Comparison of Object-Relational Mapping Frameworks on C#, Python, Go, and Node.js using MSSQL With A Case Study

1st Rand Kouatly
Department of Business
Univ. of Europe for Applied Sciences
Potsdam 14469, Germany
rand.kouatly@ue-germany.de

2nd Orkun Demirci
Department of Business
Univ. of Europe for Applied Sciences
Potsdam 14469, Germany
orkun.demirci@ue-germany.de

3rd Debopriya Das
Department of Business
Univ. of Europe for Applied Sciences
Potsdam 14469, Germany
debopriya.das@ue-germany.de

***Abstract*—Object-relational mapping (ORM) frameworks or tools have turned into a necessity for the development of modern applications. Thorough cross-language performance comparisons that particularly emphasize on Microsoft SQL Server (MSSQL) are still very rare even though ORMs have been widely used by lots of programming languages for different tasks. This paper presents a comparison of eight ORM tools that are SQLAlchemy (Python), GORM and SQLBoiler for Go, Prisma, Sequelize, and TypeORM for Node.js, and Entity Framework and Dapper for .NET Core (C#). Three separate Northwind databases containing Orders tables in different sizes (1,000, 10,000, and 100,000 records) were considered. To mimic practical situations, web API applications were employed for executing CRUD operations and reading queries. Execution time was used to evaluate the performance of ORMs. The findings reveal that the Go-based ORMs, mainly GORM and SQLBoiler, are much faster than the other ORMs. Dapper showed up as a strong candidate in the C# ecosystem. Performance is on the slow side for Node.js-based ORMs, with TypeORM obviously the least efficient one. SQLAlchemy turned out to be a good option for Python projects with medium performance through midrange performance. This study provides developers with useful information on what kind of ORM tools or frameworks they should select in respect to their application needs.**

***Index Terms*—Object-Relational Mapping (ORM), CRUD Operations, MSSQL, Entity Framework, Dapper, GORM, SQLBoiler, Prisma, Sequelize, TypeORM, SQLAlchemy, Cross-language Comparison, Web API**

I. Introduction

ORMs that manipulate databases are tools or frameworks that are used in programming languages. As previously mentioned, programming languages have their own ORM frameworks and tools to perform the same tasks that are CRUD operations. For example, SQLAlchemy ORM is a tool or a framework for the Python programming language, Dapper and Entity Framework are frameworks or tools for the .Net platform that mostly use C# programming language. The advantages of using ORMs are that ORMs provide devel- opers productivity by making database access easier through standard procedures, reducing coding time, and minimizing errors [1]. Additionally, ORMs help to avoid rewriting the code when the database is changed [1]. On the other hand, the disadvantage of using ORMs is that ORMs generate the SQL queries to be executed, which makes it difficult or even impossible to apply optimization techniques or utilize stored procedures [1]. In conclusion, ORMs are convenient to leverage CRUD operations for relational databases.

In this study, Entity Framework (C#), Dapper (C#), GORM (Go), SQLBoiler (Go), Prisma (Node.js), Sequelize (Node.js), TypeORM (Node.js), and SQLAlchemy (Python) ORMs were compared their CRUD performance. Thus, this study is crucial for developers to choose which ORM is better for their projects. While comparing their performance, used Northwind, which is popular and has a familiar structure and relationships between tables, and the MSSQL database that is supported by Microsoft. Additionally, formed a web API application as a case study for every ORM. In this case study, measured CRUD operations of the Orders table and a read operation of all tables that are all joined in a query in Northwind. Moreover, the measurements were made in 3 different databases, which are Northwind1000, which has 1000 rows in the Orders table, Northwind10000, which has 10000 rows in the Orders table, and Northwind100000, which has 10000 rows in the Orders table. While measuring their performance, separated as first-work performance and rest-performance 20 times of CRUD operations for all different databases. As the last information, all measurements are made in the exe file and stored in a table that is created and named TestData table in all Northwind databases.

A. Problem Statement

Despite the use of Object-Relational Mapping (ORM) frameworks and tools for communicating with databases, there are still small studies for having a thorough comparative analysis of these various ORMs' performance using MSSQL. Furthermore, those previous studies mostly compared ORMs in the same languages. This lack of data makes it difficult to select the optimal ORM for high-performance and scal- able solutions. Therefore, this study attempts to figure out this issue by systematically measuring the performance of EntityFramework (.Net Core), Dapper (.Net Core), GORM

(Go), SQLBoiler (GO), Prisma(Node.js), Sequelize(Node.js), TypeORM (Node.js), and SQLAlchemy (Python) ORMs in CRUD operations. Additionally, this study is intended to provide developers with insights about the work that guides their technology choices in real-world applications.

B. Research Questions

Question 1: How do ORM frameworks and tools which are Entity Framework, Dapper, GORM, SQLBoiler, Prisma, Sequelize, TypeORM, and SQLAlchemy perform in executing CRUD (Create, Read, Update, Delete) operations when used MSSQL and Northwind Database?

Question 2: Which ORM framework or tool has optimal execution time for CRUD operations across different programming languages which are .NET Core (C#), Go, Node.js, and Python?

