

Search System Optimization Using Elasticsearch with Redis-Based Caching

**Sharmila Jeyarani S.¹, Anwarsha K.², Akash S.³,
Mani Kandan R.⁴**

¹Assistant Professor, ^{2,3,4}Student, Department of Computer Science and Engineering,
NPR College of Engineering and Technology, Natham, Dindigul, India.

E-mail: ¹sharmimca85@gmail.com, ²Anwarshakaja@gmail.com, ³Akashshiva44620@gmail.com ,

⁴manikandanravichandranv@gmail.com

Abstract

Large-scale search system requires low latency and efficient management of requests in the face of large-scale data. In this research, a cache-aware search system is introduced through the use of Elasticsearch with Redis to minimize duplicate search processes. Elasticsearch is adopted as the main tool for full-text retrieval, while Redis acts as an in-memory cache through the cache-aside approach. On each request received, the backend first checks the cache via Redis. When there is a cache hit, the result already cached from previous searches is returned without the need to call Elasticsearch. The performance of the proposed system was measured on the 500,000+ US Homes Data for Sale (Properties Dataset) data set, which comprises around 600,000 house entries. Three types of queries were tested: both single and multiple result search. The average execution time was reduced from 175.39 milliseconds without caching to 0.85 milliseconds when using Redis, which makes up 99.52% decrease. The decrease in time per individual query ranged from 98.96% to 99.73%. Comparison with a similar search scheme with the implementation of PostgreSQL proves the superiority of Elasticsearch – Redis in terms of lower latency of queries.

Keywords: Elasticsearch, Redis, Search Optimization, Cache-Aside, Query Caching, Inverted Index, Response Time, Information Retrieval.

1. Introduction

The fast development of digital information has led to the need for fast and reliable search mechanisms that can provide useful information without causing delay. Searching has become an essential feature in many areas like electronic commerce, corporate information systems, digital libraries, and online shopping websites. As the size of indexed data and number of simultaneous requests grow, the repetitive execution of similar search queries may lead to redundant computations and additional delays.

Indexing in an efficient manner is another solution to decrease the search problem. The Elasticsearch system makes use of the information retrieval system with the use of inverted index that allows full-text searching for large amounts of documents [2], [4]. Through inverted indexes, the search engine can search for documents related to query terms without having to look through all the data [6]. But indexed searching still needs query processing when the query comes into the search engine.

Another alternative method is storing the results from previous searches into the cache. Rather than making use of the query every single time, it is stored as a result, which is used later on whenever there is a request that needs a response. In particular, this technique is highly useful in situations where repeated queries are being made within a workload, which makes response reuse possible and, therefore, improves search engine efficiency. For the purposes of this research, the technology used is the Redis memory-based caching tool, which sits between the application backend and Elasticsearch. In this case, the cache-aside technique has been adopted, where first the backend will search for the response within Redis [3], [5].

The new approach is assessed with the help of US Homes for Sale (Properties Datasets) that has been released by PolarTech in Kaggle [7]. The dataset contains 600,000 housing properties and is indexed in Elasticsearch to be used to assess the search quality. The experimental study compares query processing without Redis cache and with Redis cache for one result and for several results. The objective is to determine the extent to which response caching can reduce latency for repeated searches while retaining Elasticsearch as the underlying retrieval engine.

The main contribution of the study is an application-level Elasticsearch–Redis cache-aside architecture which was specially crafted to eliminate search redundancy. In addition, the study

examines the reduction in the latency caused by the solution and then makes a comparison between the implemented search algorithm and the one based on PostgreSQL.

2. Literature Survey

Search efficiency has generally been considered by using indexing structures that help minimize the number of items being searched at the time of processing queries. Scholer et al. have studied compression techniques for inverted indexes, thereby emphasizing their significance in performing efficient queries in information retrieval systems [6]. Elasticsearch makes use of this concept of information retrieval in order to perform full-text search and distributed document search [2], [4]. While these methods increase the efficiency of search operations, they cannot prevent duplication of searches.

Caching of search results has thus become an optimization technique of great significance. Markatos studied caching of search engine query results and proved that query workload properties could be used to reuse the previous search results [8]. In another study, Fagni et al. looked into caching and prefetching with historical queries to use the query properties for improving Web searches [9]. Lempel and Moran worked on predictive caching and prefetching of query results and moved on from the idea of result reuse to prediction of future queries [10].

