can have multiple roles, clusters of just a node are totally possible.

- **Ports**: Nodes need to communicate between them and with the outside, to do so, Elasticsearch uses two specific ports.
  - **9200**: Serves as the HTTP REST API, allowing to communicate with HTTP requests
  - **9300**: Used for internal communication between nodes, like synchronization, coordination, data replication etc.

- **Index/shards**: When doing a parallelism with the relational schema an Elasticsearch index is similar to a database. It can be defined as a collection of documents, and which in turn can be divided into shards, the individual instances of Lucene.

- **Replicas**: Data replication in Elasticsearch is achieved using shards, in particular, Elastic defines another type, called Replica shards which are copies of primary shards. By default, each primary shard has at least one replica. If a node fails,

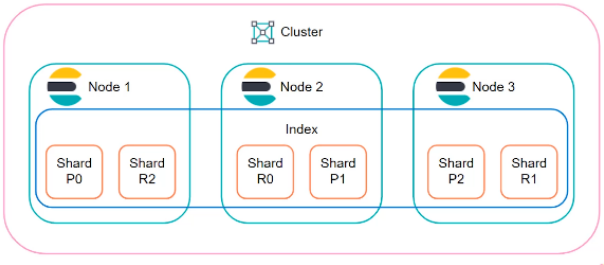


Figure 4: Replica and primary shards

they ensure that no data is lost, and that the cluster remains operational, of course if we are not talking of a single node cluster.

- **Analyzers**: An analyzer is what takes text and breaks it into tokens that are then ready to be indexed and searched. An analyzer consists of a tokenizer and possibly a set of filters. The tokenizer divides text into terms, while filters can modify these terms, like converting them to lowercase or removing specific words, like the stopping one. By default, Elasticsearch comes with some default analyzer, but a user can easily define a custom one, as we will see in section 7.

- **Documents**: A JSON object that is the basic unit we will store and index in Elasticsearch. It is defined by a set of fields, each having its own type, and is schema-less, meaning the fields and their type can change from a document to another.

- **REST API**: One of the most important feature of Elasticsearch, and what allows it to easily communicate with other platforms, is the REST API component. It allows users, micro-services or other components of bigger applications to interact with the cluster through HTTP requests. Thanks to that, one can execute operations like such as indexing, searching, updating, or deleting documents.

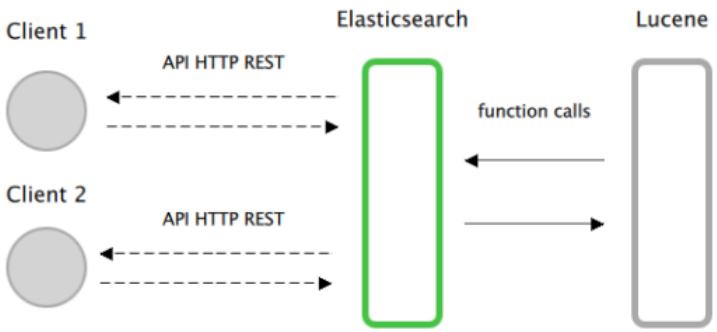


Figure 5: REST API communication

3.4 Caching

One thing that makes Elasticsearch performant, is the way it caches results. [7] Usually caching involves part of an index or the whole query result.

A peculiar caching method that Elastic is able to use it's the query caching, which allows it to cache parts of the query that are very common or popular. As an example, given the following query:

*Find the products with name **tissue** and tag **sales***

In this case, if we are managing an ecommerce, we expect the first condition on the name, to happen very rarely, while the condition on the tag = sales being the ones that is mostly applied in our search bar.

In this case Elasticsearch will be able to cache not the whole query results, but just the results coming from the second condition, as that second condition is expected to appear more frequently in other queries, while the first is rather specific.

Figure 6: Bitmap index used in part of query caching

The concept at the base of this type of caching is to have a bitmap that records what documents a certain condition hits, and puts a 1 if that document is retrieved. Bitmaps are in this way helpful, because they are both easily compressible and they allow bitwise operations like AND or OR, to understand when a document matches several conditions.

3.5 Semantic Search

> *Semantic search is a data searching technique that focuses on understanding the contextual meaning and intent behind a user's search query, rather than only matching keywords.[11]*

Semantic search allows for incredible capabilities, for example when querying our search engine with a query like **tiger**, the results we will get are those documents that have the word tiger in it. This is an example of how lexical search works. Semantic search instead is far more powerful, indeed using the same query we can now get the documents containing not only tiger as a word, but other similar words or concept like **animal**, **predator**, **feline** etc.

3.5.1 How does it work

The way it works is by creating for each paragraph, phrase or word, an embedding, which is typically a vector representation of that text.

The embedding is created using a text encoder trained on textual similarity, meaning that vectors of similar phrase will have a short distance between one and another.

Once these embeddings are created, one can now do the same for the query, and with algorithms such as KNN, find the K nearest neighbour for that query, which will be connected to the K most similar documents to the query received in input.[14]

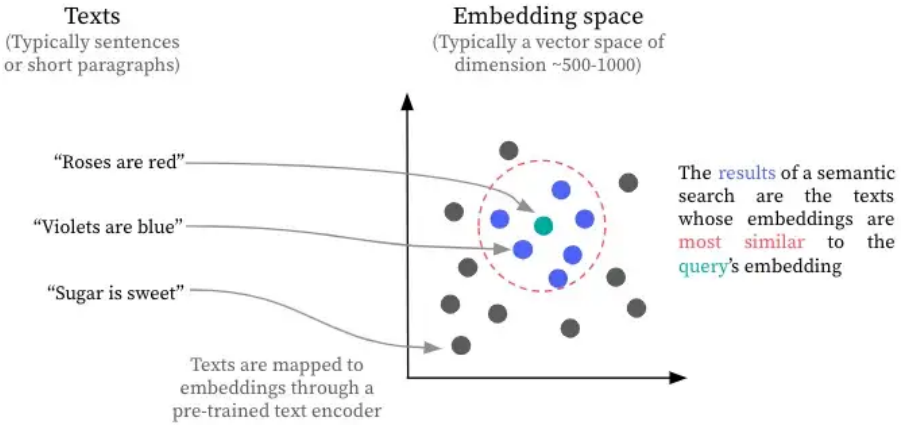


Figure 7: Semantic search

3.5.2 Advantages and Disadvantages

As said before, Semantic search is a powerful technique, which allows to broaden search results and the capabilities of a search engine. It can also be used, to cluster or find similar documents, by calculating similarities of their embedded vectors.

On the opposite side, semantic search may return more documents then needed, and usually produce a long tail of documents which have some similarity with the query but are not relevant to the user needs.

Other issues are, working with domain-specific vocabulary, which may require to build an embedding model just for the specific search task and in case of big data, comparing the query embedding with all the documents is not a scalable operation and may become more and more expensive with a bigger corpus.

4 Sphinx

Sphinx is a free search server written in C++ that focuses on query performance and search relevance. The primary client API is SphinxQL, a SQL dialect. While Sphinx is a complex technology filled with intricate details and useful functions, this project will only cover the basics. Our goal is to understand how Sphinx works and highlight the main differences between various tools, rather than aiming to become an expert. If you are interested in learning more, we encourage you to refer to the official documentation. [1]

4.1 Features overview

In this section we will list the main features that Sphinx offers as a search engine.

- SQL, HTTP/JSON, and custom native SphinxAPI access APIs
- Near Real Time and offline indexing
- Full-text and non-text searching
- Relevance ranking, from basic formulas to ML models
- Results retrieved from multiple servers

4.2 Sphinx Architecture and Concepts

Sphinx runs upon three major components as depicted in Figure 8 that interact with each other. In the graphic, a "MySQL Server" is used as the underlaying full text database, but that can differ, depending on the installation.

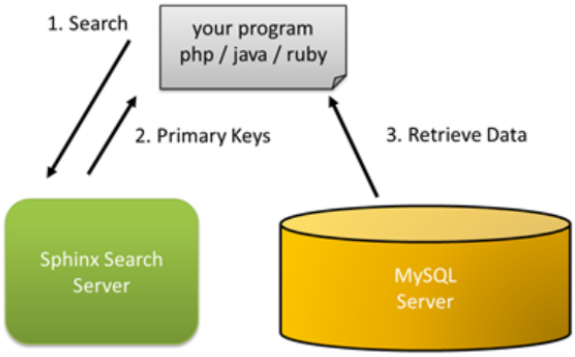


Figure 8: Sphinx Architecture

1. **Search Query:** The individual UI program gets an input query by a user and sends a search query to the *Sphinx Search Server*. The Sphinx server or "Search

Deamon" quickly searches through its pre-built index to find matching results. We will talk about the creation of the index in more detail lateron.

2. **Primary Keys:** Sphinx returns a list of *primary keys* (IDs) that correspond to the rows in the *MySQL Server* which match the search query.

3. **Retrieve Data:** Using the primary keys, the program queries the *MySQL Server* to fetch the full data, such as document content or other detailed information.

We will now look more closely into the important components in the architecture of Sphinx and its general concepts.

- **Search Server:** The central component of Sphinx, that performs the search and indexing operations and is depicted as the green rectangle in the graphic 8, is the so called "Search Deamon". It runs as a standalone application, processing queries and returning results. As one of the core principles of search engines is efficiency and fast information retrieval, it is designed for efficiency, handling both real-time and batch-based queries equally. In a typical installation, as we can see in 8, the search daemon runs as a standalone application which can be connected to in a very similar way to a MySQL server, e.g. to retrieve search indices but not search data.

- **Text Acquisition:** Sphinx does the necessary text acquisition in a pretty straightforward way, not relying on complex crawling or live data feeds like other search engines. Instead, it pulls the data directly from one or multiple existing sources like relational databases (e.g., MySQL, PostgreSQL), CSV files, etc. This makes it fairly easy to integrate Sphinx into projects that already have an existing a database. Additionally Sphinx can handle live data sources and static databases individually, which will be elaborated on in the next point "Indexing".

- **Text Transformation:** The Text Transformation process happens during the creation of the index, independently of the chosen type of index. Sphinx performs basic normalization and morphology actions, like stemming, flexible tokenization, morphology, mappings and annotations.
  Sphinx completely lacks the ability tho to understand semantics of phrases or the meaning behind single words.

- **Indexing:** The Indexing in Sphinx is one of its most outstanding features. By offering three different types of Indexing, the real-time index, the disk index and the distributed index, it covers both cases of underlaying databases, the dynamically updating data sources and the static kinds and treats them accordingly. Therefore Sphinx has the ability to connect to live data sources using real-time indexing while still supporting offline batch indexing. For every data source that

is added the index type can be reconcidered.