Question 3: Which ORM framework or tool is most suitable for high performance for applications in .NET Core (C#), Go, Node.js, and Python environments when based on performance analysis?

C. Contributions

This study contributes to the field of software development that conducts an objective comparative analysis of several used ORM frameworks and tools which are Entity Framework, Dapper, GORM, SQLBoiler, Prisma, Sequelize, TypeORM, and SQLAlchemy in four programming languages which are C#, Python, Go, and Node.js where MSSQL was used as a database. Additionally, this study investigates the performance, and execution time of these frameworks or tools while executing CRUD operations on the Northwind Database. Through the use of these measurements, developers can make wise decisions on the efficiency of the ORMs. Furthermore, this study highlights performance bottlenecks of ORMs to optimize which ORM is chosen.

II. Literature Review

Object Relational Mappers (ORMs) have evolved into a part of software development as they simplify the interaction with databases by enabling developers to use object-oriented concepts when working with data. Recent studies mostly compared .Net ecosystem ORMs which are EntityFramework, NHibernate, and Dapper. Therefore, there are a few studies comparing Node.js ORMs, which are Prisma, Sequelize, and TypeORM. Additionally, could not find a comparison study in the Go ecosystem ORMs, which are GORM and SQLBoiler. In .Net ecosystem, a performance comparison study was done to assess the execution performance of running the same eight Queries with EntityFramework and Dapper, where the average was calculated after each query was repeated 100 times in the study [2]. The study of [2] indicates that the performance of the results was significantly better for Dapper than EntityFramework. Likewise, research of [3] tried to compare the execution time taken by different ORMs when running operations using MySQL and PostgreSQL. The study indicated that Dapper was the most efficient in queries, while NHibernate and ORMLite had other results depending on the complexity of the query and the database type [3]. Familiarly, results of the study [4] were clear that Dapper is consistently faster than EntityFramework and NHibernate in terms of speed, memory, and processor usage for read, delete, search, sort, and join operations [4]. On the other hand, EF Core was found to have the best performance with respect to the insertion and updating of records while consuming only a minimal amount of RAM and CPU power [4].

For the Node.js ecosystem, the study of [5] comparing ORM tools typeORM, Sequelize, and Prisma focused on several operations such as CRUD (Create, Read, Update, Delete) that were evaluated in terms of requests per second and average response time [5]. These results revealed that TypeORM was the top performer compared to Sequelize and Prisma in CRUD operations [5]. On the other hand, [6] is a study like this study comparing ORMs in different programming languages. The study compares Django and SQLAlchemy for Python, Entity Framework and NHibernate for C#, Hibernate and JOOQ for Java, and CakePHP and Laravel for PHP and indicates that JOOQ (Java) has the best performance among other ORMs [6].

III. Methodology

A. Case Study

In this study, seven web API projects were created to measure the performance of CRUD operations of chosen ORMs in different programming languages using the Northwind database. The Northwind database was selected due to its well-known structure of moderate complexity. Therefore, it is used in academia and benchmarking, making it easier to associate this work with others. Its relational structure with several related tables also serves to test simple or complex queries. Furthermore, N-tier architecture is to be easily readable for this study. Each web API project in different ORMs allows CRUD operations with all tables in the Northwind database. Therefore, each web API project can be an example of an influencer for real-world projects. Additionally, endpoints that are possessed by web API projects are easy to understand for people who are not developers. Additionally, endpoints allow direct connection and CRUD operations to tables in the Northwind database. Used endpoints are shown in “Fig. 1”.

B. Project Structure

Each web API project in different programming languages is organized with N-Tier Architecture, which is shown in “Fig. 2”. Even if some little filename differences by languages and ORMs, for each web API project, the logic is the same. “api” (Presentation Layer) folder is the place where the API layer resides that makes available the RESTful endpoints to interact with the business layer. “bll” (Business Logic Layer) is a place where business logic is kept in the “services” folder, each service being responsible for the main operations of its associated entity. “dal” (Data Access Layer) consists of “entities” and “repositories” files. “entities” provide definitions of the data models which are tables in the database, while “repositories” handle database operations for each model.

```
Category                    Customer
GET    /Categories/{id}     GET    /Customers/{id}
PUT    /Categories/{id}     PUT    /Customers/{id}
DELETE /Categories/{id}     DELETE /Customers/{id}
GET    /Categories          GET    /Customers
POST   /Categories          POST   /Customers

Employee                    Product
GET    /Employees/{id}      GET    /Products/{id}
PUT    /Employees/{id}      PUT    /Products/{id}
DELETE /Employees/{id}      DELETE /Products/{id}
GET    /Employees           GET    /Products
POST   /Employees           POST   /Products

Region                      Shipper
GET    /Regions/{id}        GET    /Shippers/{id}
PUT    /Regions/{id}        PUT    /Shippers/{id}
DELETE /Regions/{id}        DELETE /Shippers/{id}
GET    /Regions             GET    /Shippers
POST   /Regions             POST   /Shippers

Supplier                    Territory
GET    /Suppliers/{id}      GET    /Territories/{id}
PUT    /Suppliers/{id}      PUT    /Territories/{id}
DELETE /Suppliers/{id}      DELETE /Territories/{id}
GET    /Suppliers           GET    /Territories
POST   /Suppliers           POST   /Territories

Orders                      Test Datum
GET    /Orders/{id}         GET    /TestData
PUT    /Orders/{id}
DELETE /Orders/{id}
GET    /Orders/AllTables
GET    /Orders
POST   /Orders

Order Detail
GET    /Order_Details/{orderId}&{productId}
DELETE /Order_Details/{orderId}&{productId}
GET    /Order_Details
POST   /Order_Details
```

