JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 37 (2025) 426–430
Received: 18 June 2025
Accepted: 19 August 2025

Analysis of ORM framework approaches for Node.js

Serhii Zhadko-Bazilevych*

Department of Computer Science, Kharkiv National University of Radioelectronics, Nauky Ave, 14, Kharkiv, Kharkiv Oblast, 61166, Ukraine

Abstract

This work analyzes the performance of three ORM frameworks for Node.js Sequelize, Prisma, and TypeORM under different database interaction modes: single cached and uncached queries, as well as parallel load. Testing was conducted across various usage scenarios using a simple online store system backed by a PostgreSQL database. Collected data provides insights into how each ORM behaves under different conditions and may be helpful when selecting a tool for working with databases. The results show that Prisma provides the best performance under parallel load, while Sequelize performs efficiently in single-query scenarios with low concurrency. TypeORM demonstrated stable behavior across all modes and supports more advanced features such as hierarchical data processing.

Keywords: ORM performance analysis; Node; Sequelize; Prisma; TypeORM

*Corresponding author

Email address: **serhii.zhadkobazilevych@gmail.com** (S. G. Zhadko-Bazilevych)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

Databases serve as fundamental components in information systems by providing structured storage and management of data. The integrity, availability, and consistency of stored data are critical for the correct functioning of software applications across various domains.

The interaction between a backend application and a database is quite complex due to fundamental differences between relational data models and the object-oriented approach in programming. In relational databases, data is organized in tables with clearly defined fields, and relationships are implemented using foreign keys, ensuring data integrity. In contrast, object-oriented languages operate with objects that have attributes, methods, and complex interconnections through classes and interfaces. This mismatch complicates the integration of these two data structuring methods.

To simplify this process, Object-Relational Mapping (ORM) frameworks [1] are used to automate the transformation of data between code objects and database records, significantly reducing development time. There is a wide range of ORM solutions available, each with varying functionality, levels of abstraction, and architectural features. Among the most common solutions for Node.js are Sequelize [2], Prisma [3], and TypeORM [4].

Sequelize follows the Active Record pattern, allowing each database row to be directly represented as a JavaScript object.

Prisma applies the Data Mapper pattern, separating application logic from the database layer and aiming for high performance and type safety.

TypeORM supports both Active Record and Data Mapper approaches and offers advanced features such as a query builder and support for hierarchical data structures.

The primary objective of this work is to experimentally investigate and compare the efficiency of selected ORM frameworks.

2. Materials and methods

Numerous studies have been conducted on the topic of ORM performance and usage characteristics. For example, Bäcke and Lindström compared common ORM frameworks in terms of performance, maintainability, and usability using a containerized environment and a realistic backend structure [5].

While their research provides a broader evaluation across multiple criteria, this work focuses specifically on detailed performance benchmarking across a greater variety of usage scenarios. In particular, the study isolates the impact of each ORM's internal query generation and execution behavior under three distinct load conditions.

2.1. Research object

The object of this study is the backend of an online store information system, developed in the Node.js environment using the NestJS framework [6] and the PostgreSQL database management system. The system is designed to provide realistic data interaction conditions typical for commercial web services. The structure of the system's database is shown in Figure 1.

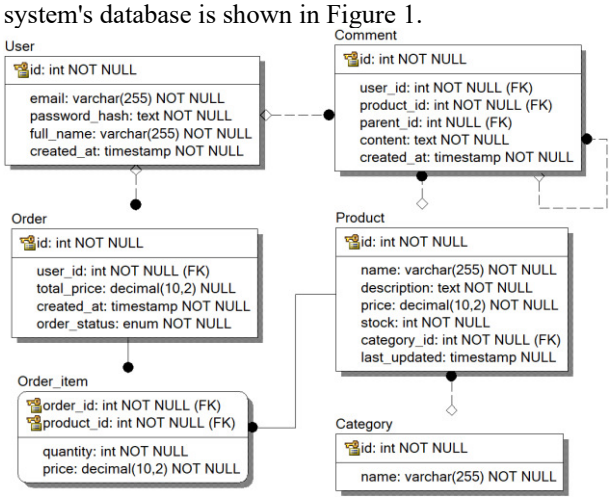

Figure 1: Physical data model.