Real-Time Index
The real-time (RT) index is designed for dynamic datasets that need frequent updates. Unlike traditional indexes, it allows you to add, update, or delete documents instantly without rebuilding the entire index. New data is written directly to memory and periodically saved to disk for persistence. This makes RT indexes perfect for applications where data changes often, like a news feed or forums.

Disk Index
Sphinx is also offering a so called "Disk Index". That is a static and read-only structure that works great for large datasets with no regular updates, like archiving historical documents or handling massive text repositories. Disk indexes are designed to provide maximum indexing and searching speed, while keeping the RAM footprint as low as possible [1]. Data is processed in a bulk, transformed into an optimized structure, and stored on a "disk". Since the disk index is read-only, it provides great performance for searching, especially when handling large amounts of data, but requires a full rebuild with every update.

Distributed Index
The distributed Indexing from Sphinx query several indexes at once, treating them as if they were a single index. Sphinx is in this indexing case in doing the sending of search requests to remote machines in the cluster, aggregating the result sets and even retrying failed requests.

- **Relevance and Ranking**: To asses if a document is relevant for the entered search query, Sphinx uses three methods.

  1. Exact word matching
  2. Attribute weighting
  3. Proximity matching

  The more often a search term appears in a document, the more relevant that document becomes. In order to prevent longer documents from being favorised, Sphinx uses a default function called "BM25" [10].

  Proximity matching, meaning, its able to recognize when search terms ar close together in a document, which indicates a higher relevance for the search query.

  For ranking the output, Sphinx uses customizable ranking formulas. So if required, you can adapt the way of calculating the ranking weight score of each document for the search output per query like in the following example:

```
SELECT *, WEIGHT() FROM myindex
WHERE MATCH('hello world')
OPTION ranker=expr('sum(lcs)*10000+bm15')
```

Here we are ranking by a value that is based on the "longest common subsequence" (lcs) for the provided two search words in order, multiplied by 10.000 to assign more weigth to that criteria and apply the "BM15" function, which is a variation of the "BM25" function, so in this case we don't take the document length into account.

4.3 Querying and Syntax

In this section, we will introduce the main syntax at a basic level to the reader. We will not delve into specific index types or explain complex functions.

SphinxQL is a SQL dialect, meaning its syntax closely resembles that of standard SQL. Therefore, we believe that providing a few examples will effectively demonstrate how this tool works at a beginner level.

Let's create an index (table) and perform some basic queries.

```
-- Let's begin by creating an index and inserting some data
CREATE TABLE test
(id bigint, title field stored,
content field stored, gid unit);

INSERT INTO test (id, title) VALUES (123, 'hello world');
INSERT INTO test (id, gid, content)
VALUES (234, 345, 'empty title');

SELECT * FROM test;
```

The result of the last query will be something like:

id	gid	title	content
123	0	hello world	
234	345		empty title

Let's perform a bit more complex queries, using the full text search capabilities of Sphinx.

```
SELECT * FROM test WHERE MATCH('hello');
```

The result will be:

id	gid	title	content
123	0	hello world	

```
SELECT * FROM test WHERE MATCH('@content␣empty');
```

id	gid	title	content
234	345		empty title

```
SELECT * FROM test WHERE MATCH('@content␣hello');
```

However, this last query will result in the empty set because the word hello is not in the column content of any document.

Before diving deeper into explaining the main operators and modifiers, we should underline some limitations of SphinxQL compared to SQL.

- SELECT always has an implicit ORDER BY and LIMIT clauses based on the weight of each document on the ranking (ORDER BY WEIGHT() DESC, id ASC LIMIT 20).
- WHERE clause only accepts for column checks basic math comparison operators and BETWEEN, IN and NOT IN operators. However, SphinxQL documentation explains some workarounds to manage more complex comparisons.

> WHERE MATCH('pencil') AND (color = 'red' OR color = 'green') is not valid because color is part of a complex expression, however we could use MATCH('pencil') AND color IN ('red', 'green'), instead.

Operators By default, full-text queries in Sphinx require that all keywords in a document match. However, text queries offer much more flexibility. You can combine keywords using operators like OR and NOT, along with brackets, to construct any boolean expression as needed. We will include a cheat sheet of the available operators based on the official cheat sheet provided by Sphinx.

Operator	Example	Description
AND	word1 word2	Match both keywords in the document
OR	word1 | word2	Match any keyword
term-OR	word1 || word2	Match any keyword, but don't take into account in-query position for ranking
NOT	word1 -word2 word1 !word2	Match 1st keyword, but exclude matches of 2nd
MAYBE	word1 MAYBE word2	Match 1st keyword, but include 2nd keyword when ranking
fields limit	@field1 word1 @field2 word2 @(field1,field2) word1 @!(field1,field2) word1 @* word1	Limit matching to a given field Limit matching to given fields Limit matching to all but given fields Reset any previous field limits
position limit	@field1[N] word1	Limit matching to N first positions in a field
phrase	"word1 word2" "word1 * * word4"	Match all keywords as an (exact) phrase
proximity	"word1 word2" N	Match all keywords within a proximity window size N
quorum	"word1 word2 word3"/N "word1 word2 word3"/(1/N)	Match any N out of all keywords Match any given fraction of all keywords
BEFORE	word1 <<word2	Match keywords in this specific order only.
NEAR	word1 NEAR/N "word2 word3"	Match in any order within a given distance

Operators should be written as shown in the cheat sheet. For example, the MAYBE operator would be used as (word1 MAYBE word2) in a query. The phrase word1 maybe word2 is a regular query that requires all three keywords to match, instead of using the operator.

Modifiers In the following cheat sheet, we display the three most common and simple modifiers used in SphinxQL.

Modifier	Example	Description
field start	$\wedge word1$	Only match at the very start of (any) field
field end	word1$	Only match at the very end of (any) field
IDF boost	word1$\wedge N$	Multiply keyword IDF by a given value when ranking

4.4 Advantages and Disadvantages

In this section we will list some of the advantages and disadvantages of Sphinx compared to the other two tools.

Sphinx' most unique selling point is its ability to connect to live data sources using real-time indexing while simultaneously supporting offline batch indexing via a disk index. In that point Sphinx combines the necessary flexibility with performance, which makes it suitable for a variety of projects with different data sources and needs. Additionally its fairly simple to connect to already existing data sources, which makes it attractive to use in for example brown field projects.

Another big advantage of Sphinx is its speed, making it a suitable choice for projects where searches need to happen instantly, especially for static data sources, using the disk index. Om the contrary, those use cases need a full rebuild of the index, for every change in the data source.
Its SQL-like query language, SphinxQL, is easy to be learned by people that are already familiar with SQL databases. The syntactic adaptions include an extra set of operators required for search engine functionalities, as we saw earlier in this chapter. Sphinx is also lightweight and doesn't need as much memory or processing power as bigger tools like Elasticsearch or OpenSearch, which also makes it more accessible. Sphinx provides great flexibility, letting its users customize how search queries operate and rank results.

On the downside, Sphinx doesn't handle large amounts of data or very high traffic as well as Elasticsearch or OpenSearch. It also doesn't have advanced features like semantic search or tools for analyzing data, which can be important for more complex projects. Another disadvantage is its smaller user community, which means there are fewer resources and plugins available. Overall, Sphinx is best for smaller or simpler projects where speed and efficiency are more important than fancy features or the ability to scale up.

4.5 Worth mentioning

In this section, we will discuss some complex features of Sphinx that we believe are important but do not fit into this beginner's introduction.

- Morphology and text-processing tools (tokenization, lemmatizers, stemmers,...)
- Geosearch support, vector indexes and memory budgets
- Ranking options
- Query suggestions and snippets builder
- Substring and wildcard searches

5 OpenSearch

5.1 Introduction and Similarities with ElasticSearch

OpenSearch is an open-source, enterprise-grade search and observability suite that brings order to unstructured data at scale. [19]

OpenSearch is a distributed search and analytics engine that supports various use cases, from implementing a search box on a website to analyzing security data for threat detection. It was created in early 2021 after Elasticsearch 7.10.2 became licensed under Elastic's SSPL (Server Side Public License), which was not OSI-approved.
In August 2024 the GNU Affero General Public Licence was added as an option to ElasticSearch, making it mostly free and open-source once again. [20].
OpenSearch is an open-source, Apache 2.0 licensed fork maintained by Amazon Web Services (AWS) and the community. It was designed to provide a fully open-source alternative to Elasticsearch and Kibana. [3]
OpenSearch maintains compatibility with Elasticsearch 7.10.2, making it easier to migrate without reconfiguring indices or APIs. Moreover, OpenSearch's development is community-led, promoting transparency and flexibility in the project's direction.

5.2 The architecture

Its architecture is similar to the architecture of ElasticSearch as both are built on Apache Lucene and defined for horizontal scalability. Similar to Elasticsearch, OpenSearch clusters consist of multiple nodes (servers that store data and process search requests) with distributed data and workload. Like in ElasticSearch, nodes in OpenSearch have specialized roles such as Cluster Manager Node (or Master Node), Data Node, Ingest Node, Coordinating Node, Dynamic Node (used for ML tasks) and Search nodes, enabling horizontal scalability and optimized resource utilization [19].

Figure 9 illustrates an example of an OpenSearch cluster.

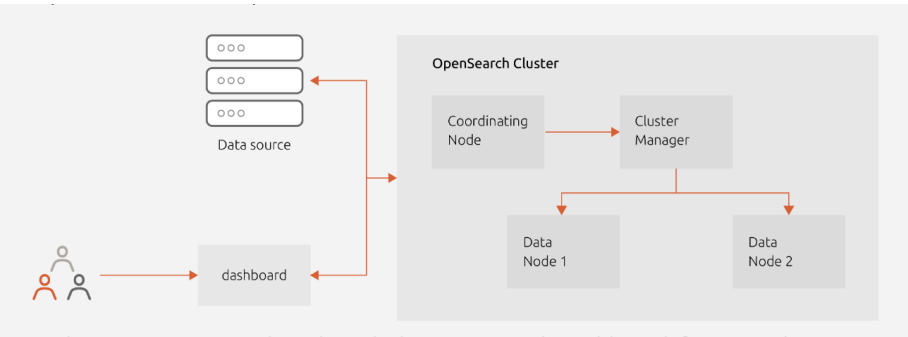


Figure 9: Structure of an OpenSearch cluster [13]

As in ElasticSearch, nodes in Opensearch have different types of data tiers, repre-

senting distinct storage levels that classify data based on criteria like access frequency, cost efficiency, and performance needs. [19].

Figure 10 presents the main differences between ElasticSearch's and Opensearch's data tiers.