Fig. 1. Endpoints

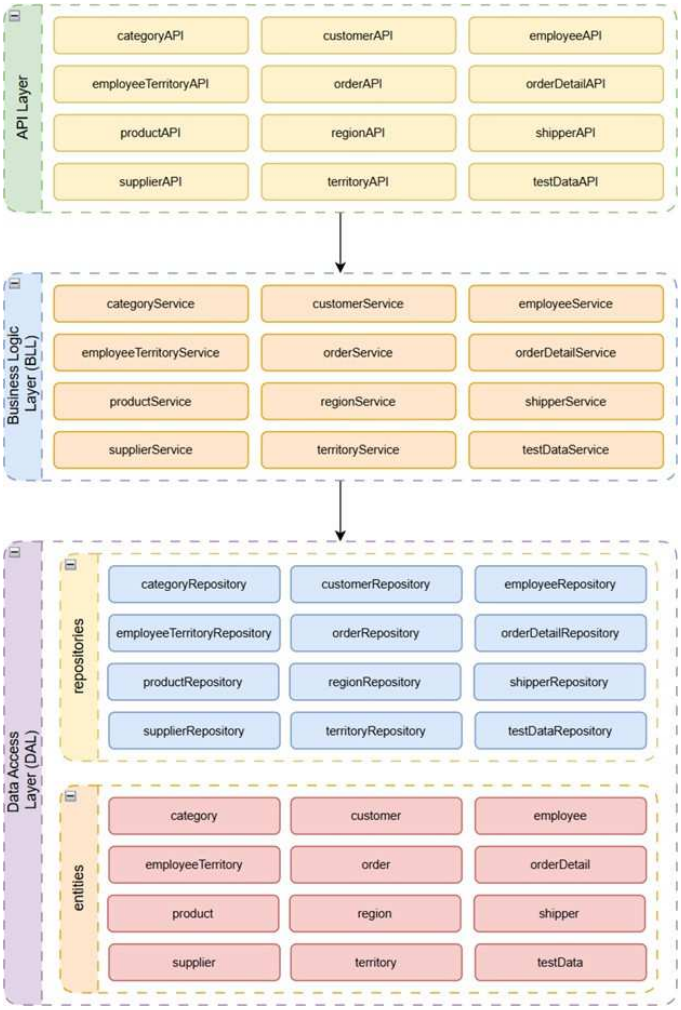

Fig. 2. Project Structure

Clean and maintainable code is made easier by the separation of these parts, which allows the decoupling of business logic from APIs and data access. Additionally, For measurements, orderAPI is used.

C. Database Structure

The Northwind database is an example database that Microsoft created. The Northwind database features several tables that illustrate relationships among key entities, including Customers, Employees, Orders, Products, Suppliers, Shippers, and Categories. Some of the Northwind database tables are presented as one-to-many, one-to-one, and many-to-many relationships. The Northwind database schema is relational with primary and foreign keys, which are used to operate business relationships in the real world, such as orders with customers and products. Therefore, this study measures ORMs' performance. Additionally, this study uses three different Northwind databases to measure ORMs' performance in various situations where the Orders table has 1000 rows, the Orders table has 10000 rows, and the Orders table has 100000 rows. An example diagram of the Northwind database is shown in "Fig. 3".

