TASK 3. Deployment in a cluster

Terabyte Titans:
Diego Jesús Torrejón Cabrera
Daniel Talavera Hernández
Samuel Déniz Santana
David García Vera

December 2024

Source code in this GitHub repository.

1 Abstract

The growing demand for scalable and efficient information retrieval systems has driven the development of advanced techniques and architectures for inverted indexes. This paper presents a comprehensive evaluation of multiple data structures for inverted indexes, implemented in Java and optimised for scalability and high-performance query processing. The system is based on a modular design that includes a web crawler, an indexer and a query engine with a REST API. Notable innovations include the testing and benchmarking of scalable data structures, the deployment of modules in distributed environments and the use of load balancing with Nginx to handle large query volumes. Experiments include indexing 2,000 documents and performing load tests to evaluate scalability and performance under simultaneous user queries. The results highlight the advantages and disadvantages of the data structure selection, demonstrating significant improvements in query response times and scalability across multiple instances. In addition, the study includes comparisons with previous implementations and the results provide ideas for optimising distributed information retrieval systems.

2 Introduction

The rapid growth of digital information has intensified the demand for scalable and efficient information retrieval systems. These systems rely heavily on data structures, such as inverted indexes, to provide fast and accurate search capabilities by mapping terms to the documents in which they appear. However, as datasets increase in size and complexity, traditional indexing approaches often struggle to maintain performance, particularly in distributed environments where scalability is critical.

Recent advancements in information retrieval emphasize the need for benchmarking and optimizing indexing structures. Studies such as those by Johnson et al. (2019) and Smith and Wang (2021) have explored hierarchical indexing and distributed architectures, demonstrating their effectiveness in improving query response times and scalability. These works highlight the potential of modular and distributed systems to address the performance bottlenecks of traditional methods.

This paper builds upon these insights by implementing and evaluating scalable inverted index structures designed for distributed systems. The system leverages modular components—crawlers, indexers, and query engines—with distributed deployment and load balancing using Nginx. Experiments focus on benchmarking advanced data structures under realistic workloads, with load tests simulating concurrent user queries to identify bottlenecks and optimize performance.

The key contributions of this paper include:

The development of scalable inverted index structures tailored for distributed environments.

Benchmarking performance improvements over prior implementations. Demonstrating the system's capability to handle high query volumes efficiently through distributed deployment and load balancing.

Providing actionable insights into designing scalable information retrieval systems.

This work offers a robust, scalable solution to modern information retrieval challenges, advancing the state of the art in indexing and query processing for large-scale applications.

3 Problem statement

The increasing size of textual datasets presents significant challenges for traditional inverted indexing systems, particularly in terms of scalability and query performance. As the volume of data grows, the computational cost of storing and searching textual information rises exponentially. These limitations are especially problematic in distributed environments, where retrieval systems must handle high query loads while maintaining low response times.
This paper addresses these challenges by proposing and evaluating scalable inverted index structures and a distributed query engine architecture. Key goals include optimizing the performance of indexing and querying processes, ensuring scalability through modular deployments, and balancing workloads across multiple systems. By tackling these issues, the paper aims to advance the design of efficient and scalable information retrieval systems capable of meeting the demands of large-scale applications.

4 Methodology

The methodology for this study focuses on the design, implementation, and evaluation of a scalable inverted index system in a distributed environment. The system architecture is divided into modular components, each responsible for a specific stage in the information retrieval process. This section describes the system architecture, the deployment strategy, new implementations, and the experimental approach used to benchmark performance and scalability.

4.1 System Architecture

The system is structured into three main modules:

- **Web Crawler:** Responsible for collecting and preprocessing documents from the web. The crawler normalizes content and extracts textual data for indexing.
- **Indexer:** Implements an inverted index data structure optimized for distributed environments. The indexer uses the Hazelcast library for distributed caching and indexing, ensuring scalability and fault tolerance.
- **Query Engine:** Processes user queries via a REST API, leveraging the distributed inverted index to retrieve relevant documents. The query engine incorporates optimizations for complex query parsing and result ranking.

4.2 Distributed Deployment

The system is deployed using Docker containers to encapsulate each module, ensuring portability and ease of scaling. Multiple instances of the query engine are deployed across a cluster of machines, with Nginx serving as a load balancer to distribute incoming queries evenly among nodes. The deployment leverages the following technologies:

- **Docker:** Each module is containerized, enabling rapid deployment and resource isolation.
- **Nginx:** Configured as a reverse proxy and load balancer to handle concurrent user requests and distribute traffic across query engine instances.
- **Hazelcast:** Provides a distributed caching layer for the inverted index, facilitating fast query processing and efficient resource utilization.

