Performance Analysis of Object-Relational Mapping (ORM) Tools in .Net 6 Environment

Araştırma Makalesi/Research Article

Abdullah Eren GÜVERCİN[1], Bilgin AVENOĞLU[2]

[1]Yazılım Mühendisliği, Ahmet Yesevi Üniversitesi, Ankara, Türkiye
[2]Yazılım Mühendisliği, Ankara Üniversitesi, Ankara, Türkiye
egvrcn@gmail.com, bavenoglu@gmail.com
(Geliş/Received:18.01.2022; Kabul/Accepted:01.10.2022)
DOI: 10.17671/gazibtd.1059516

***Abstract*—** ORM tools are frequently used in projects developed by object-oriented programming paradigm. Software developers generally look at the performances of these tools when they select an ORM tool. Most of the performance studies on ORM tools are limited to processing time and RAM usage information, and CPU usage information is not included. Moreover, no ORM performance study has been found in the literature, conducted in .NET 6, which is an open-source and platform-independent new generation .NET platform. In this study, to close the mentioned gap in the literature and guide the software developers, we conduct research for analyzing performances of certain ORM tools in .NET 6. Our study includes CPU usage information as well as processing time and RAM usage information. We develop a software for measuring processing time, RAM and CPU usage while performing read, insert, update, delete, search and sort operations with Dapper, NHibernate and Entity Framework Core (EF Core) ORM tools. As a result, while Dapper is best in terms of processing time for read, delete, search and sort operations, EF Core has the best results for insert and update operations. We conclude that Dapper has the best performance in terms of resource usage, while the rankings of EF Core and NHibernate vary among themselves according to the number of records and operation type.

***Keywords*—** orm, .net 6, dapper, nhibernate, entity framework core

Nesne-İlişkisel Eşleme (ORM) Araçlarının .NET 6 Ortamında Performans Analizi

***Özet*—** Nesneye yönelik programlamada, Nesne-İlişkisel Eşleme (Object-Relational Mapping – ORM) araçları sıklıkla kullanılmaktadır. Yazılım geliştiricilerin ORM seçimi yaparken en önemli seçim kriterlerinden birisi bu araçların sağladığı performanstır. ORM araçları üzerine yapılan performans araştırmalarının çoğu işlem süresi ve Rastgele Erişimli Bellek (Random-Access Memory – RAM) kullanım bilgileriyle sınırlı kalmış, Merkezi İşlem Birimi (Central Processing Unit – CPU) kullanım bilgilerine yer verilmemiştir. Ayrıca literatürde, platform bağımsız ve açık kaynak olarak üretilen yeni nesil .NET platformu olan .NET 6 ortamında yapılmış bir ORM performans çalışmasına rastlanılmamıştır. Bu çalışmada, belirtilen eksikliği gidermek ve yazılım geliştiricilere yol göstermek için .NET 6 ortamında belirli ORM araçlarının performans analizi gerçekleştirilmiştir. Çalışmada, işlem süresi ve RAM kullanım bilgilerinin yanında CPU kullanım bilgileri de yer almaktadır. Bu çalışmada başlıca ORM araçlarından Dapper, NHibernate ve Entity Framework Core (EF Core) ile kayıt okuma, ekleme, güncelleme, silme, arama ve sıralama işlemleri gerçekleştirilerek, işlem süresi, RAM ve CPU kullanımının ölçülebileceği bir yazılım geliştirilmiştir. Yapılan ölçümler sonucunda işlem süresi açısından; okuma, silme, arama ve sıralama işlemleri için Dapper; ekleme ve güncelleme işlemleri için EF Core en iyi sonuçları vermiştir. Kaynak kullanımı açısından Dapper'ın en iyi performansa sahip olduğu, EF Core ile NHibernate araçlarının sıralamalarının ise kayıt sayısı ve işlem türüne göre kendi aralarında değiştiği sonucuna varılmıştır.

***Anahtar Kelimeler*—** orm, .net 6, dapper, nhibernate, entity framework core

1. INTRODUCTION

Object-oriented programming paradigm is frequently used for developing software. When using this paradigm, Object Relational Mapping (ORM) tools are indispensable for converting structures in relational model to object-oriented model. There are many ORM tools developed for different platforms, and each offers different advantages and disadvantages.

The performance of ORM tools is the most important criteria for developers when they need to select one of these tools. There are several studies ([1-3]) which analyze the performance of ORM tools on different software platforms and databases. These studies make comparisons based on operation processing times. Zmaranda et. al [4], measures the RAM usage besides processing times. A study by Balcı [5], which is not performance comparison research on ORM tools but examines the performance analysis of the Entity Framework ORM tool on different databases, included all of the processing time, RAM and CPU usage information. There is a need to make performance comparison research on ORM tools by including not only processing times and RAM usage, but also the CPU usage.

Moreover, existing studies such as [6,7] perform comparisons on previous versions of .NET Framework by using Microsoft SQL Server database. Microsoft has a new .NET 6 environment which is a new generation, open-source, and cross-platform software development framework. Besides the framework, PostgreSQL database has gaining popularity and it has not been used in performance studies. These discussions show that, there is also a need for ORM tools comparison study on .NET 6 environment with PostgreSQL database.

In this study, we try to find the best performant ORM tools in .NET 6 environment with PostgreSQL database. We develop a software for measuring processing times, RAM usage, and CPU usage. We measure these for read, insert, update, delete (CRUD) [8], search and sort operations. We use Dapper, NHibernate and EF Core ORM tools in comparisons. These measurements will be analyzed to show the performance statistics of ORM tools in .NET 6 environment. The results may offer a guideline to developers for selecting the best ORM tools suitable for different operations.

2. CONCEPTUAL FRAMEWORK

There are two major methods for accessing to the databases from different programming languages or environments. The first one, which is the traditional one, is to use database providers' libraries to connect and execute operations. This method is fast since the libraries are generally optimized by the database providers to their databases. However, the application is bound to a specific database, and it is hard to switch between different databases. Moreover, the code for converting relational model to object-oriented model or vice versa must be written manually. This may need a lot of work. The second method is to use an ORM tool for enabling modularity and decreasing the workload. Using an ORM tool may decrease the performance of the application. Joshi and Kukreti [9] compare ORM tools and traditional library access methods and they found that the complex code produced by ORM tools decreases the performance. They also indicate that when the advantages of using ORM tools are considered, the performance loss can be negligible.

Since ORM tools decrease the total performance of the applications, the importance of the performance of the ORM tools is paramount. There are some performance comparison studies in the literature. In a study implemented by Cvetkovic and Jankovic [7], the two ORM tools, NHibernate and Entity Framework are compared. Zmaranda et. al. [4] compare Dapper, EF Core and NHibernate tools. These studies shed the light on the performances of different ORM tools. However, these studies use Microsoft SQL Server as database, and they do not measure the CPU performance. Additionally, these studies don't analyze search and sort operations directly without the effects of other database structures. In another study [10], authors compare eight ORM frameworks with four different programming languages. Yousaf [11] evaluates the performance of Java-based ORM tools (Hibernate, EclipseLink, OpenJPA and Ebean) and his own GlycoVault lightweight persistence tool. In these studies, authors only compare read operations and Dapper is not included in ORM lists. For these reasons, we make a performance comparison of popular ORM tools which are Dapper, NHibernate and Entity Framework in .NET 6 environment by using a popular database which is PostgreSQL. We also compare read, insert, update, delete, sort and search operations. This study does not include a performance comparison of ORM tools and traditional library access methods. Colley et al. [12] has such a study which compares the effects of Entity Framework with SQL Server 2014 database and lists the negative behaviors of ORM tools.