Caching architecture is also an important factor that affects the performance of the above optimization. Long and Suel came up with a three-tiered caching architecture design in Web search engines to lower query-processing costs at various stages of the search process [11]. In another study by Ozcan et al., they looked into the potential of a five-tiered caching architecture design in a static environment and proved its benefits for making search engines more efficient [12]. Although most of the above work focuses on search-engine-level or multi-tiered caching architecture, our current study follows a much simpler approach, that of application level caching.

The redis library offers an in-memory data storage option that works well for low latency and is commonly used for caching purposes [5]. The redis caching techniques were examined by Kaptosv in the context of high-traffic Web applications and their impact on performance [3]. Performance optimization of Redis clusters was considered by Zhu et al. in the field of distributed storage metadata management [1]. Even though these articles demonstrate the benefits of Redis in performance optimization, their application domains do not overlap with the

one in this research.

There is considerable empirical support for each of the aforementioned concepts of inverted index searching, query-result caching, and in-memory Redis storage. However, considerably lesser emphasis is paid to the combination of these principles into a cache-aside design for application-level use with Elasticsearch searches. This research aims to fill this gap by positioning Redis ahead of Elasticsearch and testing the ability of precomputed searches to improve query speed.

3. Proposed System

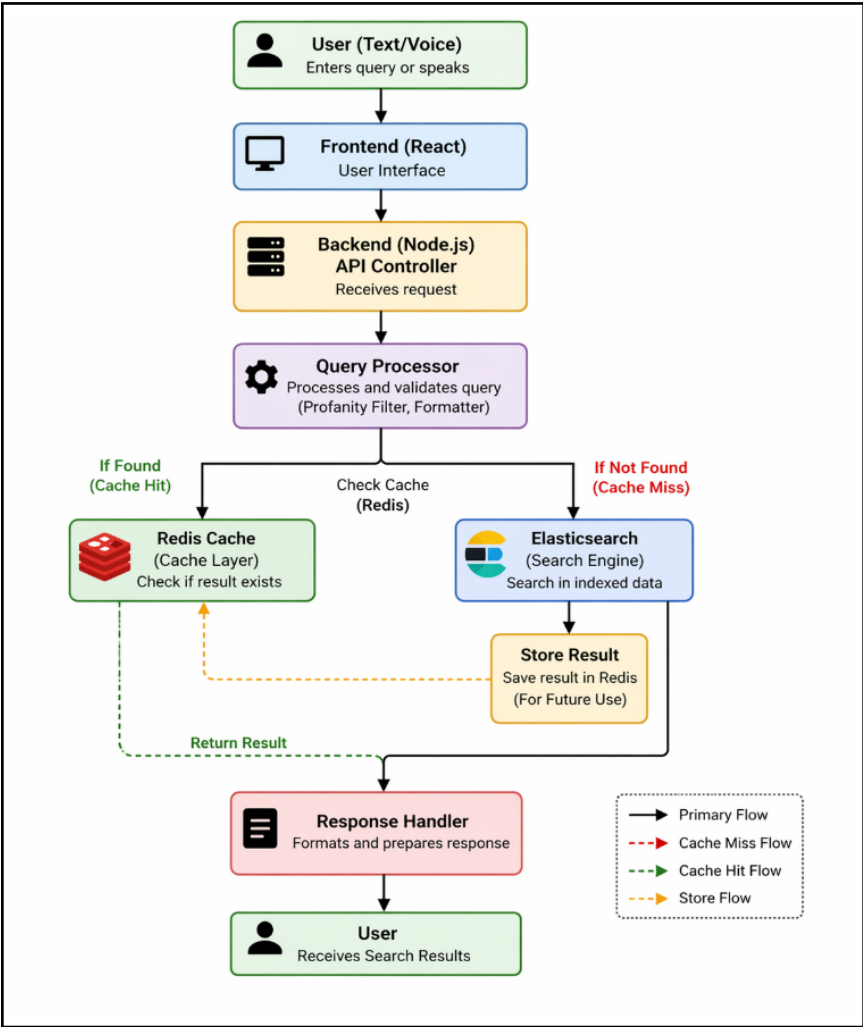


Figure 1. System architecture of search system using Elasticsearch with Redis-based caching

