

CS 582: Phase 1
Report {Team 2: Data Force}



Anamika Rajendran
NMSU, USA

Xindi Zheng
NMSU, USA

Jacob Rydecki
NMSU, USA

Abstract

Elasticsearch is an open-source distributed search and analytics engine built on top of Apache Lucene. It was first released in 2010 and has since become popular for its scalability, near real-time search capabilities, and ease of use. Elasticsearch is designed to handle a wide variety of data types, including structured, unstructured, and time-based data. Our report will provide an in-depth analysis of Elasticsearch and examine Elasticsearch's features and performance, comparing it with traditional SQL databases and other competitive technologies.

1. Architecture and Design

1.1 Architecture of Elasticsearch

Elasticsearch is built on top of Apache Lucene and uses a RESTful API for communication. It stores data in a flexible JSON document format, and the data is automatically indexed for fast search and retrieval.

The basic architecture of Elasticsearch consists of **nodes**, which are the basic building blocks of a cluster. Elasticsearch's architecture is designed for scalability, performance, and flexibility. A **cluster** is the backbone, consisting of multiple nodes that collaboratively store and manage data. Data is distributed across nodes, ensuring

high availability and fault tolerance. Each **index** within a cluster is a collection of documents that share similar characteristics and is split into shards to distribute the data across nodes. This sharding allows Elasticsearch to handle large volumes of data and balance the search load. **Replica shards** are copies of primary shards, providing fault tolerance and enhancing search speed by distributing the query load. **Documents**, the fundamental data units, are indexed according to mappings, which define their structure and how they should be processed. Elasticsearch's query and aggregation engine enables sophisticated search operations and real-time analytics by executing complex queries and aggregations on the indexed data. This architecture ensures that Elasticsearch can efficiently manage large-scale data sets, deliver fast search results, and support real-time analytics.

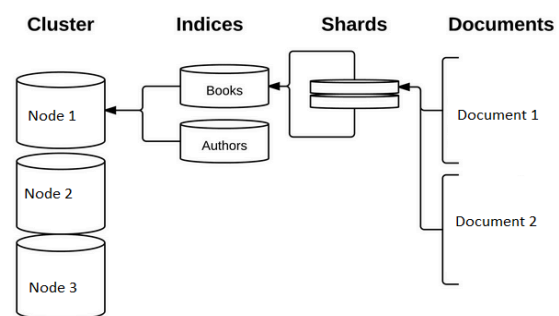


Figure 1: Architecture

| Elasticsearch | RDBMS |
|---------------|----------|
| Index | Database |
| Type | Table |
| Document | Row |
| Field | Column |

Figure 2: Comparison with relational databases

1.1.1 Novelty: Elasticsearch stands out for its distributed architecture, enabling horizontal scaling and high availability through data sharding and replication. Data is automatically divided into shards and distributed across nodes, which helps in balancing the load and improving performance. Its near real-time search capabilities allow for immediate indexing and querying, while its schema-free design with dynamic mapping provides flexibility in data ingestion. Built on Apache Lucene, Elasticsearch offers advanced full-text search features and complex query capabilities. Additionally, its RESTful API simplifies integration, and ingest pipelines allow for on-the-fly data transformation, making Elasticsearch a powerful tool for handling large-scale data and delivering real-time insights.

1.1.2 Orientation: Elasticsearch is primarily a **document-oriented** database, which sets it apart from traditional row-oriented and column-oriented databases. Instead of storing data in fixed rows or columns, Elasticsearch organizes data into flexible JSON documents. Each document is a self-contained unit that can include various fields with different data types, allowing for dynamic and schema-free storage.

Note: While Elasticsearch is not inherently a column-oriented database, it uses doc values to provide columnar storage for certain types of data. Doc values are designed to support efficient sorting, aggregations, and faceting by storing field values in a columnar format on disk, separate from the inverted index.

