CS 582: Phase 2
Report {Team 2: Data Force}

Anamika Rajendran
NMSU, USA

Xindi Zheng
NMSU, USA

Jacob Rydecki
NMSU, USA

Abstract

Our research report investigates the comparative performance of Elasticsearch and MongoDB, focusing on their capabilities as document-oriented databases. One might ask, "Which database should I use?" There's no straightforward answer to that question, which is why we're conducting a research study to understand each database's capabilities. Through a series of experiments, we assess key performance metrics, including query execution time, indexing efficiency, and overall resource consumption, to provide a clearer view of their strengths and optimal use cases. Our methodology involves various test cases that simulate common data retrieval and manipulation tasks encountered in real-world applications. The findings highlight distinct performance characteristics of each database, revealing scenarios where one outperforms the other. This research aims to deepen understanding of these database systems, guiding developers in making informed choices based on the unique requirements of their applications.

1. Introduction

Elasticsearch is primarily recognized as a search engine, and has evolved into a powerful and versatile data store. Built on top of Lucene, Elasticsearch stores data in a *JSON-based document store*. It has become a popular choice for applications that require fast and efficient full-text search capabilities. Elasticsearch's strengths lie in its distributed nature and real-time search capabilities. It is highly suitable for scenarios where quick access to large datasets is crucial, such as log analytics, monitoring, and recommendation systems. The ability to scale horizontally makes Elasticsearch a robust solution for handling increasing data loads.

Elasticsearch offers several advantages, particularly in handling large volumes of data and performing full-text searches with high speed and accuracy. Built on Apache Lucene, it's optimized for indexing and searching text, making it ideal for applications that require rapid retrieval of information from vast datasets. Elasticsearch uses an ***inverted index structure***, enabling it to return search results almost

instantly, even for complex queries. Its scalability is another strength: Elasticsearch can seamlessly distribute and balance data across multiple nodes, allowing it to handle high loads and ensure fault tolerance. Additionally, Elasticsearch supports complex queries and aggregations, allowing users to analyze large datasets with minimal latency. Its RESTful API and JSON-based responses further enhance its integration capabilities with various programming languages and platforms, making it highly adaptable in data-intensive environments.

1.1 Choice of Database for Comparison

When considering comparisons to Elasticsearch, several database types provide unique points of contrast. SQL databases like MySQL, and PostgreSQL are strong candidates; they're widely used for structured data storage, complex querying, and transactional applications. These relational databases are ideal for managing data with well-defined relationships, often performing well in e-commerce, banking, and enterprise applications. However, they're less suited to handling unstructured data or performing rapid, full-text searches—areas where Elasticsearch excels due to its inverted index and schema flexibility.

Another possibility is comparing Elasticsearch to vector databases, such as Pinecone, Weaviate, or Milvus, which are optimized for storing and querying high-dimensional vector embeddings. Vector databases excel in use cases involving AI and machine learning, including semantic search, image recognition, recommendation engines, and natural language processing. These databases support approximate nearest neighbor (ANN) search, making them ideal for applications requiring similarity matching, but they differ significantly from traditional document storage and search systems.

Apart from various database types we could consider, such as SQL databases, or specialized vector databases, a NoSQL option made more sense for our comparison with Elasticsearch. Choosing another document-oriented NoSQL database aligns well with Elasticsearch's flexible structure and unstructured data handling capabilities. ***MongoDB***, in particular, shares a similar document model but is typically used for different applications—prioritizing operational data storage rather than the rapid full-text search that defines Elasticsearch. Being NoSQL and also document-oriented, MongoDB offers a comparable yet distinct approach, making it a natural choice for this study. It's akin to "*comparing a fuji apple with a granny apple"*, allowing us to explore both the parallels and contrasts in how these two systems handle scalability, indexing, and query performance.

2. Experimental Setup

Our experimental setup includes detailed hardware and software specifications to ensure consistency and reliability in our performance tests. We used Elasticsearch 7.17.25 alongside Kibana 7.17.25, as well as the latest version of MongoDB. Python was employed to run test scripts, along with a custom script to generate additional data, ensuring a dataset substantial enough for rigorous text search evaluation.

Note: Kibana is a visualization tool that integrates with Elasticsearch, enabling users to create dynamic visualizations and dashboards for data analysis. It allows for real-time monitoring, intuitive search and filtering of data, and supports time-series analysis, making it essential for uncovering insights and tracking key performance indicators. Its user-friendly interface empowers both technical and non-technical users to explore and interpret large datasets effectively.

To observe any potential effects of operating systems on performance, we conducted tests on a virtual environment. One researcher operated a Linux virtual machine with the following specifications: Ubuntu 23.10, 4-cores, and 24GB of RAM. Notably, all tests were performed in isolated environments, with no background tasks running to ensure maximum CPU core availability and accurate measurement of database performance metrics. This configuration helped us capture the most precise performance data possible, focusing solely on the databases under test.

Also, to avoid data fluctuations due to caching, we restarted the applications for each run. This ensured that the data was not being retrieved from memory (cold cache), allowing us to accurately measure performance without any interference from previously cached data.