The proposed methodology utilizes a cache-aware architecture for full-text search which utilizes Elasticsearch combined with Redis in order to decrease latency caused by repetitive

search queries (see Figure 1). The architecture is based on the cache-aside execution model which implies that prior to executing Elasticsearch, one needs to execute Redis. A query which can be handled through the cache does not require execution of the underlying query again; otherwise, the request is sent to Elasticsearch, and its response is stored in the cache afterwards. The experimental study was conducted using the 500,000+ US Homes Data (For Sale Properties) dataset consisting of about 600,000 houses, which is called PolarTech/Kaggle housing-properties dataset [7].

3.1 Dataset Preparation and Search Index Construction

The housing data set serves as the primary repository of information for the proposed search engine application. The data set comprises records that represent houses and have searchable attributes pertaining to residential homes in the United States. Initially, the data set is processed to create an index, which then gets imported to Elasticsearch. Instead of scanning through the entire data set to fulfill every query, Elasticsearch finds candidate records using term-to-document mapping from the queries.

The indexed data set is the only reliable source that is used to conduct searches in the architecture, and it is not true that Redis replaces Elasticsearch. Redis holds onto the results of past searches to be reused. The above is critical because it means that Elasticsearch is supposed to fetch data from the index, while Redis avoids fetching duplicate requests.

3.2 Query Acquisition and Pre-processing

The query is initiated by the React-based front-end application and can be provided either as text or voice request. In case of voice request, the speech to text functionality translates the voice query to the text format prior to initiating the request towards the backend side. The resulting text query is then sent to the Node.js API server, acting as the central request handler.

Prior to cache lookup, the query goes through the query-processing phase depicted in Fig. 1. Query-processing phase is responsible for query validation and formatting as well as using the profanity filtering method integrated in the system. The main goal of this phase is to make sure that the incoming query is in the correct format suitable for processing and any offensive terms are eliminated prior to the search procedure. As a result, both text and voice queries are unified into the text format before proceeding to the cache and search phases

3.3 Cache-aside Search Strategy

The central methodological contribution is the integration of Redis with Elasticsearch using a cache-aside pattern. Let q denote a processed search query and Kq denote its corresponding cache key. The backend first evaluates whether a response associated with Kq exists in Redis.

The request is therefore divided into two execution paths:

Cache hit:

$$Kq \in \text{Redis} \Rightarrow R_q \text{ is returned} \quad (1)$$

If the requested result is present in Redis, the cached response is returned directly through the response handler. Elasticsearch is not invoked for that request. This path eliminates the search-engine execution and associated processing overhead.

Cache miss:

$$Kq \notin \text{Redis} \Rightarrow q \rightarrow \text{Elasticsearch} \quad (2)$$

In case there is no response in the cache, the backend sends the query to Elasticsearch. This service does the full-text search on the indexed housing dataset and sends the output back to the backend. It is stored in Redis prior to sending to the user. In other words, the result from the first request can be used when receiving similar requests in the future.

3.4 Elasticsearch Search Processing

Elasticsearch is the main retrieval engine used in the proposed system design. The housing records are pre-indexed prior to executing the query, hence enabling the text search to work on the inverted index format of the dataset and not having to scan the entire dataset each time. Whenever there is a cache miss, the Node.js back-end sends the query to Elasticsearch where the full text search is done. Therefore, the function of Elasticsearch complements that of Redis. In this case, the Elasticsearch solves the retrieval challenge through searching the indexed data, whereas Redis solves the problem of redundancy by caching the responses to queries that have been handled before.

3.5 Response Caching and Retrieval

Once a result is returned by Elasticsearch for the cache miss request, the result is written in Redis by the backend. The saved object corresponds to the result of the particular query. Future requests for that particular query would then be able to retrieve the result directly from memory.

The cache interaction can be represented as:

$$\text{Redis}Kq = \begin{cases} R_q, & \text{if } Kq \text{ exists,} \\ \emptyset, & \text{otherwise.} \end{cases} \quad (3)$$

where R_q represents the search response stored for query q , and $\emptyset$ represents a cache miss. The methodology consequently converts repeated searches from a search-engine-bound operation into a cache retrieval operation. This is the principal mechanism through which the architecture targets latency reduction.

