MULTIDISCIPLINARY RESEARCH INSTITUTE INDIA

Multidisciplinary Journal of Research in Engineering and Technology

ISSN: 2348-6953

Volume 13 Issue 02 (2026) | Pages: 399-403

Optimizing SQL Query Performance in Spring Boot Applications: Best Practices and Techniques

Swapnaja Patwardhan[1], Ronit Dalal[2], Bhav Bothare[3], Naman Gupta[4]

[1,2,3,4]Department of MCA, MES' IMCC, Pune

[1]*sp.imcc@mespune.in*, [2]*dalalronit131@gmail.com*, [3]*bhav9bothare@gmail.com*, [4]*nmgupta459@gmail.com*

Peer Review Information

***Type**: Article*
***Received**: 20 March 2026*
***Revised**: 03 April 2026*
***Accepted**: 21 May 2026*
***Published**: 03 June 2026*

Abstract

Modern applications heavily depend on relational databases for storing and managing structured data. In backend systems developed using frameworks such as Spring Boot, the efficiency of database interactions has a significant impact on overall system performance. Poorly designed SQL queries may lead to higher response time, higher server load, and inefficient utilization of resources.

This study focuses on exploring different techniques to improve SQL query performance in Spring Boot applications. Including issues such as the N+1 query problem, inefficient joins, lack of proper indexing, and redundant data retrieval. Various optimization strategies, including indexing, query restructuring, pagination, caching mechanisms, and connection pooling, are examined in detail.

The research also highlights the importance of using performance monitoring tools such as SQL execution plans and Hibernate statistics to detect inefficiencies. The results indicate that applying these techniques improves system performance noticeably and response time. The study concludes that careful query design is essential for building scalable and efficient backend applications.

Keywords: SQL Query Optimization; Spring Boot; Hibernate; Indexing; N+1 Problem; Pagination; Caching; Database Performance; JPA; Backend Development.

How to Cite This Article

Patwardhan, S., Dalal, R., Bothare, B., & Gupta, N. (2026). Optimizing SQL query performance in Spring Boot applications: Best practices and techniques. *Multidisciplinary Journal of Research in Engineering and Technology*, 13(2), 399–403.

Introduction

In today's digital landscape, the demand for fast and responsive web applications has grown significantly. As a result, efficient backend processing has become a critical requirement. Frameworks like Spring Boot have simplified backend development by offering features such as dependency management, RESTful API support, and seamless database integration.

Spring Boot applications commonly use technologies like Hibernate and Spring Data JPA to interact with relational databases. These tools help developers work with databases using object-oriented concepts instead of writing complex SQL queries manually. However, while these abstractions improve productivity, they can sometimes generate inefficient queries if not handled carefully.

As applications grow and data volume increases, inefficient database operations can negatively affect performance. Issues such as excessive database calls, poorly structured joins, and missing indexes often lead to slower query execution and increased system load. Earlier research indicates that optimizing database queries is essential for maintaining application performance and scalability (Stonebraker and Cetintemel) [3]. Therefore, understanding how to design efficient queries is important for developing high-performance backend systems.

This research aims to analyze common SQL performance issues in Spring Boot applications and suggest practical techniques to overcome them. In real-world applications, database performance optimization is often overlooked during early development stages, which later leads to scalability challenges.

Literature Review

Database performance optimization has gained significant attention in both academic research and industry practices.

Stonebraker and Cetintemel discussed the evolution of database systems and highlighted the importance of optimizing query execution for handling large-scale data efficiently [3]. Their work highlights how inefficient queries can reduce system throughput.

Chaudhuri and Narasayya introduced an automated index selection mechanism that helps improve query performance in relational databases [4]. Their research demonstrated that proper indexing strategies can significantly reduce execution time.

Li et al. analyzed the performance behavior of ORM frameworks and identified the N+1 query problem as a major issue affecting database efficiency [5]. They suggested using optimized joins and better query design to overcome this problem.

Mihalcea explored advanced performance tuning techniques for Hibernate-based applications and recommended using fetch joins, batch processing, and caching to reduce database overhead [6].

Kumar and Patel examined SQL optimization techniques in web applications and highlighted the importance of pagination and selective data retrieval [7].

Zhang et al. studied caching strategies and found that caching frequently accessed data can greatly enhance application performance [8].

Gupta and Sharma focused on connection pooling techniques and demonstrated how efficient connection management improves system scalability in high-traffic environments [9].

Ghodake et al. emphasized the importance of backend architecture and efficient database design in improving application performance [10].

Collectively, these studies suggest that indexing, caching, efficient query design, and proper connection handling are essential for building high-performance systems.

Problem Statement

Although Spring Boot simplifies backend development, many applications still face performance issues due to inefficient database interactions.

A key factor contributing to this issue is the automatic query generation performed by ORM frameworks such as Hibernate. While these frameworks reduce development effort, developers may not always be aware of the exact SQL queries being executed behind the scenes.

