# Performance Comparison of CRUD Methods using NET Object Relational Mappers: A Case Study

Doina Zmaranda[1], Lucian-Laurentiu Pop-Fele[2], Cornelia Győrödi[3], Robert Győrödi[4], George Pecherle[5]
Department of Computer Science and Information Technology, University of Oradea, Oradea, Romania[1, 3, 4, 5]
Faculty of Electrical Engineering and Information Technology, University of Oradea, Oradea, Romania[2]

***Abstract*—Most applications available nowadays are using an Object Relational Mapper (ORM) to access and save data. The additional layer that is being wrapped over the database induces a performance impact in detrimental of raw SQL queries; on the other side, the advantages of using ORMs by focusing on domain level through application development represent a premise for easier development and simpler code maintenance. In this context, this paper makes a performance comparison between three of the most used ORM technologies from the .NET family: Entity Framework Core 2.2, nHibernate 5.2.3 and Dapper 1.50.5. The main objective of the paper is to make a comparative analysis of the impact that a specific ORM has on application performance when realizing database requests. In order to perform the analysis, a specific testing architecture was designed to ensure the consistency of tests. Performance evaluation for time responses and memory usage for each technology was done using the same CRUD (Create Read Update Delete) operations on the database. The results obtained proved that the decision to use one of another is dependent of the most used type of operation. A comprehensive discussion based on results analysis is done in order to support a decision for choosing a specific ORM by the software engineers in the process of software design and development.**

***Keywords—ORM (Object Relational Mapper); domain-level development; performance evaluation; CRUD (Create Read Update Delete) operations***

## I. INTRODUCTION

ORM (Object Relational Mapper) is a pattern for accessing a relational database from an object-oriented language, with several implementation for almost every language. Basic features of an ORM include support for at least one specific persistence engine and CRUD operations. Some ORM features also include custom-SQL extensions for query building. Consequently, an ORM is a library that uses the object-oriented paradigm in a specific language to write a query that will return a set of data mapped into an object type that is needed [1].

Generally, the use of ORMs will have a negative impact on the application's execution time because it is being wrapped over the relational database and, in comparison to raw SQL queries, which can be stored as procedures or functions in a database, it provides slower return time values for all types of requests. However, ORM libraries could be preferred by developers in detrimental of raw SQL queries because of the easiness of writing data accessing code, faster debugging, ORMs being more readable than raw SQL [2], thus resulting better quality software. Nowadays, there are many ORM libraries free of charge or with a paid license which are offering a great set of functionalities ready to be used out of the box and with constant updates. The main advantage of using an ORM is represented by the fact that development is focused on the domain (model) level that describes at higher level of abstraction how program data is stored and retrieved from the database, leading to easier development and code maintenance. On the other side, by introducing an additional layer, performance issues arise. Depending on the ORM, this performance downsides are introduced either by internal way of entity-model design approach or, by using reflection [3].

From several ORMs that exists nowadays, this paper focuses on the three of the most used ORMs for .NET applications development: Entity Framework Core [4], nHibernate [5] and Dapper: EF Core 2.2.3 with EF Core Proxies 2.2.3 and EF Core SqlServer 2.2.2 libraries alongside with Dapper 1.50.5 library and using Dapper.Bulk 1.4.2 for bulk operations and nHibernate 5.2.3, configured with FluentNHibernate 2.1.2. A complex analysis and comparison between these ORMs impact on application's data interrogation methods performance is presented in the paper, by analyzing multiple CRUD calls with different levels of complexity. The main objective is to provide an overall experimental study that helps developers and architects when considering the trade-off between benefits of ORMs and their performance drawbacks when developing an application.

Execution time and memory footprint are considered the metric to realize the comparisons; a specific testing architecture was developed for running the tests and comparing ORM's performance results depending on different CRUD operation. This architecture implies the development of an application that targets a custom-made database and uses a specific benchmarking library, DotNetBenchmark [6], together with ORM's specific-developed repository class, to test the execution time and memory usage of multiple CRUD operations, as well as testing on multiple runtimes.

