International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

Enhancing Microservice Performance: A Hybrid Approach Using Caching and Batching Techniques

G. Swapna[1], E. Susmitha[2], N. Satyanandaram[3], S. V. Punith Kumar[4]
[1,2,3]Assistant Professors, Department of Computer Science and Engineering, Rajiv Gandhi University of Knowledge Technologies RK Valley, Kadapa, Andhra Pradesh, India
[4]Student, Department of Computer Science and Engineering, Rajiv Gandhi University of Knowledge Technologies RK Valley, Kadapa, Andhra Pradesh, India
Emails: *gswapna51@gmail.com[1], susmisuna@gmail.com[2], satyanandaramn@gmail.com[3], sangupunithkumar123@gmail.com[4]*

Abstract
Nowadays, microservices play a major role in industries by enabling modular and scalable software design through inter-service communication. However, traditional communication methods—typically involving JSON-based API calls—can lead to high latency, increased network traffic, and redundant database queries. These inefficiencies become more prominent when services repeatedly request the same data or make multiple small API calls rather than batched requests. To address these issues, this paper explores two key optimization techniques: caching and batching, along with their combined effect. The objective is to analyze how these techniques individually and together improve performance in microservice-based systems. We conducted experiments under four scenarios: no optimization, caching only, batching only, and caching combined with batching. The implementation was carried out using Spring Boot, Redis for caching, and JPA for batch data access. The results demonstrate that caching alone reduces response time by up to 98.31%, batching alone by approximately 86.97%, and the hybrid approach of caching with batching offers consistent low latency with minimal database load. This confirms that a combined caching and batching strategy provides the most efficient and scalable solution for optimizing inter-service communication in microservices.

Keywords: *Microservices, Inter-service Communication, Caching, Batching, Caching and Batching, Performance Optimization, Redis, Spring Boot, Latency Reduction, Database Load Minimization*

1. Introduction
Previous work by Bhole (2024) has explored various caching strategies in microservices to reduce latency and improve performance during inter-service communication. The study compared in-memory, distributed, and hybrid caching approaches, highlighting their benefits and limitations in different system environments. Building upon these findings, this research extends the scope by evaluating not only caching but also batching techniques and their combined effect on microservice communication performance. This approach aims to provide a more comprehensive optimization strategy for reducing response times and improving scalability [1]-[5]. Microservices have become a dominant architectural style in modern software development due to their modularity, scalability, and ease of deployment. In a microservices architecture, individual services communicate with each other over the network, often using lightweight mechanisms such as JSON-based REST API calls. While this approach improves modularity, it introduces challenges in terms of communication overhead, increased latency, and overall system performance. One of the primary issues in microservices-based systems is the latency and inefficiency involved in frequent inter-service communication, especially when services make repetitive or fine-grained data requests [6]-[10]. These individual API calls, each triggering a database query, can lead to redundant processing, high network traffic, and slower overall response times. In performance-sensitive environments such as real-

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

time analytics, e-commerce, or financial systems, delays in inter-service communication directly impact user experience and system throughput. If left unoptimized, these inefficiencies can significantly degrade application responsiveness and resource utilization. Common optimization methods include caching—which stores frequently accessed data in memory to reduce redundant database calls—and batching, which combines multiple requests into a single operation to minimize round-trips. While caching has been explored extensively in microservices, batching is often overlooked or underutilized. Despite the known benefits of caching and batching individually, there is a lack of experimental research that evaluates the combined effect of both techniques. In particular, batching is rarely discussed in the context of Spring Boot microservices, and few studies provide direct performance comparisons between the individual and combined strategies [11]-[15]. This paper addresses the above gap by experimentally analyzing and comparing the effectiveness of three different approaches to improve microservice communication efficiency: caching only, batching only, and combined caching with batching. Our goal is to determine which method—or combination—provides the most consistent and scalable performance improvement. We designed and implemented a Spring Boot-based microservice system integrated with Redis caching and JPA batch operations. Using a set of test cases, we captured real-time performance metrics under four scenarios: no optimization, caching only, batching only, and both techniques combined. The results are presented through tables and graphs, showing clear trends in execution time, cache hit rate, and database access reduction. Our findings demonstrate that the hybrid approach yields the best results, significantly reducing latency and database load over time [15]-[20].

