Search Engine and Deployment in a Cluster

Laura Lasso, Miriam Méndez, José Gabriel Reyes, Alejandro González y Andrea Mayor

December 20, 2023

Contents

Abstract

This paper presents the fundamental aspects involved in deploying an application within a computer cluster to meet our clients' demands within reasonable timeframes. Our focus is on the development of an API application designed to retrieve books containing a specified word and providing the word's frequency within each book. To accomplish this, we evolved our prior work by transitioning from SQLite to MongoDB Atlas (cloud) as our database solution. Futhermore, we have also developed a shared storage using hazelcast. This last implementation enchance our apps by allowing distributed memory instead of persinstance memory, using the idea of in-memory data grid and a cluster of computer, resulting in quicker response times for our specified task. As an alternative, we could use MongoDB atlas, although its slower because is stored as a cloud service.

1 Introduction

In the rapidly evolving landscape of modern computing, the deployment of applications within clustered environments has become pivotal to meet the escalating demands of users and clients. This paper delves into the essential strategies and methodologies essential for deploying an application within a cluster of interconnected computers, focusing on optimizing response times to ensure timely and efficient service provision.

The primary objective of this study is to illustrate the deployment process through a case study involving the development of an Application Programming Interface (API) tailored for retrieving books based on specified keywords and providing corresponding word frequencies within each book. This research extends from previous work, emphasizing the pivotal shift in database infrastructure from SQLite to MongoDB, as well as, the improvements in performance we can get with hazelcast and the use of in-memory data grid.

This paper elucidates the critical nuances of deploying applications within clustered environments, drawing insights from the transformation of database infrastructure and its tangible impact on response times and query efficiency.

2 Components and Functionality:

2.1 Crawling Process:

1. **Source Specification:** In our case, we will visit the website https://www.gutenberg.org/, to download books in a txt plain format using the batch downloader, which will try to retrieve each time 10 books.

2. **Data Retrieval:** We will store this files in a defined project structure where we store the books by the day(using the date) they were stored. Each book will have is own folder(named with the book id) to store its raw content at first. The files of this folder will vary with the appearance of the Book Cleaner.

3. **Message to queue:** The crawler will use the service activemq to send a message to a queue, where the book cleaner will be subscrited to. Once the message is deployed it will contatin the location of the recently stored book, indicating to the book cleaner that is ready to be splitted into content and metadata.

2.2 Book Cleaning

In this phase, the books obtained by the Crawler are divided into content and metadata. The license for each book is removed, ensuring that the datamart contains data ready for processing and storage, which will be performed by the datamart-builder.

The book cleaner will remove redundant spaces, stop-words or punctuation characters. This will allow the datamart-builder to store the words in a database without including repeated words that rely in tiny changes. This content will be store in a file called id(with an underscore)content.txt in the folder of the book. Moreover, the metadata will be stored in a json file named id(with an underscore)metadata.json. It will contain information about the release date, title, language and author of the book it represents.

Finally, the book cleaner will be connected to two queues, one for the content and one for the metadata. The datamart builder will be subscripted to the two queues in order to receive the messages of the locations of the content and metadata stored. This will allow to create Associate, Book and Word objects, in order to get their attributes and store them in our mongodb instance.

2.3 Datamart-builder

This image, labeled datamart-builder, plays a critical role in the Inverted Index project by facilitating the construction of the datamart. The datamart is a structured repository designed for efficient data processing and storage using a MongoDB database. This database is used because of the BSON format storage that it offers to us which increases the search speed for our API.

The datamart builder retrieves messages from both content and metadata, storing them in a MongoDB database. MongoDB, in contrast to SQL, persists data in binary format. However, relying solely on MongoDB posed challenges as it could only be accessed locally. To enhance data accessibility, we plan to establish a cluster of computers that will collectively utilize their RAM memory, leveraging Hazelcast instances for distributed storage. As a backup and for comparison purposes, we will also maintain a datamart stored in MongoDB Atlas.

Our data adheres to the following structure in MongoDB Atlas:

A words collection where each object is composed by:

- An 'id' field representing the word itself.
- Each word is associated with a dictionary, whose keys are the book ids and the values are the count of the before mentioned word in those books.

A books collection where each object is composed by:

- A 'GutenbergBookId' field which represent the book id in Gutenberg.
- An 'author' field.
- A 'bookTitle' field.
- A 'releaseDate' field.

In the Hazelcast Multimap, we maintain a dictionary where the keys represent words, and the corresponding values are dictionaries containing book fields. Each book's attributes serve as keys, with their respective values associated with each book.

2.4 Api

This image contains the API JAR file, serving as a crucial component to deliver services to both the frontend and other integral parts of the system. The API JAR encapsulates the functionality and endpoints required for communication between clients and the datamart, acting as a webservice between our project and the potential clients. The webservice could be check using the direction:

http://localhost:8080/documents/:words?from=...&to=...&author=...