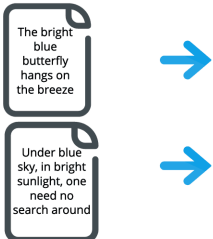
1.2 Index structure of Elasticsearch

Fundamentally, Elasticsearch organizes data into JSON-based documents, which are grouped into indices. Indices are akin to databases in relational schemas and contain logically related documents. For example, an e-commerce website might have indices for Customers, Products, and Orders. Each index is identified by a name used for querying, indexing, updating, and deleting documents.

Inverted Index:

Elasticsearch uses inverted indices to facilitate efficient searches. This data structure maps content, such as words, to their locations in documents. It works like a hashmap that links search terms to the documents where they appear. Instead of storing strings directly, it splits documents into individual search terms and maps these to their respective documents. This allows Elasticsearch to quickly find and retrieve relevant documents even from large datasets by leveraging distributed inverted indices.

For example, in the image below, the term “best” occurs in document 2, so it is mapped to that document. This serves as a quick look-up of where to find search terms in a given document.



| ID | Term | Document |
|----|-----------|----------|
| 1 | butterfly | 1 |
| 2 | blue | 1,2 |
| 3 | bright | 1,2 |
| 4 | retire | 2 |
| 5 | wind | 2 |

Figure 3: Inverted Index Table

There is one more index structure - doc value. **Doc values** are used for efficient sorting, aggregations, and faceting. They provide a

columnar storage format that is optimized for these operations. Although querying on doc values is generally slower compared to indexed fields, it presents a tradeoff between disk usage and query performance. This approach is suitable for fields that are infrequently queried and where query performance is less critical.

1.2.1 Storage of indexes: Elasticsearch uses a combination of in-memory and disk-based storage to optimize performance.

Disk: Most of the index data, including inverted indexes and doc values, is stored on disk. This is essential for handling large datasets that exceed available memory.

Main Memory: Frequently accessed data and index structures may be cached in memory to speed up search and query operations.

1.2.3 Modification to enhance performance: To speed up Elasticsearch indexing, several key strategies can be employed. First, upgrade hardware by using fast storage solutions like SSDs and ensuring sufficient memory and CPU resources. Next, adjust settings such as increasing the `refresh_interval` to reduce the frequency of index refreshes during heavy indexing. Utilize the bulk API to index large batches of documents more efficiently rather than handling them one at a time. Temporarily reduce the number of replica shards and disable automatic index refreshes during large indexing operations to lower overhead and improve throughput. Employ index templates to standardize index settings and mappings for consistency. Additionally, Elasticsearch uses a segment-based indexing approach, where documents are written to segments on disk and subsequently merged in the background to improve search performance and minimize fragmentation. Finally, continuously monitor performance and make adjustments based on system metrics and load.

2. Queries and Optimization

2.1 Queries

Elasticsearch provides a Query Domain Specific Language (DSL) that uses JSON to create queries that search your data. Official documentation describes the queries as “an AST (Abstract Syntax Tree) or queries, consisting of two types of clauses”[10]. These Elasticsearch clauses are generally divided into two categories: *leaf query clauses* and *compound query clauses*. *Leaf queries* can further be subdivided into three different categories: (1) *full-text*, (2) *term-level*, and (3) *range* queries. Whereas *compound* queries have various keywords to implement multiple *leaf* queries.

We will take time here to discuss each leaf query and compound queries. First, however, some explanations of terminology are in order. In the various types of queries, there are multiple *keywords*. These keywords are key-value pairs used to provide both necessary and optional parameters for the functionality of the query. For example, in the *range* queries (expounded below) some of the keywords include `gt` and `le` which stand for “greater than” and “less than” respectively. The data mapped to these keywords define the range for which the data is searched in these queries. Likewise, *fuzzy* searches return documents that are relatively similar to the given search term. This allows for matches even in the case of typos or misspelling at the cost of time and computational efficiency. Finally, in Elasticsearch aggregations are one “step” outside DSL queries. This means that while aggregations are not properly called queries, we will still have some discussion on their specification and use in this section.

Note: All example queries in this section are based on a hypothetical index that contains information on emails sent within an

organization including senders, timestamps, body content, IP addresses, etc.

