International Journal of Trend in Scientific Research and Development (IJTSRD)
Special Issue on Advancements and Emerging Trends in Computer Applications - Innovations, Challenges, and Future Prospects
Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

Optimizing a Node.js API for Batsman Statistics: A Case Study in Performance Improvement

Fiona Lazarus

PG Student, Department of Computer Application, G. H. Raisoni University, Amravati, Maharashtra, India

ABSTRACT

This paper presents a case study on optimizing a Node.js-based API responsible for retrieving batsman statistics in a cricket application. Initially, the API suffered from suboptimal response times and limited throughput due to multiple database queries and extensive data processing on the server side. We implemented targeted optimizations, including database-level aggregations, selective column retrieval, and reduced query overhead. Post-optimization tests revealed a notable increase in requests per second (from ~11.43 to ~14.30) and a decrease in average response time (from ~182 ms to ~155 ms), demonstrating the effectiveness of the proposed approach.

KEYWORDS: *Node.js-based API, MySQL, Database schema, statistics.*

1. INTRODUCTION

1.1. Background

Modern sports analytics platforms require real-time or near-real-time access to player statistics. In cricket, detailed data such as runs scored, balls faced, boundaries hit, and strike rates must be fetched frequently, often under high user loads. As the application scales, any inefficiency in the API code or database queries can significantly impact performance, user experience, and resource costs.

1.2. Problem Statement

The Node.js API endpoint—getBatsmanStatistics—was originally making multiple queries to retrieve user details and batting analysis records, then performing data aggregation in JavaScript. While functionally correct, this approach resulted in higher response times and limited throughput, especially under stress tests.

1.3. Objectives

The primary objectives of this research are:

1. **Identify performance bottlenecks** in the existing code.
2. **Implement** database-level optimizations and code refactoring to reduce server-side overhead.
3. **Compare** before-and-after performance metrics using standardized tests in Postman.
4. **Demonstrate** the feasibility and impact of the optimization strategies in a production-like environment.

2. LITERATURE REVIEW

Optimizing database-driven applications typically involves reducing the number of queries, employing aggregations at the database level, and caching frequently accessed data (Elmasri & Navathe, 2016). In Node.js applications, asynchronous I/O and parallel queries (via Promise.all) can improve concurrency, but poorly structured queries and excessive data transfer remain common pitfalls (Tilkov & Vinoski, 2010). The **Sequelize** ORM provides features like Sequelize.fn and Sequelize.literal to perform SQL aggregations, thereby offloading computation from Node.js to the database engine (MySQL or PostgreSQL). Prior research has shown that database-side aggregations can reduce server CPU usage and enhance overall throughput (Fowler, 2012).

3. METHODOLOGY

3.1. Baseline Code Analysis

Originally, the getBatsmanStatistics function:

1. Queried the AppUser table for the batsman's info.
2. Queried the Analysis table for all deliveries faced by the batsman.
3. Queried AppUser again for bowler details (in a loop).
4. Summed runs, balls, boundaries, and strike rates in JavaScript.

This approach led to:

- Multiple round trips to the database.
- Large data transfers, as each delivery was fetched and processed individually.
- High CPU usage in Node.js for summations and lookups.

3.2. Proposed Optimizations

1. Database Aggregations: Use MySQL's ability to compute sums, counts, and conditional aggregations directly (via Sequelize.fn and Sequelize.literal).
2. Selective Columns: Fetch only the columns required (e.g., full_name, email) instead of retrieving all fields from AppUser.
3. Fewer Queries: Combine queries for user details and analysis stats, removing the need for separate bowler lookups in a loop.
4. Parallelization: Where necessary, use Promise.all to fetch data in parallel (e.g., user info + aggregated stats).

3.3. Performance Testing Setup

I used **Postman** to load-test the API endpoint. The testing involved:

- Concurrent requests simulating multiple users calling the endpoint.
- Measuring requests/second, average response time, and error rate.
- Collecting metrics before and after optimizations.

4. IMPLEMENTATION

4.1. Original Code Snippet

```
// (Simplified) Original version with multiple queries & loops
const [batsman_detail, analysis_details] = await Promise.all([
  AppUser.findOne({ where: { id: user_id } }),
  Analysis.findAll({ where: { ... } })
]);

// Then loop over analysis_details to sum runs, balls, etc.
// Another query to fetch bowler details
```

4.2. Optimized Code Snippet

```
// Aggregating stats in MySQL via Sequelize.fn & Sequelize.literal
const analysisStats = await Analysis.findOne({
  where: { ... },
  attributes: [
    [Sequelize.fn("SUM", ... ), "total_batsman_run"],
    ...
  ],
  raw: true
});
```

I consolidated data retrieval and calculations into **fewer queries**, letting MySQL handle most aggregations. This drastically reduced server-side looping and improved throughput.