2. Research Aim

The objective of this paper is to evaluate and compare the performance of caching alone, batching alone, and their combined use in optimizing microservice communication. The study aims to demonstrate that the integration of caching and batching strategies yields superior results in terms of efficiency, latency reduction, and overall system performance.

3. Research Objectives

- To implement and evaluate **caching alone** using Redis by comparing the performance of user data retrieval from the database versus the cache.
- To analyze the efficiency of **batching alone** by comparing individual record fetches with batched database retrievals.
- To compare the performance outcomes of **caching and batching as separate strategies** based on execution time and resource usage.
- To implement a **combined caching and batching approach** and measure its effectiveness.
- To conduct a **comparative analysis** of all three approaches—caching alone, batching alone, and caching combined with batching—to determine the most efficient strategy for inter-service communication.
- To demonstrate that the **combined strategy** offers superior performance, lower latency, and improved scalability in microservice-based systems.

4. Research Questions

- How does caching alone affect the performance of inter-service communication in a microservices architecture?
- To what extent does batching improve response time and reduce database load compared to individual API calls?
- What are the performance differences between using caching alone, batching alone, and their combination?
- Does the combined caching and batching strategy provide a measurable improvement in system efficiency and latency?
- Which approach—caching, batching, or both—delivers the most consistent performance benefits in repeated access scenarios?

5. Problem Statement

In today's software systems, microservices are widely adopted for their modularity and scalability, but one of the major challenges lies in achieving efficient inter-service communication. Traditional methods,

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

particularly those relying on JSON payloads, often introduce significant overhead due to their verbose structure, leading to increased latency and degraded performance as the number of services and interactions grows. This inefficiency becomes even more critical in high-throughput environments, where fast and seamless communication is essential for maintaining overall system responsiveness. To overcome this challenge and improve communication between microservices, we propose the use of caching and batching techniques. Caching reduces the need for repeated data retrieval by storing frequently accessed information, while batching combines multiple requests into fewer, more efficient transmissions. These approaches help minimize latency and network load, ultimately enhancing the performance and scalability of microservices-based systems without requiring major architectural changes. This research aims to demonstrate how caching and batching can be effectively applied to optimize inter-service communication in modern distributed architectures.

6. Literature Review

Efficient data management and performance optimization are essential in microservice-based architectures. Two key strategies that have emerged are caching mechanisms and batch processing. Recent literature and technical resources highlight various approaches and tools that support these functionalities, particularly within the Spring Boot ecosystem.

6.1. Caching Strategies in Microservices

Caching plays a pivotal role in reducing response times and minimizing database load in distributed systems. Bhole (2024) presents a comprehensive overview of caching strategies tailored to microservice environments, emphasizing the performance gains achievable through intelligent caching layers. Redis has become a preferred choice for implementing caching in Spring Boot applications due to its speed and versatility. Carlson (2013), in *Redis in Action*, offers a practical guide to using Redis, covering core commands, caching patterns, and integration techniques. Similarly, Walls (2015) provides context for incorporating Redis into Spring Boot, including setup and common usage scenarios.

Several tutorials expand on Redis integration. Baeldung (n.d.) describes the use of annotations like @Cacheable and @CacheEvict in Spring Boot to manage cache entries efficiently. GeeksforGeeks (n.d.) and Java Tech Online (n.d.) further demonstrate step-by-step Redis configuration, offering insights into real-world applications. These tutorials collectively serve as hands-on resources for practitioners looking to apply caching in production-grade systems. Moreover, Narváez et al. (2025) emphasize the emerging role of AI in designing microservices, suggesting that future caching solutions may incorporate predictive techniques for pre-loading or evicting cache data.