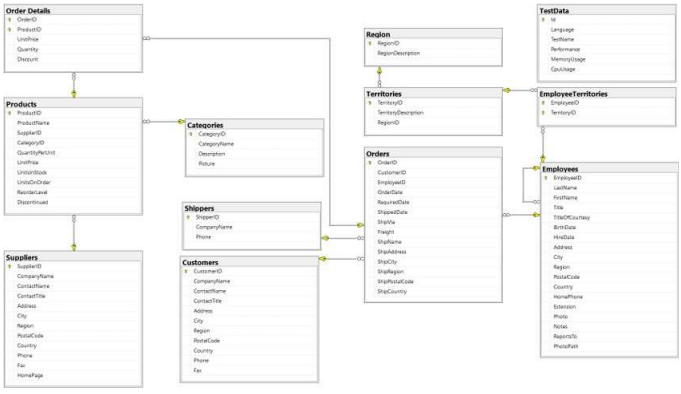

Fig. 3. Northwind Database

D. Measurement System

All measurements were measured in OrderRepository file with running a published executable (.exe) file for each ORM and executed on a computer which has 12th Gen Intel® Core™ i7-12700H processor running at 2.30 GHz, 16.0 GB of RAM, the operating system was Windows 11 Professional, and a 500GB Samsung SSD. Results of measurements were stored in the TestData table which was created in each Northwind database. Before storing measurement results, measurement systems were separated into two sections that are exe runs the first-time for an operation, and exe runs nineteen times for an operation (as shown in "Fig. 4". Then, the results were recorded in each TestData table in their Northwind databases. The performance queries used different methodologies that demonstrate the average, min, and max times of ORMs except for their first-time performances, because demonstrat- ing their first-time performance in different methodologies. Additionally, this study used five operations to measure the performance of ORMs by sending request endpoints of the Orders tables. Those endpoints are GET .../Orders/id, PUT .../Orders/id, DELETE .../Orders/id, GET .../Orders/AllTables, GET .../Orders, and POST .../Orders. Moreover, shown below to publish approach and used libraries, as well as code snippets, which are examples of how tests are provided.

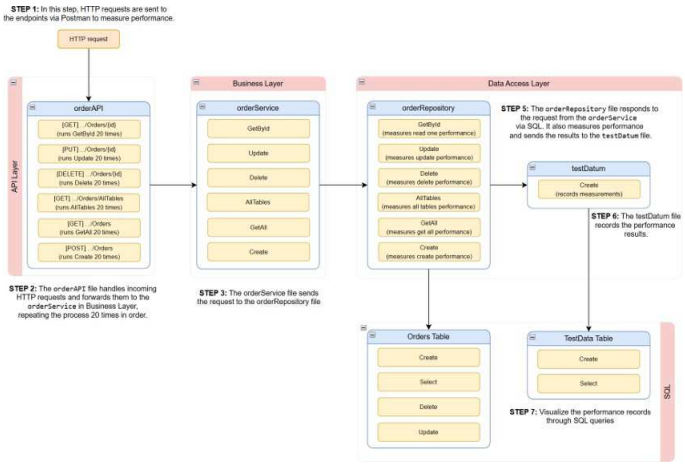

Fig. 4. Measurement System Diagram

IV. Results and Discussions

A. Results

1) Create Performance: As shown in "Fig. 5 (A, B, C)", across all tested scales Go-based ORMs (SQLBoiler and GORM) and C#'s Dapper consistently show the fastest cre- ate operation times, which are significantly above the times demonstrated by other languages. SQLBoiler (GO) leads the pack (e.g., 0.22 ms, 0.46 ms, 0.35 ms) followed closely by Dapper (C#) (e.g., 0.36 ms, 0.36 ms, 0.44 ms) and GORM (GO) (e.g., 0.44 ms, 0.46 ms, 0.49 ms). C#'s Entity Framework is slower, but still leads over Node.js/Python, while Node.js ORMs (Sequelize, Prisma) are much slower (typically 1.9- 2.8 ms), but they are stable. TypeORM (Node.js) remains the slowest set of times.

2) Update Performance: As shown in "Fig. 5 (D, E, F)", comparing all the tested scales Go-based ORMs (SQLBoiler and GORM) were the fastest in update performance through- out with SQLBoiler leading. C#'s Dapper was very competi- tive, especially at bigger scales (0.26 ms for 100k rows), while Entity Framework (C#) was slower (0.70-0.80 ms). Node.js ORMs (Sequelize, Prisma) were in the middle range (1.33- 1.53 ms), but TypeORM (Node.js) was way slower (3.51- 6.31 ms) totally going least efficient lot consistently. Python's SQLAlchemy (1.91-2.89 ms) was still between Node.js and C#/Go. Generally, performances either remained stable or slightly increased for Go and Dapper when moving up to 10k and 100k.

3) Delete Performance: As shown in "Fig. 5 (G, H, I)", GO ORMs dominated with SQLBoiler being the fastest (0.06 ms, 0.16 ms, 0.35 ms) while GORM was 0.37 ms, 0.24 ms, 0.19 ms. Dapper (C#) also showed strong performance with 0.32 ms, 0.38 ms, 0.26 ms and outperforming Entity Framework (C#). Among Node.js ORMs, Prisma was the fastest (1.16 ms, 1.26 ms, 1.94 ms), followed by Sequelize (2.69 ms, 2.78 ms, 2.59 ms) while TypeORM improved significantly at larger scales (7.44 ms, 2.71 ms, 3.03 ms). SQLAlchemy (Python) was slowest at lower scales (3.26 ms, 3.50 ms) but improved at 100k rows (2.80 ms).

4) Read One Performance: As shown in "Fig. 6 (A, B, C)", across all tested data scales, Entity Framework (C#) consistently dominates read one operations with 0.01 ms latency, even at 100,000 rows. Go-based ORMs (GORM, SQLBoiler) delivered competitive speeds (0.07–0.45 ms) and outperforming Dapper (C#, 0.22–0.31 ms) in larger datasets. For Node.js, Prisma was the fastest (0.61–0.75 ms) while Sequelize (1.12–1.85 ms) and notably TypeORM (2.59–6.28 ms) lag behind. SQLAlchemy (Python) maintained midlle performance (0.70–1.0 ms) and positioning Python between C#/Go and Node.js ecosystems. TypeORM emerges as the slowest across all benchmarks.