3.6 Latency Model and Performance Mechanism

The expected latency behavior of the proposed architecture can be expressed by distinguishing the cache-hit and cache-miss paths. For a cache hit, the response latency is approximately determined by query processing, backend communication, Redis lookup, and response handling:

$$T_{\text{hit}} = T_{\text{qp}} T_{\text{redis}} T_{\text{response}} \quad (4)$$

For a cache miss, the Elasticsearch retrieval and cache-storage operations are additionally introduced:

$$T_{\text{miss}} = T_{\text{qp}} T_{\text{redis}} T_{\text{es}} T_{\text{store}} T_{\text{response}} \quad (5)$$

Each business instance corresponds to a predictor vector in 19 dimensions,

$$x_i = x_{i1}, x_{i2}, \dots, x_{i19} \quad (6)$$

where:

- T_{hit} = total cache-hit response time
- T_{qp} = query-processing time
- T_{redis} = Redis lookup time
- T_{miss} = total cache-miss response time
- T_{es} = Elasticsearch search time
- T_{store} = time required to store the result in Redis
- T_{response} = response-handling time

The methodology therefore does not claim that Redis makes the underlying full-text search itself faster. Instead, the principal optimization is avoiding the Elasticsearch execution altogether for repeated queries. This distinction is particularly important when interpreting the experimental results because the reported sub-millisecond measurements correspond to cached responses, whereas the non-cached measurements include Elasticsearch retrieval.

3.7 Integration of Supporting Modules

Two additional modules have been added without affecting the basic functioning of the cache-aside search structure. The speech to text module adds to the input functionality by providing support for oral queries to be inputted through the text-processing pipeline. The profanity filter module is used during the query processing phase where any offensive words are filtered out before proceeding to cache or searching phases.

The modular structure depicted in Figure 2 ensures that these operations are separate from the Redis and Elasticsearch tiers. Therefore, the core querying system can deal with both typed and transcribed queries using the same backend channel.

The proposed methodology relies on the complementarity of the role distribution. The framework for interactive components is provided by React, request handling is done via Node.js, Redis stores pre-computed responses, and Elasticsearch handles the indexed full-text search of the housing dataset. The cache aside policy defines which of those resources have to be engaged to serve requests. The performance of the proposed solution is analyzed in the comparison of cache and non-cache execution on the dataset that contains roughly 600,000 records. The results demonstrate that the cache hit scenario takes considerably more time compared to the

non-cached execution via Elasticsearch.