2.1.1 Full-Text Queries: *Full-text* queries are used to search for specific strings within your documents' analyzed text fields. These are fields that have been “analyzed”, meaning that the text within these fields have been tokenized for efficient search and lookup within the database. These queries have various keywords for different types of searches of your data. The most common include `match`, `match_phrase`, and `multi_match`. These keywords look for text with fuzzy matching and phrase queries in a single field, `match_exact_phrases`, and fuzzy matching across multiple fields respectively. Although they look similar, they have subtle differences – a recurring theme in Elasticsearch queries. As Elasticsearch is most often used in the space of text-analysis these queries are very commonly seen as they are the natural way to search for text in your documents.

Example: This query returns a list of all documents where the string of “critical information” (or similar text) is found in the body of the email.

```
GET /_search
{
  "query": {
    "match": {
      "body": "Critical information"
    }
  }
}
```

2.1.2 Term-Level Queries: *Term-level* queries are used to find **exact matches** within a field and **do not analyze search terms**. This exact matching in predictably structured data without analysis is the main feature and use case for *term-level* queries. Some common keywords in these queries include `term`, `exists`, and `wildcard`. These query keywords return docs

that have the exact value in the specified field, docs that contain the specified field, and docs that match the given wildcard string respectively. According to the official Elasticsearch documentation, some common use cases include IP Address matching or product IDs [11].

Example: This query returns a list of all documents where the source IP is 172.16.16.1.

```
GET /_search
{
  "query": {
    "term": {
      "ip.source": "172.16.16.1"
    }
  }
}
```

2.1.3 Range Queries: While *range* queries are technically a type of *term-level* query, we found it helpful to give *range* its own section as it is relatively dissimilar to the other *term-level* query keywords and functions. *Range* queries return documents where a specified field falls within a specified range. By default, this query is only allowed to be used on numeric or time data, but it can be used on text data if allowed in the database; however, according to the official Elasticsearch documentation [12], it is an extremely expensive operation and is not recommended.

Example: This query will return all documents that were sent since the start of 2024.

```
GET /_search
{
  "query": {
    "range": {
      "timestamp": {
        "gte": "2024-01-01T00:00:00",
        "lte": "now"
      }
    }
  }
}
```

```

    }
  }
}

```

2.1.4 Compound Queries: *Compound* queries are the queries/keywords that surround and wrap the leaf queries. The most common of these queries is the *bool* query, which allows you to string together various leaf queries with logical keywords. These include *must*, *should*, and *must_not*. Where *must* functions as a logical AND and *should* functions as a logical OR. *Compound* queries are most often created when you need to combine various leaf queries. It should be noted that *compound* queries can certainly be nested; however, the deepest level of nesting will always consist of a leaf query.

Example: This query will return all documents that contained the previous “Critical information” in the body, but only from the year 2024. This query combines the *full-text* and *range* leaf queries through the use of a *bool* *compound* query.

```

GET /_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "body": "Critical information"
          }
        },
        {
          "range": {
            "timestamp": {
              "gte": "2024-01-01T00:00:00",
              "lte": "2024-12-31T23:59:59"
            }
          }
        }
      ]
    }
  }
}

```

2.1.5 Aggregations: As mentioned in Section 1, Elasticsearch is optimized for efficient aggregations and thus is an important area to discuss and understand. In Elasticsearch, aggregations are used to “summarize data as

metrics, statistics, or other analytics” [13]. Essentially, this means that aggregations typically analyze a single field across very many docs. The efficiency of this operation is inversely proportional to the number fields per doc you are attempting to analyze. As mentioned in this section’s introduction, aggregations are one “step” outside queries proper, meaning that aggregations are performed at a different time (usually after) queries. Aggregations either run on all data, or on data filtered on one or more queries. In Elasticsearch, aggregations are considered *metric*, or *bucket* aggregations. These are aggregations that calculate metrics from field values, and aggregations that group docs into “buckets” or “bins” based on field values respectively. Therefore the output can be summarized as statistical/numeric, or buckets containing information on groups of docs.