The paper is organized as following: in the first chapter, a short introduction emphasizing the motivation of the paper is presented, followed by chapter two that reviews related work. The method and testing architecture are illustrated in chapter 3 and the obtained experimental results are presented in chapter 4. An overall analysis and discussion about the results is described in chapter 5 and finally some conclusions are drawn.

## II. Related Work

Several comparisons were done in the literature between different .NET ORM technologies, but they are generally targeting only two at the time. Solutions are generally analyzed in terms of performance, as in [7], but also in terms of their impact on application development [8].

Translation overhead for persistence operation is analyzed in [9] from the perspective of the additional layer introduced by the ORMs, by making a comparative analysis from the software development point of view of the two most used ORM tools in .NET programming environment: Entity Framework and nHibernate. Other studies analyze the benefits of using an ORM versus the drawback performance induced by ORM [10], by giving an insight look to the generated code. A study of the performance of Entity Framework and nHibernate for different types of databases (MS SQL Server and PostgreSQL) and using different query languages (lambda expressions and LINQ for Entity Framework and HQL and Critera API for NHibernate) in comparison with using SqlClient queries is presented also in [11]. Another arising issue is related to energy efficiency of ORM approaches, as it is described by the authors in [12], a study that experimentally evaluates energy efficiency of three different approaches to programmatically access SQL databases from PHP applications.

When coming to the recent versions of the .NET ORMs practical performance issues (caching, lazy loading, future queries) when building robust and scalable data access layer using NHibernate's are described in [13]; also, in [14] an approach for detection of ORM performance anti-patterns in the source code regarding database access details is presented. From the performance point of view, in [15] a fetch performance comparison by conducting experiments on common test data set of selected data access libraries: ADO.NET, Dapper and Entity Framework Core with tracking and no-tracking change is investigated. A common conclusion that results is that generally, using ORMs for application development introduce several benefits when compared to a plain SQL approach. On the other side, these techniques have well known disadvantages; but, as outlined in [8], the latest versions of skilfully developed ORMs is likely to generate well-tuned code that minimizes the performance impact on modern applications. The simpler ORMS, as Dapper, tend to work faster but exhibits fewer functionalities than the most complex ones, like EF Core or NHibernate. However, since performance issues could depend on the type and complexity of operation and on the volume of entries, none of these studies presents a comparison between all three and analyze multiple CRUD calls with different levels of complexity.

## III. Method and Testing Architecture

The method used for testing performs experimental tests for all three different ORMs: Entity Framework Core, nHibernate and Dapper. For each ORM, different type of queries (Insert, Get, Update and Delete) with different degree of complexity were run on the same database. Execution time and memory consumption were monitored over different number of entries implied in the operation. The testing architecture, used to realize the comparisons in the present study, is presented in Fig. 1. The testing architecture implies a custom developed targeted database used for benchmarking testing.

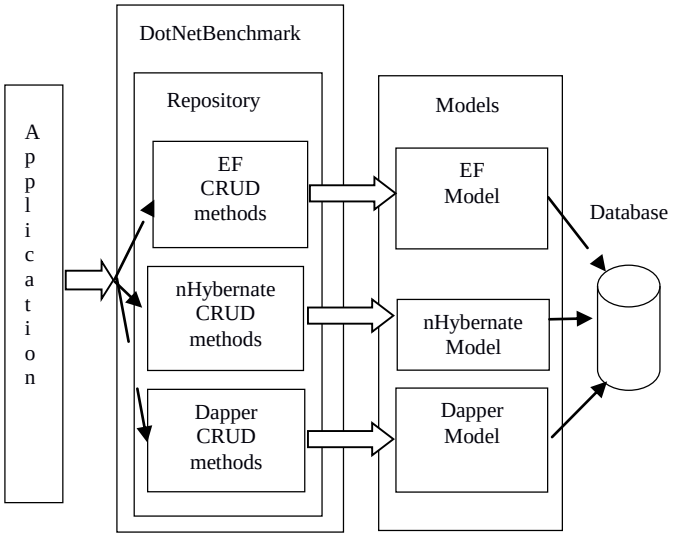


Fig. 1. Testing Architecture.