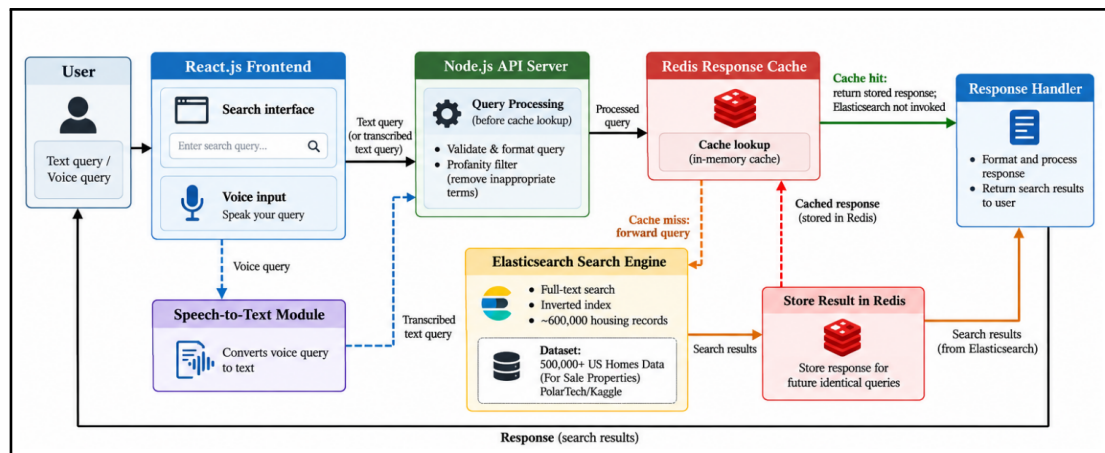


Figure 2. Implementation workflow of the proposed system

4. Results And Discussion

4.1 Experimental Evaluation

The proposed search solution was tested using the dataset called US Homes Data (For Sale Properties) which consists of more than 500,000 data entries of US homes for sale. The test was performed to analyze how Redis cache affects response time. There were three queries which were chosen as representative for the experiment: Q1 and Q2 return one answer while Q3 returns several answers. The response time was compared between Elasticsearch without cache and Elasticsearch with Redis cache where the same query results were used.

The values reported in the original evaluation are 288.07 ms, 121.10 ms, and 117.02 ms for non-cached execution, compared with 0.78 ms, 0.55 ms, and 1.22 ms for cached execution. The corresponding reported averages are 175.39 ms and 0.85 ms, respectively.

Table 1. Response time comparison on large dataset (600K records)

Query Type	Without Cache (ms)	With Redis Cache (ms)	Reduction
Q1 – Single Result	288.07	0.78	99.73%
Q2 – Single Result	121.10	0.55	99.55%
Q3 – Multiple Results	117.02	1.22	98.96%
Average	175.39	0.85	99.52%

4.2 Effect of Redis Caching on Response Latency

The most important finding is that the execution process in the Elasticsearch case differs from the Redis cache hit case. In Q1, the execution time goes down from 288.07 ms to 0.78 ms, which represents about 99.73% reduction in latency. The execution time for Q2 goes down from 121.10 ms to 0.55 ms, while in Q3, it goes down from 117.02 ms to 1.22 ms. Thus, the improvement is observed across both single-result and multiple-result queries rather than being limited to one query type.

The average response time drops from 175.39 ms to 0.85 ms, which is a decrease of 99.52%. Relative to each other, the average response time for cached requests is 206 times faster than that of non-cached requests according to the average values. This proves the key feature of the architecture proposed: If the result of the query is stored in Redis, then the query doesn't have to be performed in Elasticsearch again.

However, the various reductions obtained for Q1-Q3 prove that the impact of caching is not a consistent response time irrespective of the results. Q3 takes 1.22 ms, which is more than the other two queries since the answer that Q3 gives is more than one. However, despite being higher, it is still considerably less than the time taken by non-cached execution in Elasticsearch. This shows that the major gain is the reduction of repeated search execution.

4.3 Analysis of the Search Execution Path

The obtained performance results are in line with the described above cache-aside algorithm shown in Figures 1 and 2. When there is a cache miss, the query goes through the backend and is delivered to Elasticsearch, where the dataset gets indexed. After that, the received result gets saved in Redis. In case of subsequent queries with the same data, the backend will be able to fetch it from Redis.

Thus, the obtained performance gain is due to reduced processing time for repeated queries, not due to improved Elasticsearch index processing performance. Indeed, Elasticsearch is still in charge of processing non-cached queries, while Redis returns cached results. This is vital to understand since our system makes performance gains by altering the path of processing for repeated queries rather than by altering the Elasticsearch indexing process.

Additionally, these findings validate the choice of architecture of performing the caching

check prior to Elasticsearch queries. The use of the cache following each Elasticsearch request means that regardless of whether or not the response is found in the cache, the cost of performing the search will still be incurred. In the proposed architecture, a successful Redis check ends the process of fetching the data.

4.4 Comparison with PostgreSQL

A second experiment compares the proposed approach with a PostgreSQL-based search implementation on the same dataset. The measurements reported in Table 2 are shown below.

Table 2. Performance comparison with postgresQL (600K records)

Query Type	PostgreSQL (ms)	Elasticsearch (ms)	Redis Cache (ms)
Q1 – Single Result	278	288.07	0.78
Q2 – Single Result	762	121.10	0.55
Q3 – Multiple Results	1038	117.02	1.22

In the case of Q2 and Q3, the use of Elasticsearch leads to significant reduction in the response time in comparison with PostgreSQL: 121.10 ms as against 762 ms for Q2 and 117.02 ms as against 1038 ms for Q3. However, Q1 forms an exception wherein the response time registered by PostgreSQL is 278 ms as against 288.07 ms recorded by Elasticsearch. It can therefore be concluded that the experimental data does not validate the claim that Elasticsearch is always faster than PostgreSQL.