Example: This aggregation is a *bucket* aggregation (known through the use of the *terms* keyword) that groups docs based on the email senders. This will create “buckets” for each of the email senders and give the count of each bucket in this example.

```

GET /_search
{
  "aggs": {
    "emails_per_sender": {
      "terms": {
        "field": "from.keyword"
      }
    }
  }
}

```

Some example output for this aggregation would look like this:

```

...
{
  "buckets": [
    {
      "key": "johndoe@company.com",
      "doc_count": 12
    },
    ...
  ]
}
...

```

2.1.6 Joins: Elasticsearch is not a relational database and does not support traditional joins as found in SQL databases. Instead of using joins, it is recommended to denormalize the data. This involves embedding related data within a single document structure, often as nested objects or arrays. By doing so, you create a doc that can be queried directly without any types of joins. This denormalization allows for efficient querying of the data in the future, as any other attempts at joining multiple documents together is very expensive in Elasticsearch, and not what this database was designed for.

2.2 Optimization

Elasticsearch is most often used for text analysis and searching, and aggregation data. In this section, we will discuss some optimizations that have been mentioned in passing that allows text to be searched more efficiently, and aggregations to be performed more efficiently. Text searches are optimized by a process of “analyzing” and “tokenizing” the text. While aggregations are optimized based on the columnar format of fields for docs on the disk.

2.2.1 Analyzing & Tokenization: In this context, the term “analyzing” specifically refers to the processing that Elasticsearch does on both the indexed text (docs) and your search terms. During this process, Elasticsearch takes the inputted text and performs various operations to improve the efficiency of search. The process

begins with “tokenization”, where the text is split into various “tokens”. For example, splitting a paragraph of text into a token for each word found. Then, Elastic will lowercase all the text, which reduces the amount of text that needs to be searched in the future. The next step of the process is stemming and lemmatization of the tokens. This is mainly attempting to reduce words to their root or stem in order to reduce the amount of terms to search through. For example, the token “running” or “runner” may be reduced to the stem “run”, so then if searching for the term “run”, both areas with the term “running” and “runner” will be returned without actually comparing all the strings with each other. Finally, Elasticsearch can perform “synonym expansion”, which is a process of treating words as synonyms during search to find similar ideas without actually comparing more strings. For example, “sprint” may be a synonym to “run”, so a search for “run” could return both text for “sprint” and “running” without searching for the text “sprint” as well.

While this analysis process may take some extra computational overhead during the ingestion of data, its benefits far outweigh this small overhead because of the reduction of terms and strings that need to be compared during future searches on the data. This process is what allows Elasticsearch to be efficient for finding textual matches in large amounts of data.

2.2.2 Columnar Field Blocks: As mentioned in Section 1, Elasticsearch stores each field in a separate column for various docs. This allows aggregations to be performed much quicker because aggregations typically run only one on field across many docs, so that all the data can be read sequentially from the disk (as the fields of across different docs are located in the same place).

3. Use Cases and Practical Applications

Elasticsearch is the core of the Elastic Stack, an open-source suite for data ingestion, storage, analysis, and visualization. Built on a distributed architecture, it allows data to be stored and searched across multiple nodes, providing fast, near real-time search and analysis of large datasets. Originally a search engine, it became popular for real-time log data processing and visualization. Elasticsearch supports real-time data updates, complex aggregation queries, and full-text search, making it suitable for environments requiring large-scale, real-time data handling. With a simple REST API, it's easy to integrate across systems, making it accessible to developers for use cases like log analytics, application monitoring, web and enterprise search, and business analytics.

3.1 Primary use cases

3.1.1 Application search: For large e-commerce platforms, users can input keywords, and Elasticsearch will quickly search through millions of product entries, returning the most relevant information based on user needs.

3.1.2 Website Search: When a user searches for news keywords on a news website that has stored articles for decades, Elasticsearch can filter through all the articles using an inverted index, returning relevant news reports quickly.

3.1.3 Enterprise search: In the internal network of large enterprises, employees may need to quickly search for company documents, employee information, project files, etc., making information retrieval within the company more efficient.

3.1.4 Logging and log analytics: Cloud service companies can use Elasticsearch to monitor their server logs in real time. When anomalies occur, analyzing log data helps the operations team quickly locate the source of the problem and take immediate corrective action.