Before describing the performance comparison methodology, we give general information about ORM technique and specific information about the ORM tools we use in this study.

2.1. Object Relational Mapping

An ORM tool is a bridge between a relational database and object-oriented programming language. It allows developers to work directly on the object-oriented programming concepts without thinking the details of the conversion of the components to relational tables or constraints. ORM tools have some advantages:

- They allow developing applications without being tied to a specific database. Different databases can be used with the same source code.
- Developers may execute database operations without writing SQL statements.

- Developers can easily concentrate to the OOP concepts.
- They decrease the time for writing database code.
- They increase the code readability.

Besides these advantages ORM tools have also some disadvantages:

- Writing direct SQL statements allows better performance.
- It is hard to write complex queries with ORM. Writing them with SQL may be easier.

2.2. ORM Tools

There are many ORM tools for object-oriented programming: Hibernate, TopLink and OpenJPA are used with Java; Django, Peewee, and SQLAlchemy are used for Phyton; and RedBeanPHP, Doctrin, and Propel are used for PHP. In .NET environment, Dapper, NHibernate and EF Core are highly used and, in this study, performances of these ORM tools are compared.

Dapper: Dapper is an open-source micro ORM tool developed for .NET environment. The main aim of Dapper is to provide performance to applications and to allow developers to decrease the effort of mapping operations.

Entity Framework Core (EF Core): EF Core is an open source and cross-platform ORM tool for ADO.NET data access library. It is a new version of Entity Framework ORM tool which has been distributed within .NET Framework. Starting from the Entity Framework version 6, Microsoft decided to deliver EF Core separately [13]. Because of this, EF Core, a more modern and sustainable ORM tool, is used in this study.

NHibernate: NHibernate is a .NET version of Hibernate which is frequently used ORM tool in Java environments. NHibernate is an open-source tool and includes almost all features of current Hibernate.

ORM tools are classified as full-featured ORM tools and micro ORM tools based on the features that they support. Micro ORM tools have limited capabilities according to the full-featured ones but they perform faster. A micro ORM tool may not support some caching capabilities, e.g., second level cache. Moreover, when a query is executed and an object is loaded from the database, other objects which are in relationship with this object are not automatically loaded. The programmer has to write special queries to load related objects. Besides these, micro ORM tools generally do not have graphical modelers and automatic database object creation capabilities [14].

ORM tools apply caching techniques for repetitive database operations. These techniques provide performance gains. EF Core has three types of caching: object caching, query plan caching and metadata caching. Object caching is known as first level caching and it stores objects retrieved from database to memory. Query plan caching is used for storing queries executed more than once. This allows skipping the parsing and compiling operations of the query for later executions. EF Core supports metadata caching which is used for different connections to share the type and mapping information. NHibernate also supports first level caching to maintain objects in memory when they loaded first time. NHibernate has a second level cache for storing query plans and query results. EF Core and NHibernate, since they are full-features ORM tools, provide first level cache by default. However, Dapper only caches information for queries to materialize objects and process parameters quickly [15].

Another performance concern with ORM tools is loading related data with queries. This concern generally known as related data loading or fetching. Different ORM tools have different default characteristics for related data loading. Some ORM tools such as EF Core support eager loading by default [16]. Eager loading allows loading all the required entities with one query. Objects in relation with the parent object are also automatically loaded. Some ORM tools, such as NHibernate use lazy loading by default [17]. In this method, related objects are not loaded unless they are really needed. Dapper uses a multi mapping technique which is almost similar with eager loading [18]. However, because Dapper is micro ORM tool, third party libraries are needed for adding lazy loading property.

ORM tools use different mapping techniques between objects and tables and fields. Using an XML file, inserting annotations to source code or writing code to generate mappings are examples of metadata mapping. EF core and Hibernate support variety of these methods as shown in Table 1. In Dapper queries, we execute SQL statements by passing parameters. Beside the metadata mapping, ORM tools have capabilities for reflecting structure changes in object models to databases. If object model frequently changes, then these changes can be reflected to the database by executing a loading procedure. Executing a loading procedure frequently can be tedious. Because of this, some ORM tools use a reflection technique for the objects at runtime to reflect the changes. In this technique, the mapping between the object and table is stored in cache and upcoming calls use this mapping. Change reflection is only applied on first call [19].

Table 1. Properties of ORM Tools

	EF Core	NHibernate	Dapper
Mapping for Metadata	Code based, Attributes based	.HBM, .XML, Code based, Attributes based	SQL Statement
API	ADO.NET	ADO.NET	ADO.NET
Model Change Reflection Type	Automatic	With 3rd party tools	-

Another concern with the ORM tools is transaction support. In EF Core, all the changes are tracked and handled in memory and when the "SaveChanges" method of "DbContext.Database" class is called the changes are applied to the database. This operation is atomic and all the changes are either committed or rollbacked. We call this method after inserting, updating and deleting all the records. NHibernate has the similar methods. We use the "Save" method of "ISessionFactory" class in NHibernate API to save the changes permanently. However, Dapper has a different technique. Because Dapper, does not use object caching, it directly applies SQL statements through the classes of the related database. EF Core and NHibernate also have support for locking mechanisms for concurrent operations since they have object caches. However, our evaluation does not include concurrent access of the data and we have not utilized locking mechanisms.

2.3. .NET 6

.NET Framework is a software development environment produced by Microsoft and supports many languages like C#, Visual Basic, and F#. .NET 6, the latest version of this framework, is a platform for unifying web, mobile, desktop, games and IoT applications under a single framework. .NET 6 is released in November 2021 and targets cross-platforms from iOS, Mac OS, Windows, WatchOS, Android, tvOS etc. Programs written with different .NET compatible languages are compiled to platform-neutral Common Intermediate Language (CIL). Common Language Runtime (CLR), a platform specific runtime environment for .NET, compiles CIL to machine readable code.

2.4. PostgreSQL

PostgreSQL is an object-relational database management system developed by the University of California at Berkeley [20]. It is assumed that, PostgreSQL is the most advanced open-source relational database system [21]. This claim is supported by statistics of usage of PostgreSQL in high-level projects implemented by public and private organizations.

2.5. BenchmarkDotNet

BenchmarkDotNet is an open-source performance measurement library supported by .NET Foundation. BenchmarkDotNet, creates an isolated project for each method which are to be measured and executes them without other side-effects. By this way, processing time and resource consumption of each method can be measured precisely within their private processes [22].