5) Read All Performance: As shown in "Fig. 6 (D, E, F)", performance trends across different ORMs indicate that SQLBoiler (GO) is the fastest for all dataset sizes, followed by SQLAlchemy (Python), which achieves 1.05–1.18 ms. GORM among Go ORMs moves at a very slow pace. Dapper is the most efficient C# ORM and it is especially evident when compared to Entity Framework. Node.js ORMs (TypeORM, Sequelize, Prisma) exhibit the slowest performance, worsening with larger datasets—Prisma is the least efficient. In general, compiled languages (Go) and optimized libraries (Python's SQLAlchemy, C#'s Dapper) are highly efficient, while Node.js ORMs are facing difficulties in scaling up.

6) Read All Joined Tables Performance: As shown in "Fig. 6 (G, H, I)", in all the test runs, SQLBoiler (GO) and GORM (GO) are good and unchanging with the lowest latencies, whereas Prisma (Node.js) is the slowest. Dapper (C#) is faster than Entity Framework (C#). ORMs for Node.js

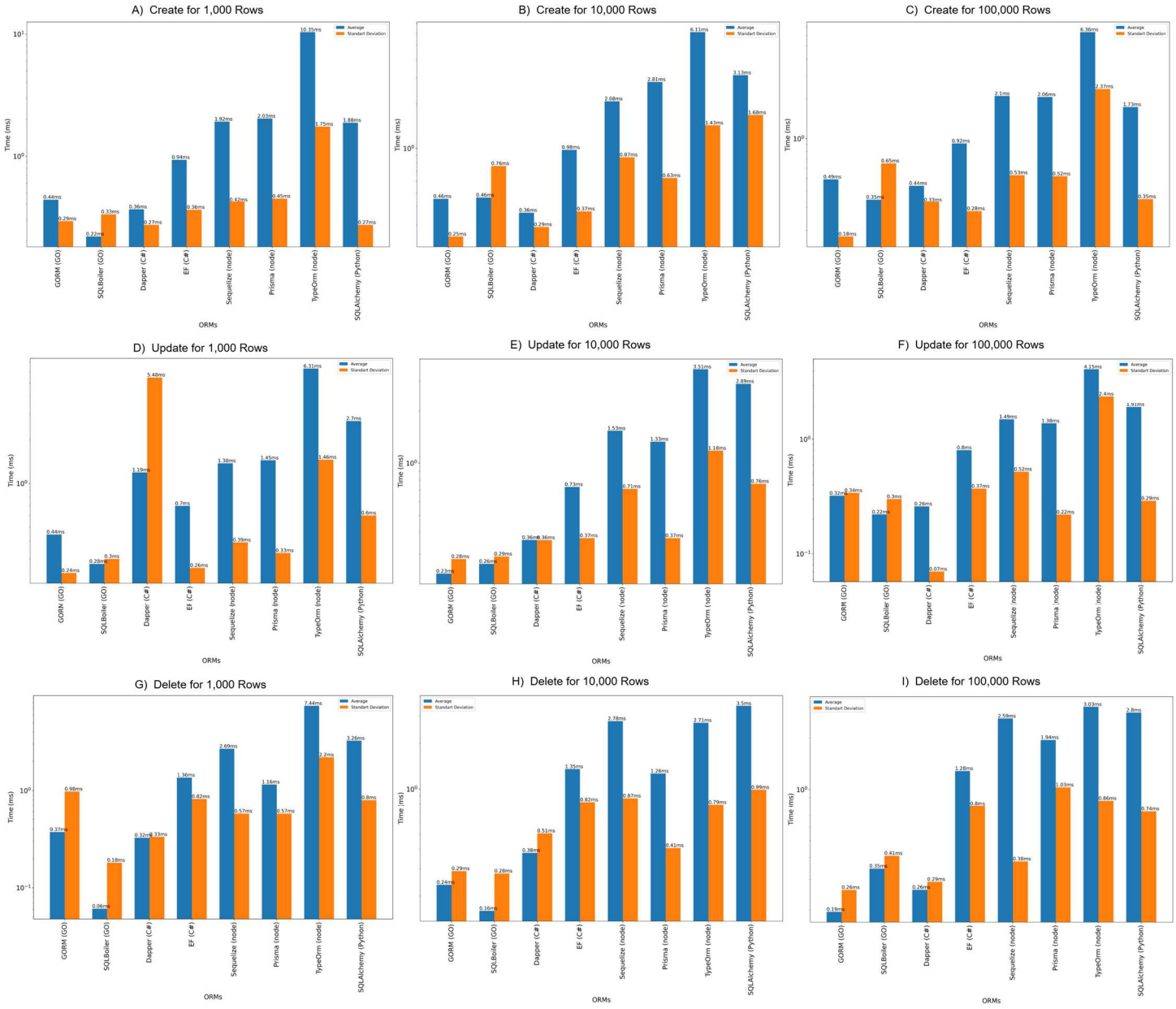

Fig. 5. Create, Update, Delete Performances of ORMs

(Sequelize, TypeORM) are considerably behind, but still they are faster than Prisma.