2.1 Dataset Selection

To ensure a relevant comparison between Elasticsearch and MongoDB, we selected datasets with practical use cases for each database. Elasticsearch is widely used by organizations such as ***Netflix*** and ***Quora*** for its powerful text search capabilities and scalability, and MongoDB is frequently applied in similar high-demand data retrieval contexts. With this in mind, we identified two datasets that align well with the strengths of both databases: a text-based dictionary dataset (`**dictionary.txt**`) and a **Spotify dataset** containing the top 200 songs (Refer Appendix B).

The dictionary dataset, provided in `.txt` format, represents a large volume of searchable text entries, making it ideal for evaluating text indexing and retrieval in both databases. To increase its size and create a more challenging search environment, we used a Python script to generate additional data and convert the expanded dataset to `.json` format for consistent testing in both systems. The advantage to using such a generated dataset is we get to create large documents for testing and comparison. We generated a 900MB file with 10,000 documents, with each document containing (on average) 2,500 random 5-word sentences.

The Spotify dataset, a 3GB file in `.csv` format, contains data on the top 200 songs. 9 columns containing various data types such as song name, date, region, URL, and numeric rankings, was particularly suited for a wide range of querying. This dataset's significant size and structured format allowed us to test performance under high-load conditions. We used a Python script to convert the `.csv` file into `.json` format for compatibility with MongoDB and Elasticsearch. By using two distinct datasets—a text-heavy dictionary and a large, structured music database—we ensured a robust comparison that highlights the performance, indexing, and retrieval capabilities of each database across varied data structures. To assess performance under different data loads, we divided our experiment into two phases. Details of each experiment phase, including setup, query types, and performance metrics, are provided in the next section.

Experiment Phase	Description
Experiment 1	Testing with a small dataset (expanded dictionary dataset in JSON format) to simulate scenarios where the database handles more dense documents and evaluate rapid text search needs.
Experiment 2	Testing with a large dataset (Spotify Top 200, converted to JSON format) to examine performance metrics in high-volume data environments and evaluate database handling of larger, structured datasets.

Table 1: Experiment details: This table shows the different experiment phases and the respective dataset used in the experimental phase.

2.2 Criteria Selection

2.2.1 Experiment 1 - Criteria and Query Processing

We evaluated the search functionality on both Elasticsearch and MongoDB on a smaller dataset with much more text per document. In this dataset we focused on two types of queries: **full-text search**, and **phrase search**.

For both queries, we queried for two specific words from the body of the document to assess each database's handling of searching through vast amounts of each in each document. We believe this is a good test to assess the capability, scalability, and optimization of the inverted indices, as terms likely to appear in numerous documents challenge the system's retrieval efficiency.

```
{
  "queries": [
    {
      "query": {
        "match": {
          "body": "repulsive friction"
        }
      }
    },
    {
      "query": {
        "match_phrase": {
          "body": "repulsive friction"
        }
      }
    }
  ]
}
```

Figure 1:Full-Text Search (Top) and Phrase Search (Bottom) Queries in Elasticsearch

```
// Full-Text Search
db.music.find({
  $text: {
    $search: "repulsive friction"
  }
})

// Phrase Search
db.music.find({
  $regex: "repulsive friction",
  $options: "i"
})
```

Figure 2:Full-Text Search and Phrase Search Queries in MongoDB

For the full-text search query, each of the words is searched for within all the documents, regardless of their order or relation to each other within the document – they simply both have to be in a document for it to be returned. This

query specifically tests the efficiency of each database in finding and retrieving documents that contain multiple specific terms, irrespective of context or positioning. By requiring both terms to appear in each returned document, we assess how well each database optimizes search speed, indexing performance, and resource usage when handling high-frequency term combinations across a large dataset.

For the phrase search query, the phrase itself is searched for within all the documents, where both words need to be found next to each other and exactly as seen in the query for that document to be returned. This query specifically tests the efficiency of each database in handling exact phrase matching, assessing both the speed and accuracy with which the database can locate contiguous word sequences. By requiring an exact match, this test evaluates how well each database's indexing structure supports phrase-based searches, as well as its capability to optimize for precise, context-sensitive queries within large documents and datasets.

2.2.2 Experiment 2 - Criteria and Query Processing

We evaluated search functionality on both Elasticsearch and MongoDB using specific criteria within Experiment 2. We focused on three types of queries: **a text-based search**, a **non-text-based filter**, and **sorting by a non-unique attribute**.

For the text-based search, we queried the *artist's name* to assess each database's handling of text data and retrieval accuracy with partial or full matches. For the non-text-based filter, we chose the *album rank*, which allowed us to evaluate numerical filtering and comparison. Finally, we sorted the results by *rank*, a non-unique attribute, to identify duplicates within the dataset and compare each system's ability to handle ordering effectively. As discussed in class, ordering can reveal duplicates by clustering identical values together, making them more apparent.

```
GET /music/_search
{
  "query": {
    "term": {
      "artist.keyword": "Taylor Swift"
    }
  }
}
```

```
GET /music/_search
{
  "query": {
    "term": {
      "album_rank": 6
    }
  }
}
```

```
{
  "sort": [
    { "album_rank": "asc" }
  ],
  "size": 1000
}
```

Figure 3: Sample Elasticsearch queries for a large dataset, demonstrating a text-based search, a non-text-based filter, and sorting by a non-unique attribute.