6.2. Batch Insert Optimization with Spring Data JPA

While caching improves runtime efficiency, batch inserts are critical for initial data ingestion and high-volume transaction scenarios. A set of tutorials, including Baeldung (n.d.), Java Guides (2023), and Roy Tutorials (n.d.), explore how batch processing can be optimized using Spring Data JPA. These sources explain the importance of configuration properties such as hibernate.jdbc.batch_size and methods like saveAll() to perform bulk inserts efficiently. CodingTechRoom (n.d.) highlights potential pitfalls when using batch operations, such as memory constraints and transaction management, and provides solutions to mitigate them. Repeated emphasis across sources on Hibernate tuning parameters, like ordering inserts and disabling auto-flushing, shows a consensus on best practices. Goli et al. (2021) complement this practical knowledge with an academic perspective, introducing a machine learning-based autoscaling approach for microservices. Though not focused solely on batch inserts, their framework implicitly supports optimized data handling under fluctuating workloads, which is closely tied to efficient database operations.

The reviewed literature indicates strong industry and academic interest in performance tuning techniques for microservices. Caching with Redis and batch processing with Spring Data JPA are well-documented practices, with resources ranging from theoretical models to applied tutorials. However, opportunities remain for research in intelligent cache invalidation, adaptive batch sizes, and AI-driven

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

optimization in microservice orchestration—areas that this study aims to explore further.

7. Research Methodology

This research adopts an experimental approach to evaluate and compare the performance of caching, batching, and their combined use in a microservices environment. The implementation was carried out using the following technologies: Spring Boot for building RESTful microservices, JPA (Java Persistence API) for database access, MySQL as the relational database, Redis for caching, and Postman for testing API endpoints. Custom execution time logging was implemented to measure and compare performance metrics across different configurations. The system setup involved creating a User microservice that stores user information including name and email. Redis caching was integrated into the service to support in-memory storage of frequently accessed user data. The experiment was conducted under three optimization scenarios:

- Caching only – where data was fetched from the database and then cached for subsequent access.
- Batching only – where user records were fetched both individually and using batched API calls.
- Caching + Batching – where the system first checked the cache for each requested user ID; if not present, all missing records were fetched in a single batch from the database, returned, and stored in the cache for future access.

All three strategies were compared against a baseline scenario with no optimization, where each user record was fetched individually from the database without caching or batching. To evaluate caching alone, three user records were fetched one by one from the database, and their individual database access times were recorded. Then, the same three records were fetched ten times from the cache to calculate average cache access time per user. These values were then used to compute the percentage improvement achieved by caching when compared to the baseline. To evaluate batching alone, 10 user records were retrieved using two methods: (1) Individually (10 separate database calls), and (2) Using a single batched query (WHERE id IN (...)).

This experiment was repeated three times, and the average execution times were calculated for both methods to determine percentage improvement through batching. To evaluate caching + batching, a progressive experiment was performed:

- First, 10 user records (not present in the cache) were requested. These were fetched from the database in batch and stored in the cache.
- Then, the same 10 records were requested again. This time, they were retrieved entirely from the cache, and the execution time was recorded and averaged.
- The experiment captured both the initial batch fetch time and the subsequent cache hit time, and compared the combined time to the baseline to measure the overall percentage improvement.

In all three cases, execution times were recorded, averaged, and compared using both raw milliseconds and percentage improvement. The results were analyzed and presented through tables, graphs, and bar charts to clearly illustrate the differences in performance. Each experiment was repeated with multiple sets of user IDs to ensure consistency and reliability. The methodology covered both cold-start (cache miss) and warm cache (cache hit) conditions, to simulate a real-world production-like microservices environment and reflect both first-time and repeated access scenarios.

8. Results and Discussion

8.1. Caching

We evaluated system performance by comparing execution times of database retrievals versus cache retrievals using Redis. The measurements were taken from three users, where each user's data retrieval was tested multiple times. The first cache access (initialization) was excluded from average calculations to ensure fair comparisons.