2.6. Chinook Database

Chinook database is a sample database that can be created by a sample script file. It can be used by different databases such as, PostgreSQL, Oracle, SQL Server, and MySQL. The Chinook database has a data model which includes a digital media store, including tables for artists, albums, media tracks, invoices, and customers. In this study, we use the "Track" table from this model in performance measurements since it includes almost 3,500 records. We also use "Album" table for join operations.

3. METHODOLOGY

In this study, we get measures for read, insert, update, delete, sort and search operations. We test a one table select statement and a join statement for read measurements. We measure the processing times, RAM usage, and CPU usage of these operations by different ORM tools in .NET 6 environment. Dapper 2.0.123, EF Core 6.0.6 and NHibernate 5.3.12 versions are used for performance comparisons. Version 14.1 of PostgreSQL database is used. The records of the "Track" table of Chinook database are used in measurements. Additionally, we join "Album" table to "Track" table for measuring read operations of joined tables.

3.1. Architecture

We develop a software, ORMPY, for measuring processing times, RAM, and CPU usages of ORM tools in .NET 6 environment. We make the software open-source and publish it in GitHub[1]. The software is developed by a layered architecture including a model layer (Entity Layer), a persistence layer (Data Access Layer) and two application layers. Business layer is integrated within the application layer for executing the operations in isolation to get most accurate results.

There are two application layers in ORMPY. The first application layer measures processing times and RAM usage by BenchmarkDotNet library. In this application, each method is executed 100 times iteratively by BenchmarkDotNet, and averages are calculated. The second application layer measures CPU usage by Microsoft Diagnostics library [23]. Even though the methods are run in isolation, CPU usage is very fluctuating due to operating system processes. We execute each method 500 times and calculate the averages to normalize the CPU usage times. Moreover, we take the test computer in airplane mode and close the internet connections and all other applications. Measurements are implemented by using the computer with hardware properties given in Table 2. The detailed architecture of ORMPY software is given in Figure 1.

[1] https://github.com/egvrcn/ORMPY

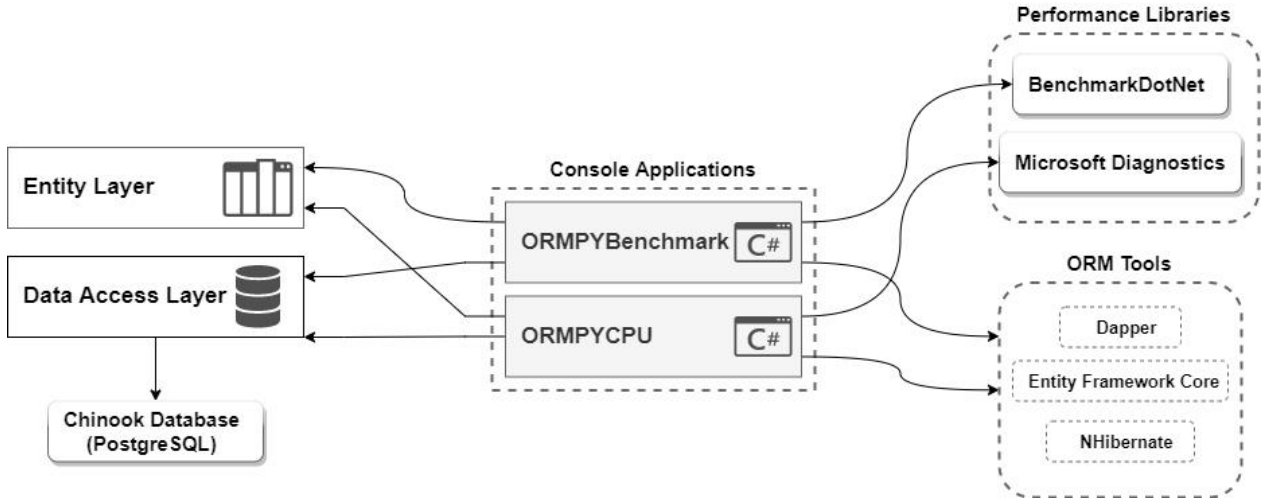

Figure 1. ORMPY Architecture

Table 2. Hardware/software information of test computer

Hardware/Software	Property
CPU	Intel Core i5-7300HQ CPU 2.50GHz (Kaby Lake), 4 logical and 4 physical cores
RAM	16 GB
Disk	Samsung SSD 860 Evo 250GB
Operating System	Windows 10 Pro

3.2. Data Collection

ORMPY is designed to execute read, insert, update, delete, search and sort operations. These operations are applied on the "Track" and "Album" tables of Chinook database for three ORM tools. The total number of records used for a process can be different depending on the process. The following list shows the total number of records used in different operations.

- Read operation is executed by 10,000, 50,000, and 100,000 records.
- Insert operation is executed by 1,000, 10,000, and 25,000 records.
- Update operation is executed by 1,000, 10,000, and 25,000 records.
- Delete operation is executed by 1,000, 10,000, and 25,000 records.
- Sort operation is executed by 10,000, 50,000, and 100,000 records.
- Search operation is executed by 10,000, 50,000, and 100,000 records.
- Read operation from joined tables is executed by 10,000, 50,000, and 100,000 records.

We store the results of processing times, RAM, and CPU usage data for each ORM tool into files after executing each operation. In addition, we extract the execution plans of the queries for understanding the background database operations triggered by ORM tools.

4. RESULTS

We implement 7 operations and measure processing times, RAM usage, and CPU usage. Totally, we collect data from 21 test scenarios. Each scenario includes measurements for three ORM tools and three different record count groups. We measure processing times through calculating the seconds needed for completing the tasks. RAM usage is measured by total MBs or KBs consumed by ORM tools. CPU usage is measured by getting the percentage of the total CPU usage throughout the process.

4.1. Read Operation

Read operations are executed by reading 10,000, 50,000 and 100,000 records on Dapper, EF Core and NHibernate ORM tools. The results of these operations are given in Table 3.

Table 3. Results of "Read" operations

ORM Tool	Record Count	Processing Time (sec)	RAM Usage (MB)	CPU Usage (%)
Dapper	10,000	0.063	4	1.34
EF Core		0.091	12	2.09
NHibernate		0.097	12	3.14
Dapper	50,000	0.143	19	4.76
EF Core		0.265	60	7.79
NHibernate		0.400	64	11.64
Dapper	100,000	0.245	37	8.21
EF Core		0.465	121	11.87
NHibernate		0.803	128	16.6

Figure 2.a shows the processing time results of 10,000, 50,000, and 100,000 record reading operations of different

ORM tools. This figure shows that, Dapper is the fastest ORM tool for reading operations, whereas NHibernate is the slowest. These results are similar for 10,000, 50,000, and 100,000 record reading operations.

Figure 2.b shows that Dapper uses the least amount of memory, whereas NHibernate uses the highest. The difference between NHibernate and EF Core is small according to the difference between them in processing times.

Similarly, Figure 2.c shows that Dapper uses the least CPU percentage, whereas NHibernate uses the highest. The difference between ORM tools for CPU usage is more significant according to the difference between them in RAM usage.