Common performance challenges include:

- The N+1 issue occurs when multiple additional queries are executed unnecessarily
- Lack of proper indexing, leading to full table scans
- Retrieval of large datasets without filtering
- Inefficient joins that increase processing overhead
- Fetching complete entities instead of only required fields

These problems increase database workload and reduce application responsiveness. Research suggests that poorly optimized queries are one of the major reasons behind performance degradation in database-driven systems (Hellerstein and Stonebraker).

Objectives

The main objectives of this research are:

- To identify common SQL performance bottlenecks in Spring Boot applications
- To analyze the impact of inefficient queries on system performance
- To examine various methods for improving query efficiency
- To evaluate optimization strategies through practical implementation
- To provide useful practical suggestions for developers working with database-driven systems

Methodology

To evaluate SQL query optimization techniques, A sample Spring Boot application was developed.

The following steps were followed:

- A Spring Boot application was created using Spring Data JPA
- MySQL was used as the database system
- Entities such as User, Product, and Order were designed
- Sample data was generated to simulate real-world scenarios
- Performance was monitored using tools such as:
  - Hibernate Statistics
  - SQL Execution Plans (EXPLAIN)
  - Spring Boot Actuator

These tools helped in identifying inefficient queries and analyzing the impact of optimization techniques.

SQL Query Optimization Techniques

Several techniques were implemented to improve database performance.

Indexing

Indexing improves database performance by enabling faster data retrieval and avoiding full table scans.

Example:

Without index:

```
SELECT * FROM users WHERE email = 'user@example.com';
```

With index:

```
CREATE INDEX idx_user_email ON users(email);
```

This reduces the time required to locate records.

Avoiding N+1 Query Problem

The N+1 issue arises when one query retrieves parent data and additional queries fetch related data.

Example:

```
SELECT * FROM users;
SELECT * FROM orders WHERE user_id = 1;
SELECT * FROM orders WHERE user_id = 2;
```

Optimized query:

```
SELECT u FROM User u JOIN FETCH u.orders;
```

This approach retrieves all required data in a single query.

DTO Projection

DTO projections help reduce unnecessary data transfer by fetching only required fields.

Example:

SELECT name, price FROM products;

This reduces memory use and also it improves performance.

Pagination

Pagination limits the number of records retrieval using a single SQL query.

Example:

```
Pageable pageable = PageRequest.of(0, 10);
Page<User> users = userRepository.findAll(pageable);
```

This is useful when handling large datasets.

Caching

Caching keeps frequently used data in memory, reducing repeated database queries.

Common tools include:

- Redis
- EhCache
- Hazelcast

Connection Pooling

- Connection pooling improves performance by reusing database connections instead establishing new connections for each request.
- Spring Boot uses HikariCP for efficient connection management.

Experimental Results

A prototype application was implemented to evaluate the effectiveness of the proposed.

Initially, multiple queries were executed due to lazy loading.

Before optimization:

- Multiple queries executed
- Average response time: 900 ms

After optimization:

- Reduced query count
- Response time improved to 480 ms

This shows an improvement of approximately 45–50%.

The results says that combining multiple optimization techniques will lead to provide better performance than using a single approach.

Conclusion

Efficient query design is essential for building scalable backend systems. Poorly optimized queries can lead to increased latency and higher resource consumption. The study shows that techniques such as indexing, caching, pagination, and query restructuring can significantly improve SQL query performance in Spring Boot applications.

It also highlights the importance of analyzing queries generated by ORM frameworks instead of relying entirely on abstraction. Developers should actively monitor and optimize database operations to ensure better system performance. Further research may focus on advanced techniques such as distributed databases, sharding, and AI-based query optimization. Combining multiple optimization strategies generally provides better results compared to relying on a single technique.

References

1. "One Size Fits All: An Idea Whose Time Has Come and Gone," IEEE International Conference on Data Engineering, 2005.
2. "Readings in Database Systems," MIT Press, 2005.
3. "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," VLDB Conference, 1997.
4. "Performance Analysis of ORM Frameworks in Web Applications," International Journal of Computer Applications, 2018.
5. "High Performance Java Persistence," 2017.
6. "SQL Query Optimization Techniques for Web Applications," International Journal of Computer Science and Engineering, 2019.
7. "Database Caching Techniques for High Performance Applications," IEEE Transactions on Knowledge and Data Engineering, 2020.
8. "Efficient Database Connection Pooling for Web Systems," International Journal of Advanced Computer Science and Applications, 2018.
9. "Efficient Query Processing in Modern Database Systems," VLDB Journal, 2011.
10. "Backend Development for E-Commerce Application," 2024 8th International Conference on Computing, Communication, Control and Automation (ICCUBEA), IEEE, 2024.
11. "Optimizing Database Queries for Large Scale Web Applications," IEEE Software Engineering Conference, 2021.
12. "Query Optimization Techniques for Modern Databases," ACM Computing Surveys, 2018.