To evaluate the performance and capabilities of different ORM frameworks, a set of nine endpoints was developed to simulate typical interactions between a backend system and a relational database. Each endpoint corresponds to a common scenario found in real-world applications – from simple Create, Read, Update, Delete (CRUD) operations to more complex interactions such as nested data retrieval, transactional operations, and hierarchical data processing. All endpoints were implemented using Sequelize, TypeORM, and Prisma to enable direct comparison under identical conditions. This approach allows for a comprehensive assessment of each framework's behavior, efficiency, and query generation strategies in a variety of usage contexts.

The implemented endpoints are as follows:

- **Read user data by a given ID** – a simple data reading operation.
- **Create user** – a simple data insertion operation.
- **Update user data** – a simple data update operation.
- **Delete user** – a simple data deletion operation.
- **Get list of products** – a data reading operation involving filtering, sorting, and pagination.
- **Read order data** – a read operation involving multiple tables to fetch nested records.
- **Create order** – an operation that inserts nested records across multiple tables.
- **Confirm order** – a set of operations wrapped in a transaction:
  - Update the order status to "Confirmed".
  - Retrieve the list of ordered products.
  - Decrease the stock quantity of the ordered products.
- **Get comment tree by the given parent comment ID** – a set of operations to work with a simple hierarchical structure.

2.2. Test methods

The testing methods focused on measuring the execution time of database queries using each of the ORM frameworks.

During the study, three types of testing were applied:

- **Single cached query execution**: Each subsequent query is executed only after the previous one has completed.
- **Single uncached execution**: After executing a query and before sending the next one, the PostgreSQL cache is cleared.
- **Parallel query execution**: 50 queries are sent to the database simultaneously; after the completion of each, a new one is immediately created until the target number is reached.

For each endpoint and each testing type, 1000 queries were sent to ensure a sufficient data volume and obtain statistically reliable results.

With the use of logging tools, the raw SQL queries generated by the ORM frameworks were obtained. Comparing the execution time of queries through the ORM and the directly generated raw SQL queries allowed evaluating the overhead introduced by the ORM during query formation.

The obtained raw query was further analyzed using PostgreSQL EXPLAIN (ANALYZE) command [7]. This made it possible to assess the quality of queries generated by each ORM framework both in terms of execution speed and the efficiency of the execution plan created by PostgreSQL.

The database was populated with more than 5 million records for testing purposes (Table 1).

Table 1: Number of generated test records for each table

Table name	Number of records
User	300 000
Profile	150 000
Category	20
Product	600 000
Order	600 000
Order_item	1 800 000

To ensure stability and isolation of the testing environment, the NestJS server and the database were deployed in two separate Docker containers [8].

Docker containers have next versions:

- Database container: Debian 17.4-1.pgdg120+2 by using postgres:latest image.
- Server container: Alpine Linux v3.21 by using node:18-alpine image.

Testing was conducted on a device with the following technical specifications:

- Processor: Intel Core i5 8265U.
- RAM: 16 GB DDR4 2400MHz SODIMM.
- Storage: 256 GB SSD Seagate BarraCuda 510.

3. Results

The data collected for each endpoint is visualized in four diagrams. Three of them show the average time spent on SQL query creation, sending/receiving and database execution, across three modes: cached, uncached and with parallel load. The fourth chart summarizes the total request time per framework for direct comparison.

3.1. Endpoint "Read user data"

During the testing of the endpoint (Figure 2), under cached query conditions, TypeORM and Sequelize demonstrated comparable performance, whereas Prisma exhibited approximately 30% longer execution times.

In the uncached query mode, Sequelize showed the best performance. Prisma once again proved to be the slowest, with execution times nearly three times longer than results of Sequelize.

However, under parallel load conditions, Prisma showed the highest processing efficiency, while Sequelize demonstrated the worst performance among all frameworks.

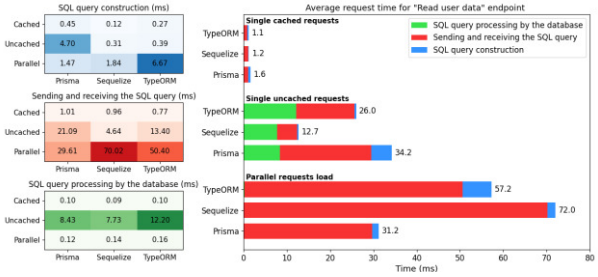

Figure 2: Results of testing the "Read user data" endpoint.