The cached results are significantly lower than those of both PostgreSQL and Elasticsearch directly for all three queries. There is thus a double performance gain in this particular workload: Elasticsearch provides the indexing mechanism to support the cache miss, and Redis helps lower the latency even more by virtue of having the result already computed.

4.5 Discussion of Performance Implications

The above results confirm that Redis caching works best when there is a lot of redundancy or similarity among the incoming requests. When there is high redundancy or similarity among the incoming requests, each hit on the cache eliminates one query from being processed by Elasticsearch

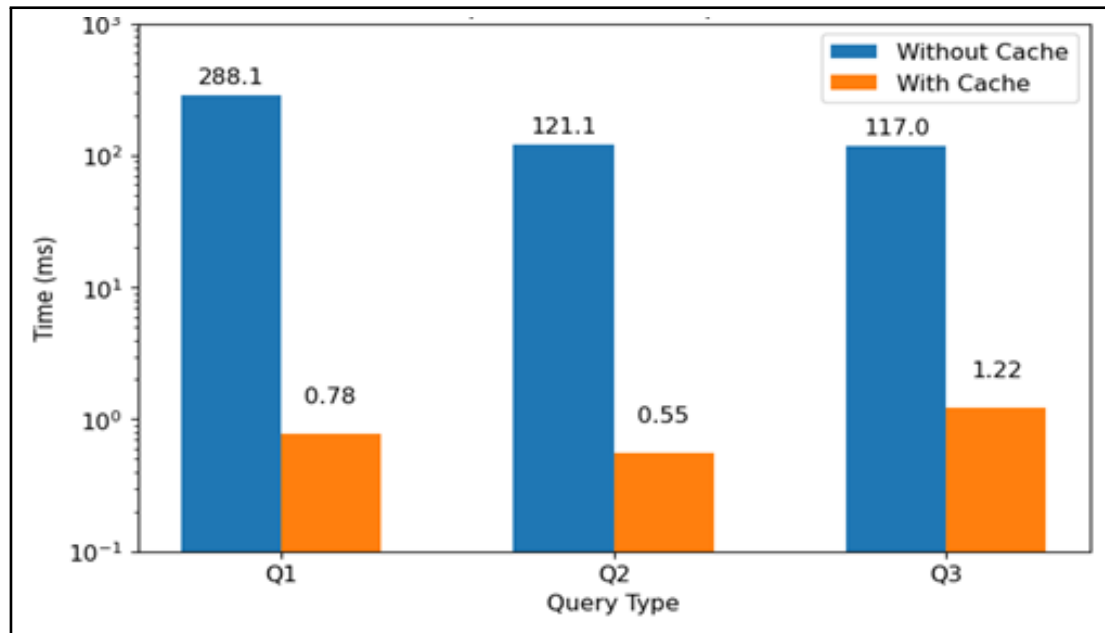


Figure 3. Response time comparison between cached and non-cached queries

The experiment proves the latency gain offered by the cached response for the chosen set of queries (see Figure 3). However, it does not, in itself, prove the performance for arbitrary query distribution, size of the cache, cache expiration policy, and concurrent user load. A query which has not been cached previously still uses the Elasticsearch approach and thus enjoys no sub-millisecond cache hit latency.

Thus, the cache works as an additional acceleration layer without replacing the search engine. The search engine becomes essential for producing responses to the new queries, whereas Redis offers performance gain for the queries whose responses may be cached.

4.6 Overall Findings

From the evaluation results, a decrease in response latency for using Redis caching is evident. The average response time decreases from 175.39 ms without Redis caching to 0.85 ms with caching, which translates into 99.52% decrease. The individual query results also show the same pattern, with decreases from about 98.96% to 99.73%.

PostgreSQL analysis also reveals that using indexed Elasticsearch lookup and Redis query cache can help reduce latency significantly for the analyzed use cases. The main achievement from this experiment, however, is the ability to avoid repeating Elasticsearch queries due to the existence of query cache. Hence, this architecture will be especially useful for applications

where requested searches have already been executed before and can be reused.