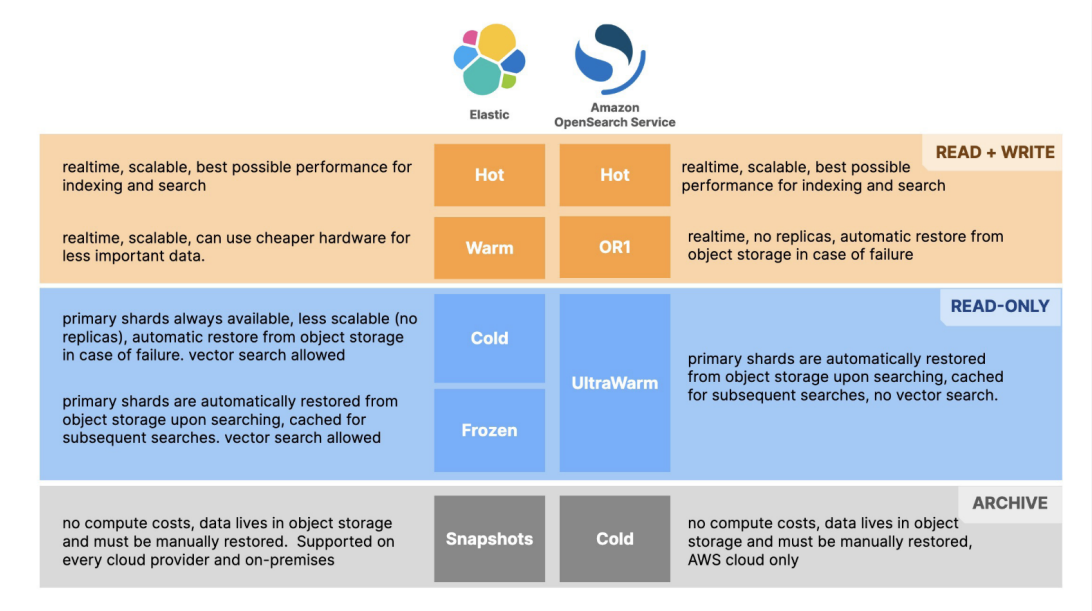


Figure 10: Difference between OpenSearch and ElasticSearch data tiers [17]

In OpenSearch, documents are stored in JSON format. The data is stored in indices. As in ElasticSearch, these indices can be split into primary shards to help scale the system and replica shards to protect against data loss if something goes wrong. Shards make it possible to run multiple searches at the same time and ensure that the data is always available, even if some parts fail.
OpenSearch can be run locally on a laptop—its system requirements are minimal [19]—but can also scale a single cluster to hundreds of powerful machines in a data center.

OpenSearch retains the same segment merging process as Elasticsearch, where deleted documents are removed during the merging process to optimize storage and improve search performance [19]. The segment merging process is used to optimize storage and improve search performance by:

- Merging smaller segments into larger ones and the larger segments become immutable.
- Expunging deleted documents: In Lucene-based search engines (like Elasticsearch and OpenSearch), when documents are deleted, they are not immediately removed from disk. Instead, they are marked as deleted, and the actual removal

happens during the next segment merge, improving disk space efficiency and search performance.

OpenSearch introduced concurrent segment search as an optional flag, and does not use it by default, it must be enabled using a special index setting. Elasticsearch on the other hand searches on segments concurrently by default.

5.3 Features Overview

OpenSearch offers a wide range of features that make it a competitive choice in the search engine space [3] [17]. Some of its most notable features include:

- Apache Lucene-Based: OpenSearch inherits Apache Lucene's robust indexing and full-text search capabilities, allowing for highly performant and scalable search.
- RESTful API Support: Provides a powerful REST API for seamless interaction with other systems using JSON.
- Near Real-Time Indexing: Supports near real-time indexing, enabling updates to be searchable almost immediately.
- Full-Text Search and Analytics: Includes advanced text search capabilities and built-in support for aggregations, log analytics, and anomaly detection.
- Visualization with OpenSearch Dashboards: OpenSearch Dashboards, a fork of Kibana, allows users to visualize data with customizable dashboards, offering insights from indexed data.
- Vector Search: Built-in support for vector-based similarity searches, crucial for applications like semantic search or recommendations.
- Security Features: OpenSearch integrates advanced security capabilities, such as fine-grained access control, authentication plugins, and role-based permissions that don't require a subscription (like in Elasticsearch).

5.4 Queries and Syntax

OpenSearch uses a Query DSL similar to Elasticsearch's JSON-based structure for creating powerful and complex queries. Queries are separated in two main types: Leaf queries and Compound queries. Leaf queries search for a specified value in a certain field or fields. Common leaf query types include: [19]

- Full-text queries: full-text queries split the query string into terms using the same analyzer that was used when the field was indexed.
- Term-level queries: to search documents for an exact term, such as an ID or value range.

- Joining queries: to search nested fields or return parent and child documents that match a specific query.

Compound queries serve as wrappers for multiple leaf or compound clauses, either to combine their results or to modify their behavior. They include the Boolean, disjunction max, constant score, function score, and boosting query types.

Elasticsearch and OpenSearch both divide search execution into two phases: the query phase (where each shard processes the search locally) and the fetch phase (where the search results are aggregated and the actual documents are retrieved) [2].

5.5 Advantages and Disadvantages

Advantages:

- Fully open-source with an Apache 2.0 license.
- Backward compatibility with Elasticsearch 7.10.2.
- Enhanced security features available out-of-the-box.
- Open-source and actively developed with a growing ecosystem of plugins.
- Supports modern search capabilities like vector-based semantic search.

Disadvantages:

- Smaller community compared to Elasticsearch.
- Some Elasticsearch plugins and features (from post-7.10 versions) are unavailable.
- Slower than ElasticSearch: "The results show that Elasticsearch is up to 12x faster than OpenSearch for vector search and therefore requires fewer computational resources." [13]

5.6 Worth mentionning

While Elasticsearch is more feature-rich, with premium offerings like X-Pack, OpenSearch provides a strong, community-driven, open-source alternative with all the core features and compatibility with Elasticsearch 7.10.2, making it ideal for users seeking a fully open-source search engine without licensing restrictions.

6 Benchmarking

In this section, we will explain how we benchmarked the various tools and discuss our findings. Since this project aims to compare three different tools, we could not find a single official benchmark that was compatible with all of them. Therefore, we conducted the following tests:

1. We conducted an official benchmark to compare OpenSearch and Elasticsearch because their underlying technologies are similar, making benchmarking straightforward.

2. We created a benchmark to compare the three tools with SQL Server. This allows us to see how this technology performs in full-text searches compared to relational databases.

3. We executed the benchmark we created in the previous step using only the three search engine technologies. However, this time we ran each query 50 times. This approach allowed us to compare the three technologies more effectively.

6.1 Official Benchmarking Discussion

In this subsection, we will explain in detail why an official benchmark cannot be applied to all three tools. First, it is important to note that there are two key metrics for evaluating search engines: performance and ranking. Performance refers to the speed of query processing, while ranking benchmarking involves comparing the results of query rankings against an established, pre-labeled ranking that the queries are expected to return. So before choosing a benchmark you have to decide which metric do you want to evaluate. In our case, to facilitate cooperation with relational systems and for standardization purposes we decided to benchmark performance.

Now that we know what we are looking for, we should establish a performance benchmark that allows us to compare the three different tools, and if possible, include some type of relational system as well. This is a complex task, as Elastic Search is the most widely used search engine, and there are not many other tools that are commonly used. Consequently, finding standard tools for comparison is not an easy endeavor, since many tools just focus on comparing different configurations of Elastic Search. In addition, the huge difference between query syntaxes made it harder, making reusing a benchmark and completely translating every query a harder job than creating a new benchmark from scratch.

Some examples of the benchmarks considered for this project with no luck are:

1. Full-Text Search Benchmark (FTSB): `https://shorturl.at/ZgyJ4`

2. DB Benchmarks: `https://shorturl.at/MObzI`

6.2 Official Benchmark

OpenSearch is search engine tool which is a fork of Elasticsearch, built by Amazon web services(AWS) in 2021 after Elastic NV changed its license from open-source to server side public license(SSPL). In September 2021 AWS and Linux Foundation, created the OpenSearch software foundation, and part of the managers of these companies moved from there to this newly created entreprise.
After researching amongst the various benchmarks distributed by companies and third parties, an incompatibility between Sphinx and Elasticsearch (and consequently OpenSearch) was observed.
In this part of the paper we will observe the application of a standard benchmark on the two most compatible tools, Elasticsearch and OpenSearch, while in subsequent sections an attempt will be made to develop a custom benchmark for comparing all three tools.

In this section, a standard benchmark was used to test the performance of the various search engine softwares. It is called geoname and is based on a public database that internally includes 11 million documents with various attributes of cities around the world. Such attributes are listed below and presented in Figure 11.

- Geoname ID
- Name
- ASCII Name
- Country Code
- Feature Class
- Feature Code
- DEM
- Location (lat, lon)

Geoname ID	Name	ASCII Name	Alternatenames	Feature Class	Feature Code	Country Code	Admin1 Code	Admin2 Code	Population	DEM	Timezone	Location (lat, lon)
386792	ʻAbd Ṭāhir	ʻAbd Tahir	Abd Tahir	P	PPL	IQ	9	90509	0	5	Asia/Baghdad	(46.59873, 30.89012)
386796	ʻArab ar Ruṭlah	ʻArab ar Rutlah	Arab ar Rutlah	P	PPL	IQ	9	90509	0	1	Asia/Baghdad	(46.64605, 30.83042)
386797	Ḥājj Daḥḥām	Hajj Dahham	Hajj Dahham,haj dham	P	PPL	IQ	9	90509	0	4	Asia/Baghdad	(46.62902, 30.8346)
386802	Sayyid Majīd	Sayyid Majid	Sayyid Majid,Sayyid Majīd,syd mjyd	P	PPL	IQ	2		0	4	Asia/Baghdad	(47.72667, 30.74802)
386813	ʻAbbās Rādī	ʻAbbas Radi	`Abbas Radi,ʻAbbās Rādī	P	PPL	IQ	6	99040	0	40	Asia/Baghdad	(44.19917, 32.885)
386826	Az Ẕahrah	Az Zahrah	Az Zahrah,Az Ẕahrah,alzhrt	L	AREA	IQ	3		0	279	Asia/Baghdad	(45.12764, 29.87057)
386831	Sharīʻat al Ḥammār	Shariat al Hammar	Shariat al Hammar,Sharīʻat al Ḥammār	H	MRSH	IQ	9		0	0	Asia/Baghdad	(46.65081, 30.68133)
386834	Ṣufnān	Sufnan	Sufnan,Ṣufnān	H	WLL	IQ	3		0	303	Asia/Baghdad	(45.57472, 29.22861)
386836	Buqāq	Buqaq		P	PPL	IQ	15	9166618	0	314	Asia/Baghdad	(42.93806, 36.69028)
386838	Qalīb aṣ Ṣulaylī	Qalib as Sulayli	Jalib as Sulayli,Qalib as Sulayli	S	PMPW	IQ	2	98247	0	49	Asia/Baghdad	(46.86988, 30.48477)