By conducting these queries on both Elasticsearch and MongoDB, we gained insights into the strengths of each database in managing text, numeric filters, and ordered data.

```
// 1. Text-based Search (Artist Name)
db.music.find({
  "artist": { $regex: /artist name/i }
})

// 2. Non-Text-based Filter (Album Rank = 6)
db.music.find({
  "album_rank": { $eq: 6 }
})

// 3. Sorting by Album Rank
db.music.find().sort({
  "album_rank": 1
})
```

Figure 4: Sample MongoDB queries for a large dataset, demonstrating a text-based search, a non-text-based filter, and sorting by a non-unique attribute.

2.3 Metrics Evaluation

To analyze database performance on metrics like query execution time, CPU usage, and memory, we used the following methods:

1. Query Processing Time Measurement: In our study, we measured query execution times using both Python's ***time*** function and Elasticsearch's internal ***took*** parameter. Python's ***time*** function records the entire duration from when the query is initiated to when the response is received, including network latency. In contrast, Elasticsearch's ***took*** parameter focuses solely on the server-side processing time, excluding any network delays. This difference makes direct comparisons challenging, as MongoDB does not have an equivalent to the ***took*** parameter. To simplify our analysis and make a fair comparison, we opted to use Python's ***time*** function, which accounts for both processing time and network latency. However, we also analyzed the ***took*** time in our script, which is typically faster because it excludes network delays, and we have included these findings in the results section.

2. CPU and Memory Usage: For measuring memory and CPU usage, we relied on Elasticsearch's JVM metrics and Python's ***psutil*** library. Elasticsearch provides detailed JVM node stats, which are useful for tracking heap usage and understanding how memory is managed during query execution. Unfortunately, MongoDB does not offer a direct equivalent of these JVM metrics. As a result, we used Python's ***psutil*** library to monitor overall system memory and CPU usage. While we did not directly compare JVM metrics for MongoDB, we did record these metrics for Elasticsearch in our script and discussed their relevance in the results section.

Note: Why Not Use Kibana? *While Kibana is a powerful tool for visualizing Elasticsearch metrics, we chose a direct approach with Python and Elasticsearch's inbuilt metrics for more detailed insights. Python's timing functions provided network latency measurements, while `psutil` allowed us to monitor cache usage directly, which is not always accessible in Kibana. This setup enabled us to track system-wide memory usage and caching in real-time, ensuring that no other background processes interfered with the resource consumption measurements during our testing.*

2.4 Indexing

In Elasticsearch, indexing is fundamental to optimizing search speed and scalability. Each document in Elasticsearch is processed to create an ***inverted index***, which maps each unique term in the data to the documents where it appears.

Figure 5: Shard and replica settings on Elasticsearch

In our project, we initially opted for a ***single shard*** and ***single replica*** to store all our data as shown in Figure 5. This choice was made due to the size and structure of our dataset, which could be efficiently managed within a single shard without requiring additional distribution across multiple shards. Using a single shard simplified our setup and maintained performance without the added complexity of cross-shard coordination. For indexing, we used the ***default indexing*** setup, which provided a good balance between performance and ease of use, allowing us to perform the required queries without complex configuration as shown in Figure 6. Default indexing in Elasticsearch automatically indexes fields, enabling straightforward search queries without defining specific mappings.

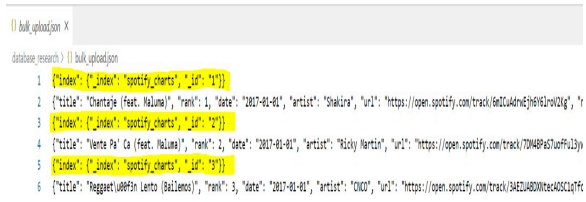

Figure 6: Default index created for each field in Elasticsearch (increasing order of index numbers)

In MongoDB, indexing functions differently, and indexes are not created by default on every field. Instead, MongoDB allows us to define indexes selectively, typically based on the fields that will be most frequently queried. For instance, in MongoDB, we can create an index on the "artist" field with the following command as shown in Figure 7.

```
db.spotify_charts.createIndex({ artist: 1 })
```

Figure 7: Indexing in MongoDB

This index would improve search performance on the "artist" field, allowing MongoDB to retrieve relevant documents faster when that field is used in queries. MongoDB's indexing structure is highly customizable, making it ideal for specific query optimization, which we discuss further in the Discussion section.
In our initial testing with MongoDB, we did not use indexing on the fields being queried, resulting in slower query performance, especially for large datasets. To make our comparisons consistent and improve performance, we added an index to the "artist" field, as this was the primary field used for filtering in our queries.

3. Result Analysis

The results of our experiments show significant performance differences between Elasticsearch and MongoDB.