7) First Time Performance: Based on the data from Ta- bles I, II, III, analysis of ORMs performance characteristics indicate that first runs of operations during various languages and types of tasks parallel to latencies span at. Ops using Go-based ORMs (GORM, SQLBoiler) are frequently faster and complete in single-digit milliseconds at 100k rows for example, they also consistently accomplish in speeds of the same order that are better than other ORMs. The inverse is,
.NET's Entity Framework (EF) and Node.js's Prisma have the longest latency. Python's SQLAlchemy is a non-commited leader between the two ranks of performance, it outperforms most of the non-Go ORMs and especially in bulk reads. Also performance scalability is different, while Go and Python ORMs keep stable latencies basically as the number of rows increases 100x, Node.js ORMs (Prisma, Sequelize) show exponential degradation in "Read All" operations.

B. Discussions

When comparing the performance of different Object Relational Mappings (ORMs) for operations such as creating, updating, deleting, reading one row in the Orders table, reading all rows in the Order table, and reading all linked tables as a consistent throughput advantage is evident for Go- based ORMs that are SQLBoiler and GORM. These Go-based frameworks or tools dominate the space for any dataset size

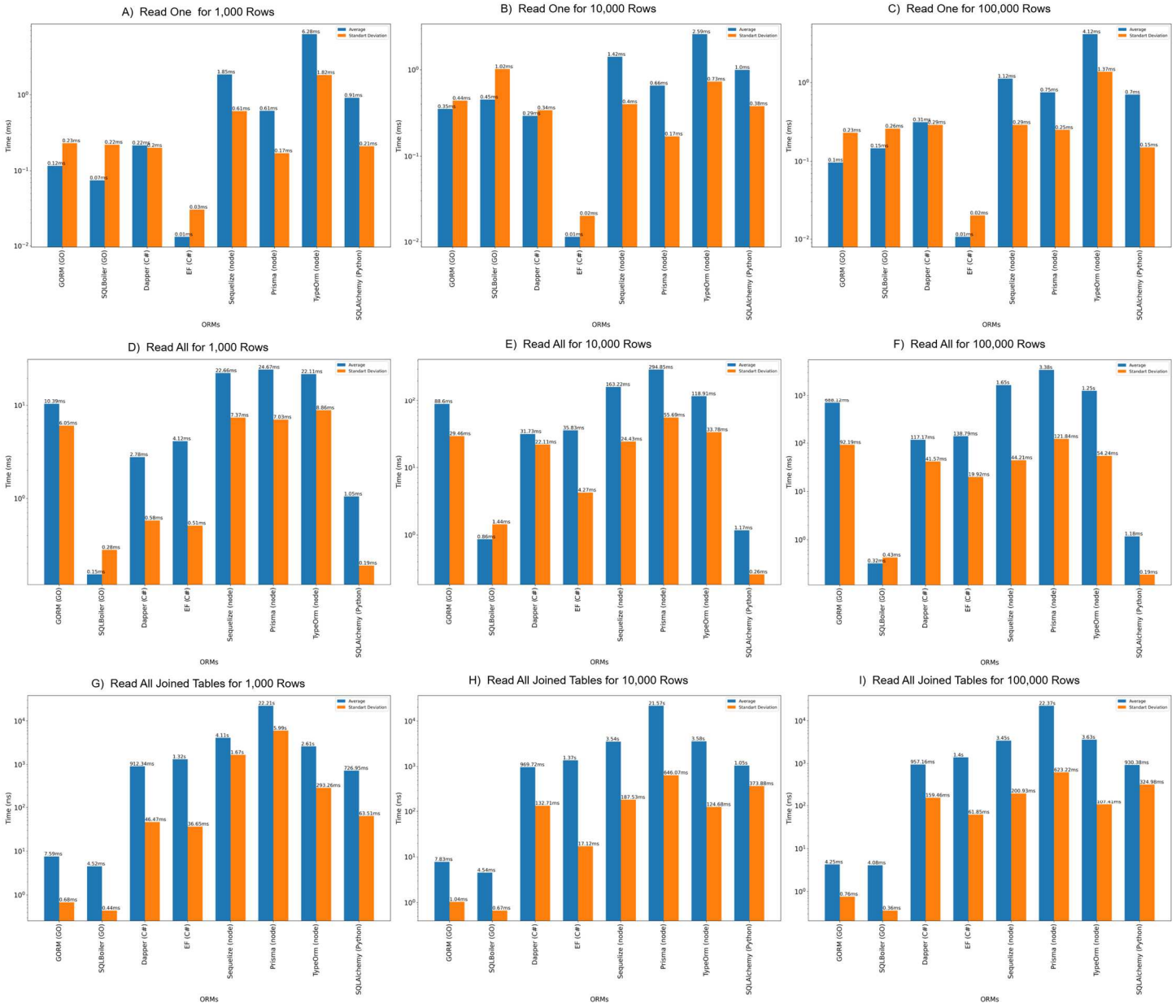

Fig. 6. Read One, Read All, Read All Joined Tables Performances of ORMs

TABLE I
FIRST TIME PERFORMANCE OF ORMS FOR 1,000 ROWS

Languages	ORMs	First Create	First Update	First Delete	First Read One	First Read All	First Read All Joined Tables
GO	GORM	9.01ms	5.62ms	4.64ms	3.59ms	29.28ms	29.59ms
	SQLBoiller	14.94ms	11.08ms	12.69ms	14.4ms	14.21ms	30.93ms
.Net	Dapper	41.52ms	5.61ms	35.53ms	67.22ms	37.82ms	1.22s
	EF	603.02ms	567.47ms	692.76ms	1.09s	610.73ms	2.61s
Node.js	Sequelize	52.58ms	18.34ms	45.11ms	87.95ms	75.38ms	4.01s
	Prisma	35.16ms	27.61ms	35.23ms	29.87ms	35.35ms	10.57s
	TypeORM	45.79ms	10.86ms	45.02ms	13.62ms	86.85ms	3.82s
Python	SQLAlchemy	8.47ms	8.56ms	24.24ms	17.99ms	7.46ms	915.3ms

TABLE II
FIRST TIME PERFORMANCE OF ORMS FOR 10,000 ROWS

Languages	ORMs	First Create	First Update	First Delete	First Read One	First Read All	First Read All Joined Tables
GO	GORM	4.74ms	6.21ms	2.8ms	2.73ms	76.56ms	17.1ms
	SQLBoiller	16.33ms	11.39ms	9.46ms	21.34ms	10.6ms	22.77ms
.Net	Dapper	45.19ms	55.44ms	64.71ms	50.65ms	100.32ms	1.22s
	EF	579.83ms	566.86ms	691.99ms	650.88ms	816.86ms	2.62s
Node.js	Sequelize	45.83ms	39.75ms	46.85ms	45.06ms	379.23ms	3.7s
	Prisma	41.35ms	31.07ms	37.01ms	27.33ms	339.29ms	21.76s
	TypeORM	14.34ms	9.08ms	9.09ms	7.93ms	312.66ms	3.66s