Figure 11: Example of Geonames table

6.2.1 Queries

This benchmark runs several queries that perform some of the most commonly used tasks in the search engine environment. In fact, one can observe queries that perform basic search, or queries that perform aggregation and sorting. Examples of these queries are shown below.
The following queries that are observed bellow play the role of search within OpenSearch. The first one focuses on searching for terms such as the term "AT" here, the second query on the other hand focuses on searching for phrases, in this case it will search for the phrase "Sankt Georgen."

```
{
  "name": "term",
  "operation-type": "search",
  "body": {
    "query": {
      "term": {
        "country_code.raw": "AT"
      }
    }
  }
},
```

Figure 12: Search term query

```
{
  "name": "phrase",
  "operation-type": "search",
  "body": {
    "query": {
      "match_phrase": {
        "name": "Sankt Georgen"
      }
    }
  }
},
```

Figure 13: Search phrases query

Below we observe a different typology of queries, in fact it focuses on the aggregation to compute the summation of the population for each country. In this specific case caching has also been enabled, so it will allow data to be saved in memory so that it does not have to compute it every time and optimizes future calculations with the same requirements.

```
{
  "name": "asc_sort_population",
  "operation-type": "search",
  "body": {
    "query": {
      "match_all": {}
    },
    "sort" : [
      {"population" : "asc"}
    ]
  }
},
```

Figure 14: Sort query

There are also queries that allow the search engine to be tested in its sorting functions, and below is an example of such queries. In fact, it allows cities to be sorted by increasing population numbers.

```
{
  "name": "country_agg_cached",
  "operation-type": "search",
  "cache": true,
  "body": {
    "size": 0,
    "aggs": {
      "country_population": {
        "terms": {
          "field": "country_code.raw"
        },
        "aggs": {
          "sum_population": {
            "sum": {
              "field": "population"
            }
          }
        }
      }
    }
  }
},
```

Figure 15: Aggregation query

Finally, there are also complex functions such as the one in Fig 16, which provide a score to each document by means of mathematical functions. The one observed in Figure 16 in particular calculates the logarithm of the population and multiplies the geographic coordinates (latitude and longitude). This type of query is called painless_static, since it is calculated through attributes that do not vary in the document.

```
{
  "name": "painless_static",
  "operation-type": "search",
  "body": {
    "query": {
      "function_score": {
        "query": {
          "match_all": {}
        },
        "functions": [
          {
            "script_score": {
              "script": {
                "source": "Math.abs(Math.log(Math.abs((int)((List)doc.population).get(0)) + 1) + (double)(doc.location.lon) * (double)(doc.location.lat))/_score",
                "lang": "painless"
              }
            }
          }
        ]
      }
    }
  }
},
```

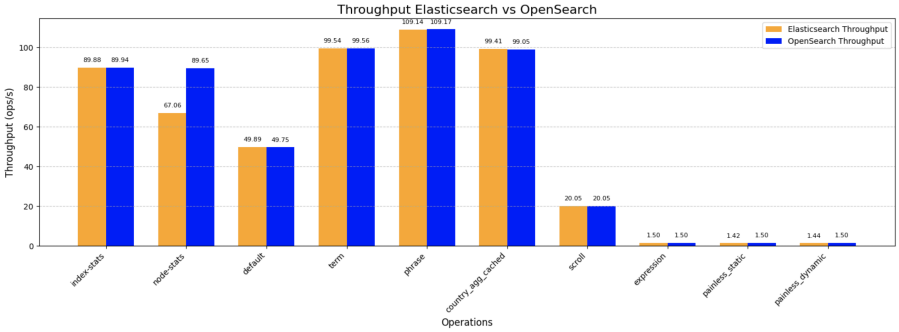
Figure 16: Painless_static query

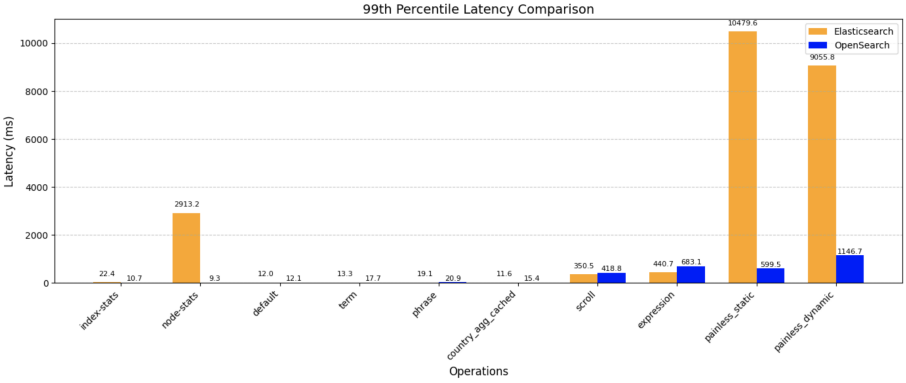
6.2.2 Results

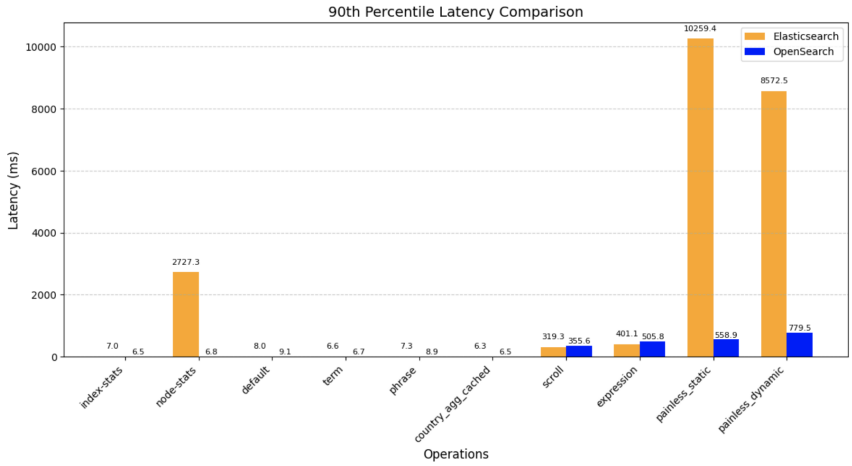
A technology provided by Opensearch called OpeanSearch-Benchmark was used to perform the process. With it, the benchmark introduced in part 1 was then run. To get a comparison of the performance of the tool, the same benchmark was run with the same "Opensearch-benchmark" technology for Elasticsearch. In order to obtain more reliable data for comparison, the same number of nodes with the same memory dedicated to each were given. The two tools were also executed using Docker Compose, in which the nodes and the memory allocated to them are specified below.

- Shards: 5
- Nodes: 2
- Memory:512MB

Below (in Figure 17) are graphs of the performances of the two tools after running geonames. In particular, two attributes are considered that are critical to consider the efficiency of the tools in individual tasks: the throughput and latency values.
Specifically throughput represents the workload on units of time, while latency represents the response time of the tools from task arrival to actual execution.


Throughput Elasticsearch vs OpenSearch
Elasticsearch Throughput
OpenSearch Throughput
Throughput (ops/s)
Operations
index-stats
node-stats
default
term
phrase
country_agg_cached
scroll
expression
painless_static
painless_dynamic
89.88
89.94
67.06
89.65
49.89
49.75
99.54
99.56
109.14
109.17
99.41
99.05
20.05
20.05
1.50
1.50
1.42
1.50
1.44
1.50


99th Percentile Latency Comparison
Elasticsearch
OpenSearch
Latency (ms)
Operations
index-stats
node-stats
default
term
phrase
country_agg_cached
scroll
expression
painless_static
painless_dynamic
22.4
10.7
2913.2
9.3
12.0
12.1
13.3
17.7
19.1
20.9
11.6
15.4
350.5
418.8
440.7
683.1
10479.6
599.5
9055.8
1146.7


90th Percentile Latency Comparison
Elasticsearch
OpenSearch
Latency (ms)
Operations
index-stats
node-stats
default
term
phrase
country_agg_cached
scroll
expression
painless_static
painless_dynamic
7.0
6.5
2727.3
6.8
8.0
9.1
6.6
6.7
7.3
8.9
6.3
6.5
319.3
355.6
401.1
505.8
10259.4
558.9
8572.5
779.5

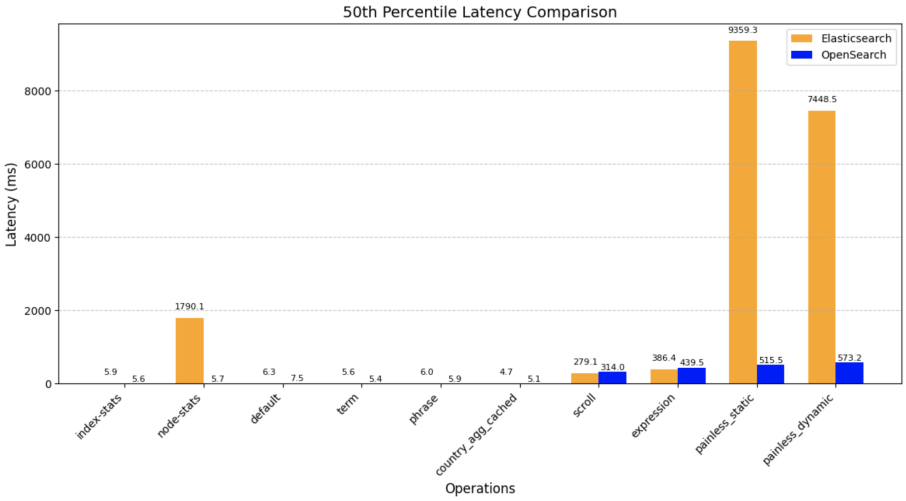


Figure 17: Performance of the tools after running geonames

In the first graph, it can be observed that the throughput between the two tools is very similar, indicating that they have a fairly comparable operation management. However, in the latency graphs, it is evident that Elasticsearch performs faster in search or aggregation queries, which suggests it may be optimized for these types of operations.
On the other hand, a clear difference is visible between the two tools, painless_static and painless_dynamic. This could be due to two reasons: first, Elasticsearch may not be optimized for executing complex equations like those present in painless-type queries; second, the tests might have been executed on a platform developed by OpenSearch. As we know, OpenSearch originates from Elasticsearch, and indeed, OpenSearch-Benchmark is a fork of Rally, the technology used to benchmark Elasticsearch. Therefore, while they are compatible, they still have internal differences, as they are currently managed by different companies.
It is possible that the technology dedicated to OpenSearch could be optimized for it, unlike Elasticsearch.