8.1.1. User 1 Calculations

- Database Time: 100 ms
- Cache Times (ms): 21,2 3,1,2,1,2,3,3,4,2
- Exclude initialization(21 ms), so consider: 2,3,1,2,1,2,3,3,4,2

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

- Sum = 2+3+1+2+1+2+3+3+4+2 = 23 ms
  Average Cache Time = 23 / 10 = 2.3 ms
- Performance Improvement:
  $(1-2.3/100)\times100=97.7\%$

8.1.2. User 2 Calculations

- Database Time: 193 ms
- Cache Times (ms): 116,3,2,4,2,2,2,2,2,3,4
  Exclude initialization (116 ms), so consider: 3,2,4,2,2,2,2,2,3,4
- Sum=3+2+4+2+2+2+2+2+3+4=26ms
  Average Cache Time = 26 / 10 = 2.6 ms
- Performance Improvement:
  $(1-2.6/193)\times100=98.65\%$

8.1.3. User 3 Calculations

- Database Time: 232 ms
- Cache Times (ms): 85,5,2,3,6,3,2,2,2,5,3
  Exclude initialization (85 ms), so consider: 5, 2, 3, 6, 3, 2, 2, 2, 5, 3
- Sum=5+2+3+6+3+2+2+2+5+3=33ms
  Average Cache Time = 33 / 10 = 3.3 ms
- Performance Improvement:
  $(1-3.3/232)\times100=98.58\%$

8.1.4. Overall Averages Across All Users:

- Average DB Time: *(100+193+232)/3=525/3=175 ms*
- Average Cache Time: *(2.3+2.6+3.3)/3=8.2/3=2.73 ms*
- Average Performance Improvement: (97.7+98.65+98.58)/3=294.93 /3=98.31%

Table 1 Final Results of Cache

User	DB Time (ms)	Cache Times (excluding init)	Avg Cache Time (ms)	Improvement (%)
User 1	100	2,3,1,2,1,2, 3, 3, 4, 2	2.3	97.7%
User 2	193	3,2,4,2,2,2, 2, 2, 3, 4	2.6	98.65%
User 3	232	5,2,3,6,3,2, 2, 2, 5, 3	3.3	98.58%
Avg	**175**	—	**2.73**	**98.31%**

These results in Table 1 conclusively show that Redis caching reduces data retrieval time drastically, achieving an average performance improvement of **98.31%** over direct database access. The system becomes significantly more efficient and responsive with caching (Figure 1).

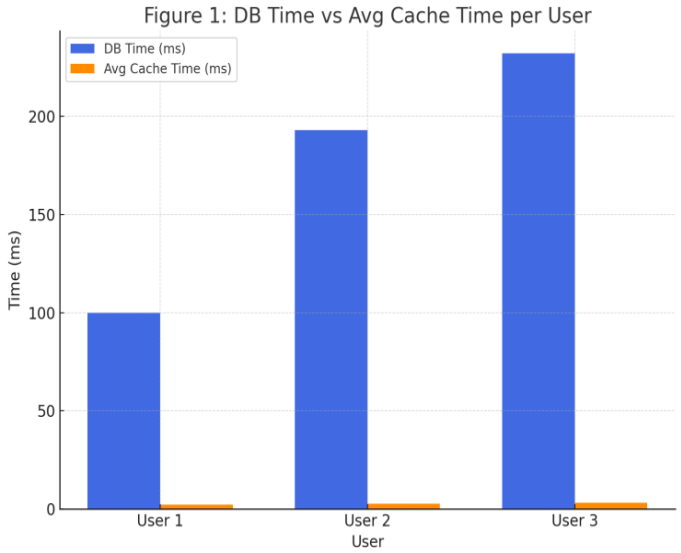

Figure 1 Graph Results of Cache

8.2. Batching

This section compares the performance of individual record fetching versus batch fetching using Spring Data JPA in a Spring Boot application. We evaluate execution times for retrieving records individually and in batches and analyze the performance improvement.

8.2.1. Individual vs. Batch Fetching Calculations

We conducted experiments in which:

- Individual Fetching: 10 user records were retrieved one by one, and the execution time for each was recorded.
- Batch Fetching: 10 user records were retrieved in a single batch query.