3.1 Experiment 1 - Smaller Dataset

For our smaller dataset experiment, we conducted two main queries to analyze the performance and memory usage differences between Elasticsearch (ES) and MongoDB. This dataset is the dataset with fewer documents, but has a very large amount of text per document.

3.1.1 Query 1 - Full-Text Search

In this query, we filtered data based on a two word phrase that we knew was in the dataset, "repulsive friction". This first query was one

that specifically looked for either “repulsive” and “friction” *anywhere* within the document field we searched without any relation to each other. In other words, if a document field contained the word “repulsive” somewhere, and the word “friction” somewhere, that document was returned. This search will be called “context-free text search”.
This was the only query where MongoDB substantially outperformed ES. As seen in the figure, on average ES took 15.4 seconds to find and return all the documents that contained the searched words (9462 documents), while MongoDB on average only took about 1.8 seconds. These results were consistent over different strings searched as well, Figure 8.

We believe there are a couple of different reasons MongoDB so significantly outperformed ES in this one query. One reason could be that the dataset simply didn’t contain enough documents to allow ES to shine (see Experiment 2). MongoDB uses a B-tree-like-structure for text indexing, which is optimized for smaller datasets and requires less overhead than Elastic’s inverted index structure and therefore for the smaller dataset Mongo pulls ahead. This also could be indicative of MongoDB being more better optimized not only for smaller datasets but also for context-free text searches generally.

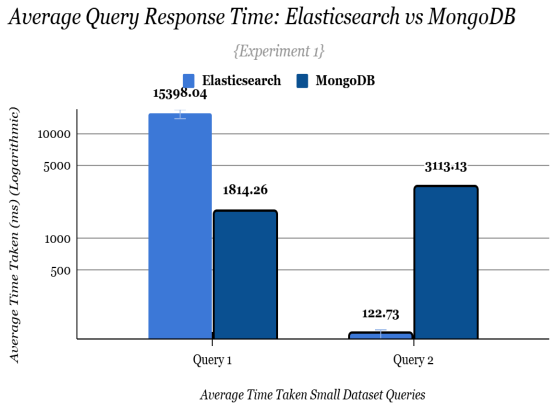

Figure 8: Comparison of Average Query Response Times Across Different Queries in Elasticsearch and MongoDB (Smaller Dataset)

The second reason Elasticsearch might have performed significantly worse is because of some external factors we were unable to measure. We believe this because after querying ES, Elastic’s own query-response time reporting field, **took**, reported that this context-free text search took, on average, 2.3 seconds. This number is quite similar to the MongoDB query time. This means that there is about 13 seconds of something else happening before our testing scripts register the true end of the query. This could be the time of transferring the original data from the database to the client program, this could be Elasticsearch attempting to “score” the returned values, or it might be extra data being transferred for each of the documents. Overall, we are not confident in asserting any of these reasons as *the* reason for the worse performance and perhaps it is even a combination of them all.
In terms of memory, MongoDB reportedly used 45% less than ES, Figure 9. This would lend to the reason that ES is either transferring more metadata or scoring data, or the B-trees indices are generally smaller than the inverted indices in these high-word count, low-document datasets.

3.1.2 Query 2 - Phrase Search

This query filtered data based on the same two word phrase, “repulsive friction”, but this time the query returns documents where those two words are found specifically in that order next to each other. Essentially, it looks for the two-word phrase in the document field, not for the mere existence of the individual words. These will be called “phrase searches” or “context-sensitive searches”.
In these phrase searches, Elasticsearch substantially outperformed MongoDB. As seen in the figure, on average ES took 122ms for the query, and Mongo took on average 3.1 seconds. This means that ES was about 25 times faster than Mongo. These results were more in-line with what we expected. We believe the reason

Elasticsearch did so much better in this experiment is twofold. The first is because there are no efficient operations in Mongo to do this context-sensitive searching. The only option for the user is to do a regular-expression search of the field, which is relatively slow. This is in contrast to Elastic's "match_phrase" keyword that efficiently uses the inverted index to find all the instances of the phrase appearing in a document. We were expecting this query to be more difficult for ES due to the sheer amount of text in each document, resulting in each word in the inverted index to map to many documents, but we were pleasantly surprised by the speed of Elastic's context-sensitive lookup. We learned this was because in the inverted index, ES also stores the position of the word within the document [15]. With this knowledge, ES is able to quickly see if two words are next to each other without ever having to access the documents themselves, just the index data. Mongo does not store positional information, and therefore has to go look in the documents. We believe this is one of the main reasons ES is significantly faster in this test.

In comparing the memory, neither database used a significant amount, but ES did use less than Mongo according to our tests.