7 Our Application

In this section, we will explain the application considered for these tools. We proposed to work with Stack Overflow and created a benchmark to compare the different tools for this purpose.

7.1 Benchmarks description, setups and database description

In this subsection we will describe the benchmark that we created, describe the different setups that were used to perform the benchmark and the database used for it.

We will begin by describing the database utilized for this benchmark. For this assessment, we used the Stack Overflow database. Stack Overflow provides various versions of their database, each containing different amounts of data, all of which can be downloaded from their website.

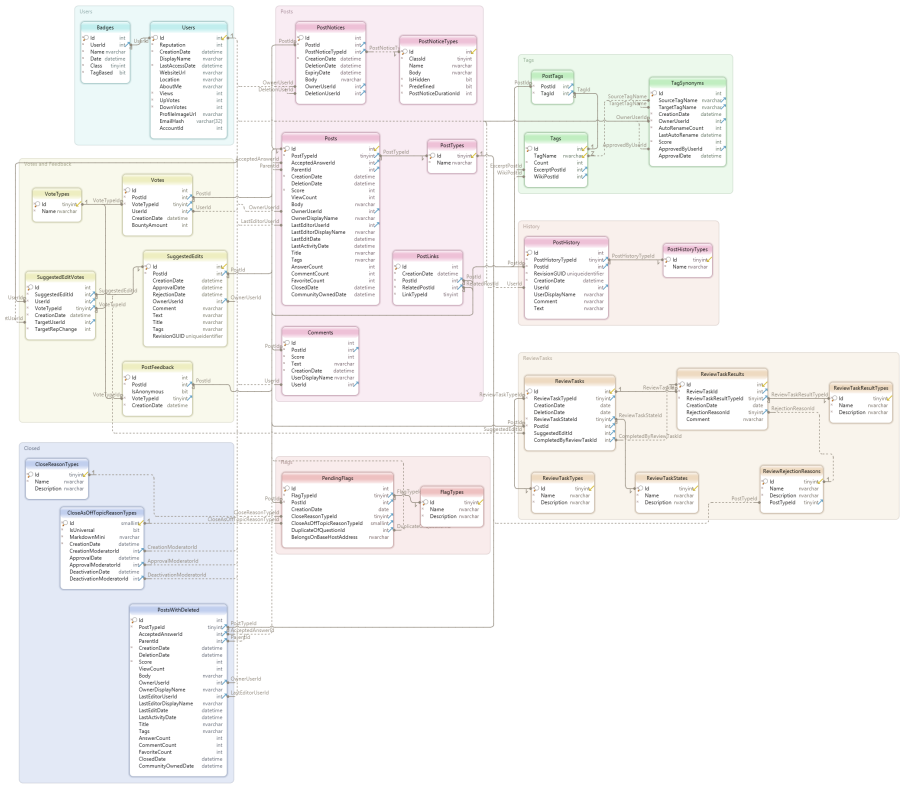


Figure 18: The logical model of our benchmark

The database schema is illustrated in Figure 18; however, for this benchmark, only the fact tables Posts, Users, and Comments were utilized, as these are the only tables

containing full-text attributes. Although the official webpage offers different database scale factors, we decided to perform our own scaling, since the ones offered by them were too big for our machines and purposes.

In addition, we have written 18 queries for this benchmark and translated them into the syntax used by the different tools. It is important to note that some of these queries were not compatible with SQL Server, as they involved advanced full-text search techniques that could not be implemented using relational technology. All these queries were run in every technology and timed to compare the results.

Now we will proceed to explain the different setups utilized in the different tools.

SQL Server We utilized SQL Server as a reference of how a relational model would address this type of problems. The setup was not especially complex, we downloaded the technology, loaded the database, and used the default setup.

Sphinx: Setting up Sphinx was significantly more challenging than configuring SQL Server. We began by downloading and loading the database into MySQL, as Sphinx tends to perform better when loading the data from this database system. Next, we wrote a configuration file that outlined the various indexes Sphinx should create and specified the data sources. Finally, we completed the setup by creating joined indexes, which enabled us to test queries that involved joins.

Open Search: To implement the benchmark in OpenSearch, the first step was to set up the server. This was achieved by creating a Docker image to run the database system. The benchmark was designed to run locally. For that reason the scale factors used were generated: 0.5x and 0.3x and 1x. Similar to the Sphinx case, for the join tables had to be created before executing the queries.

Elastic Search: The setup for Elastic Search will be explained carefully in the following section due to the amount of detail that we think that is important to explain.

Furthermore, we scaled the dataset down to perform comparisons between the performance of the tools with different workloads. The different scale factors used in the benchmarks are 1, 0.5 and 0.33. Of course to perform this comparations we did not run the queries involving joins, since the inorganic scale down performed by us make the join operations lose their true meaning.

To conclude, we performed two different benchmarks with this setup. First, we started by running all the queries in all the tools once. This resulted in a tremendous time gap between search engines and relational databases. This gap make us think that

to perform a more rigorous analysis and comparison between the search engines we should run every query multiple times. For time reasons and because the performance difference was already obvious we left SQL Server out of this second experiment. The results of these benchmarks will be discussed in the following sections.

7.2 Elasticsearch setup

7.2.1 Cluster

Our cluster is made up of 3 nodes, which are all both master and data, we've done so by using a docker-compose, which initialize them under the same network, and assigns each of them a volume, so that they are persistent.

Then we've defined for each index we will create in Elasticsearch, to have 5 shards and 0 replica. In case one wants to add more resilience to its application, he could easily add more replicas and Elasticsearch will automatically distribute them among the cluster's nodes.

One example is given by figure 19, where the posts index is divided into 6 shards, and Elasticsearch's cluster, divided the data so that every node, contains approximately the same amount in bytes.

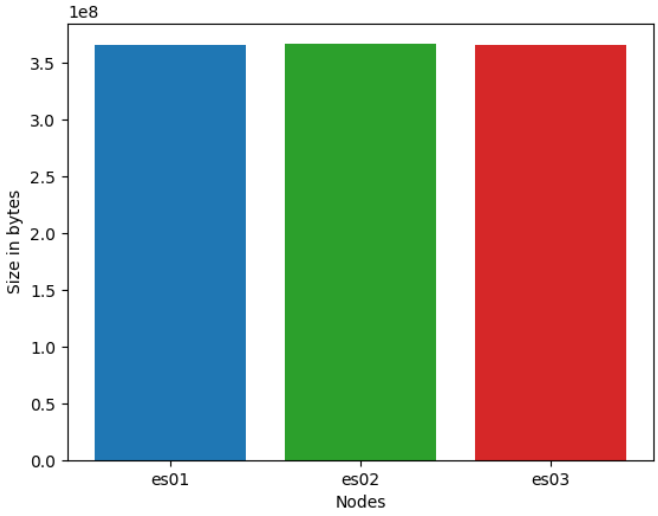


Figure 19: Elasticsearch load balancing

We did all of that just to show how the distributed architecture is deeply in the roots of Elastic, and how easy it is to scale. In case the nodes become too heavy or more replica shards needs to be inserted, one can instantiate a new node in a new or in the same machine, able to use other resources, that will allow the cluster to keep up with the new demands.

7.2.2 Analyzer

The analyzer as described in chapter 3 is what takes the input text, decides how to split it into tokens and what tokens needs to be filtered out.

```
"analysis": {
    "analyzer": {
        "custom_text_analyzer": {
            "type": "standard",
            "stopwords": "_english_"
        },
        "html_analyzer": {
            "type": "custom",
            "tokenizer": "standard",
            "char_filter": ["html_strip"]
        }
    }
}
```

Figure 20: Analyzers

In figure 20 we can see that we've defined two analyzers, one for the normal text and the other for the body of the posts.

Those are not indeed normal text but are rather stored as html contents which contains HTML tags like , , <href> etc... which we clearly don't want to index, as they are not the part of the text that we will query for.

To create this analyzers we've followed the documentation available at this page that clearly explains what every options does and what are other possible options. [6]

The "custom_text_analyzer" is so defined:

- **type**: standard, meaning that it will separate the text into tokens using the word boundaries.
- **stopwords**: this will filter out the stopwords, using as stopwords those defined in the English language (e.g. the, of, from...).

The "html_analyzer" is instead defined like:

- **type**: custom allows to create a custom analyzer, not using hence, the predefined ones.
- **tokenizer**: as a tokenizer we are using the standard one, which will separate words using the words boundaries.
- **char_filter**: passing html_strip, will filter out all the html tags and elements, which will then not be indexed, achieving the goal we wanted.

7.3 Our Benchmark with Relational

In this benchmark, each query was executed only once, with the primary objective of comparing search engines to relational database systems. The results, including execution times for each query across various database scale factors, are displayed in the following graphs. Let us begin by examining the results for scale factor 1.

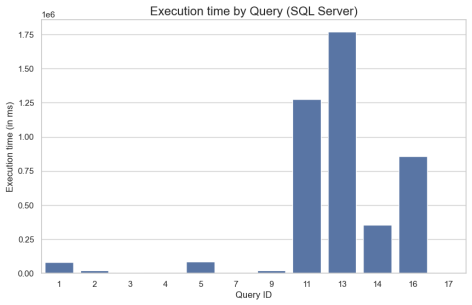


Figure 21: SQL Server

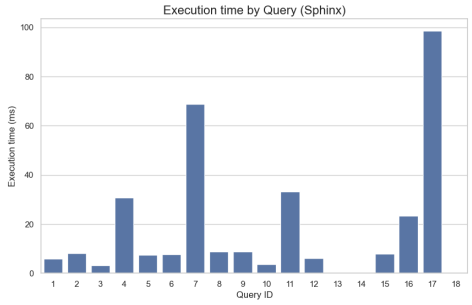


Figure 22: Sphinx

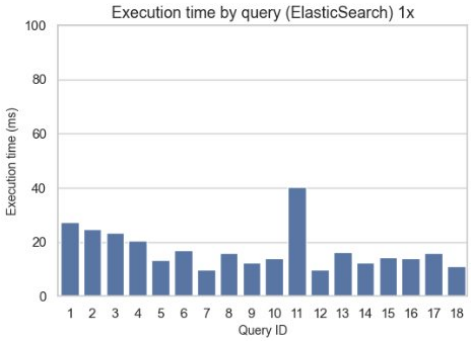


Figure 23: Elastic Search

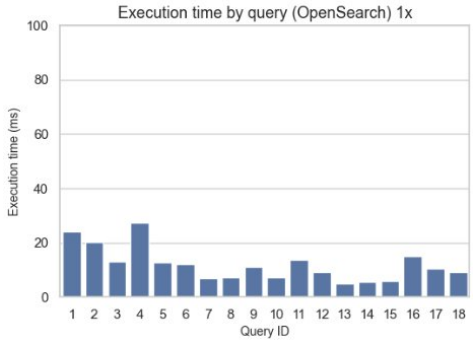


Figure 24: Open Search

Observe the y-axis of the graphs closely. The performance difference between the search engines and the relational database system is substantial. SQL Server took 74 minutes and 34 seconds to execute all the queries, whereas Sphinx required only 0.3 seconds. Similarly, for OpenSearch and Elasticsearch, execution times ranged from 10 to 40 milliseconds for the same queries. This outcome aligns with expectations: systems specifically optimized for such operations demonstrated significantly superior performance. Next, we will review the results for scale factor 0.5.