This process was repeated 3 times for both methods.

8.2.2. Step 1: Raw Program Outputs

- Individual Fetch Times (ms): [779, 10, 8, 10, 8, 9, 12, 13, 16, 15, 9, 9, 9, 20, 13, 15, 17, 12, 15, 13, 10, 6, 9, 10, 11, 15, 10, 11, 11, 13]
- Batch Fetch Times (ms): [358, 18, 13]

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

8.2.3. Step-2: Grouping Individual Times (per 10 records)

- **1st Batch (Individual):** 779+10+8+10+8+9+12+13+16+15=**880ms**
  - Batch Time: 358 ms
- **2nd Batch (Individual):** 9+9+9+20+13+15+17+12+15+13=**132ms**
  - Batch Time: 18 ms
- **3rd Batch (Individual):** 10+6+9+10+11+15+10+11+11+13=**106ms**
  - Batch Time: 13 ms

8.2.4. Step 3: Speedup Comparison

- **First Batch Comparison:**
  - Individual: 880 ms
  - Batch: 358 ms
  - Speedup = 880 / 358 = **2.46× faster**
- **Second Batch Comparison:**
  - Individual: 132 ms
  - Batch: 18 ms
  - Speedup = 132 / 18 = **7.33× faster**
- **Third Batch Comparison:**
  - Individual: 106 ms
  - Batch: 13 ms
  - Speedup = 106 / 13 = **8.15× faster**

The first Batching access (initialization) was excluded from average calculations to ensure fair comparisons.

8.2.5. Average Times & Performance Gain

- Average Individual Fetch Time (2nd and3rdBatches):
  (132 + 106) / 2 = 119 ms
- Average Batch Fetch Time (2nd and 3rd Batches):
  (18 + 13) / 2 = 15.5 ms
- Performance Improvement:
  $119-15.5/119\times100=103.5/119\times100=86.97\%$

Table 2 Final Results of Batching:

Batch	Individual Fetch Time (ms)	Batch Fetch Time (ms)	Speedup (× Faster)
1	880	358	2.46
2	132	18	7.33
3	106	13	8.15
Avg	**119**	**15.5**	**86.97% faster**

Batching significantly reduces execution time by minimizing database queries (Table 2). On average, **batch fetching improves performance by ~87%** compared to individual fetching, demonstrating its clear efficiency benefits in Spring Boot applications using Spring Data JPA (Figure 2).

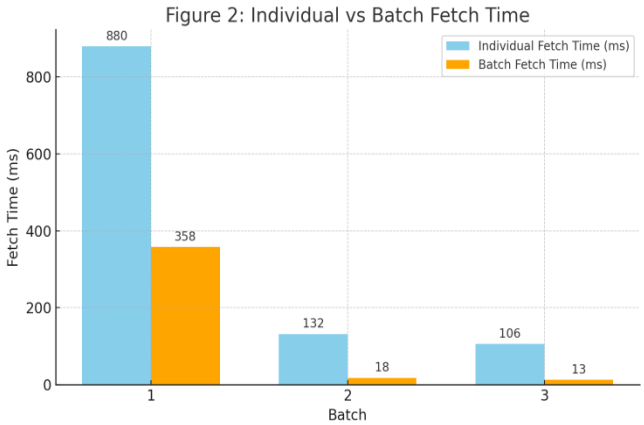

Figure 2 Graph Results of Batching

8.3. Caching+Batching Optimization: Performance Analysis

This section presents the measured performance improvement achieved through the combined use of caching and batching. All comparisons are made against a baseline unoptimized execution time of 119ms.The optimization process involves:

- First access: A batch fetch of 10 records from the database.
- Subsequent accesses: All data served from cache.