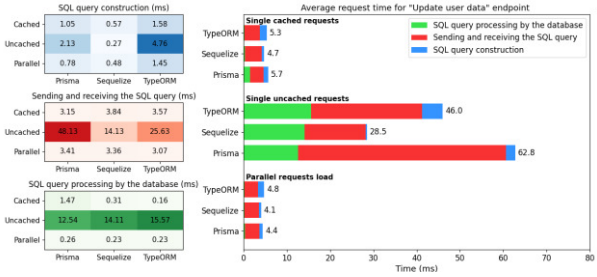

Figure 4: Results of testing the "Update user data" endpoint.

3.2. Endpoint "Create user"

During the testing of the "Create user" endpoint (Figure 3), the results differed from the previous findings.

In the cached query mode, Sequelize demonstrated the highest performance, while TypeORM was approximately 20% slower.

In the uncached query mode, the results generally followed the same patterns observed earlier during user data reading. It is worth noting that at the level of raw SQL execution, the database exhibited the highest latency for Sequelize. However, due to more efficient query generation, and result processing, this ORM framework turned out to be the fastest overall.

Under parallel load conditions, all ORM frameworks demonstrated better processing time than in the cached single-query mode. This may indicate the influence of connection pool behavior and asynchronous request handling, which enable more efficient resource distribution under high concurrency.

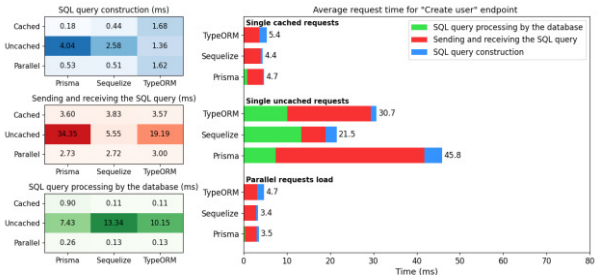

Figure 3: Results of testing the "Create user" endpoint.

3.3. Endpoint "Update user data"

During the testing of the user data update endpoint (Figure 4), the results did not differ significantly from those obtained during user creation.

In the cached query mode, all ORM frameworks demonstrated approximately the same execution speed, with Sequelize performing slightly better.

In the uncached query mode, the situation was similar to previous tests; however, the queries generated by TypeORM proved to be less optimized in terms of database processing efficiency.

Under parallel load conditions, the performance was comparable to the level demonstrated during single cached queries.

3.4. Endpoint "Delete user"

During the testing of the user deletion endpoint (Figure 5), a number of interesting results were obtained.

In the cached query mode, all ORM frameworks executed the operation at nearly identical speeds, with Sequelize performing slightly faster due to more efficient query generation.

In the case of uncached queries, the performance of the frameworks was also nearly equivalent across all execution stages.

Under parallel load conditions, the results resembled the user data retrieval scenario: Prisma demonstrated the highest efficiency, while Sequelize exhibited the lowest performance, slightly trailing TypeORM.

The list of endpoints presented above provides comparative statistics for basic CRUD operations. The following endpoints will test more complex usage scenarios of ORM frameworks.

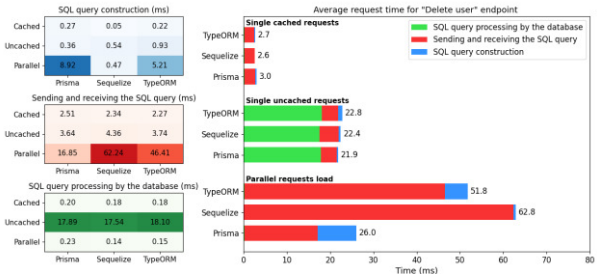

Figure 5: Results of testing the "Delete user" endpoint.

3.5. Endpoint "Get list of products"

This endpoint operates on product records, performing complex operations such as sorting by name, filtering by selected category, and pagination, which results in performance outcomes (Figure 6) that differ significantly from previous cases.

In the cached query mode, results are similar to those observed in user data retrieval: TypeORM and Sequelize demonstrate comparable execution speeds, while Prisma lags behind by approximately 30%.

In the uncached query scenario, the majority of the time is spent on query processing by the database itself, accounting for over 90% of the total time. Since this portion is roughly the same across all frameworks, the key factor is the speed of query generation and submission. In this regard, Prisma again showed the poorest performance.