Figure 25: SQL Server

Figure 26: Sphinx

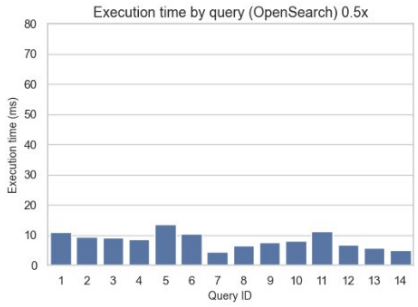


Figure 27: Elastic Search

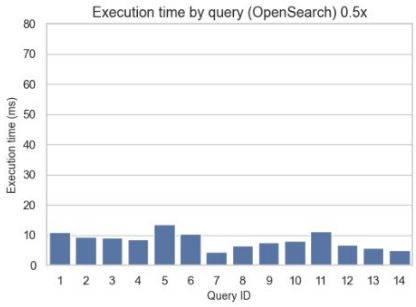


Figure 28: Open Search

With a scale factor of 0.5, SQL Server executed all queries in 32 minutes and 26 seconds. The performance disparity observed in the previous scale factor persisted here. Sphinx exhibited remarkable efficiency, completing all queries in less than 20 milliseconds, except for Query 7, which took nearly 80 milliseconds. For Elasticsearch and OpenSearch, execution times ranged from 8 to 13 milliseconds. Notably, we observed a reduction in execution time compared to the 1x scale factor, which was expected. Finally, let us analyze the results for scale factor 0.33.

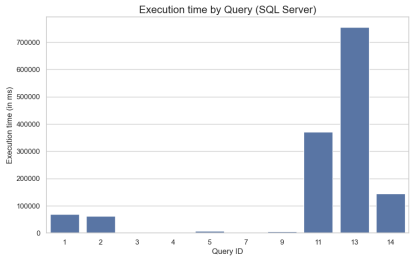


Figure 29: SQL Server

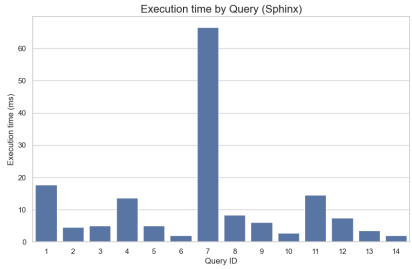


Figure 30: Sphinx

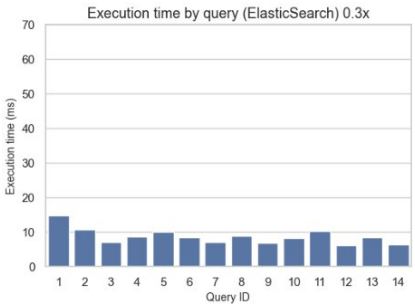


Figure 31: Elastic Search

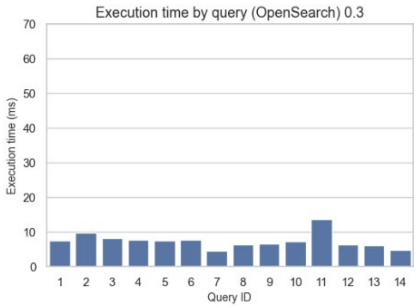


Figure 32: Open Search

At the smallest scale factor, 0.33, a noticeable reduction in execution times for SQL Server was observed for some queries. However, the performance remained significantly slower than that of the search engines. On the other hand, all queries in the search engines demonstrated further reductions in execution time at this smaller scale factor, highlighting the efficiency of these systems in handling such workloads.

The graphs vividly illustrate the significant time differences between SQL Server and Sphinx. Additionally, SQL Server was unable to execute all the queries, as some were not applicable to relational databases. While query syntax is not the primary focus of this benchmark, it is worth noting how specific queries translate between systems. For example, Sphinx's syntax is quite similar to standard SQL, making it interesting to compare its queries with their SQL Server counterparts.

```
--Search for posts that have any 2 languages
--of programmation and order them by score

 SELECT id, Score
 FROM comments
 WHERE
 MATCH('"sql␣python␣c++␣r␣ruby␣php␣java␣javascript"/2')
 ORDER BY Score DESC
```

```
--Search for posts that have any 2 languages
--of programmation and order them by score

 SELECT TOP 20 id, Score
 FROM comments
 WHERE
 (text LIKE '%sql%' AND text LIKE '%python%') OR
 (text LIKE '%sql%' AND text LIKE '%c++%') OR
 (text LIKE '%sql%' AND text LIKE '%r%') OR
 (text LIKE '%sql%' AND text LIKE '%ruby%') OR
 (text LIKE '%sql%' AND text LIKE '%php%') OR
 (text LIKE '%sql%' AND text LIKE '%java%') OR
 (text LIKE '%sql%' AND text LIKE '%javascript%') OR
 (text LIKE '%python%' AND text LIKE '%c++%') OR
 (text LIKE '%python%' AND text LIKE '%r%') OR
 (text LIKE '%python%' AND text LIKE '%ruby%') OR
 (text LIKE '%python%' AND text LIKE '%php%') OR
 (text LIKE '%python%' AND text LIKE '%java%') OR
 (text LIKE '%python%' AND text LIKE '%javascript%') OR
 (text LIKE '%c++%' AND text LIKE '%r%') OR
 (text LIKE '%c++%' AND text LIKE '%ruby%') OR
 (text LIKE '%c++%' AND text LIKE '%php%') OR
 (text LIKE '%c++%' AND text LIKE '%java%') OR
 (text LIKE '%c++%' AND text LIKE '%javascript%') OR
 (text LIKE '%r%' AND text LIKE '%ruby%') OR
 (text LIKE '%r%' AND text LIKE '%php%') OR
 (text LIKE '%r%' AND text LIKE '%java%') OR
 (text LIKE '%r%' AND text LIKE '%javascript%') OR
 (text LIKE '%ruby%' AND text LIKE '%php%') OR
 (text LIKE '%ruby%' AND text LIKE '%java%') OR
 (text LIKE '%ruby%' AND text LIKE '%javascript%') OR
 (text LIKE '%php%' AND text LIKE '%java%') OR
 (text LIKE '%php%' AND text LIKE '%javascript%') OR
 (text LIKE '%java%' AND text LIKE '%javascript%')
  ORDER BY Score DESC;
```

We can see a huge difference. This is not the only query with this syntax problem, however since they can be found in our repo, we don't think that it adds value to add more examples here.

7.4 Our Benchmark

In this instance, we ran the 18 queries again 50 times, actually 51 but we disregarded the first run for cache generalization, now only in the three search engines. We will present the results in a manner similar to the previous example. As you will notice the last four queries, 15-18, were only used for the scale factor 1, for the reasons that we have already explained previously.

Let's start by analyzing the results of Sphinx in this second benchmark.

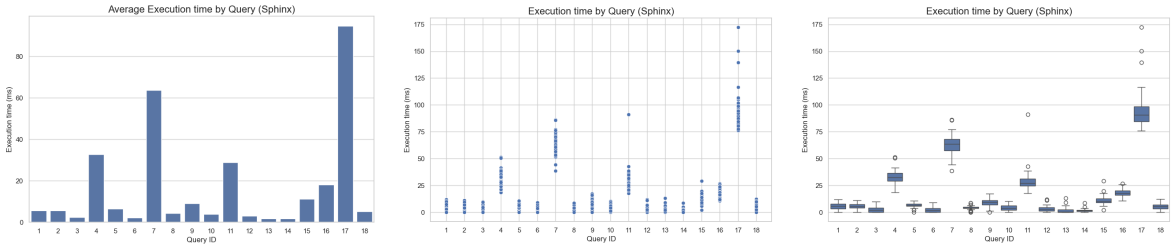


Figure 33: Sphinx sf 1

The first graph, which displays the averages, closely resembles the one from the initial experiment. This comparison allows us to deduce that although the first time a query is run may take longer, the variation in execution time is relatively minimal. We can further analyze these time variations by examining both the second and third plots. They visually illustrate how execution times fluctuate between iterations. It is evident that queries that take longer to run tend to exhibit greater variations in their execution times. However, variariation in general is quite small. Let's continue the analysis taking a look at the scale factor 0.5 results:

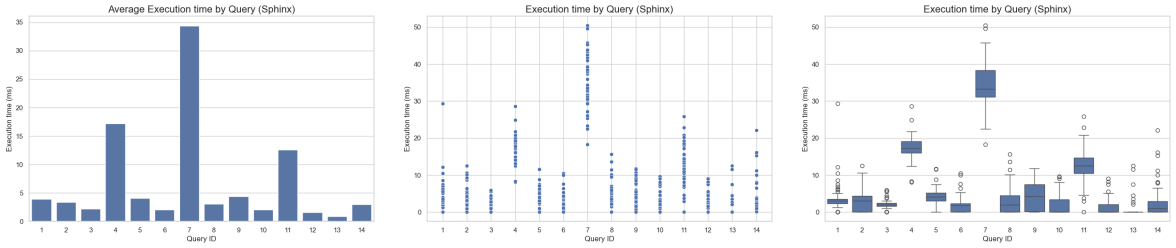


Figure 34: Sphinx sf 0.5

We can see how the times are considerably smaller in this second run. We can also see how the time pattern is the same (taking into account that the join queries are not being run). Also looking at the y-axis we can see how the times have decreased almost by half, so maybe some kind of linearity could be deduced from the relation data amount execution time for Sphinx. Let's finish analyzing Sphinx by discussing the scale factor 0.33 results:

Figure 35: Sphinx sf 0.33

As we can see the linearity hypothesis that we presented in the last paragraph still holds for this third scale factor. It is really interesting to also highlight how the time variance also seems to scale down with the scaling factor.

