Search Engine and Deployment in a Cluster

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

December 10, 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 as our database solution. This transition notably enhanced our system's responsiveness because MongoDB's storage of data is in BSON (Binary JSON) format, which contributes significantly to the efficiency of searches, resulting in quicker response times for our specified task.

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, signifying a deliberate strategy to enhance system responsiveness.

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 Dockerization

2.1 Docker Compose

Docker presents a feature that provides the execution of images and services on which they depend on. We have used this feature due to problems with connection of the broker between images and with a mongodb instance. The main point of using docker compose to execute is the interconnectivity between images. As the execution on each application depends on broker messages, we should have all the images the use the same broker service, specified in the same docker-compose.yml.

The usage of individual images for execution lead to several fails, including not connection with the host of the services(activemq or mongodb) and incompatibilities between each module. It helps by providing a shared store between all the images.

Referring to the docker compose we have developed two files. One will be used during the defence in order to make our api work with the mongodb instance. In this docker compose file we specify the images that should be running with our images like the MongoDB and the ActiveMQ images. This option was chosen because it enabled us to set the connection between this images and made it easier to run these containers.

Components and Functionality:
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 reads messages from both content and metadata, storing them in a MongoDB database. MongoDB, unlike SQL, stores data in binary format. To visualize the stored data in a more intuitive manner, MongoDB Compass becomes essential.

Our data adheres to the following structure:

- An 'id' field representing the word itself.
- Each word is associated with a list of JSON objects, each containing various attributes of the books such as author, release year, or the frequency of the word's appearance in the 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=...

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.

  Essentially, the associate is a type which allows us to represent the links between words and books. Moreover, this query could be filtered with the query parameters from, to and author. The first two will set a date range to look for books where the search word appears. The author is pretty self explanatory, and its search for books of a determined 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 Conclusion

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.

4 Docker images

The images of the project: https://hub.docker.com/r/mmdezr/inverted_index/tags. To use them we have used the command docker-compose up –build -d alongside the docker-compose.yml that can be found on the github repository.

5 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.

5.1 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.

5.2 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.