In order to separate the main concepts for each ORM technology, a project was developed as a Console Application in Visual Studio 2017 Professional. Furthermore, DotNetBenchmark specialized library was used by the application for benchmarking on the database by using different CRUD operations on multiple runs [6]. An Intel Core i7-6700HQ CPU 2.6GHz (Skylake) with 8 logical and 4 physical cores, 16 GB RAM and 256 GB SSD was used as underlying hardware support. It runs Windows 10 Pro operating system. For this purpose, a repository class was created inside the application for defining the CRUD methods for each ORM. Finally, all CRUD operations were tested through this unique application in a consistent way.

### A. The Database

MSSQL Server 2018 and SQL Server Management Studio were used for direct access and table visualization in the development process. The database schema can be observed in Fig. 2; it was created so that it includes all types of table relationships (one-to-one, one-to-many and many-to-many), in order to be able to test and observe the performance impact of those relation types on different operations.

### B. Application Project Structure

An application project *UniversityDBenchmark* was created for analysing the three different ORM technologies (Dapper, EntityFramework Core 2.2 and nHibernate); application structure was designed specifically to separate the main concepts for each used technology. The project structure is presented Fig. 3a and consists of the following modules:

- Benchmark–represents container with a class developed to configure the methods used for benchmarking
- Context–a container with a class used to configure the Entity ramework context class (UniversityContext.cs)

- Helpers–container for a class that is returning the database connection string from appsettings.json file
- Logger–container for class that provides logic to log the SQLs generated by the entity framework methods
- Repository–a container with repository classes that contains CRUD methods for accessing the database for each ORM
- Program.cs–represents the main gate to the Console application, used to call to run benchmark analysis methods

### C. Benchmark Analysis

The benchmarking was realized using the specialized library DotNetBenchmark which is offering means of testing the execution time of multiple methods as well as testing different runtimes, as described in [6]. The BenchmarkAnalysis class provides also access to each repository CRUD methods that will be dynamically called; each method was marked with the [Benchmark] attribute, by this approach telling the library which methods to include in the current benchmark session (Fig. 3b). The RankColumn defines a column in the results table, after the benchmarking process has been finished, that contains the execution time ranks in ascending order of the declared methods.

MinIterationCount/MaxIterationCount attributes are forcing the benchmarking library to execute between 10 and 20 times each method. MemoryDiagnoser is an attribute by which memory surveillance during the actual method calls is enabled, so that in the results table, the memory used to execute these methods will be shown. MinColumn and MaxColumn have the purpose of displaying the observed minimum and maximum analysis values for each tested method that will be used to compute the average final benchmark analysis value. An example about how to declare the repository methods so that it will be included into benchmarking analysis is the following (for InsertStudents method):

```
[Benchmark]
   public void InsertStudentsWithNHibernate() =>
_NHibernateRepository.InsertStudents(iterationNumber);

[Benchmark]
public void InsertStudentsWithEF() =>
   _EFRepo.InsertStudents(iterationNumber);
[Benchmark]
 public void InsertStudentsWithDapper() =>
   _DapperRepo.InsertStudents(iterationNumber);
```

For all three ORMs used for testing, the repository class implements all specific CRUD methods targeting the same tables from the database. Thus, a series of CRUD calls to the database using each targeted technology (Entity Framework Core, nHibernate and Dapper) were created, with two types of method calls: one for simple scenarios (in which 3 tables in one-to-one relationship, having 2 one-to-one relationships are targeted) and one for a more complex scenario (in which 4 tables which are in one-to-one and one-to-many relationship were targeted - 3 tables linked by 2 one-to-one relationship and 2 tables linked by a one-to-many relationship). In order to obtain the results, all these methods will be called by a defined number of times: 500, 1000, 2000, 5000 and 10000. The targeted operations are:

- INSERT–with the corresponding methods insert students and insert teachers with affiliated courses
- GET–with the corresponding methods getting a number of students and teachers and getting all students participating to a teacher's courses. This method, was run on a high number of entities
- UPDATE–with the corresponding methods update student's address and update teacher's address and courses description; before the actual update a get to take a specific number of students/teachers is run, these calls being also monitored and decreased from the overall update execution time
- DELETE–with the corresponding methods delete students and delete teachers and corresponding courses; in this case a get method to take a specific number of entries which will be targeted for deletion was run prior to deletion and its execution time was decreased from the overall deletion execution time.