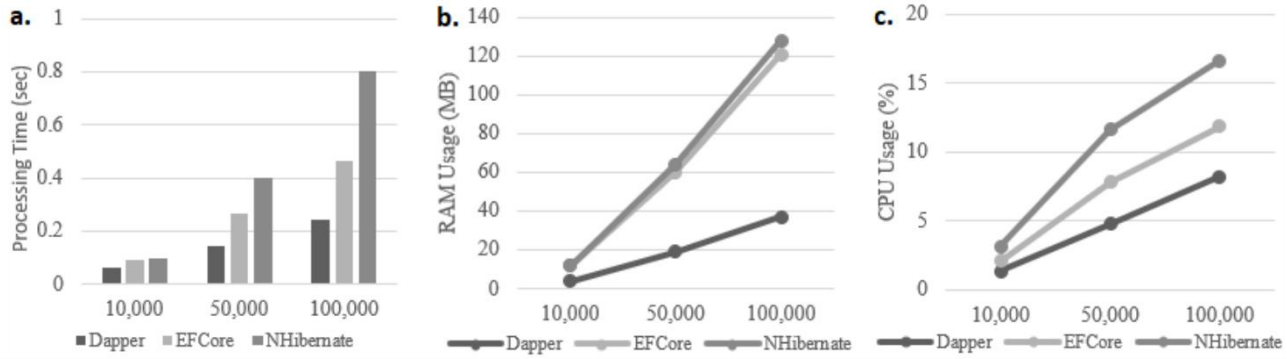

Figure 2. Results of "Read" operations

4.2. Insert Operation

Insert operations are executed by inserting 1,000, 10,000 and 25,000 records on Dapper, EF Core and NHibernate ORM tools. The results of these operations are given in Table 4.

Figure 3.a shows that EF Core is the fastest ORM tool in all the record count groups for insert operation. However, Dapper is the slowest one. According to Figure 3.b, ORM tools show quite the opposite performance in terms of RAM usage. Dapper uses the smallest amount of RAM whereas EF Core uses the largest amount of RAM. CPU usage performance of ORM tools in Figure 3.c, shows interesting results. EF Core performs better than NHibernate for 1,000 records. However, for 10,000 and 25,000 records NHibernate performs better than EF Core. Dapper has the best performance for CPU usage in all record counts.

Table 4. Results of "Insert" operations

ORM Tool	Record Count	Processing Time (sec)	RAM Usage (MB)	CPU Usage (%)
Dapper	1,000	0.207	2	1.75
EF Core		0.110	15	2.75
NHibernate		0.174	9	3.89
Dapper	10,000	2.023	20	4.35
EF Core		1.012	146	9.35
NHibernate		1.642	94	8.84
Dapper	25,000	5.249	49	4.43
EF Core		2.483	362	11.23
NHibernate		4.283	236	9.93

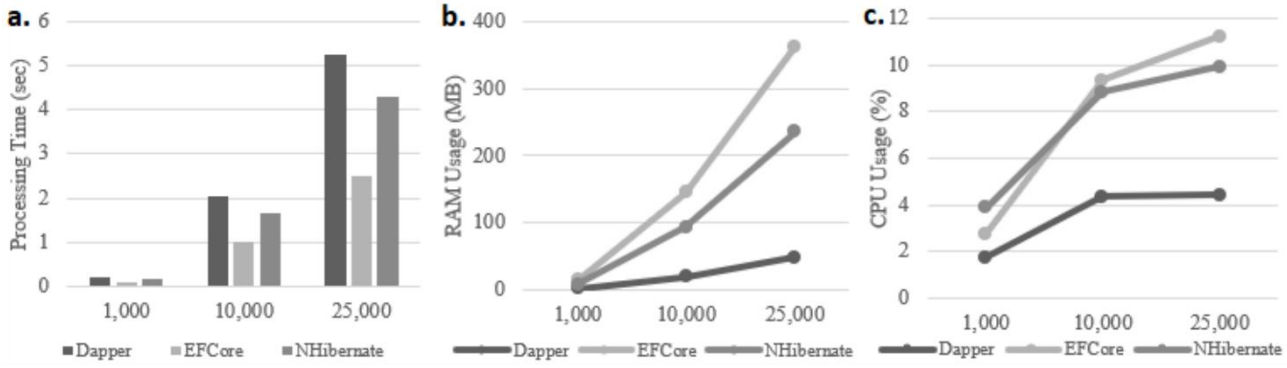

Figure 3. Results of "Insert" operations

4.3. Update Operation

We test Dapper, EF Core and NHibernate ORM tools for update operation by updating 1,000, 10,000, and 25,000 records. The results are shown in Table 5.

According to Figure 4.a, EF Core is the fastest ORM tool in all record count groups for update operations. Dapper shows the worst performance in terms of processing times. This shows that, while Dapper has a very good performance for reading operation, it doesn't have good performance for insert and update operations. However, Dapper is the best tool for RAM usage of update operations. NHibernate also consumes almost similar amount of RAM with Dapper. On the contrary to the performance on processing time, EF Core uses much memory, especially for 10.000 and 25.000 records (Figure 4.b). EF Core shows the worst performance in terms of CPU usage. Dapper is very efficient in CPU usage for all record count groups (Figure 4.c).

Table 5. Results of "Update" operations

ORM Tool	Record Count	Processing Time (sec)	RAM Usage (MB)	CPU Usage (%)
Dapper	1,000	0.244	2	1.50
EF Core		0.126	14	2.49
NHibernate		0.175	3	2.45
Dapper	10,000	2.398	22	2.98
EF Core		1.310	127	7.44
NHibernate		1.726	25	6.56
Dapper	25,000	6.186	54	3.21
EF Core		3.130	313	8.56
NHibernate		4.514	64	7.4

4.4. Delete Operation

Dapper, EF Core and NHibernate ORM tools are used for deleting 1,000, 10,000 and 25,000 records. Table 6 shows the results of these delete operations.

According to the results in Figure 5.a Dapper is very fast in all record count groups. Even though EF Core is the slowest for 1,000 records, NHibernate performs worst with 10,000 and 25,000 records. The processing time of NHibernate for 25,000 record deletion is almost five times higher than deleting 10,000 records. Dapper is also the best ORM tool in terms of RAM usage for 10,000 and 25,000 records. Interestingly, for 1,000 records NHibernate uses less RAM than Dapper and EF Core (Figure 5.b). It is beyond any doubt that Dapper is the best tool in terms of CPU usage (Figure 5.c). However, EF Core and NHibernate tools produce different results for RAM usage and CPU usage. While NHibernate is better than EF Core for RAM usage, EF Core performs better in terms of CPU usage. NHibernate shows a drastic change in CPU usage when record count increase from 1,000 to 10,000.