Python	SQLAlchemy	13.38ms	9.22ms	29.64ms	30.75ms	25.7ms	649.37ms

TABLE III
FIRST TIME PERFORMANCE OF ORMS FOR 100,000 ROWS

Languages	ORMs	First Create	First Update	First Delete	First Read One	First Read All	First Read All Joined Tables
GO	GORM	6.03ms	1.58ms	5.31ms	2.97ms	675.72ms	16.85ms
	SQLBoiller	8.85ms	10.65ms	15.86ms	12.42ms	18.96ms	21.45ms
.Net	Dapper	43.79ms	5.85ms	32.49ms	49.25ms	316.27ms	1.1s
	EF	591.92ms	573.53ms	692.37ms	657.5ms	1.52s	2.7s
Node.js	Sequelize	46.47ms	43.45ms	49.59ms	39.11ms	1.84s	3.81s
	Prisma	36.86ms	34.16ms	37.7ms	51.8ms	3.58s	21.53s
	TypeORM	11.82ms	24.91ms	11.54ms	7.36ms	1.54s	3.86s
Python	SQLAlchemy	7.22ms	8.63ms	28.15ms	23.14ms	19.97ms	606.56ms

due to their very low latency. This high performance can be attributed to the advantages Go has as being a compiled language, which results in faster execution times and lower overhead compared to interpreted or just-in-time compiled languages. In contrast, Dapper (C#) performed well as a non-Go ORM. Unfortunately, it lags a bit behind Go-based ORMs. For example, Entity Framework (C#) still offers very good performance even under heavy load, but the problem is that it becomes increasingly slower as the dataset size increases, especially during read operations. It is also important to contrast Entity Framework's slow first-run performance with its sub-millisecond averages during subsequent operations, highlighting the impact of caching and initialization overhead. Node.js ORMs, that are Prisma, Sequelize, and TypeORM not well performing with each operation, while TypeORM generally has the worst performance. SQLAlchemy (Python) bridges between Go/C# and Node.js ORMs by offering moderate but consistent throughput. Therefore, it is an acceptable choice for Python-based applications. However, it never reaches the efficiency of Go. These observations confirm that Go-based ORMs are undoubtedly the best alternative for highly efficient database operations.

C. Limitations of the Test

It is important to acknowledge that the performance measures of ORMs for this study were managed on one hardware and operating system. The results included running the 12th Gen Intel® Core™ i7-12700H processor at 2.30 GHz, with 16.0 GB RAM, a 500GB Samsung SSD, and Windows 11 Professional as the OS. Therefore, the results derived from the performance tests most probably would not reflect the behavior experienced on systems with distinct hardware specifications or operating systems. This limitation should be kept in mind while interpreting the findings and comparing them with others from different environments.