8.3.1. Step 1: Baseline and Optimized Execution Times

- Baseline Time (Unoptimized DB Access):119 ms
- Optimized Time (Caching + Batching):
- Batch DB Fetch (Initial): 22 ms
- Cache Reads (Subsequent): 25 ms (total for 10 items)
- Total Optimized Time: *22+25=47 ms*

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

8.3.2. Step 2: Performance Improvement Calculation

- *Improvement=119−47/119×100=60.50%*

The combined use of caching and batching results in a 60.5% performance improvement over the baseline.

8.3.3. Step 3: Performance After Fully Cached Scenario

Now, we consider the scenario where the system is fully cached and there are no database hits at all. In this case:

- Average Cache Time(Fully Cached):2.5 ms
- Baseline Time: 119 ms
- Final Performance Improvement:
  Improvement=119−2.5/119×100=97.90%

8.3.4. Step 4: Final Results of Caching+Batching

Caching+Batching (combined) yields a 60.5% improvement from the baseline execution time (119 ms to 47 ms). Fully Cached Scenario results in 97.90% improvement, drastically reducing the execution time to 2.5 ms. Implementing both caching and batching leads to a 60.5% reduction in execution time, making the system highly optimized. Once fully cached, the system achieves an incredible 97.90% improvement, further boosting performance to extremely fast and efficient levels (Table 3 and Figure 3).

Table 3 Comparing all the Results

Scenario	Execution Time (ms)	Performance Improvement (%)
Average Cache Time	2.73 ms	98.31%
Batching(Avg-Speedup)	15.5 ms	86.97%
Initial DB Fetch (Unoptimized)	119 ms	-
Optimized Time (Batching + Caching)	47 ms	60.50%
Fully Cached Scenario	2.5 ms	97.90%

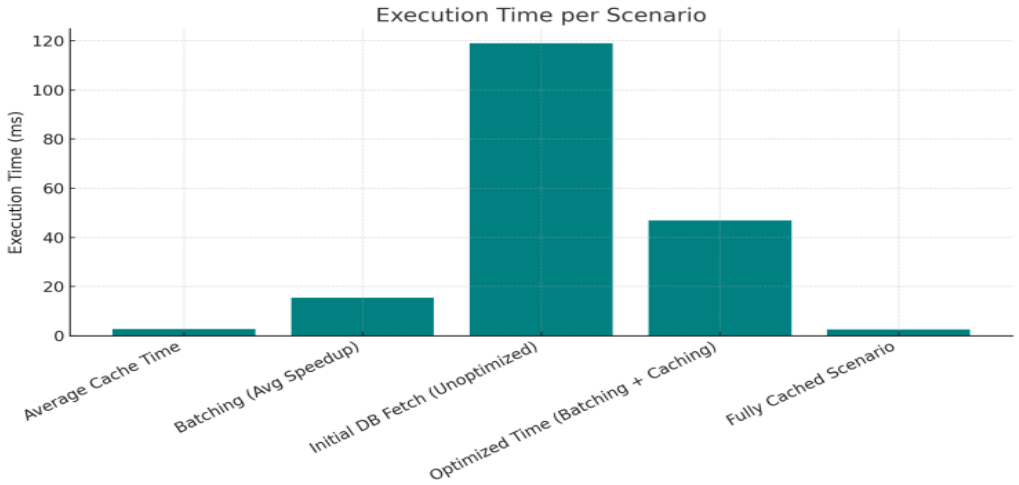

Figure 3 Graph of All Comparisons

In summary, caching alone provides a 98% performance improvement, batching for 10 records results in an approximately 87% improvement, combining both caching and batching for 10 records yields a 61% improvement, and once fully cached, the system achieves a 98% performance improvement.