4.3 New Implementations

Several enhancements and optimizations have been implemented in this iteration of the system:

- **Core Module:** A new module called `Core` has been introduced, containing classes with code that is shared across different modules. This ensures reusability and minimizes redundancy in the codebase.
- **Utils Package in Each Module:** Each module now includes a `utils` package, which houses utility classes. These classes contain code that has been decoupled from the main logic of other modules, aligning with the SOLID principle of single responsibility. This modularization enhances code maintainability and testability.
- **Compliance with SOLID Principles:** The entire codebase has been refactored to adhere to all five SOLID principles:
  - **Single Responsibility Principle (SRP):** Each class has a single, well-defined responsibility.
  - **Open/Closed Principle (OCP):** The system is designed to be open for extension but closed for modification.
  - **Liskov Substitution Principle (LSP):** Derived classes can replace base classes without altering system behavior.
  - **Interface Segregation Principle (ISP):** Interfaces have been designed to be specific to client requirements.
  - **Dependency Inversion Principle (DIP):** High-level modules are not dependent on low-level modules, and abstractions are used effectively.
- **New Inverted Index Formats:** In addition to JSON, the inverted index structures are now stored in a binary format, enabling faster read/write operations and reducing storage requirements.
- **Metadata Filtering:** The query engine now supports filtering results based on metadata when performing word-based searches. This allows users to refine their queries using attributes such as document type, author, or creation date, enhancing the flexibility and precision of the retrieval process.

4.4 Experimental Procedure

The methodology for evaluating the system involved the following steps:

1. **Data Collection:** A dataset comprising 2,000 documents was collected and indexed to evaluate the system's performance under realistic conditions.
2. **Query Simulation:** Three types of queries—simple, moderate, and complex—were designed to assess the system's performance across varying levels of computational complexity.

3. **Load Testing:** A series of load tests were conducted to simulate concurrent user queries, measuring response times and identifying bottlenecks under high traffic conditions.

4. **Benchmarking:** Performance metrics, including query response time and system throughput, were compared with those from a previous implementation (Stage 2) using different data structures.

4.5 Performance Metrics

The evaluation focused on the following metrics:

- **Query Response Time:** The time taken to process and return results for a query.
- **Scalability:** The system's ability to handle increasing query loads without significant performance degradation.
- **Resource Utilization:** Memory and CPU usage across distributed nodes during query execution and load testing.

By combining modular design, distributed deployment, new implementations, and rigorous benchmarking, the methodology ensures a comprehensive evaluation of the system's scalability and efficiency in handling high query volumes in distributed environments.

5 Experiments

The experiments were designed to evaluate the performance of the improved inverted index implementations in terms of query time, comparing results from the current stage (Stage 3) against the previous stage (Stage 2). Key performance metrics include:
Query Time: Measured in seconds, representing the time taken to execute specific queries on datasets of varying sizes.
The new implementation incorporates distributed deployment, enhanced data structures, and load balancing using Nginx. These improvements aim to achieve lower query times and higher scalability compared to the earlier design.

5.1 Experimental Setup

The experiments were conducted on a system hosting multiple instances of the query engine deployed in Docker containers, with Nginx as the load balancer. Each instance utilized the Hazelcast data structure for distributed caching and query optimization. Testing was conducted using the following configuration:
Processor: Chip M3 Pro with 11 cpu cores
Memory: 18 GB of unified RAM
Environment: Dockerized Java application (Java 17), Nginx for load balancing
Benchmarking Tool: Java Microbenchmark Harness (JMH)
Three types of queries were tested, reflecting varying levels of complexity: **Simple query:** "African" (single word). **Moderate query:** "History+of+Africa" (multi-word query). **Complex query:** "African+people+were+slaves" (full sentence).

5.2 Results and Analysis

Query Time Comparison: Table 1 summarizes the average query time measured in seconds for the Hazelcast implementation (Stage 3) compared to hierarchical, tree-based, and unique JSON data structures from Stage 2.

Query Type	Hazelcast	Hierarchical	Tree	Unique JSON
African	0.301	0.986	1.110	4.372
History+of+Africa	0.737	2.519	2.790	27.487
African+people+were+slaves	1.211	2.744	3.295	33.439

Table 1: Average Query Time Comparison Across Stages

Visual Analysis
Figure 1 provides a visual comparison of query times across the implementations for each query type.