3.1.5 Infrastructure metrics and container monitoring: In companies deploying containerized applications, Elasticsearch can be used to monitor container performance, such as CPU usage, memory consumption, and network traffic. Operations teams can visualize this data through Kibana to identify performance bottlenecks or failure points.

3.1.6 Security analytics: Elasticsearch can be used to analyze user access logs in a company, checking for abnormal behaviors such as repeated login attempts or unauthorized data access. By monitoring this real-time data, enterprises can quickly detect and prevent potential security threats.

3.1.7 Business analytics: It can monitor product sales across all time periods for certain companies, and by collecting and analyzing real-time data, management can adjust marketing strategies or inventory management based on data trends.

3.2 Company Use Cases

3.2.1 Netflix: Netflix uses Elasticsearch within its ELK Stack to monitor customer service operations and analyze security logs. Netflix benefits from Elasticsearch's automatic sharding and replication, flexible schema, and extensive plugin ecosystem. As a result, Netflix has scaled its usage to over a dozen clusters, consisting of hundreds of nodes, demonstrating Elasticsearch's ability to handle high-volume data effectively.

3.2.2 eBay: eBay relies on Elasticsearch for various business-critical search and analytics tasks. They created an internal "Elasticsearch-as-a-Service" platform, enabling easy provisioning of Elasticsearch clusters on their cloud platform. This setup enhances search efficiency and supports large-scale text search requirements for eBay's numerous applications.

3.2.3 Walmart: Walmart uses Elasticsearch to gain insights into customer purchasing patterns, track store performance, and perform real-time holiday analytics. The ELK Stack also supports Walmart's security needs, offering features like single sign-on (SSO), alerting for anomaly detection, and monitoring for DevOps, making it an ideal solution for their large-scale operations.

Summary: Elasticsearch's distributed architecture, scalability, real-time search performance, and ability to handle both structured and unstructured data made it the ideal solution for these companies. Its flexibility, powerful API, and open-source nature allow organizations like Netflix, eBay, and Walmart to tailor the platform to their specific needs, while its near real-time capabilities enhance decision-making and operational efficiency.

4. Competitors and Market Analysis

4.1 Solr

Elasticsearch is schema-less, meaning it doesn't require predefined structures like index, type, or field types, allowing for flexibility in inserting and updating records [15]. In contrast, Solr requires fields and field types to be specified in advance, making changes more rigid, such as needing to re-transfer records when adding fields. Sharding and replication in Solr are also fixed, while Elasticsearch allows dynamic changes. Furthermore, Elasticsearch supports automatic cluster discovery when nodes share the same network and cluster name, simplifying distributed setups. Solr requires manual configuration for these features.

Novelty: Elasticsearch stands out with its user-friendly REST API, offering simpler integration than Solr's more complex API [15]. Its flexible Query DSL allows powerful full-text searches and complex aggregations. It scales horizontally with ease, automatically managing replication, sharding, and load balancing.

Additionally, Elasticsearch includes native machine learning features for anomaly detection and predictive analytics, which many competitors like Solr lack.

As discussed in paper [16], a comparison of Elasticsearch with SQL and NoSQL databases are as follow:

4.2 Elasticsearch vs. SQL Databases

Elasticsearch, optimized for full-text search, outperforms traditional SQL databases like MySQL and PostgreSQL in handling large volumes of unstructured text data. While SQL databases focus on structured, relational queries and often require plugins for text search, they struggle to match Elasticsearch's speed and relevance. Elasticsearch's distributed architecture enables faster queries and better scalability, avoiding bottlenecks common in SQL databases when handling complex text searches.