Table 6. Results of "Delete" operations

ORM Tool	Record Count	Processing Time (sec)	RAM Usage (KB)	CPU Usage (%)
Dapper	1,000	0.005	3.0	0.33
EF Core		0.091	6.5	0.99
NHibernate		0.030	1.9	1.29
Dapper	10,000	0.040	3.0	0.33
EF Core		0.789	61.4	3.92
NHibernate		1.841	19.1	19.31
Dapper	25,000	0.112	3.0	0.35
EF Core		1.820	150.8	5.2
NHibernate		11.567	47.6	23.58

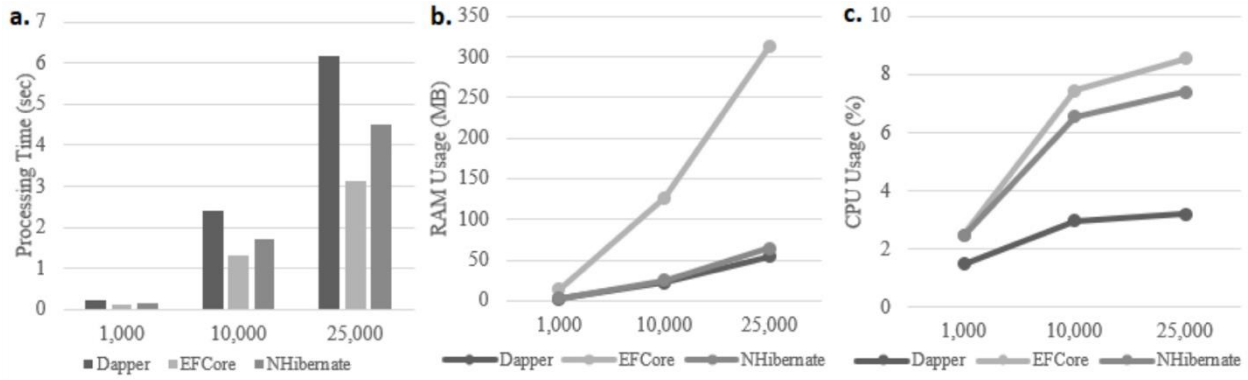

Figure 4. Results of "Update" operations

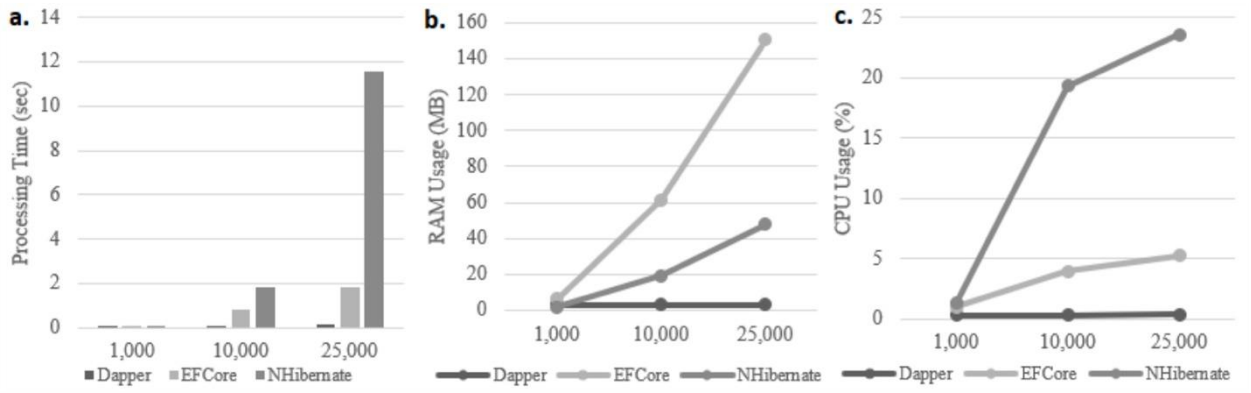

Figure 5. Results of "Delete" operations

4.5. Search Operation

We measure search operations with Dapper, EF Core and NHibernate ORM tools by using 10,000, 50,000, and 100,000 records. We search "Song" word in the "Track" table. The results are shown in Table 7.

The results in Figure 6.a shows that Dapper implements the fastest search in all record count groups. The slowest ORM tool is EF Core. Dapper is also better in RAM usage (Figure 6.b). However, for RAM usage, EF Core and NHibernate are close to each other. EF Core and NHibernate tools use RAM almost 3-4 times higher than Dapper depending on the record counts. On the contrary to processing time and RAM usage, Dapper, even though it is still the best, uses CPU time almost similar with the EF Core and NHibernate. EF Core is slightly better than NHibernate in terms of CPU usage (Figure 6.c).

Table 7. Results of "Search" operations

ORM Tool	Record Count	Processing Time (sec)	RAM Usage (KB)	CPU Usage (%)
Dapper	10,000	0.003	34	0.36
EF Core		0.013	136	0.38
NHibernate		0.005	115	0.40
Dapper	50,000	0.009	154	0.38
EF Core		0.021	492	0.44
NHibernate		0.011	493	0.46
Dapper	100,000	0.016	307	0.44
EF Core		0.032	952	0.52
NHibernate		0.019	979	0.52

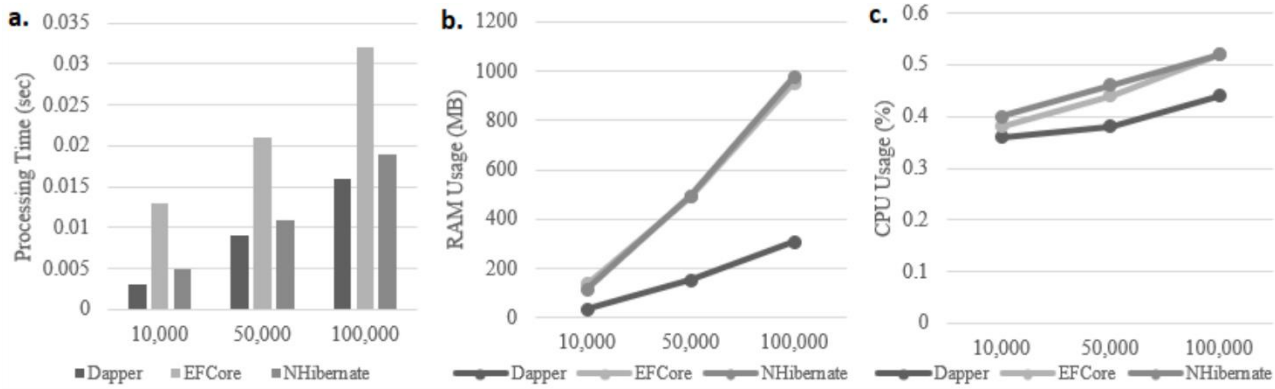

Figure 6. Results of "Search" operations

4.6. Sort operation

We test Dapper, EF Core and NHibernate ORM tools for sorting 10,000, 50,000 and 100,000 records. A descending (from Z to A) sort operation is applied to "Name" field of the "Track" table. The results are shown in Table 8.