For ElasticSearch, similarly to Sphinx, we plotted the averages and fluctuations between iterations for each query. We equally see some linearity in the performance, as the average execution time shortens with each smaller scale factor.

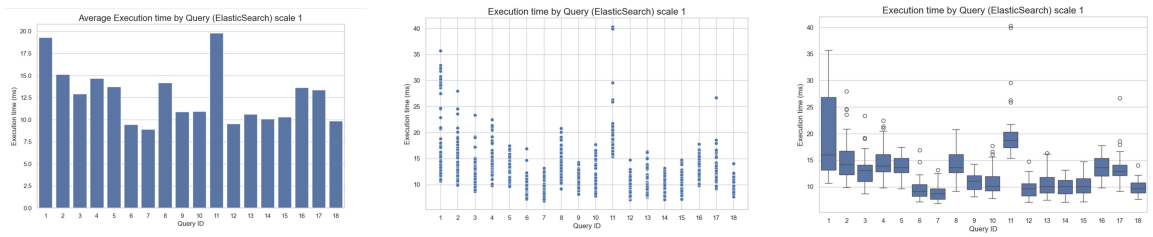


Figure 36: ElasticSearch sf 1

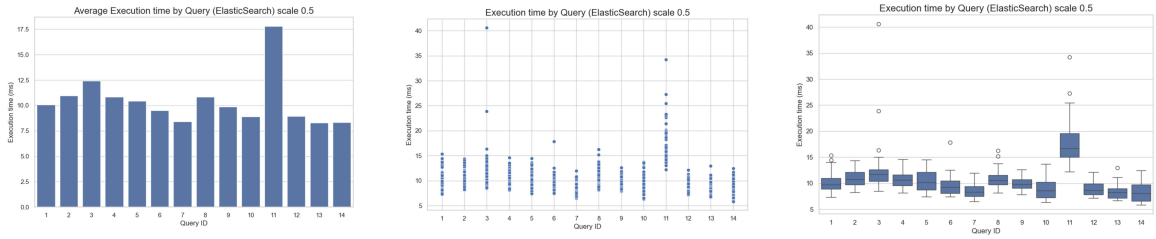


Figure 37: ElasticSearch sf 0.5

Figure 38: ElasticSearch sf 0.33

For OpenSearch, we also plotted the averages and fluctuations between iterations for each queries. Linerarity also seems to hold for these results, and we can see that they are highly similar to those of ElasticSearch.

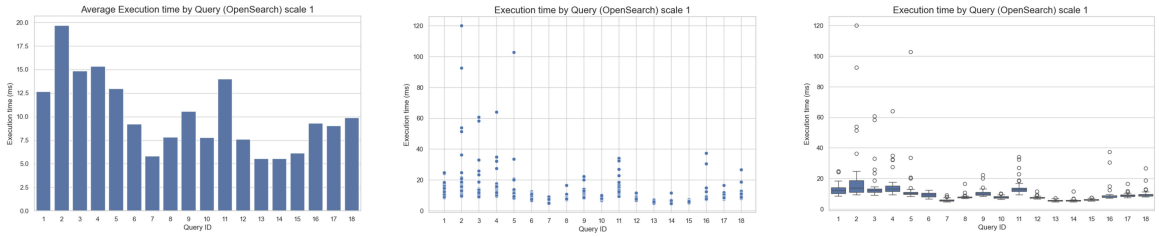


Figure 39: OpenSearch sf 1

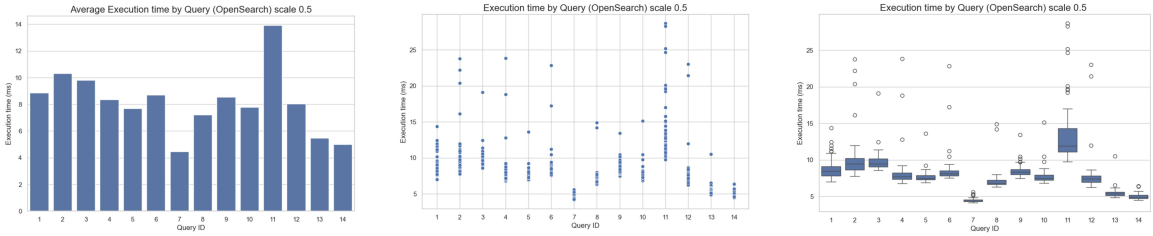


Figure 40: OpenSearch sf 0.5

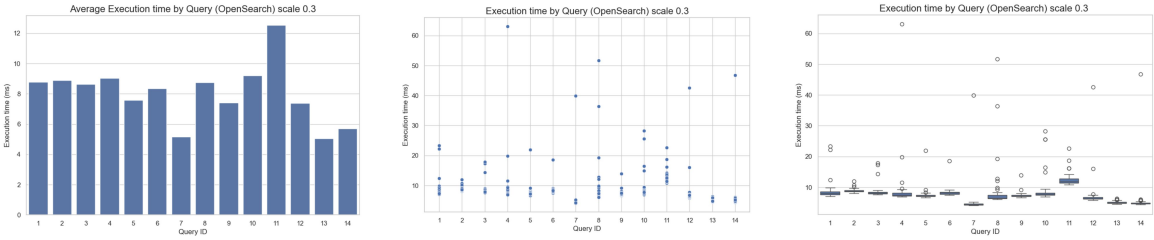


Figure 41: OpenSearch2 sf 0.33

7.5 Semantic Search with Elasticsearch

For the semantic search we've enhanced our posts' index to support semantic matching. To do so, we've decided to focus on the title of the posts, and we've done so by creating an index that both encodes the title as text and as vectors for semantic search. Unfortunately, creating an embedding and storing it for every document is expensive, so for this demo, we've decided to limit our dataset to 50% of the total number of tuples.

```
mappings={
    "properties": {
        "Body": {
            "type": "text",
            "analyzer": "html_analyzer"
        },
        "Title": {
            "type": "text"
        },
        "Semantic_Title": {
            "type": "dense_vector",
            "dims": 384,
            "index": "true",
            "similarity": "cosine"
        },
        "CommentCount": {
            "type": "integer"
        },
        "CreationDate": {
            "type": "date",
            "format": "yyyy-MM-dd HH:mm:ss.SSS"
        }
    }
}
```

Figure 42: Semantic search mappings

As we can see from 42 we have to introduce a new field for our post document.

We've called it Semantic_Title. This is a dense vector of size 384 which is due to the model we've used, "all-MiniLM-L6-v2" whose description can be seen in figure 43.

We also need to set index to true, otherwise, you won't be able to search using this field, and lastly, we have to define what is the distance function we will use to calculate similarity. We've decided to use the cosine distance.

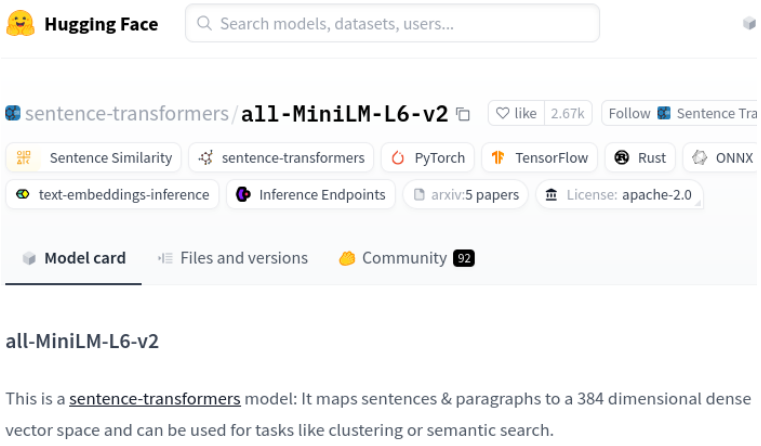


Figure 43: Hugging face model[15]

This model is trained on a generic english corpus, to have better results, it may be needed to use a model trained on a more technical language, more computer science oriented, as Stack Overflow posts usually contains such language.

Next, to assess the capabilities of semantic search we've run the same query using first the syntactic matching, using the normal Title of our posts and then we've used the same query but using the embedded semantic title field of our index.

```python
query = "Python book"

synt_resp = es.search(
    index="semantic_html_posts",
    size=10,
    query={
        "match": {
            "Title": {
                "query": query
            }
        }
    }
)

sem_resp = es.search(
    index="semantic_html_posts",
    knn={
        "field": "Semantic_Title",
        "query_vector": model.encode(query),
        "k": 10,
        "num_candidates": 10000,
    }
)
```

Figure 44: Syntactic vs Semantic search

To do so we have defined a KNN query matching, that will basically finds the k

nearest neighbors to our input and return them as a result. We also need to define the number of candidates to compare. Basically in this way, every shard will compare their top num_candidates with the query, meaning that increasing this, will increase the search space, leading to higher latency but hopefully better results.

Clearly, to run the query against our embedded title, we need to encode our query, using the same model we've used to encode our Titles in the index (Figure 44)

	syntactic_hits	semantic_hits
0	What is the best quick-read Python book out there?	What is the best quick-read Python book out there?
1	Microsoft CryptoAPI Book	Which Python book would you recommend for a Linux Sysadmin?
2	Online Java Book	Learning Python
3	Which Python book would you recommend for a Linux Sysadmin?	Python, beyond the basics
4	Book Store Database Design	Blender- python
5	boolean algebra article/book	Resources for Python Programmer
6	Advanced LaTeX Tutorial/Book	Python Version for a Newbie
7	Test driven development book	Introducing Python
8	Recommendations for IIS book	Python for mathematics students?
9	Address book DB schema	Python code that needs some overview

Figure 45: Syntactic vs Semantic search results

The results are quite impressive as we can see from Figure 45, where we've obtained the top 10 ranked results of both methods. We can notice, how syntactic search gave us uninteresting results like "Online Java Book" or "Microsoft CryptoAPI Block", while semantic search was able to give us very interesting matches with the query such as "Resources for Python Programmer", "Learning Python" etc.

8 State of the art and interesting applications

In this section of the project, we will provide a brief summary of several papers related to search engines that we found interesting.

8.1 Optimizing Retrieval-Augmented Generation with Search Engines for Enhanced Question-Answering Systems

Hypothesis In today's information era, extracting valuable insights from large datasets is crucial. Large Language Models (LLMs) excel at text understanding and generation but often struggle with up-to-date or specific knowledge. To address this challenge, researchers have introduced Retrieval-Augmented Generation (RAG), which combines external document retrieval with LLMs to enhance response quality and accuracy. [18]

RAG operates by first retrieving relevant documents from a large collection and then using them as context for the LLM, allowing it to generate more precise answers. The traditional RAG framework relies on keyword matching or semantic similarity for document retrieval but often encounters difficulties with complex queries, recall rates, and diverse results.