While this asynchronous programming was implemented in .NET-based ORMs like Entity Framework and Dapper as well as Node.js-based ORMs such as Prisma, TypeORM, and Sequelize. On the other hand, the asynchronous approach was not implemented in Golang-based ORMs (GORM, SQLBoiler) and Python-based ORM (SQLAlchemy). Additionally, this study does not include any concurrent or load testing. All performance measurements are considered a single request execution. Therefore, these results did not account for the performance under heavy concurrency or a real multi-user environment.

Although memory and CPU usage metrics were included in the measurement outputs and displayed in the code snippets, these values were assigned "0 MB/ms". Because memory and CPU usage metrics were not within the intended scope of this study. The test focus on the individual execution performance in CRUD operations.

V. CONCLUSIONS

This study researched the performance of various object-relational mapping (ORM) frameworks and tools for multi- ple programming languages, including Go, C#, Node.js, and Python, specifically in the context of Microsoft SQL Server (MSSQL) and the Northwind database. Numerous efficiency patterns and growth opportunities were identified by performing create, read, update, and delete operations across a range of record sizes. Go-based ORMs, particularly SQLBoiler and GORM, proved to be fast across all operations. SQLBoiler's delete operations showed the best performance and lowest latency across all record sizes, while GORM showed impressive performance for both read and delete operations. Go is

undoubtedly the best solution for scenarios where high speed and low latency are critical for database interactions.

In the C# ecosystem, Dapper was found to be really effective, especially for reads, and managed to keep up with the Go-based ORMs. Although Entity Framework had its position as a stable, the data size led to very significant helplessness in handling large-scale datasets well. Node.js ORMs like Prisma, Sequelize, and TypeORM also have the same results as Go and C#. Nevertheless, they are the last preference, especially when talking about dataset sizes. Prisma can be considered faster than Sequelize by only a little bit, nevertheless, still, both have problems in the latency area. TypeORM turns out to be the slowest one that makes use of the inefficient way of dealing with large data. These results also have the implication that there is a need for SQL ORM to be optimized within the Node.js environment so as to shorten the performance gap. By comparing it with the other two, Python ORMs represented by SQLAlchemy performed neither best nor worst taking an intermediate position regarding their capabilities, of which the extreme points were the Node.js platform and C#. Although it did not manage to reach the same velocity as Go, it showed that it could be a consistent alternative for read-heavy applications where high latency is not a big issue.

Overall, this study supports the idea that Go-based ORMs are the most suitable for applications. C# ORMs, particularly Dapper, come forth as potent solutions with quite a high level of read performance. Node.js ORMs can actually get a lot from the improvement of data management and concurrency operations in terms of achieving the same efficiency level that the Go, C#, and Python possess.

VI. Future Works

While this study provides meaningful results on ORM performance, numerous future research opportunities have not yet been fully explored. One example could be to measure distributed databases (e.g. CockroachDB, Cassandra) more broadly to test the actual performance of ORMs in highly scalable environments. Furthermore, ORM efficiency tests can be performed under real-world data processing conditions to optimize latency-sensitive applications. Measuring the impact of microservices on ORM efficiency can help organizations determine the average overhead incurred by inter-service communication and database transaction behavior. An in-depth examination of the caching mechanisms of individual ORMs can reveal significant improvement opportunities, especially when reading biased applications. A closer look at different ORMs on how they ensure the integrity and security of information processed during CRUD operations can be useful, especially in regulated industries. Equally important is to investigate the impact of multi-threading on ORM performance, especially for Node.js and Python ORMs. This can lay the groundwork for innovative solutions. These steps will bring into focus more advanced approaches to increasing ORM efficiency and scalability, which can lead to more robust applications that have databases.

References

[1] D. A. Alvarez Eraso, A Framework for Evaluating Maintainability and Performance of Object-Relational-Mapping Tools in Web Application Frameworks, Ph.D. dissertation, 2017.

[2] M. M. Forsberg, An evaluation of .NET Object-Relational Mappers in relational databases: Entity Framework Core and Dapper, 2022.

[3] M. Klimiuk, P. Karabowicz, and M. Plechawska-Wo´jcik, ”Comparative analysis of database mapping frameworks available in NuGet Manager,” Journal of Computer Sciences Institute, vol. 32, pp. 231–238, 2024.

[4] A. E. Gu¨vercin and B. Avenoglu, ”Performance analysis of object-relational mapping (ORM) tools in .NET 6 environment,” Bilis¸im Teknolojileri Dergisi, vol. 15, no. 4, pp. 453–465, 2022.

[5] V. C. Jungers, E. A. de Oliveira, and V. A. De Souza, ”Comparac¸a˜o de desempenho entre os ORMs TypeORM, Prisma e Sequelize em aplicac¸o˜es Node.js,” Revista Eletroˆnica e-Fatec, vol. 14, no. 2, 2024.

[6] V. Sivakumar, T. Balachander, R. Jannali et al., ”Object relational mapping framework performance impact,” Turkish Journal of Computer and Mathematics Education, vol. 12, no. 7, pp. 2516–2519, 2021.