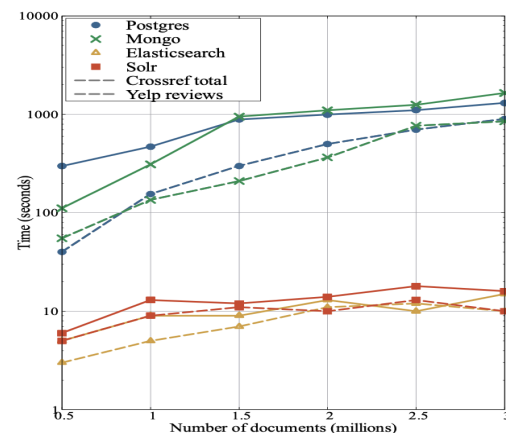


Figure 4: Exact phrase matching [16]

As shown in Figure 4, ElasticSearch and Solr though, seem to achieve the best performance compared to MongoDB and PostgreSQL, which both need significantly more time to execute the issued queries.

4.3 Elasticsearch vs. NoSQL Databases

Elasticsearch outshines other NoSQL databases like MongoDB, Cassandra, and Couchbase in full-text search due to its use of Apache Lucene. While NoSQL databases focus on high-speed transactions and key-value storage, Elasticsearch is designed specifically for efficient text indexing and search. It offers more advanced query capabilities, faster performance, and better ranking algorithms. Additionally, Elasticsearch supports distributed indexing, real-time search, fuzzy searching, and query optimization, making it ideal for search-heavy applications.

Elasticsearch is purpose-built for this task and consistently outperforms in scenarios involving large-scale, unstructured data and complex search queries.

4.4 Algolia

One competitor to Elasticsearch is “Algolia”. Algolia is self-described as software that “help[s] you implement efficient, flexible, and insightful search on your sites and applications.” [17] Generally, it is a search-as-a-service product. Algolia and Elasticsearch have similar use cases – both are heavily used for full-text searches and querying. Algolia is said to be specifically created for real-time text searches. The two notable differences described here are implementation and performance.

4.4.1 Implementation: The method of implementing each software is quite different. Elasticsearch is an open-source (typically) self-hosted database solution, whereas Algolia is a commercial cloud-based product. Therefore Algolia is far easier to get up and running efficiently as all the infrastructure is hosted and managed by the company, while with Elasticsearch you have to manage the infrastructure and optimizations. With this trade-off comes the trade-off of flexibility. Elasticsearch is highly flexible, giving the user many configuration, optimization, and indexing

options to perfectly fit their specific use case; Algolia is much more generic without as many configuration options. Likewise, the only limit to the amount of data you can store in Elasticsearch is physical storage and processing times; for Algolia the subscription costs scales directly the the number of queries performed, potentially making it far more expensive.

4.4.2 Performance: In terms of performance, there is not very detailed data. Algolia has performed their own tests comparing the two text-search solutions, and claim their product is “between 12 and 200 times faster than Elasticsearch – for every search query we performed.” [19] Yet, this document does not describe what configurations were used in Elasticsearch, and has limited to strictly simple text searches. Also, the data was said to be 2.4 million documents, so the performance is unclear for larger datasets. Although we found no other performance comparisons between the two products, we note that Algolia is optimized for real-time text searches (which was reflective in their selected test cases), whereas Elasticsearch has optimized capabilities for both text searches and statistics/analytics.

Generally, Algolia is a competitor to Elasticsearch which excels in its “plug-and-play” capabilities for end users, and real-time text search speeds when compared to Elasticsearch, at the expense of flexibility in data and searches.

5. Drawbacks and Limitations

While Elasticsearch offers many powerful features, it also comes with its share of drawbacks.

1. Lack of ACID Transactions: Elasticsearch doesn’t support ACID properties (atomicity, consistency, isolation, durability), making it

unsuitable for operations needing strong data integrity like traditional OLTP databases. This limitation impacts reliability when it comes to transactional guarantees or complex joins required in some applications. It's important to remember that Elasticsearch is an OLAP database and doesn't have the required consistency guarantees.

2. Data Retention Challenges: Due to Elasticsearch's append-only model, data grows continuously, especially in log and monitoring use cases. This means that the original and existing data is more or less immutable, and any new data that is written is merely appended. This leads to potential performance degradation and higher storage costs. Users must implement effective data retention and archiving strategies to prevent these issues.

3. Fault Tolerance: Elasticsearch clusters need careful setup to avoid data loss. Although its distributed nature supports replication, without proper configuration (e.g., multiple data nodes, replication across zones), the risk of losing data remains. Setting up fault-tolerant infrastructure is essential when using Elasticsearch as the primary datastore.