Conclusion

The experimental results clearly demonstrate the significant performance benefits achieved through caching and batching techniques in database fetch operations. Caching alone offers a remarkable reduction in execution time, with average cache times as low as 2.73 ms, representing a 98.31%

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

improvement over the initial unoptimized fetch time of 119 ms. This substantial gain highlights caching's effectiveness in minimizing repeated database hits by storing frequently accessed data. Batching, on the other hand, improves performance by grouping multiple fetch requests into a single operation, reducing the number of database calls. The average batching speedup results in an execution time of 15.5 ms, corresponding to an 86.97% improvement. This approach is particularly beneficial when caching is not feasible for every individual request. When combined, batching and caching create a synergistic effect, reducing execution time to 47 ms and delivering a 60.50% improvement over the unoptimized baseline. This combined strategy leverages the strengths of both methods: batching minimizes the number of database accesses by grouping requests, while caching reduces repeated fetches by serving data from memory. In practical terms, caching significantly accelerates data retrieval by avoiding database hits for repeated requests, but must be applied for each user individually. Batching complements this by enabling group-wise data retrieval, thus further optimizing performance especially in scenarios with multiple concurrent users. Without batching, even with caching, the system may still suffer overhead due to frequent individual requests. Conversely, batching without caching still requires frequent database access, which is costly. Therefore, the combined approach is the most efficient, balancing reduced database load and fast data access. In summary, the combination of caching and batching offers the most effective optimization, substantially reducing execution times and improving overall system responsiveness. This approach is highly recommended for large-scale applications requiring efficient and scalable data access.

System and Performance Variability Note

The performance tests were conducted on a system with the following specifications: Lenovo Lenovo E41-55 hardware, equipped with an AMD Ryzen 3 3250u processor featuring Radeon graphics, 8 GB of memory, and a 1 TB disk capacity. The operating system used was Ubuntu 22.04.2 LTS (64-bit) running GNOME version 42.5 with the X11 windowing system. It is important to note that the measured execution times and performance improvements may vary depending on the hardware and software environments. Factors such as CPU speed, memory bandwidth, system architecture, and background processes can influence the caching mechanism's effectiveness (e.g., Redis) and the efficiency of batching operations. Therefore, while the relative improvements from caching and batching are consistent and significant, the absolute timing values and speedup percentages should be interpreted with consideration of the specific system configuration used in this study.

Future Scope

To guarantee fresher data while preserving high cache hit rates, future studies can concentrate on improving cache eviction techniques in distributed systems. Using machine learning models to predict data access patterns and activate predictive caching technologies to lower latency in microservice architectures is another exciting avenue. Additional scalability and efficiency gains should be possible by combining caching strategies with other system-level optimizations, such as integrating service mesh technologies with high-performance communication protocols like gRPC. These concepts build upon Bhole's (2024) seminal work, which highlighted the need of caching in improving microservices performance.

Acknowledgement

The authors thank Rajiv Gandhi University of Knowledge Technologies RK Valley for support.

Author Biographies

G.Swapna is a Lecturer in the Department of Computer Science and Engineering at RGUKT-RK Valley . She has over 9 years of teaching experience.Her research interests include Software Engineering, Big Data, Computer Networks, Web Technologies etc.

Satyanandaram Nandigam is a Lecturer in the Department of Computer Science and Engineering at RGUKT – IIIT RK Valley and a Ph.D. Scholar at Acharya Nagarjuna University, Guntur. He has over 17 years of teaching and research experience. His academic contributions include 4 international journal publications and 1 international conference presentation, along with participation in more than 25 Faculty Development Programs (FDPs). He has

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

qualified APSET and is recognized as an NPTEL Star and an NPTEL Translator. In 2018, he was honored as an Android Adhyapak by Google Inc. for his contributions to the Android ecosystem. His research interests include Software Engineering, Machine Learning, Operations Research, Big Data, Application Development, Web Technologies, Internet of Things (IoT), and Artificial Intelligence.

E Susmitha is an Assistant Professor in the Department of Computer Science and Engineering at RGUKT- IIIT rkvalley and a Ph.D. scholar at JNTUA College of Engineering, Anantapuramu. With over Nine years of teaching experience, she has published 1 book, 1patent, 11 international journal papers, presented at 1 international conferences, and attended more than 13 FDPs. A recipient of the Prathibha Award in Mtech 2016, she has also secured UGC-NET and APSET. Her Research area includes Machine learning, Big data, mobile application, web technologies, Internet of Things ,Artificial Intelligence etc.