Under parallel load conditions, Prisma proved to be the most stable and fastest, whereas Sequelize once again exhibited the lowest performance.

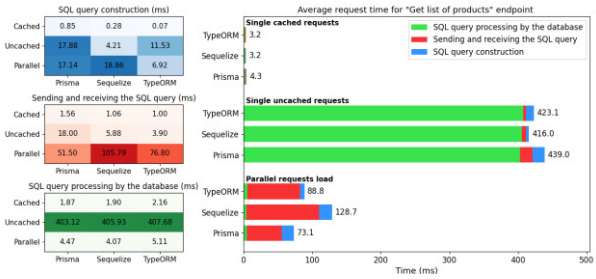

Figure 6: Results of testing the "Get list of products" endpoint.

3.6. Endpoint "Read order data"

Next endpoint operates on two tables simultaneously: Order and Order_item. The scenario requires retrieving order information along with the list of ordered items. The implementation of this scenario varies depending on the chosen ORM framework [9]:

- Sequelize performs the database query using a single SQL statement with a LEFT OUTER JOIN.
- Prisma executes two separate sequential queries to each table.
- TypeORM uses a nested subquery to avoid order duplication due to mismatches between relational and object data structures.

According to the test results (Figure 7), such complex implementation for TypeORM negatively impacted performance: across all three testing modes, this framework exhibited the longest SQL query execution time. It also had the longest query generation and submission time, except for Prisma, which was predictably slow in the uncached query mode.

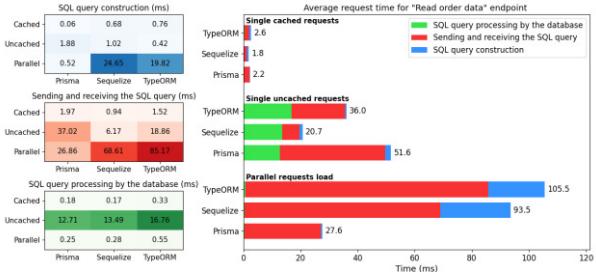

Figure 7: Results of testing the "Read order data" endpoint.

3.7. Endpoint "Create order"

Next endpoint also requires working with two tables simultaneously, but this time for data insertion operations.

According to the obtained results (Figure 8), Sequelize's performance was significantly lower compared to the other ORM frameworks.

In the cached single-query mode, Sequelize showed the worst results.

Nevertheless, in the uncached mode, it remained the fastest, although only slightly ahead of TypeORM.

Under parallel load conditions, Sequelize's query execution time was approximately three times longer than that observed for Prisma.

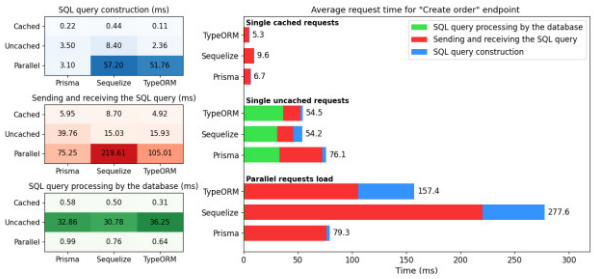

Figure 8: Results of testing the "Create order" endpoint.

3.8. Endpoint "Confirm order"

The results of the transaction tests (Figure 9) largely depend on the efficiency of executing nested queries and should therefore be interpreted with caution, as they may not universally apply to all transaction cases. The transaction execution time was calculated as the sum of the durations of each of its constituent queries.

The most notable difference is the significant increase in Prisma's query execution time under parallel load conditions, which contrasts with its typical stability in other scenarios. Meanwhile, TypeORM exhibited the poorest performance in this mode, primarily due to the considerable time spent generating the raw SQL query.

Based on the obtained results, it can be concluded that the majority of resources consumed by each ORM framework are dedicated to transaction formation and management, as evidenced by the proportion of time each ORM spends processing queries during parallel load, in contrast to the other endpoint testing results.

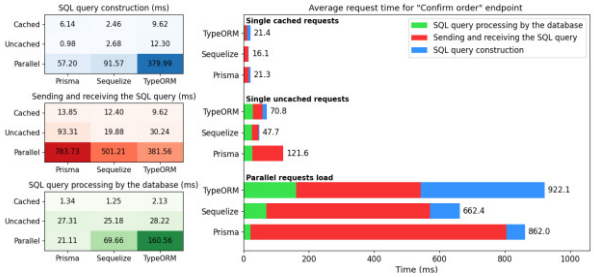

Figure 9: Results of testing the "Confirm order" endpoint.