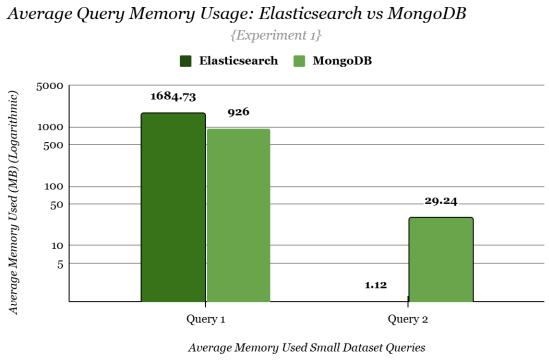

Figure 9: Comparison of Average Memory Usage Across Different Queries in Elasticsearch and MongoDB (Smaller Dataset)

3.2 Experiment 2 - Large Dataset

For our large dataset experiment, we conducted three main queries to analyze the performance and memory usage differences between Elasticsearch and MongoDB.

3.2.1 Query 1 - Filtering by Artist Name (Text-Based Filter)

In this query, we filtered data based on the artist name. Using Python's timing libraries, we measured total query time, which included network latency. In Elasticsearch, we also recorded the ***took*** parameter and JVM memory statistics (heap memory usage), providing a deeper view into the internal memory management. However, MongoDB lacks a directly comparable internal measurement, so for consistency, we used Python's timing and memory functions across both databases for the main comparison graphs. We repeated this experiment 10 times and took the average for both time and memory to minimize variability.


```
[
  {
    "Query": "Filter By Artist Elastic",
    "Total Time Taken Average": 104.41000000000001,
    "Elastic 'Took' Time Average": 52.6,
    "Memory Usage Average": 0.426,
    "JVM Heap Used Average": 451.855
  },
  {
    "Query": "Filter By Rank Elastic",
    "Total Time Taken Average": 307.7990000000001,
    "Elastic 'Took' Time Average": 197.8,
    "Memory Usage Average": 3.5089999999999995,
    "JVM Heap Used Average": 525.814
  },
  {
    "Query": "Sort By Rank Elastic",
    "Total Time Taken Average": 359.111,
    "Elastic 'Took' Time Average": 249.9,
    "Memory Usage Average": 3.623,
    "JVM Heap Used Average": 508.046
  },
```


Figure 10: Runtime results from Elasticsearch

As shown in Figures 10 and 11, Elasticsearch outperformed MongoDB in both speed and memory efficiency for handling large volumes of text data, thanks to its inverted index, which

is specifically designed for high-speed, full-text search.

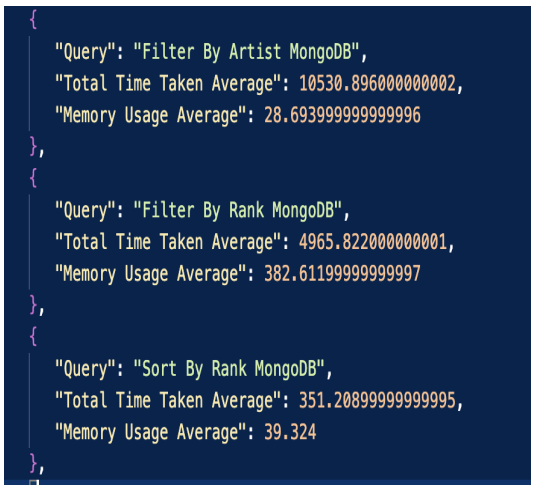

Figure 11: Runtime results from MongoDB

This index structure maps terms directly to documents, allowing Elasticsearch to locate relevant entries without scanning each document individually. Its tokenization and normalization processes further optimize searches by breaking text into searchable units and standardizing them (e.g., lowercasing).

In contrast, MongoDB's document-based indexing, though effective for general queries, lacks the specialized efficiency for large-scale text search that Elasticsearch's inverted index provides. As depicted in the graph Figure 13, this results in significantly higher memory consumption and slower processing times in MongoDB when handling full-text search on large datasets. The average memory usage graph reveals MongoDB's increased resource demands, while the average time graph (measured in milliseconds) highlights Elasticsearch's performance advantage, Figure 12. This is primarily due to Elasticsearch's ability to map terms directly to documents, enabling faster keyword lookups and more efficient memory utilization.

3.2.2 Query 2 - Filtering by Album Rank (Non-Text-Based Filter)

To test non-text-based filtering performance, we chose to filter by *album rank*. Even with numerical data, Elasticsearch continued to outperform MongoDB, both in speed and memory usage. The primary reason is Elasticsearch's efficient filtering algorithms that quickly identify and retrieve numerical values using specialized data structures. In contrast, MongoDB's document-based approach often requires scanning more extensively, especially with larger datasets, leading to higher memory consumption as seen in Figure 10 and 11.

For numeric filtering, Elasticsearch outpaces MongoDB largely because of its **columnar data storage** and specialized data structures, such as **doc values**. Doc values store field values in a way that is optimized for sorting and aggregating, allowing Elasticsearch to access and filter numeric data efficiently. This method minimizes memory consumption, as shown in Figure 13, and enables faster access to sorted or filtered values by keeping them in memory-efficient columnar format, as shown in Figure 12. MongoDB, while capable of handling numerical data, primarily uses B-tree indexes, which are efficient for general lookups but not specifically optimized for high-speed filtering on large numeric datasets. This difference allows Elasticsearch to handle non-text filtering operations with less overhead.