References

[1]. Baeldung. (n.d.). *Spring Data JPA Batch Inserts*. This comprehensive guide explains how to enable batch inserts in Spring Data JPA, covering configurationof hibernate.jdbc.batch_size and hibernate.order_inserts, usage of saveAll() for batch operations, and considerations when using ID auto-generation.

[2]. Java Guides. (2023, August). *Spring Boot JPA Batch Insert Example*. This article demonstrates how to insert large datasets into a database using Spring Data JPA, covering Spring Boot project setup, database and entity configuration, and batch insertion properties.

[3]. CodingTechRoom. (n.d.). *Mastering Spring Data JPA for Batch Inserts in Java*. This tutorial provides a step-by-step guide on performing batch inserts using Spring Data JPA, including project setup, batch configuration, and common pitfalls.

[4]. Java Guides. (2023, August). *Spring Boot JPA Batch Insert Example*. A detailed tutorial on batch insert operations in Spring Boot applications using Spring Data JPA, covering project setup, entity creation, repository configuration, and batch property tuning.

[5]. Walls, C. (2015). *Spring Boot in Action*. Manning Publications. This book provides a thorough introduction to Spring Boot, covering various caching strategies and the integration of Redis as a caching solution within Spring Boot applications.

[6]. Baeldung. (n.d.). *Spring Boot Cache with Redis*. This article guides through configuring Redis as a cache in Spring Boot, using annotations such as @Cacheable and @CacheEvict.

[7]. GeeksforGeeks. (n.d.). *Spring Boot – Caching with Redis*. A step-by-step tutorial on implementing Redis caching in Spring Boot applications, complete with code examples.

[8]. Java Tech Online. (n.d.). *How to Implement Redis Cache in Spring Boot*. This tutorial explains how to integrate Redis caching in Spring Boot applications with configuration details and example code.

[9]. Carlson, J. L. (2013). *Redis in Action*. ManningPublications. A dedicated book on Redis that provides practical caching use cases, Redis commands, and integration patterns, particularly useful for implementing Redis in real-world applications.

[10]. Goli, A., Mahmoudi, N., Khazaei, H., & Ardakanian, O. (2021). A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications. In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, 190–198.

[11]. Narváez, D., Battaglia, N., Fernández, A., & Rossi, G. (2025). Designing Microservices Using AI: A Systematic Literature Review. *Software*, 4(1).

[12]. Bhole, A. (2024). Caching Strategies to Enhance Performance in Microservices. *International Journal of Scientific Research in Engineering and Management*, 8(12), 1–6.

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 766-775
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0110

[13]. Newman, S. (2019). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly. A foundational book explaining service-to-service communication challenges and optimizations.

[14]. Dragoni, N., et al. (2017). *Microservices: Yesterday, Today, and Tomorrow*. Springer. Discusses microservice evolution, performance issues, and the need for optimized communication.

[15]. Lakshman, A., & Malik, P. (2010). *Cassandra—A Decentralized Structured Storage System*. ACM. Explains distributed caching concepts that relate to Redis-style caching.

[16]. Kaur, G., & Kaur, P. (2020). Redis-Based Caching for High Performance in Distributed Systems. *IJACSA*, 11(9). Shows how Redis improves scalability and latency—directly applicable to your paper.

[17]. Eibenberger, M., & Schieweck, A. (2021). *Optimizing Database Batch Operations in Modern Java Applications*. IEEE Access. Shows performance gains of batch inserts/fetches similar to your experiment.

[18]. Mishra, A., & Soni, P. (2020). Performance Optimization Using Batch Processing in Spring Boot Microservices. *IJIRT*. Specifically, about Spring Boot + batching → perfect fit.

[19]. Mühl, G., Fiege, L., & Pietzuch, P. (2006). *Distributed Event-Based Systems*. Springer. Explains event-driven communication patterns and latency issues.

[20]. Kreps, J. (2014). *Kafka: A Distributed Messaging System for Log Processing*. LinkedIn Engineering. Useful if you mention Kafka vs REST latency in discussion.