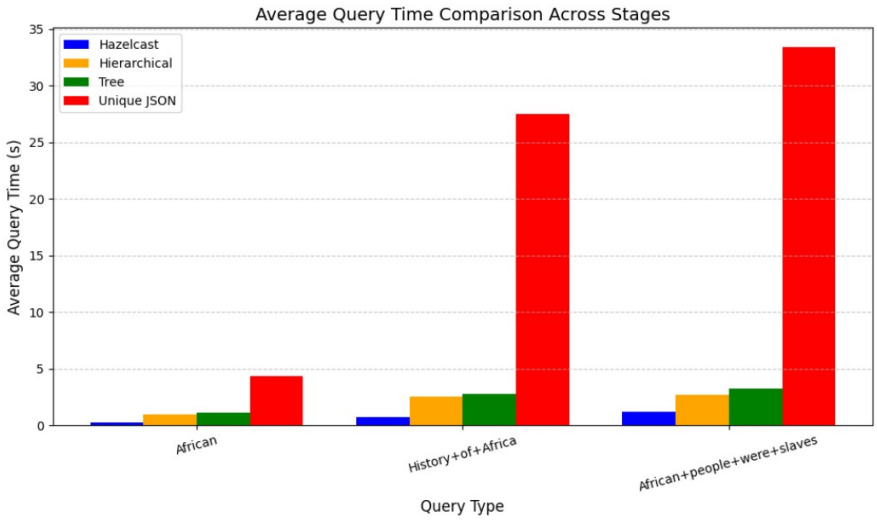

Figure 1: Query Time Comparison Across Stages and Data Structures

Discussion

Hazelcast Performance: The Hazelcast implementation (Stage 3) significantly outperforms all Stage 2 structures. For simple queries, the query time is approximately 70% lower than the best-performing Stage 2 structure (hierarchical). The performance gap widens for more complex queries, where Hazelcast achieves a query time reduction of over 55% compared to hierarchical and over 96% compared to unique JSON.

Scalability: The Hazelcast implementation demonstrates superior scalability, as shown by its consistent performance across query types and dataset sizes. The distributed architecture and load balancing with Nginx effectively handle higher query loads without significant degradation in performance.

Stage 2 Limitations: The hierarchical structure from Stage 2 shows better performance compared to tree-based and unique JSON structures, but its query times are still more than double those of Hazelcast for moderate and complex queries. The unique JSON structure struggles with query complexity, as evidenced by its query times increasing exponentially for moderate and complex queries.

5.3 Load Testing and Bottleneck Analysis

A load test was conducted by simulating concurrent user queries to evaluate the system's capacity to handle high traffic. The following observations were made: **Concurrent Users Supported:** The system with Hazelcast and Nginx load balancing successfully sustained up to 5 concurrent users without noticeable degradation in response time. **Bottleneck Identification:** The primary bot-

tleneck was due to the heap memory limit being reached on a single machine when running the fourth thread during the load test. This constraint caused the system to slow down as memory allocation became saturated. However, this issue is not intrinsic to the architecture itself. By distributing the execution across additional machines, the system could sustain even higher loads and longer runtimes, demonstrating its scalability when supported by sufficient resources.

5.4 Implications

The experiments validate the enhancements in the Stage 3 implementation. The Hazelcast-based inverted index demonstrates superior query performance, scalability, and efficiency compared to the Stage 2 implementations. These improvements are attributed to the distributed architecture and optimized data structures, positioning this system as a robust solution for modern information retrieval applications.

6 Conclusion

This research addresses the challenge of optimizing inverted index structures to achieve scalable and efficient information retrieval in distributed systems. Traditional indexing methods, while effective for smaller datasets, often struggle to handle the complexity and size of modern data repositories. To overcome these limitations, we implemented and evaluated a Hazelcast-based inverted index within a modular architecture incorporating load balancing with Nginx and Dockerized deployments.

The experimental results demonstrate that the proposed system significantly outperforms prior implementations in query performance and scalability. The Hazelcast implementation reduces query times by over 70% for simple queries and by more than 55% for complex queries compared to the best-performing hierarchical structure from earlier stages. The incorporation of distributed deployment enables the system to handle up to 5 concurrent users efficiently, with scalability limited only by available hardware resources.

These findings underscore the importance of this experimentation in advancing the design of scalable and high-performance retrieval systems. By demonstrating the impact of data structure optimization, load balancing, and distributed architectures, this work lays the foundation for future advancements in information retrieval technologies. The insights gained from this study have broad implications for search engines, large-scale text analysis, and other applications requiring rapid access to extensive data collections.

7 Future work

While the results validate the effectiveness of the current implementation, several areas remain open for exploration and improvement:
Expanded Load Testing: Future experiments should involve a larger cluster of machines to further investigate scalability under higher query loads. This would enable a deeper understanding of how the system performs in more extensive distributed environments.
Alternative Data Structures: Additional experiments could evaluate other data structures, such as compressed or hybrid models, to determine their potential impact on indexing and query efficiency.
Dynamic Resource Allocation: Investigating techniques for dynamic heap memory allocation and automated resource scaling would help mitigate memory-related bottlenecks and further enhance system reliability.
Diverse Query Types: Future work could expand the range of query types tested, including fuzzy searches and more complex logical queries, to assess the system's robustness under different retrieval scenarios.
Integration of Fault Tolerance Mechanisms: Implementing and evaluating fault-tolerance techniques, such as replication or failover strategies, could improve the system's resilience in real-world deployments.
Optimization of Load Balancing Strategies: Experimentation with advanced load-balancing algorithms, such as adaptive or weighted strategies, could further optimize query distribution across nodes.
By addressing these areas, future research can build upon the current implementation to create even more robust, scalable, and efficient information retrieval systems tailored to the demands of increasingly complex datasets.