We can use http://localhost:8080/stats?type=count to obtain the number of words for each book.

Components and Functionality:

- **Get a word appearance:** The information about the books where a determined book appears, can be retrieved using documents/:words, where each word that we desire to obtain should be separated with a "+". The result of this query will be a Json indicating the timestamp when the query was launched, and a list as value. This list will be composed by associate objects, each associate object will contain:
  - **Word object**: which contains the id of the word.
  - **Book object**: which contains the id, author, release year and title of a book.
  - **Count**: which represent the number of times a word appears in a book.

  Fundamentally, the 'associate' is a type that enables us to depict the connections between words and books. Furthermore, this query can be refined using the parameters 'from,' 'to,' and 'author.' The first two establish a date range for searching books where the queried word appears. The 'author' parameter is self-explanatory, focusing on books by a specific artist.

2.5 Nginx

This image provides an Nginx server configured to act as a reverse proxy, playing a crucial role in managing service exposure within the Inverted Index system. The Nginx server serves as an intermediary between client requests and backend services, optimizing traffic flow, enhancing security, and enabling efficient load balancing.

Components and Functionality:

1. **Nginx Server:** The image includes a fully configured Nginx server, ready to operate as a reverse proxy to handle incoming requests.

Service Management:

1. **Reverse Proxy:** The Nginx server operates as a reverse proxy, forwarding client requests to the appropriate backend services based on defined routes and configurations.
2. **Load Balancing:** In scenarios with multiple backend services, Nginx facilitates load balancing, distributing incoming requests across the available service instances to optimize performance.
3. **Security Features:** Nginx incorporates security measures such as SSL termination, access control, and request filtering, ensuring a secure and reliable service exposure.

Benefits:

1. **Efficient Traffic Handling:** The reverse proxy efficiently manages incoming traffic, optimizing the distribution of requests to backend services.
2. **Scalability:** Nginx supports load balancing, enabling the system to scale by efficiently distributing incoming requests among multiple backend service instances.
3. **Security Enhancement:** The Nginx server enhances security by implementing features like SSL termination, access controls, and request filtering.

3 Experiment

As we have two implementations oriented for distributed programming, we have decided to compare the times for indexing and searching with both implementations:

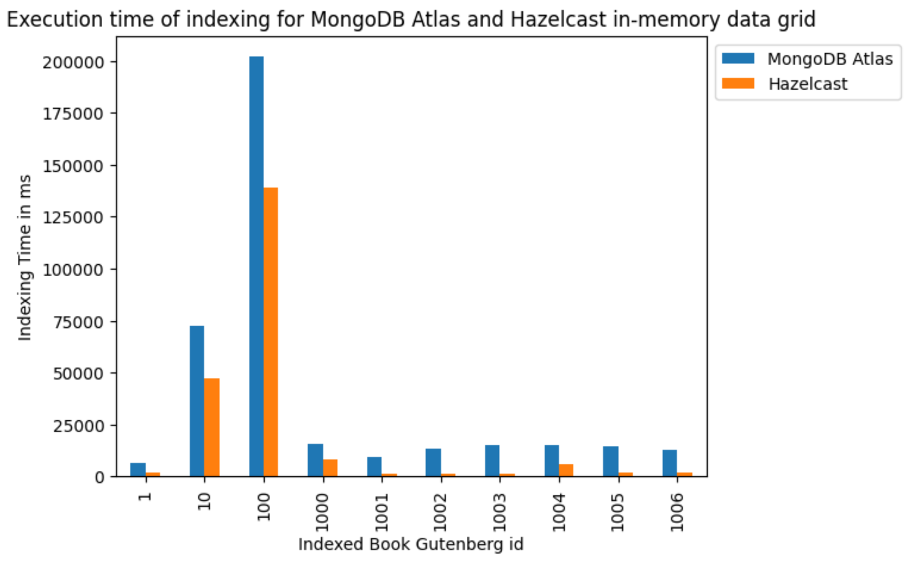

Figure 1: Indexer Execution Time

Hazelcast demonstrates faster insertion times than MongoDB Atlas, potentially attributed to quicker local connections compared to the cloud-based connections of MongoDB Atlas after indexing. In addition to this, it is worth mentioning that we were using the free tier for Mongodb Atlas, which also gave us minimum resources. And finally, due to the location of the server (which is in Paris) we also suffered from latency issues for all the operations.

In this example, the average indexing time in MongoDB Atlas is 37,837.4 milliseconds, while the Hazelcast implementation averages 21,131.4 milliseconds. Translating these times to books indexed per minute, MongoDB completes approximately 1.5858 books per minute, whereas Hazelcast indexes approximately 2.8368 books per minute. However, it's worth mentioning that books with IDs 10 and 100 are heavier compared to the rest, impacting the overall mean, which tends to converge to a lower value with the remaining iterations.