In summary, the experimental data clearly support the research goal of decreasing search latency by the use of Elasticsearch and Redis. The obtained data present the quantitative proof that the cache-aside design allows conversion of frequent search queries from search engine processing to low-latency in-memory access while maintaining Elasticsearch as the basis of full-text retrieval in case of cache misses.

5. Conclusion and Future Work

In this research, the use of an application-level Elasticsearch–Redis cache-aside architecture is proposed to minimize the redundant search processing of a big housing-property dataset. In this system, Elasticsearch is used as the main search engine that indexes full-text data, whereas Redis is used to save previous results in order to serve future queries without calling Elasticsearch again. This architecture uses query preprocessing and voice-to-text feature as well, and the same way to process search queries. Through the experiment, it was shown that the latency of the queries is significantly reduced by the use of caching. The reduction percentages obtained from the individual query cases are between 98.96% and 99.73%, which prove that response reuse works for single-answer and multi-answer workloads. The comparison with PostgreSQL also revealed that Elasticsearch performed better response times on the analyzed Q2 and Q3 workloads, while Q1 workload results were generally similar. However, it is especially noteworthy that Redis delivered the lowest latency on all analyzed query cases. Generally speaking, the results obtained confirm that Redis acts as an efficient acceleration component, rather than a substitute for Elasticsearch. The main advantage stems from the fact that Elasticsearch does not have to perform repeatedly on cacheable queries. It will be interesting in future research to analyze larger query workloads, cache expiration policies, cache size, cache hit ratio, and concurrent users.

References

- [1] Y. Zhu, T. Xia, T. Zhu, Z. Zhao, K. Li, and X. Hu, “RAPO: An Automated Performance Optimization Tool for Redis Clusters in Distributed Storage Metadata Management,” *IEEE Access*, vol. 13, pp. 58060–58071, 2025, doi: 10.1109/ACCESS.2025.3556240
- [2] N. Kathare, O. V. Reddy, and V. Prabhu, “A Comprehensive Study of Elasticsearch,” *Journal*

- of Research in Science and Engineering (JRSE), Nov. 2022, vol. 4, no. 11, 34-38
- [3] L. Kaptosv, “Using Redis for Caching Optimization in High-Traffic Web Applications,” *International Journal of Advanced Multidisciplinary Research and Studies* 2025, vol. 5, no. 4, pp. 1714–1722.
- [4] Elasticsearch B.V., “Elasticsearch Documentation,” [Online]. Available: <https://www.elastic.co>
- [5] Redis Labs, “Redis Documentation,” [Online]. Available: <https://redis.io>.
- [6] ReEF. Scholer, H. E. Williams, J. Yiannis, and J. Zobel, “Compression of Inverted Indexes for Fast Query Evaluation,” *Proc. ACM SIGIR*, 2002.
- [7] PolarTech, “500,000+ US Homes Data for Sale (Properties Dataset),” Kaggle,[Online]. Available:<https://www.kaggle.com/datasets/polartech/500000-us-homes-data-for-sale-properties>. Accessed: Apr. 2026.
- [8] E. P. Markatos, “On Caching Search Engine Query Results,” *Computer Communications* 2001, vol. 24, no. 2, pp. 137–143.
- [9] T. Fagni, R. Perego, F. Silvestri, and S. Orlando, “Boosting The Performance of Web Search Engines: Caching and Prefetching Query Results by Exploiting Historical Usage Data,” *ACM Transactions on Information Systems* 2006, vol. 24, no. 1, pp. 51–78.
- [10] R. Lempel and S. Moran, “Predictive Caching and Prefetching of Query Results in Search Engines,” in *Proc. 12th International Conference on World Wide Web (WWW)*, 2003, pp. 19–28.
- [11] X. Long and T. Suel, “Three-Level Caching for Efficient Query Processing in Large Web Search Engines,” *World Wide Web* 2006, vol. 9, no. 4, pp. 369–395.
- [12] R. Ozcan, I. S. Altıngövd, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy, “A Five-Level Static Cache Architecture for Web Search Engines,” *Information Processing Management* 2012, vol. 48, no. 5, pp. 828–840.