3.9. Endpoint "Get comment tree"

Since most ORM frameworks do not support hierarchical data structures, the Adjacency List model was used to store comments in the database. This approach relies on recursively querying each descendant in order to reconstruct the full comment tree. As the method primarily involves simple read operations, the resulting performance metrics (Figure 10) were similar to those observed when retrieving user data.

Although TypeORM does not demonstrate the highest performance results, unlike other ORM frameworks, it supports more advanced hierarchical data structures

such as Closure Table, Nested Set, and Materialized Path [10]. These approaches offer significant advantages over the basic Adjacency List method, particularly in terms of query efficiency and flexibility when working with deeply nested or frequently accessed hierarchies.

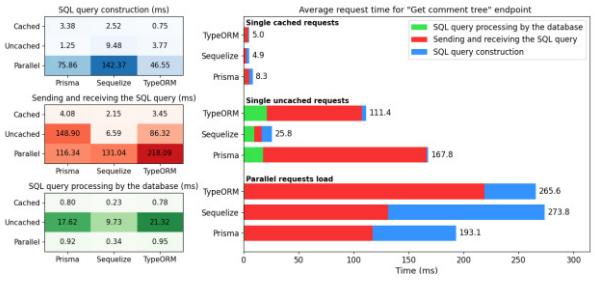

Figure 10: Results of testing the "Get comment tree" endpoint.

4. Conclusions

Prisma proved to have the best scalability under parallel load, particularly in scenarios involving high query concurrency. It demonstrates stable performance when handling a large number of simultaneous requests – for example, during mass user or order creation. In parallel execution modes, Prisma consistently outperforms other ORM frameworks due to its efficient connection pooling, fast SQL query generation, and minimal resource locking. However, in single uncached query scenarios, Prisma often exhibits increased latency, especially when reading complex structures. This makes it less suitable for systems with low throughput and predominantly sequential access patterns.

Sequelize performs best for simple or single queries, particularly in read or create operations targeting individual records. Its compact SQL generation model allows it to handle nested data structures efficiently, for example, when retrieving an order with its associated items. Nevertheless, under parallel load, Sequelize experiences a notable decline in performance. The increased overhead in query construction and response transmission leads to reduced throughput, limiting its effectiveness in high-concurrency systems.

TypeORM offers the most balanced performance profile across various operational modes without critical regressions. Its main strengths lie in handling complex data structures, nested queries, transactions, and implementing hierarchical trees using more advanced methods. TypeORM delivers consistent results in both read and update operations, ensuring predictable behavior even in multi-level processing scenarios. While it may trail behind Prisma under intense parallel workloads and is not always faster than Sequelize for basic operations, its versatility and support for advanced storage patterns make it a strong candidate for complex application architectures.

References

[1] J. Barnes, Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications, Macalester College, Saint Paul, 2007, https://digitalcommons.macalester.edu/mathcs_honors/6/.

[2] Sequelize documentation, https://sequelize.org/docs/v6/, [26.07.2025].

[3] Prisma documentation, https://www.prisma.io/docs/orm, [26.07.2025].

[4] TypeORM documentation, https://typeorm.io/docs/, [26.07.2025].

[5] A. Bäcke, E. Lindström, Evaluation of ORM frameworks for Node.js applications, Master thesis, Linnaeus University, Växjö, 2024, https://www.diva-portal.org/smash/record.jsf?pid=diva2:1881324.

[6] NestJS documentation, https://docs.nestjs.com/, [26.07.2025].

[7] PostgreSQL documentation: EXPLAIN, https://www.postgresql.org/docs/17/sql-explain.html, [26.07.2025].

[8] C. Boettiger, An introduction to Docker for reproducible research, ACM SIGOPS Operating Systems Review 49(1) (2015) 71–79, https://doi.org/10.1145/2723872.2723882.

[9] P. Mishra, M. H. Eich, Join processing in relational databases, ACM Computing Surveys 24(1) (1992) 63–113, https://doi.org/10.1145/128762.128764.

[10] P. Novotný, J. Wild, Modeling hierarchical structures in biodiversity databases, Database (2024) https://doi.org/10.1093/database/baae107.