Figure 7.a shows that the fastest ORM tool is Dapper. NHibernate and EF Core show almost similar processing time performance for 10,000 records. However, for 50,000 and 100,000 records EF Core is faster than NHibernate. Dapper is still better for RAM usage (Figure 7.b) and CPU usage (Figure 7.c) according to EF Core and NHibernate. Dapper's RAM usage 3-4 times lower than EF Core and NHibernate. EF Core and NHibernate show similar performances for RAM usage. This is not the case for CPU usage, because EF Core has significantly lower CPU usage than NHibernate.

Table 8. Results of "Sort" operations

ORM Tool	Record Count	Processing Time (sec)	RAM Usage (MB)	CPU Usage (%)
Dapper	10,000	0.115	4	1.20
EF Core		0.144	12	2.06
NHibernate		0.146	12	2.83
Dapper	50,000	0.401	19	3.14
EF Core		0.539	60	5.65
NHibernate		0.638	64	8.96
Dapper	100,000	0.884	37	4.62
EF Core		1.097	121	7.50
NHibernate		1.407	128	11.76

Table 9. Results of "Join" operations

ORM Tool	Record Count	Processing Time (sec)	RAM Usage (MB)	CPU Usage (%)
Dapper	10,000	1.449	5	0.42
EF Core		1.456	5	0.34
NHibernate		1.511	14	1.1
Dapper	50,000	1.478	25	2.04
EF Core		1.535	23	1.18
NHibernate		1.618	68	5.88
Dapper	100,000	1.594	50	3.88
EF Core		1.610	46	2.25
NHibernate		1.945	137	10.27

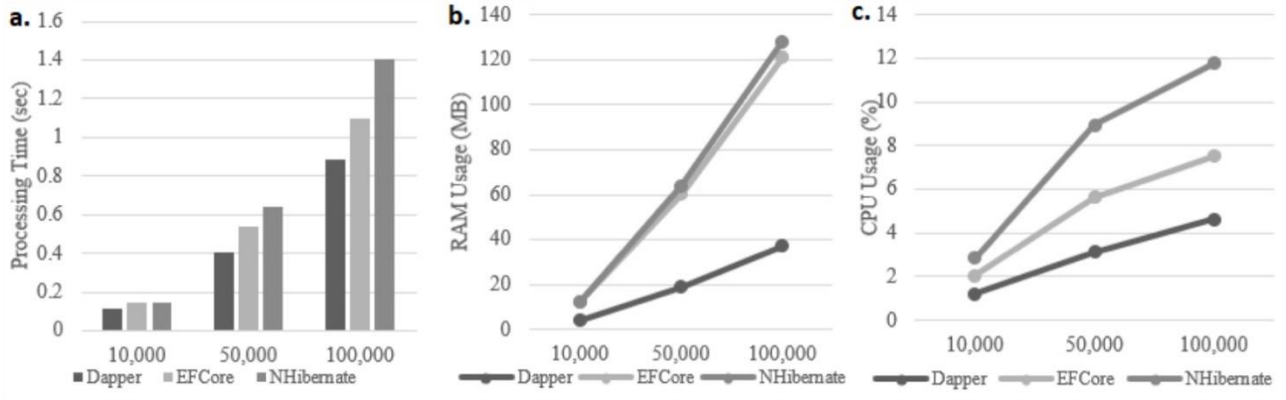

Figure 7. Results of "Sort" operations

4.7. Read operation from joined tables

We test Dapper, EF Core and NHibernate ORM tools for a join operation by reading 10,000, 50,000 and 100,000 records. We execute an inner join between "Track" and "Album" tables based on the album identifier attribute. The results are shown in Table 9

According to the results shown in Figure 8.a all ORM tools show similar performances for the processing time of 10,000 and 50,000 record readings. However, NHibernate reading operation takes longer than others for 100,000 records. Dapper and EF Core use almost similar RAM resources as shown in Figure 8.b. NHibernate consumes much memory especially for 50,000 and 100,000 records. CPU usage results are almost similar to the RAM usage. While NHibernate has the worst performance, EF Core is a little bit better than Dapper which is shown in Figure 8.c.

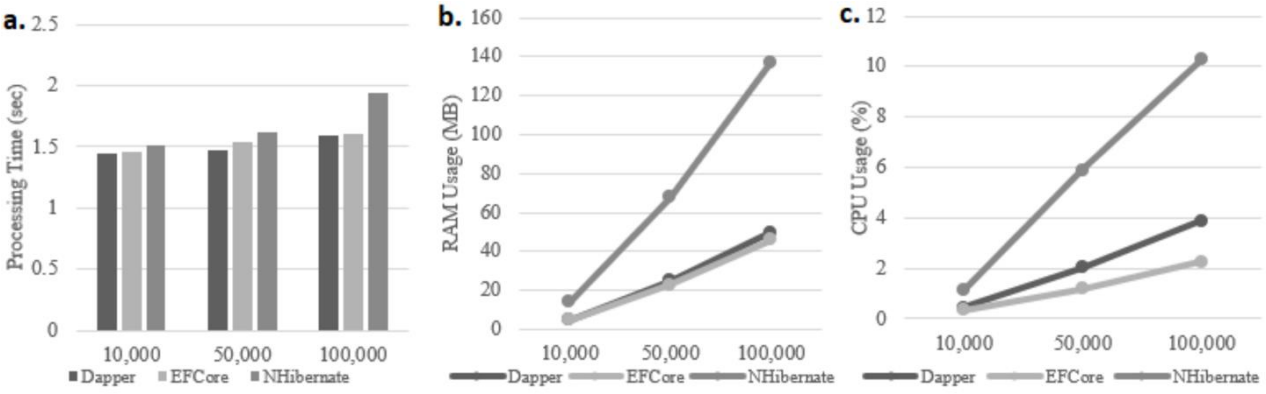

Figure 8. Results of "Join" operation

4.8. Background operations

We examine the execution plans of the queries for understanding the background database operations triggered by ORM tools. We extract these execution plans by using the Session Manager of Dbeaver and Server Monitor of Navicat (see Table 10). We successfully access the execution plans for read, sort, search and reading from joined tables operations. However, for insert, update and delete operations we are able to access only to the execution plans of Dapper. For EF Core and NHibernate, we are able to catch only commit/rollback operations. In reading from one table operation, all ORM tools apply sequential scans. Similarly, for search operation, all ORM tools apply sequential scan by adding conditions. In sorting operation, all ORM tools first apply sorting and then apply sequential scan. For reading from joined tables operation, all ORM tools apply a nested loop first. Then, a sequential scan is applied on "Track" table. After then, "Momoize" operation, which is an optimization technique for making efficient computation, is applied. The plan is completed by applying index scan on "Album" table. The analysis of execution plans shows that the difference between ORM tools is not due to the operations at the database side. The techniques applied by the ORM tools such as mapping, caching etc. make the difference.

5. CONCLUSION AND DISCUSSION

In this study, we implement a performance analysis of specific ORM tools in .NET 6 environment. We measure the processing time, RAM usage and CPU usage for 7 operations including read, insert, update, delete, search, sort and a read from joined tables. We develop a software, ORMPY, for implementing these measurements. We give results of 21 different scenarios implemented with Dapper, EF Core, and NHibernate ORM tools and discuss the results with graphics.