3.2.3 Query 3 - Sorting by Album Rank

For sorting, we set up both Elasticsearch and MongoDB to order results by album rank. By default, Elasticsearch returns only the first 10 results, so we had to adjust the ***size*** parameter in the query to retrieve a larger dataset for an accurate comparison. For sorting, both databases showed similar query times, likely due to MongoDB's indexing capabilities, which helped

it achieve comparable speeds. However, Elasticsearch still used significantly less memory as shown in Figure 13. This difference is due to Elasticsearch's efficient memory handling and optimized storage for sorted data, whereas MongoDB's in-memory sorting can be more resource-intensive when dealing with large amounts of data.

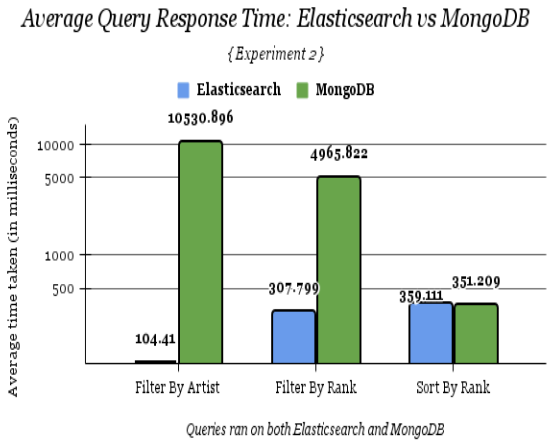

Figure 12: Comparison of Average Query Response Times Across Different Queries in Elasticsearch and MongoDB

In the case of sorting, MongoDB's use of **B-tree indexing** allows it to achieve similar speeds to Elasticsearch. B-trees are particularly effective for operations that require ordered data, such as sorting, as they allow the database to quickly traverse the index in order. While B-trees are not as optimized as Elasticsearch's inverted index for text-based searches, they work well for sorting tasks because MongoDB can quickly locate and retrieve ordered data, leveraging existing indexes on fields. In filtering tasks, however, MongoDB's B-tree indexing is less specialized, leading to more overhead as it processes each document to identify matching entries. Therefore, while indexing in MongoDB speeds up sorting, it does not provide the same efficiency boost in filtering, especially when handling large text-based datasets.

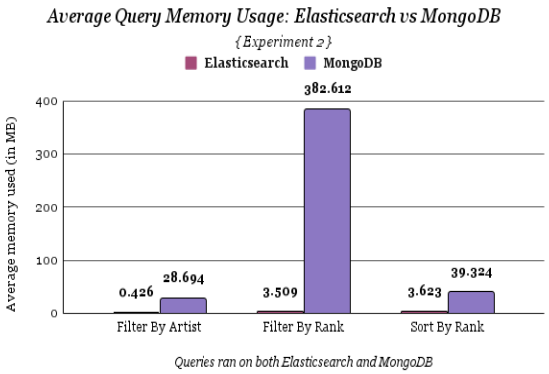

Figure 13: Comparison of Average Memory Usage Across Different Queries in Elasticsearch and MongoDB

In summary, Elasticsearch demonstrated a distinct performance advantage over MongoDB, particularly with larger datasets. Its specialized indexing structures are optimized for high-speed text searches and numeric filtering, making it both significantly faster and more memory-efficient for these tasks. Unlike MongoDB, Elasticsearch's inverted index allows it to rapidly locate documents containing specified terms without scanning every entry, which is especially useful in real-time applications and scenarios that demand quick access and retrieval.

For smaller datasets, MongoDB's general-purpose indexing provided an advantage in certain tasks, especially simple full text search and a slight advantage in sorting, but it lacked the filtering efficiency needed for both context-specific text search and high-performance scenarios. Overall, Elasticsearch's architecture excels with larger datasets, where it consistently showed much faster response times and lower memory usage. This makes it well-suited for real-time applications, Quora, Netflix, Uber or Walmart, where speed and memory optimization are crucial.

4. Discussion

In this section, we aim to address key questions that were raised during the submission of Report 1.

Question 1: Are inverted indices the most popular index structures for document searching? Are there any other index structures better than inverted indices for text searching that are employed by other analytics engines?

Inverted indices are indeed among the most popular index structures for text searching, particularly in search engines like Elasticsearch, Apache Solr, and databases like PostgreSQL with full-text search capabilities. Inverted indices map terms to the documents in which they appear, enabling quick retrieval of documents based on specific keywords, which is essential for high-performance full-text search.

Other index structures can also be effective, depending on the use case:

Trie (Prefix Tree): Useful for prefix matching and autocomplete by efficiently finding terms that start with specific prefixes, although less common for document search.

B-tree/B+ tree: Often used in databases, like MongoDB, for range queries or equality-based searches, providing efficient sorting and access for non-text data.

SSTable (Sorted String Table): Used in NoSQL systems like Cassandra, optimized for write-heavy environments with large ordered datasets.

N-gram indexing: Useful in applications requiring fuzzy search and typo tolerance by breaking text into smaller chunks, though it can require more memory than inverted indices.

While each indexing method has specific strengths, inverted indices remain the go-to choice for general document search due to their balance of speed and efficiency in handling large volumes of text data [14].