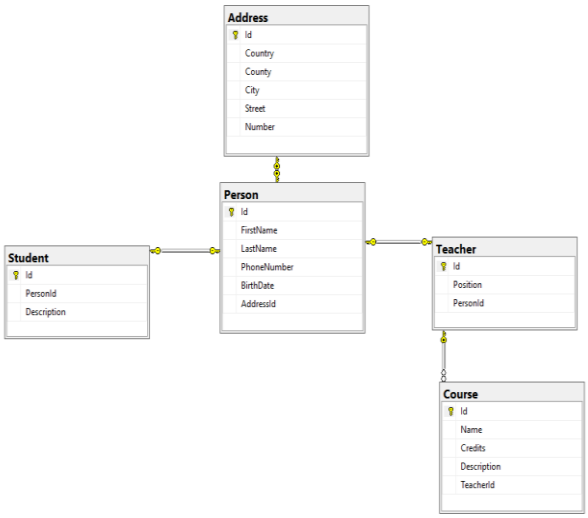


Fig. 2. University Database Schema.

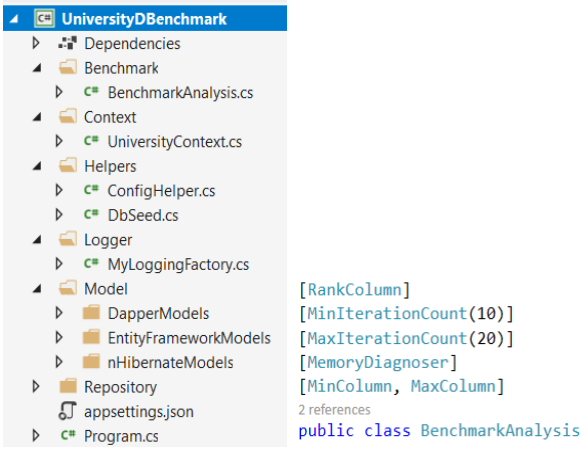


Fig. 3. (a) Project Structure. (b) Benchmark Attribute.

## IV. Experimental Results

Performance comparison was using DotNetBenchmark benchmarking library for execution time and memory allocation for each called method. Each framework will store in memory the results after running the generated SQL statements. The logic that translates the methods into equivalent SQL will also cost memory. EF Core for example is using reflection and thus, the first call will be slower and more time consuming than the rest of them because of the caching used. Dapper is not using reflection but is having logic for mapping results from the SQL that was generated and ran and the expected entities. nHibernate is also using first level of cache for optimization, but the mappings behind the scenes will put a mark on the memory consumption.

### A. Insert

Two scenarios were used for testing INSERT operation with the targeted technologies. First scenario represented by insert students method implies inserting entities that have two one-to-one relationship (Student-Person-Address) that means the logic will add first an address, person and finally a student. The second scenario, represented by insert teachers with affiliated courses method, implies inserting entities which have besides the one-to-one relationship also one-to-many (Teacher-Person-Address, Teacher->Courses), that means the logic will add first an address, person, teacher and then a list of courses for each teacher entity.