According to performance analysis results, Dapper shows the best performance in terms of processing times for read, delete, search and sort operations. EF Core performs best for insert and update operations in terms of processing times. The worst performance for processing times is handled by NHibernate on read and sort operations and Dapper on insert and update operations. The ORM tools that complete the delete operation in the longest time differ according to the number of records. EF Core performs the worst for 1,000 records, and NHibernate for 10,000 and 100,000 records based on delete operation processing time. For reading from joined tables operation, Dapper, again, has the best results while EF Core and NHibernate come later. However, the gap between ORM tools for reading from joined tables operation is very small according to reading from one table operation. According to the RAM usage results of ORM tools, Dapper performed the best by consuming the lowest RAM resources in all operations except reading from joined tables operation. NHibernate showed almost similar performance with Dapper in terms of RAM usage in update operations. The worst performances of RAM usage are handled by NHibernate for read, search, sort and reading from joined tables operations and by EF Core for insert, update, and delete operations. In RAM usage, Dapper has not been the worst performing ORM tool for any operation. NHibernate uses less RAM than Dapper for only deleting 1,000 records. Interestingly, EF Core outperforms other tools for reading from joined tables operation. This means that, if the query becomes more complicated, EF Core starts performing better than others in terms of RAM usage. This may occur due to advanced coding techniques of the EF-Core. We examined the logs produced by BenchmarkDotNet for the Garbage Collector (GC) operations. GC executes to release the memory for objects that are no longer used by the applications. If the memory is not used efficiently, GC performs frequently. Logs show that, for EF Core operations GC performs less than other ORM operations. According to the CPU usage results of ORM tools, Dapper performs best for all operations except the reading from joined tables operation. The worst performances of CPU usage are handled by NHibernate for read, delete, and sort operations and by EF Core for update operation. For insert operation, while EF Core is better than NHibernate for 1,000 records, NHibernate is better than EF Core for other record counts. For search operation, while EF Core is slightly better than NHibernate for 10,000 and 50,000 records, NHibernate and EF Core show the same performance for 100,000 records. EF Core uses less CPU for reading from joined tables operation. Again, this means that, if the query becomes more complicated, EF Core starts performing better than others in terms of CPU usage. This result may also be related with the GC operations. When GC starts, it suspends the application, releases the unnecessary objects and resumes the application. Less GC execution means less suspension and faster execution times for EF Core.

It is very hard to compare the results of our study with other studies. The test environments, databases that are used and operations are not standard within these studies. For example, in [4], one of the insert scenarios includes three insert operations to the tables in one-to-one relationship. In this scenario NHibernate performs better in terms of processing time with all record count groups including 500, 1,000, 2,000, 5,000, and 10,000. On the contrary to our insert operation results, the results of [4] may show that for complex scenarios, NHibernate may have better performances. Similarly with processing times, memory usage statistics in [4] are not coherent with our study. While Dapper is the best tool in terms of memory usage in our study, it is not the case in [4], especially for complex operations. We can also compare our joined reading operation with the get operation of [4]. In our one-to-many joined select statement, Dapper is the fastest tool while NHibernate is the fastest in [4]. Because of these test environment and operation structure differences, we mostly use simple queries and don't apply any configuration parameter changes to ORM tools and PostgreSQL database to get results without any side effects.

Table 10. Execution plans of the operations

Oper.	ORM	Execution Plan					
		Node Type	Entity	Cost	Rows	Time (ms)	Condition
Read	Dapper	Seq Scan	track	0.00 - 271836.60	100000	1.505.325	[NULL]
	EF Core	Seq Scan	track	0.00 - 271836.60	100000	1.495.283	[NULL]
	NHibernate	Seq Scan	track	0.00 - 271836.60	100000	1.472.832	[NULL]
Insert	Dapper	ModifyTable	track	0.00 - 0.01	0	0.030	[NULL]
		Result	[NULL]	0.00 - 0.01	1	0.006	[NULL]
	No accessible data for EF Core and NHibernate						
Update	Dapper	ModifyTable	track	8.43 - 12.44	0	0.033	[NULL]
		Bitmap Heap Scan	track	8.43 - 12.44	1	0.016	[NULL]
		Bitmap Index Scan	PK_Track	0.00 - 8.43	1	0.009	(track_id = 1)
	No accessible data for EF Core and NHibernate						
Delete	Dapper	ModifyTable	track	11.51 - 15.53	0	0.074	[NULL]
		Nested Loop	[NULL]	11.51 - 15.53	1	0.058	[NULL]
		Aggregate	[NULL]	3.08 - 3.09	1	0.038	[NULL]
		Subquery Scan	[NULL]	0.00 - 3.08	1	0.034	[NULL]
		Limit	[NULL]	0.00 - 3.07	1	0.029	[NULL]
		Seq Scan	track	0.00 - 271836.60	1	0.028	[NULL]
		Bitmap Heap Scan	track	8.43 - 12.44	1	0.010	[NULL]
		Bitmap Index Scan	PK_Track	0.00 - 8.43	1	0.007	(track_id = "ANY_subquery". track_id)
	No accessible data for EF Core and NHibernate						
Search	Dapper	Seq Scan	track	0.00 - 272058.00	742	1.507.293	((name)::text ~~ '%Song%'::text)
	EF Core	Seq Scan	track	0.00 - 272279.40	742	1.549.108	(strpos((name)::te xt, 'Song'::text) > 0)
	NHibernate	Seq Scan	track	0.00 - 272058.00	742	1.533.420	((name)::text ~~ '%Song%'::text)
Sort	Dapper	Sort	[NULL]	283051.24 - 283272.64	100000	2.381.069	[NULL]
		Seq Scan	track	0.00 - 271836.60	100000	1.610.319	[NULL]
	EF Core	Sort	[NULL]	283051.24 - 283272.64	100000	2.212.061	[NULL]
		Seq Scan	track	0.00 - 271836.60	100000	1.486.284	[NULL]
	NHibernate	Sort	[NULL]	283051.24 - 283272.64	100000	2.188.023	[NULL]
		Seq Scan	track	0.00 - 271836.60	100000	1.471.869	[NULL]
Join	Dapper	Nested Loop	[NULL]	0.16 - 274098.82	100000	1.497.240	[NULL]
		Seq Scan	track	0.00 - 271836.60	100000	1.453.257	[NULL]
		Memoize	[NULL]	0.16 - 0.18	1	0.000	[NULL]
		Index Scan	album	0.15 - 0.17	1	0.001	(album_id = t.album_id)
	EF Core	Nested Loop	[NULL]	0.16 - 274098.82	100000	1.619.155	[NULL]
		Seq Scan	track	0.00 - 271836.60	100000	1.577.679	[NULL]
		Memoize	[NULL]	0.16 - 0.18	1	0.000	[NULL]
		Index Scan	album	0.15 - 0.17	1	0.001	(album_id = t.album_id)
	NHibernate	Nested Loop	[NULL]	0.16 - 274098.82	100000	1.537.926	[NULL]
		Seq Scan	track	0.00 - 271836.60	100000	1.496.500	[NULL]
		Memoize	[NULL]	0.16 - 0.18	1	0.000	[NULL]
		Index Scan	album	0.15 - 0.17	1	0.001	(album_id = track0_.album_id)

In terms of processing time, Dapper gives the best results in 5 of 7 operations including read, delete, search sort and reading from joined tables. However, Dapper could not win the feature of being the best ORM tool in processing times due to the worst results in insert and update operations. EF Core gives the best result in all record count groups to the nearest ORM tool, with a margin of about 70% for inserts and about 40% for updates. In terms of transaction times, the NHibernate ORM tool doesn't have the best performance for any operation. As a result, Dapper ORM tool should be used to get the fastest results in applications where operations such as reading, searching, and sorting will be done intensively. In applications which adding and updating operations will be carried out intensively, the EF Core ORM tool should be used to get the fastest results.