Question 2: Is there any drawback of Elasticsearch especially when compared with MongoDB?

Complexity: Elasticsearch's distributed setup can be complex to manage and scale, requiring more expertise than MongoDB, especially for large-scale deployments.

Consistency vs. Availability: Elasticsearch uses an eventual consistency model, which might not work well for applications that need strict data consistency. MongoDB offers flexible consistency levels, which may be preferable in scenarios requiring stronger consistency.

Speed vs. Flexibility: In Elasticsearch documents are immutable [16], whereas in Mongo they are not [17]. In Elasticsearch, anytime you "update" or "change" a document, the DB actually just deletes the original and rewrites the entire updated document. This leads to MongoDB being better for situations where documents are being updated or changed. Elasticsearch excels in being highly available with quick text searching at scale – at the cost of having no efficient way to update data. This tradeoff is exemplified in the popular use case of log storage, search, and analysis for Elastic – because when analyzing logs you are never changing them, but you do need quick searches.

Storage Requirements: Elasticsearch generally needs more storage due to its index structures, which optimize for fast search performance.

General Purpose: As seen in our experimental results, for smaller datasets, MongoDB's general-purpose indexing proved beneficial for tasks like basic full-text search and sorting, giving it a slight edge in simpler operations. However, it struggled to match Elasticsearch's specialized indexing when filtering was required for high-performance, context-specific text searches, as also shown in other studies [13].

Question 3: Why is Elasticsearch faster than MongoDB? Is this comparing distributed Elasticsearch vs non-distributed MongoDB?

In the paper we cited [13] in our report 1, the authors compare multiple data stores, including PostgreSQL, MongoDB, Elasticsearch, and Apache Solr. For Elasticsearch, they use its distributed setup capabilities, emphasizing its ability to handle large-scale, distributed environments with features like real-time search and indexing. However, for MongoDB, they appear to use a single-node configuration rather than a sharded or distributed setup, which might limit the comparison's scalability and performance aspects. So, while Elasticsearch's distributed strengths are assessed, MongoDB's distributed capabilities are not fully represented in this specific setup.

Elasticsearch is designed for fast, scalable search and optimized for **full-text search** and analytics, especially in large datasets. Its **distributed architecture** and specialized indexing structures (like inverted indices and custom analyzers) provide a significant performance advantage when it comes to query execution, particularly in search-heavy scenarios.

However, in our experiment the comparison is between two non-distributed instances. With only a single shard in Elasticsearch and no sharding in MongoDB, both systems operated without distributed scaling. This setup allowed us to observe their indexing and query capabilities in a more controlled environment, focusing on the performance differences due to their indexing mechanisms alone, rather than any distributed clustering effects. Elasticsearch's advantages in handling large datasets stem primarily from its inverted index, which is optimized for search, rather than from a distributed architecture. MongoDB's performance on smaller datasets benefited from its document-based indexing, but it struggled with larger data sizes where Elasticsearch's specialized search index was much more effective.

These are some of the extra research questions that we came across and wanted to address.

Question 4: Can we only use default indexing in Elasticsearch, or is custom indexing also an option, and when should each be used?

Custom indexing in Elasticsearch is typically used when specific field mappings or tokenizations are needed to support complex queries, custom analyzers, or data structures beyond what the default settings provide. For instance, if a project requires detailed filtering or precise control over how text fields are broken down and searched (like handling multilingual text or specific numeric ranges), custom indexing enables the adjustment of mappings to support these requirements. Default indexing provides an efficient, general-purpose setup that was sufficient for our experiment's needs, as our queries focused on basic text search and sorting. This simplified approach saved time and reduced complexity in our setup, aligning well with the goals of our experiment. Custom indexing, however, may be explored further in other contexts where fine-tuning is necessary, especially as MongoDB's approach to indexing differs significantly in its flexibility and setup requirements.

Question 5: Does MongoDB support inverted indexing for full-text search, and how does it compare to Elasticsearch in terms of performance?

MongoDB Atlas, a managed cloud service, includes full-text search powered by inverted indexes, which are essential for efficient text retrieval. It uses the open-source Lucene library to implement these indexes, allowing features like phrase matching, tokenization, fuzzy search, and text ranking. This indexing capability helps MongoDB Atlas perform well for smaller datasets with text-based queries.

However, while MongoDB Atlas uses inverted indexes for text search, it is less advanced than Elasticsearch, which is specifically designed for large-scale, distributed search applications. Elasticsearch's inverted indexing is more customizable and optimized for complex queries and high-performance environments. MongoDB's text search, though useful, does not offer the same level of sophistication or scalability as Elasticsearch, particularly in handling large datasets or intricate search requirements. Therefore, MongoDB Atlas is suitable for many applications, but Elasticsearch excels in more demanding search scenarios.

5. Limitations

This report acknowledges several limitations that affect the generalizability of our findings. Due to time and resource constraints, we focused on a limited set of query types and specific experimental setups, which may not fully capture the broader performance characteristics of MongoDB and Elasticsearch.