The insert method used for the first insertion scenario (*insert students*) is the same for all three technologies; the one-to-one relationship between Student-Person-Address implies that 3 inserts for each student entity insert process will be made:

```
foreach Student
    create Address;
        create Person;
        add Address to Person
            create Student;
            add Person to Student
    save Student;
```

However, some particularities were considered for each ORM:

- Entity Framework Core: since behind the scenes it uses transactions, there is no need to include the simple save statement into a new transaction;
- nHibernate: in order to make calls to the database using this technology it is necessary to manually open a session
- and a transaction and all operations to be made inside this opened transaction. After each transaction has ended, it is recommended to clear (flush) the session;
- Dapper: to benefit of using a library that is adding bulk data, the approach of separately adding entities was chosen.

The obtained results after the executing the benchmark analysis are represented in Fig. 4 and Table I. Is obviously that the best timing results for insert students scenario are obtained by nHibernate, mainly because of its simple underlying logic and also because of the simplicity of the next SQL statement which is called after an INSERT, that for nHibernate has a simple form: SELECT scope_Identity().

Dapper is also very close to nHibernate's results, being a bit slower because under the hoods, Dapper is running a lightly different SQL in the form of: SELECT CAST (scope_identity()), and that particular cast operation will mark its effect upon the total result. EF Core is on the last position because, after the INSERT, it is running a much more complex SQL query: SELECT where @@rowcount = 1 and [id] = scope_identity(), which is time-costly compared to the other two.

For the second insertion scenario (represented by insert teachers with affiliated courses) for each teacher inserted, a new address, a new person and a list of courses will be added (one-to-one/one-to-many relationships). Consequently, when adding a new teacher, a new address and a new person entity are required. The insert method used is the following:

```
foreach teacher create Address;
            create Person;
            add Address;
                create Teacher;
                add Person;
            create Courses;
            add Courses to Teacher;
    save teacher;
```

The obtained results are presented in Fig. 6 and Table I.

When adding more complex entities, nHibernate has still be best values than EF Core and Dapper, just the same as for the last scenario. Thus, from the time results that overall, nHibernate would be the best option for realising *Insert* operations followed closer by Dapper; if memory usage is considered, the best is still nHibernate followed by EF Core. Memory allocation for each technology is presented in Fig. 5 and Table II. An explanation for EF Core time results is that behind the scene, is adding the Teacher entity and then all courses are being saved into a temporary table that is merged with the Course table using a select on Course table joined with the temporary table. In comparison, Dapper and nHibernate are realizing simple inserts with the given values.

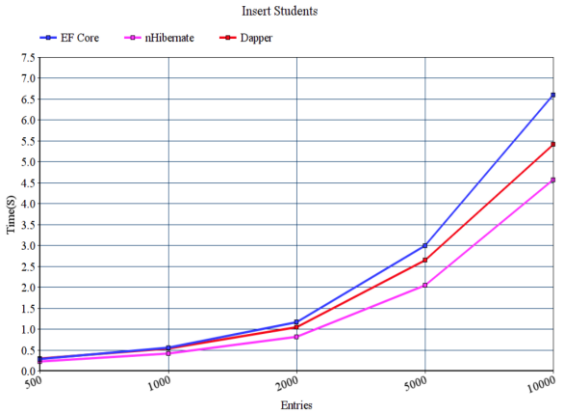


Fig. 4. Insert Students (One-to-One).

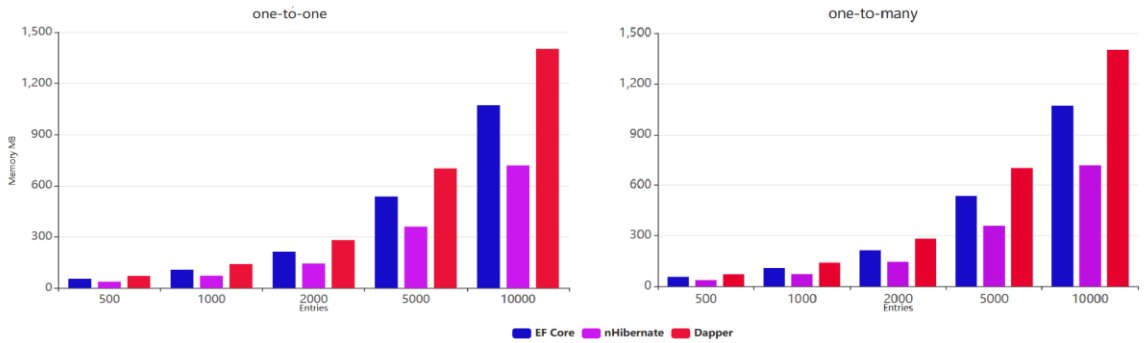


Fig. 5. Memory used – Insert.