To tackle these issues, this article proposes integrating a search engine with advanced configuration options, specifically Elasticsearch, as the core component for RAG. This integration offers several improvements:

1. **Improved Retrieval Speed and Efficiency:** Search engines utilize an inverted index structure, which allows them to handle large volumes of data while maintaining high response times.

2. **Enhanced Relevance and Diversity of Search Results:** Search engines facilitate advanced word segmentation and adaptive weight distribution based on user needs, leading to more relevant and varied results.

3. **Scalability:** Elasticsearch's architecture enables easy addition of nodes to accommodate business growth and includes monitoring tools for real-time cluster management. This adaptability makes the RAG solution based on Elasticsearch suitable for both current and future enterprise needs.

Experiment To test the previously presented thesis, this paper introduces a conducted experiment. The experiment was carried out using the Stanford Question Answering Dataset (SQuAD), a well-known English dataset developed by Stanford University's Natural Language Processing Group. This dataset includes numerous paragraphs from Wikipedia along with corresponding questions and answers, which evaluate the model's ability to comprehend text. SQuAD version 2.0 contains over

100,000 question-answer pairs and introduces unanswerable questions to assess the model's capability to recognize when no answer is available.

The experiment involved analyzing 536 articles, and the dataset was preprocessed to improve the search engine's performance. This preprocessing included removing HTML tags, standardizing capitalization, and eliminating stop words to enhance data quality. Additionally, Elasticsearch was configured with appropriate word segmenters and mapping rules to optimize search performance. To evaluate the method, a benchmark was created that encompassed common knowledge questions, terminology explanations, and long-tail information searches. In this benchmark, Elasticsearch was compared against other architectures: traditional RAG, BM25 scoring function, and TF-IDF vector representation.
The results of the benchmark were the following:

Model	Acc	F1	Recall
RAG	65.51	66.11	65.72
BM25-RAG	66.32	66.56	66.47
TF-IDF-RAG	67.78	67.63	67.23
ES-RAG	68.29	68.42	68.13

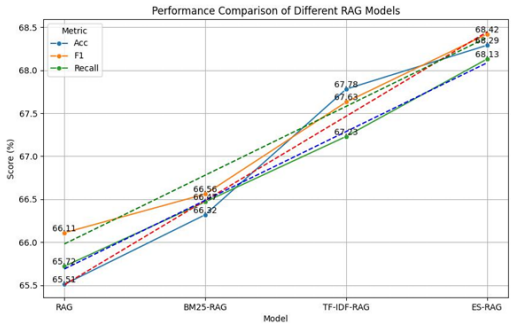


Figure 46: Results of the experiment

This result show that integrating Elasticsearch into the RAG framework significantly improves the overall performance of the question-answering system.

8.2 Local Geographic Information Storing and Querying using search engines

The Ottawa Resource Collection at Carleton University's Archives holds approximately 3,000 print items and maps related to the history of Ottawa. Most of the documents are in English, and local researchers, including historians and architects, often seek information about properties, communities, or buildings within larger documents. [16]

Although staff members have made note of some locations, this approach is not scalable given the size of the Collection. Current digitization efforts are exploring the use of Geographic Information Retrieval (GIR) to extract relevant information. While creating a full-text searchable repository could assist researchers in finding location keywords, it would not provide geographic context for neighborhoods unless this information is explicitly mentioned. The project requirements include minimizing hardware and software costs, enabling library staff to perform quality control with easily readable output, and focusing specifically on locations within the Ottawa area. Previous attempts to use GIR methods to access historical documents have not been successful.

The solution found for task was implementing Elastic Search as the main engine. Let's step by step explain in detail the architecture and processes performed.

1. **Digitalization and conversion:** The first step in this process involved scanning all documents and converting them into PDF files. Each PDF was named based on the title and publication year of the document. Additionally, a spreadsheet was created that included the title, author, publication details, call number, and description of each document. The PDFs contained embedded OCR text, which was used to convert every page of each PDF into a TXT file.

2. **Elastic search setup:** To prepare TXT files for Elasticsearch, a JSON metadata file was created for each TXT file based on the previously mentioned CSV. The records were exported as individual JSON files, and a Node.js script was used to match these files with their corresponding TXT files. Finally, an Elasticsearch index was created, and the data was bulk uploaded into it.

3. **Lexicons:** To search for geographic locations in TXT files, lexicons of Ottawa locations were created using geospatial datasets. These datasets included the necessary spatial information, such as latitude and longitude coordinates. Relevant data was processed, such as neighborhoods and landmarks, using QGIS to ensure compliance with the WGS84 datum. After exporting the processed data as CSV files, into was converted into JSON format and loaded into Elasticsearch. The lexicon index was mapped manually to code the location field as a geo_point before it was uploaded. This index is used to query locations within the Collection documents and provide corresponding geospatial coordinates.

Preliminary searching in Elasticsearch was developed. The process starts by retrieving all lexicon items from the lexicon index, including neighborhood names, aliases, and coordinates. After obtaining this data, a full-text phrase match query is conducted for each location name in the TXT files of the documents. If the initial match fails, a second query is performed using available aliases for that neighborhood.

Search results are returned in JSON format and stored in a new Elasticsearch results index. The original document title, page number, and geo_point coordinates are retrieved from item metadata. Elasticsearch also offers a highlight field to show matching and surrounding text for disambiguation. Work is ongoing to create human-readable query output formats.

9 Conclusion

The goal of this project was to compare three major search engines—Elasticsearch, Sphinx, and OpenSearch—to understand their functionalities, architectures, and performance. We began by exploring the core architecture of these search engines, detailing their differences in architecture, features, advantages and disadvantages.
We discussed Elasticsearch as a distributed search engine built on Apache Lucene, with features like semantic search and advanced caching. Sphinx was presented as a search engine optimized for query performance and search relevance, with its own SQL-like query language, SphinxQL. OpenSearch, a fork of Elasticsearch, shared a similar architecture while being an open-source project.
The project also included 3 different benchmarking tests. The first one, an official benchmark for OpenSearch, *geoname*, was used to compare OpenSearch and ElasticSearch. The conclusion was that ElasticSearch is faster for latency, meaning that it performs faster in search or aggregation queries. But the benchmark also seemed optimized for built-in OpenSearch tools, and we think it is possible that this benchmark dedicated to OpenSearch could be optimized for it.

The second benchmark compared OpenSearch, Elasticsearch, Sphinx, and SQL Server. Due to a lack of pre-existing compatible benchmarks for all of our three tools, we created a benchmark with 18 search engine queries, and ran each query only once for each tool (because running queries in SQL server took a very long time).
The results showed that SQL Server performs poorly for search engine queries, which makes sense. On the other hand, queries that require JOIN are optimized for SQL server, and not for our 3 search engines.
For our third benchmark, we ran the queries 50 times for each of the 3 search engines. The results showed that Sphinx is faster for some queries but that ElasticSearch and OpenSearch's performances are more constant accross all queries. We also hypothesized some linearity in the performance of the queries with respect to the scale factor of data.
This project offers a comprehensive comparison, showcasing each search engine's strengths and potential for different use cases.

References

[1] Andrew Aksyonoff. "Sphinx documentation v 3.0.1". In: (2017). URL: https://sphinxsearch.com/docs/sphinx3.html#sphinx-3.

[2] Andrew Hopp. "A query, or There and Back Again". In: (2021). URL: https://opensearch.org/blog/a-query-or-there-and-back-again/.

[3] Carl Meadows. "Introducing OpenSearch". In: (2021). URL: https://aws.amazon.com/blogs/opensource/introducing-opensearch/.

[4] Clicking. "ElasticSearch, Sphinx, Lucene, Solr, Xapian. Which fits for which usage? - Stack Overflow". In: (2024). URL: https://stackoverflow.com/questions/2271600/elasticsearch-sphinx-lucene-solr-xapian-which-fits-for-which-usage.

[5] Coralogix. "Elasticsearch Architecture: 8 Key Components and Putting Them to Work - Coralogix". In: (2024). URL: https://coralogix.com/guides/elasticsearch/elasticsearch-architecture-8-key-components-and-putting-them-to-work/.

[6] Developers. "Create a custom analyzer — Elasticsearch Guide [8.17] — Elastic". In: (2024). URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-custom-analyzer.html.

[7] Developers. "Elasticsearch caching deep dive: Boosting query speed one cache at a time — Elastic Blog". In: (2024). URL: https://www.elastic.co/blog/elasticsearch-caching-deep-dive-boosting-query-speed-one-cache-at-a-time.

[8] Developers. "Elasticsearch: The Official Distributed Search Analytics Engine — Elastic". In: (2024). URL: https://www.elastic.co/elasticsearch.

[9] Developers. "Nodes — Elasticsearch Guide [8.17] — Elastic". In: (2024). URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html.

[10] Elastic. "The BM25 model". In: (2024). URL: https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables.

[11] Considering Various Factors. "What is semantic search, and how does it work? — Google Cloud". In: (2024). URL: https://cloud.google.com/discover/what-is-semantic-search?hl=en.

[12] Jeevan George John. "(23) Elasticsearch: Understanding the basic architecture — LinkedIn". In: (2024). URL: https://www.linkedin.com/pulse/elasticsearch-understanding-basic-architecture-jeevan-george-john/.

[13] Ugo Sangiorgi. "Elasticsearch vs. OpenSearch: Vector Search Performance Comparison". In: (2024). URL: https://www.elastic.co/search-labs/blog/elasticsearch-opensearch-vector-search-performance-comparison.

[14] Vivien Tran-Thien. "Semantic Search: An Overlooked NLP Superpower". In: (2024). URL: https://blog.dataiku.com/semantic-search-an-overlooked-nlp-superpower.

[15] Hugging Face. We. "sentence-transformers/all-MiniLM-L6-v2 · Hugging Face". In: (2024). URL: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2.

[16] Rebecca Bartlett. "Local Geographic Information Storing and Querying using Elasticsearch". In: (). URL: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://dl.acm.org/doi/pdf/10.1145/3371140.3371144.

[17] Canonical.com. "What is OpenSearch?" In: (). URL: https://canonical.com/data/opensearch/what-is-opensearch.

[18] Jiajing Chen. "Optimizing Retrieval-Augmented Generation with Elasticsearch for Enhanced Question-Answering Systems". In: (). URL: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://arxiv.org/pdf/2410.14167.

[19] OpenSearch.org. "OpenSearch documentation". In: (). URL: https://opensearch.org/docs/latest/about/.

[20] Wikipedia. "Elasticsearch". In: (). URL: https://en.wikipedia.org/wiki/Elasticsearch#:~:text=Due%20to%20potential%20trademark%20issues,and%20open%2Dsource%20once%20again..