Caching effects were significant, as both databases use caching to improve repeated query performance, though results varied based on system memory and configuration. This impacted the reliability of our performance assessments for one-time or real-time queries. Our focus on text-based and rank-based queries also limits applicability, particularly for MongoDB, which excels in structured data and aggregation tasks that were not fully explored.

Hardware and environmental factors further influence database performance. Elasticsearch typically performs better in high-resource setups, and our single-node experiment did not test its scalability in a multi-node environment. Additionally, both databases offer extensive indexing and tuning options, which we kept standard for simplicity; optimized configurations could yield different results. Overall, these limitations provide context for our conclusions, suggesting that further experiments are needed to capture a fuller picture of database performance.

Under the hardware limitations, we ran into several memory-related issues where our test system did not have enough memory to complete certain tasks. The main example of this was in the sorting tests. Elasticsearch is unable to return larger than 10,000 document chucks without special “scrolling” queries and handlers. Because there is nothing similar in Mongo, we did not want to implement and analyze such techniques in Elasticsearch. Therefore, when sorting, Elastic was only returning 10,000 documents, while Mongo was attempting to return the entire dataset sorted. Our test system did not have enough memory for Mongo to sort, and our testing automation to receive the data. Therefore we had to limit Mongo (and therefore all our test results) to the 10,000 document cap from Elasticsearch. In a system with large amounts of memory, we would be able to get more accurate test results.

6. Conclusion

This study compared Elasticsearch and MongoDB, focusing on their performance in text-based search tasks. Our findings show that Elasticsearch outperforms MongoDB when handling larger datasets, primarily due to its efficient use of inverted indexing, which enables faster text search and filtering. On the other hand, MongoDB performed well with smaller datasets, where its general-purpose indexing provided satisfactory performance for queries, especially simple full text search and a slight advantage in sorting. While we used default indexing in Elasticsearch for our experiment, custom indexing could offer further optimization for more complex use cases. Overall, Elasticsearch is better suited for large-scale, high-performance text search applications, whereas MongoDB remains effective for smaller, simpler workloads.

References

[1]https://blog.quarkslab.com/mongodb-vs-elasticearch-the-quest-of the-holy-performances.html

[2]https://medium.com/@emmaw4430/mongodb-vs-elasticsearch-deciding-the-right-database

[3]https://bitninja.com/blog/comparing-mongodb-with-elasticsearch/

[4]https://www.dragonflydb.io/faq/mongodb-performance-vs-elasticsearch

[5]https://logz.io/blog/elasticsearch-vs-mongodb

[6]https://www.scaler.com/topics/elasticsearch-vs-mongodb/

[7]https://granulate.io/blog/elasticsearch-vs-mongodb-5-key-differences-how-to-choose/

[8]https://www.mongodb.com/resources/basics/full-text-search

[9]https://shambhavishandilya.medium.com/elasticsearch-understanding-full-text-search-eb08d272f97e

[10]https://www.dragonflydb.io/faq/forgot-mongodb-cluster-password

[11]https://chatgpt.com/share/672bebd7-f504-8003-8539-b5de42e45fb9

[12] X. -M. Li and Y. -Y. Wang, "Design and Implementation of an Indexing Method Based on Fields for Elasticsearch," 2015 Fifth International Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC), Qinhuangdao, 2015, pp. 626-630, doi: 10.1109/IMCCC.2015.137.

[13] Fotopoulos, George & Koloveas, Paris & Raftopoulou, Paraskevi & Tryfonopoulos, Christos. (2023). Comparing Data Store Performance for Full-Text Search: to SQL or to NoSQL?. 10.5220/0012089200003541.

[14] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. 1998. Inverted files versus signature files for text indexing. ACM Trans. Database Syst. 23, 4 (Dec. 1998), 453–490. https://doi.org/10.1145/296854.277632

[15]https://www.elastic.co/blog/found-indexing-for-beginners-part3

[16]https://www.elastic.co/guide/en/elasticsearch/guide/current/update-doc.html

[17]https://www.mongodb.com/docs/manual/core/document/

Appendix

A. Contributions

1. Anamika R.: Worked on large dataset experiments with Elasticsearch, including setting up Elasticsearch and writing the scripts for querying. She researched why Elasticsearch performed better with larger datasets, focusing on its indexing structures and internal optimizations.

2. Jacob R.: Managed the smaller dataset experiments for both Elasticsearch and MongoDB. He was responsible for running all the scripts on the VM, ensuring the setup was properly executed. Jacob also contributed to analyzing the performance of the systems.

3. Xindi Z.: Focused on large dataset experiments for MongoDB, running the scripts to gather the necessary output. She researched MongoDB's indexing mechanisms and how they contributed to query performance.

As a Team: We collaborated on creating the graphs, writing the results, and structuring the document. We also worked together to understand and explain why Elasticsearch behaved the way it did in specific scenarios. This involved researching blogs, data, research papers, and collecting relevant quotes to support our findings.

B. Dataset Sources

Dictionary dataset
https://github.com/sujithps/Dictionary

Spotify Top 200
https://www.kaggle.com/datasets/dhruvildave/spotify-charts