TABLE. I. INSERT METHOD EXECUTION TIME

| Entries /Time(s) | Insert methods Time(s) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** |
| 500 | 0.28 | 0.82 | 0.22 | 0.56 | 0.29 | 0.67 |
| 1000 | 0.55 | 1.44 | 0.41 | 1.24 | 0.53 | 1.51 |
| 2000 | 1.16 | 3.21 | 0.81 | 2.29 | 1.04 | 2.56 |
| 5000 | 2.99 | 7.28 | 2.04 | 5.92 | 2.64 | 7.98 |
| 10000 | 6.6 | 15.01 | 4.75 | 11.76 | 5.41 | 13.65 |

TABLE. II. INSERT METHOD MEMORY USAGE

| Entries /MB | Insert methods memory usage (MB) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** |
| 500 | 21.93 | 53.45 | 13.1 | 29.16 | 30.83 | 66.68 |
| 1000 | 43.83 | 106.74 | 26.2 | 57.9 | 61.65 | 133.34 |
| 2000 | 86.8 | 12.85 | 52.43 | 115.88 | 123.28 | 266.67 |
| 5000 | 217.45 | 535.5 | 130.66 | 291.04 | 261.16 | 666.65 |
| 10000 | 435.57 | 1071 | 261.16 | 582.05 | 616.35 | 1333.29 |

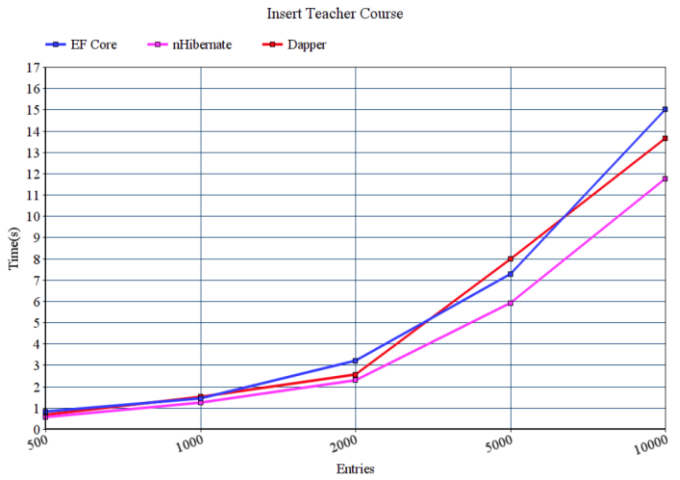


Fig. 6. Insert Teachers with Corresponding Courses (One-to-many).

## B. Update

The same testing approach from Insert was applied also to Update methods, by using two scenarios: updating students addresses and updating teacher's address and corresponding courses in order to test the update process of entries being in a one-to-one respectively one-to-many relationship. Each method will update a specific number of students and teachers that will be updated: 500, 1000, 2000, 5000, 10000. The database on which update statements were called has a constant 10.000 entries in Student, Person and Address table for the first scenario and a constant of 10.000 entries in Teacher, Person and Address and 40.000 in Course table for the second one. During benchmarking process, the Get methods will also be considered so that the exact time for the update statement alone is computed. For the first scenario of updating student's addresses, the same update statement is used for all three technologies:

*get Students (with Person and Address)*
*foreach Student*
*change Address; update Student*

All three technologies have the same logic flow, with minor differences:

- EF Core: uses the Include option to retrieve the related one-to-one entities; afterwards, a simple build-in Update method is called upon the retrieval of Student entities which will detect all changes then update them accordingly. Even there are multiple Update calls, all updates effectively take place only when context's SaveChanges() method is hit;
- NHibernate: just like EF Core, firstly it retrieves a given number of Students alongside with all related entities by using the call QueryOver; then each student's address will be changed, and the build-in Update call is triggered. Just as for EF Core, there are multiple Update calls, but only when committing the opened transaction, the statements will be triggered;
- Dapper: in particular for Dapper, all the auto-generated SQL calls that EF Core is generating behind the scenes when is getting the Student entity were re-created in order to provide a much more accurate analysis between these technologies. After getting all students and related entities, the addresses were modified then the Update method was called.