4. Storage Choices: While local storage offers better speed, it risks data loss if the node fails. Networked storage options like AWS EBS provide resilience but compromise speed. It's important to balance performance with data safety when deciding on storage methods.

5. Limited Support for Complex Joins: Elasticsearch doesn't natively support complex SQL-style joins. Handling relational data often requires workarounds like denormalization or using parent-child relationships, which may lead to performance hits and added complexity.

6. Lack of Schema Alteration: Fields defined in Elasticsearch are difficult to modify or rename

without requiring a full reindex. This makes schema planning critical when using Elasticsearch as a primary datastore. Changes to data types, for example, can force reindexing, a process that can be time-consuming and resource-heavy.

7. Performance Bottlenecks: Elasticsearch excels at handling aggregations and small result sets but struggles with large data sets and operations requiring frequent access to the entire dataset. Reindexing tasks can cause memory strain and degrade performance, making it essential to plan for scalability and optimize queries.

8. Huge Result Sets: Elasticsearch is optimized for aggregations and small result sets but is less efficient when dealing with queries returning large volumes of data or requiring frequent access to the entire dataset. This can create performance issues for use cases needing large result sets.

9. Complex Querying: Elasticsearch's Query DSL can be difficult to learn, particularly for users familiar with SQL. The learning curve for creating even simple queries can be steep, which may be a barrier for new users or teams transitioning from traditional relational databases.

10. Storage Efficiency: Compared to columnar databases, which are designed for read-heavy workloads and analytics, Elasticsearch can be less efficient in terms of storage, especially for large-scale data or analytics use cases.

References

1. <https://www.linkedin.com/pulse/elasticsearch-understanding-basic-architecture-jeevan-george-john/>

2. <https://betterprogramming.pub/system-design-series-elasticsearch-architecting-for-search-5d5e61360463>
3. <https://www.geeksforgeeks.org/elasticsearch-arch-architecture/>
4. <https://www.elastic.co/guide/en/elasticsearch/reference/current/doc-values.html>
5. <https://www.elastic.co/blog/what-is-an-elasticsearch-index#>
6. <https://www.knowi.com/blog/what-is-elasticsearch/>
7. <https://www.sentinelone.com/blog/improve-your-elasticsearch-performance/>
8. <https://www.elastic.co/guide/en/elasticsearch/reference/7.17/tune-for-indexing-speed.html>
9. <https://bigdataboutique.com/blog/using-elasticsearch-or-opensearch-as-your-primary-datastore-1e5178>
10. <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html>
11. <https://www.elastic.co/guide/en/elasticsearch/reference/current/term-level-queries.html>
12. <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-range-query.html>
13. <https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html>
14. <https://www.knowi.com/blog/what-is-elasticsearch/>
15. Kılıç, Uğur & Karabey Aksakalli, Isil. (2016). Comparison of Solr and Elasticsearch Among Popular Full Text Search Engines and Their Security Analysis. 10.13140/RG.2.2.24563.32803.
16. 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.
17. <https://www.algolia.com/doc/guides/getting-started/what-is-algolia/>
18. <https://josipmisko.com/algolia-vs-elasticsearch>
19. <https://www.algolia.com/blog/engineering/full-text-search-in-your-database-algolia-versus-elasticsearch/>

Appendix

A. Contributions

1. **Anamika:** Responsible for “Architecture and Design and Drawbacks” questions, contributed to reviewing the team's work, conducted literature research, and helped format the final document.

2. **Xindi Zheng:** Focused on “Use Cases and Practical Applications” questions, assisted in reviewing group contributions, participated in the literature review, and worked on document formatting.

3. **Jacob Rydecki:** Addressed “Queries and Optimization” questions, provided feedback on the group's work, conducted literature review, and contributed to formatting the report.

As a Team: As a team, we worked collaboratively on the “Competitors and Market Analysis”, as it required a literature survey from both academic research papers and industry sources. Each member contributed by gathering relevant data and insights, ensuring that our analysis was covering both scholarly research and current market trends.