In terms of RAM usage, Dapper gives the best results in 6 out of 7 operations while EF Core performs better only in the reading from joined tables operation. However, in reading from joined tables operation, there are very close consumption results between EF Core and Dapper in terms of RAM usage.

Dapper gives the best results in terms of CPU usage in all operations except reading from joined tables operation. Moreover, it performs 2 times or better than other ORM tools in many operations. However, from these results, we can infer that when queries become more complicated, EF Core outperforms others in terms of CPU and RAM usage.

If we evaluate the RAM usage and CPU usage results together, Dapper is still the ORM tool that gives the best results in terms of resource usage. Dapper ORM tool should be preferred in software projects developed with .NET 6 where resource consumption is higher.

As a result of our research, Dapper provides the best performance within the ORM tools in .NET 6 environment, considering the processing time, RAM usage, and CPU usage. When choosing Dapper, it should be considered that some of its features are clipped as a micro ORM tool. If a full-featured ORM tool needs to be used, the ORM tool that we can get the best performance is EF Core.

According to the execution plan results, the differences between ORM tools are not due to the operations at the database side. Databases apply almost similar execution plans for simple operations. The main reason for the differences is the techniques applied by the ORM tools such as mapping, caching etc.

Different conclusions about different ORM tools can be obtained by doing similar research to ours. The following list can be given as research recommendations in this regard:

- Analysis studies can be done with different or new ORM tools.
- Performance analyzes can be made with different databases other than PostgreSQL.
- ORM tools performance analysis can be performed for different software development environments.
- A benchmark suite can be developed by standardizing operations, software configuration parameters and hardware characteristics of the test computers.

REFERENCES

[1] A. Gruca, P. Podsiadlo, "Performance Analysis of. NET Based Object–Relational Mapping Frameworks", **10th International Conference Beyond Databases, Architectures, and Structures (BDAS 2014)**, Ustron, Poland, 40-49, May 27-30, 2014.

[2] S. M. Bhatti, Z. H. Abro, F. R. Abro, "Performance Evaluation of Java Based Object Relational Mapping Tool", *Mehran University Research Journal of Engineering and Technology*, 32(2), 159–166, 2013.

[3] M. Kopteff, "The Usage and Performance of Object Databases compared with ORM tools in a Java environment", **1st International Conference on Objects and Databases (ICOODB'08)**, Berlin, Germany, 199-218, 13-14 March, 2008.

[4] D. Zmaranda, L.-L. Pop-Fele, C. Győrödi, R. Győrödi, G. Pecherle, "Performance comparison of crud methods using net object relational mappers: A case study", *International Journal of Advanced Computer Science and Applications*, 11(1), 55–65, 2020.

[5] T. Balcı, **Entity framework'ün farklı veri tabanlarındaki performans analizi,** Yüksek Lisans Tezi, Kırıkkale Üniversitesi, Fen Bilimleri Enstitüsü, 2018.

[6] B. Irakli, A. Bardavelidze, K. Bardavelidze, "Study And Analysis Of The .Net Platform-Based Technologies For Working with the Databases", **33rd International Conference on Information Technologies**, Bulgaria, 1-8, 19-20 September, 2019.

[7] S. Cvetkovic, D. S. Janković, "A comparative study of the features and performance of orm tools in a. net environment.", **3rd International Conference on Object and Databases (ICOODB'10)**, Frankfurt am Main, Germany, 147–158, 28-30 September, 2010.

[8] J. Martin, **Managing the Database Environment**, Prentice Hall, New Jersey, 1983.

[9] A. Joshi, S. Kukreti, "Object Relational Mapping in Comparison to Traditional Data Access Techniques", *International Journal of Scientific & Engineering Research*, 5(6), 540–543, 2014.

[10] V.Sivakumar, T.Balachander, Logu, R. Jannali, "Object Relational Mapping Framework Performance Impact", *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12(7), 2516-2519 2021.

[11] H. Yousaf, **Performance evaluation of java object-relational mapping tools**, Yüksek Lisans Tezi, The University of Georgia Graduate Faculty, 2012.

[12] D. Colley, C. Stanier, and M. Asaduzzaman, "The Impact of Object-Relational Mapping Frameworks on Relational Query Performance", **International Conference on Computing, Electronics & Communications Engineering (iCCECE)**, Southend, UK, 47-52, August 2018.

[13] Internet: Microsoft technical documentation, Compare EF Core & EF6, https://docs.microsoft.com/tr-tr/ef/efcore-and-ef6/, 25.01. 2021.

[14] Internet: A. Shapovalov, Micro ORM vs ORM, https://www. yaplex.com/blog/micro-orm-vs-orm/, 24.06.2022.

[15] Internet: Dapper - a simple object mapper for .Net, https://github .com/DapperLib/Dapper, 24.06.2022.

[16] Internet: Eager Loading of Related Data, https://docs.microsoft .com/en-us/ef/core/querying/related-data/eager, 24.06.2022.

[17] Internet: NHibernate Reference Documentation, Chapter 21 Improving performance, https://nhibernate.info/doc/nhibernate-reference/performance.html#performance-fetching, 24.06.2022.

[18] Internet: D. Paquette, Loading Related Entities: Many-to-One, https://www.davepaquette.com/archive/2018/02/06/loading-related-entities-many-to-one.aspxreference/performance.html#performance-fetching, 24.06.2022.

[19] Internet: A. Chan, High Performance Reflection ORM Layer, https://www.codeproject.com/Articles/22188/High-Performance-Reflection-ORM-Layer, 24.06.2022.

[20] Internet: PostgreSQL 14.1 Documentation, What is PostgreSQL? https://www.postgresql.org/docs/current/intro-whatis.html, 25.01 .2021.

[21] Internet: The World's Most Advanced Open-Source Relational Database, https://www.postgresql.org/, 25.01.2021.

[22] Internet: BenchmarkDotNet Powerful .NET library for benchmarking, https://benchmarkdotnet.org/index.html, 25.01.20 21.

[23] Internet: .NET documentation, Diagnostics client library, https://docs.microsoft.com/en-us/dotnet/core/diagnostics/diagnos tics-client-library, 25.01.2021.