In conclusion, for indexing purposes, Hazelcast outperforms MongoDB due to its consistently faster performance.

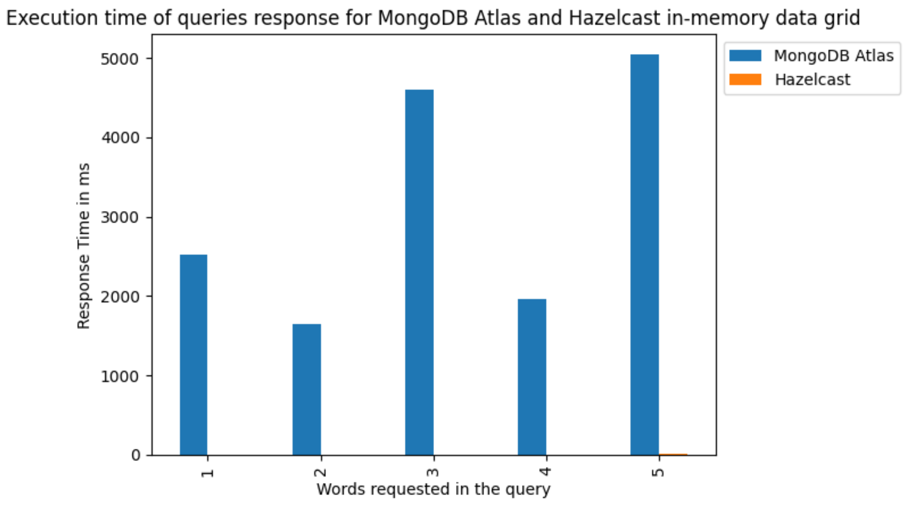

Figure 2: Queries response Execution Time

We observe a significant disparity in response times between MongoDB Atlas and Hazelcast storage. Query times using Hazelcast instances range between 2 and 5 milliseconds, making them challenging to discern clearly in 2. MongoDB Atlas, on the other hand, incurs inherent latency due to its connection to cloud services.

Hazelcast demonstrates a mean query response time of 3 milliseconds, while MongoDB Atlas averages 3156.4 milliseconds. This implies that with Hazelcast, approximately 20000 queries can be executed in one minute, whereas MongoDB Atlas achieves approximately 19 queries per minute.

In summary, Hazelcast outperforms MongoDB Atlas in both processes, particularly in search operations. Accessing data from a cloud service inherently introduces latency compared to local storage, unless the alternative utilizes RAM, as is the case with Hazelcast. While MongoDB Atlas can be optimized by creating indexes, this comes at the expense of the indexer, resulting in more pronounced negative effects.

4 Conclusion

Distributed programming improves our applications capabilities. Recognizing the demands of big data, which require high availability and rapid responses, we must establish data structures for resource sharing. Storing our datamart using a cloud service is an option; however, this choice tends to be slower due to reliance on external resources.

For efficient indexing and querying, Hazelcast is a superior alternative, particularly excelling in query performance. The sole requirement is a set of computers connected to the same network, offering an easily scalable solution for expanding our computer set and discovering new nodes to create data structures.

In striving to meet the challenge of satisfying numerous clients within reasonable timeframes, a variety of solutions can be explored. In this paper, our approach involves clustering our web service across multiple machines to handle numerous requests concurrently. To achieve this, we synchronize our machines using the Nginx load balancer, aiming to distribute the workload evenly among the computers.

5 Docker images

The images of the project: `https://hub.docker.com/r/mmdezr/inverted_index/tags`. To utilize these components, we simply execute docker run with the image name, enabling the required ports for each (5107 for Hazelcast and 8080 for the Sparkweb service). Failure to specify the port will result in isolated images. Initially, we explored the concept of Docker Compose while employing a local MongoDB database. However, this approach encountered compatibility issues with services like MongoDB and ActiveMQ. Furthermore, it produced a datamart version confined to a single computer.

The images we could execute are:

For Hazelcast and MongoDB Atlas respectively:

- mmdezr/inverted_index:api-hazelcast for webservice.
- mmdezr/inverted_index:datamart-hazelcast for indexer. We need to specify a volume with the datalake.
- mmdezr/inverted_index:api-mongo for webservice.
- mmdezr/inverted_index:datamart-builder-mongo for indexer. We need to specify a volume with the datalake.

6 Future Work

User Experience and Interface

Exploration of user-centric enhancements and interface improvements could be a future focus. Understanding user behavior patterns and preferences to tailor the system's interface and functionality accordingly would contribute to a more satisfying user experience.

6.1 Explore other load balancing alternatives

To increase the performance of our application it could be convenient to check the performance of different load balancing alternatives, like creating our own module for load balancing the load of requests.

6.2 Increase the functionality of the API

It could be interesting to add more query params to our API so the clients could request words depending on, for example, the author of the book. Additionally, it could be good if we give our clients the option of requesting several words and give them or a union or an intersection of results.