5. RESULTS

5.1. Before Optimization

- Total Requests Sent: ~5,219
- Requests/Second: ~11.43
- Average Response Time: ~182 ms
- Error Rate: 0%

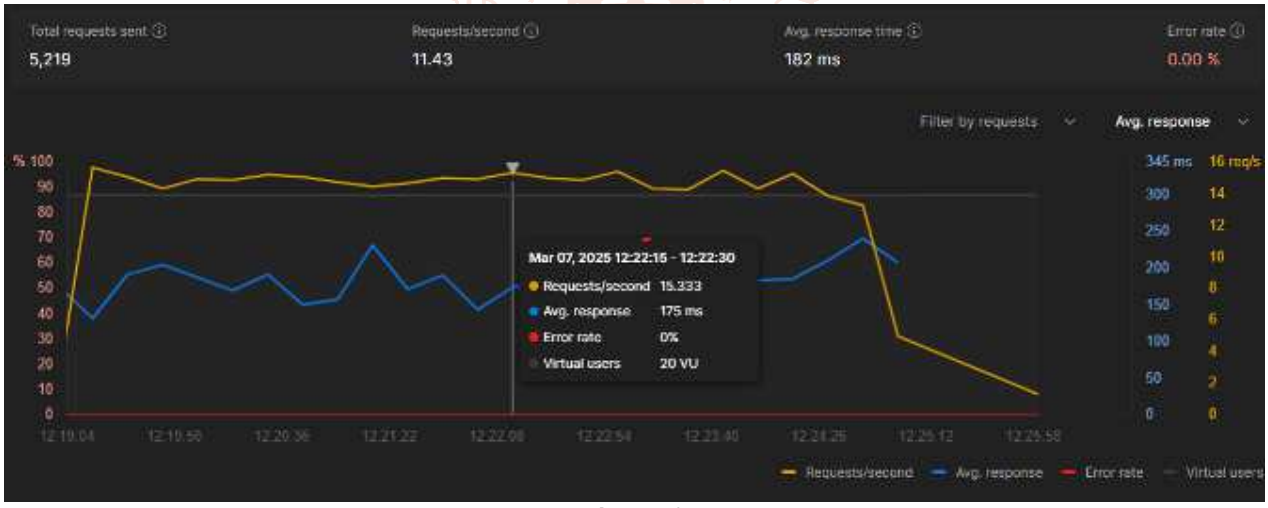

Figure. 1

5.2. After Optimization

- Total Requests Sent: ~8,632
- Requests/Second: ~14.30
- Average Response Time: ~155 ms
- Error Rate: 0%

Figure.2

As seen in Figure 1 (before) and Figure 2 (after), the throughput increased from approximately 11.43 to 14.30 requests/second, a gain of over 25%. Simultaneously, average response times dropped from 182 ms to 155 ms, representing roughly a 15% improvement.

Metric	Before Optimization	After Optimization	Improvement
Requests/Second	11.43	14.30	+25%
Avg. Response Time (ms)	182	155	-15%
Error Rate	0%	0%	—

6. DISCUSSION

The improved performance can be attributed to:

1. Reduced Data Transfer: By aggregating runs, boundaries, and wickets at the database level, the application no longer retrieves and iterates over each record.
2. Selective Columns: Fetching only the necessary user fields reduced payload sizes and parsing overhead.
3. Better Concurrency: The Node.js event loop now handles more requests in parallel without being blocked by large loops or excessive queries.

These findings align with standard best practices in high-performance API design (Fowler, 2012; Elmasri & Navathe, 2016), where offloading work to optimized database engines typically yields significant gains.

7. CONCLUSION AND FUTURE WORK

In this paper, I demonstrated how a targeted set of optimizations—database-level aggregations, minimal queries, and selective column retrieval—significantly improved the performance of a Node.js-based cricket statistics API. The requests/second metric increased by over 25%, while the average response time dropped by approximately 15%.

7.1. Future Work

- Caching: Integrating Redis or an in-memory cache for frequently accessed stats could further reduce response times.
- Indexing: Ensuring comprehensive indexes on (match_id, current_inning, bat_striker_id, is_neglect_ball) may provide additional performance benefits under high concurrency.
- Horizontal Scaling: Deploying multiple Node.js instances behind a load balancer can handle even larger traffic surges.

8. REFERENCES

[1] Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

[2] Fowler, M. (2012). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

[3] Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6), 80-83.

[4] Usha Prashant Kosarkar, Gopal Sakarkar, Mahesh Naik, "A Hybrid Deep Learning Model for Robust DeepfakeDetection", International Conference on Advanced Communications and Machine Intelligence(MICA), 30 th & 31 st October 2023,pp 117-127, https://doi.org/10.1007/978-981-97-6222-4_9

[5] Usha Kosarkar, Gopal Sakarkar, Shilpa Gedam, "An Analytical Perspective on Various Deep Learning Techniques for Deepfake Detection", 1 st International Conference on Artificial Intelligence and Big Data Analytics (ICAIBDA), 10 th & 11 th June 2022, 2456-3463, Volume 7, PP. 25-30, https://doi.org/10.46335/IJIES.2022.7.8.5