The obtained results after the execution time benchmark analysis of updating student's addresses are presented in Fig. 8 and Table III. The best timing results for *update students* scenario are obtained by EF Core that uses by default eager loading when nHibernate is using lazy loading by default: when realising a student entity update, firstly it is needed to return from database all data that did not have yet been loaded. EF Core has better timing results also in comparison with Dapper mainly because of the underlying logic from EF Core Update method, because from the code perspective, EF Core Update Students and Dapper Update Students are logically the same.

The second update scenario is approaching to update entities via both one-to-one and one-to-many relationships. In this case, Teacher's table is selected, which is linked with a one-to-one relationship with Person that is linked with Address also with a one-to-one relationship just as for Student table. Besides this link, it has a one-to-many link to Course table. During this scenario tests, the Address and a Course from each Teacher considered will be changed. A get statement needs to be run first in order to obtain all Teachers that need update and simulate through this multiple update calls.

*get Teachers;*
*foreach Teacher get Courses;*
*foreach Teacher;*
*update Address;*
*update Course.Description; update Teacher;*

As it results from Fig. 9 and Table III, all three technologies are having higher time results for updating teacher's courses in comparison with updating the Student table, but nHibernate has the biggest time result from all three. This can be explained also by the fact that nHibernate is using by default lazy loading and so, even if all teachers were returned with an initial *Get* call (simple Select from database) a call the database every time other inner-entities from Teacher object are accessed is needed.

Therefore, the overall timing for the nHibernate update call is increasing. Entity Framework and Dapper are using eager loading and therefore they do not need to access the database each time an object which was not previously loaded is changed. Nevertheless, EF Core, in this case, is slightly overcome by Dapper but the difference is relatively small.

The update methods memory allocation for each technology used is presented in Fig. 7 and Table IV. For the one-to-one approach, the the consequences of using lazy loading by default for nHibernate can be seen, that implies extra memory usage when realising the Update statement because of the need to return and save the entities in the same Update call before saving the changes.

When also one-to many relationships are involved, EF Core and Dapper are having higher memory usage when updating entities than nHibernate because of the used underlying logic. Consequently, EF Core is generally having the best results from both time and memory perspectives followed closely by Dapper, except the situation of memory usage, where slightly higher memory consumption can be seen when updating entities with one-to-many relationship to EF Core.

TABLE. III. UPDATE METHOD EXECUTION TIMES

| Entries /Time(s) | Update methods Time(s) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** |
| 500 | 0.081 | 0.703 | 0.176 | 2.61 | 0.175 | 0.45 |
| 1000 | 0.191 | 1.625 | 0.361 | 5.78 | 0.339 | 0.88 |
| 2000 | 0.38 | 3.23 | 0.726 | 11.56 | 0.654 | 1.75 |
| 5000 | 1.03 | 7.36 | 2.03 | 25.27 | 1.66 | 2.48 |
| 10000 | 1.85 | 14.53 | 3.96 | 56.31 | 3.51 | 5.12 |

TABLE. IV. UPDATE METHOD USED MEMORY

| Entries /MB | Update methods used memory (MB) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** |
| 500 | 7.6 | 52 | 19.2 | 35 | 8.84 | 60.24 |
| 1000 | 15.2 | 104.72 | 38.4 | 69.61 | 17.68 | 120.46 |
| 2000 | 30.4 | 211.57 | 76.8 | 138.77 | 35.37 | 240.97 |
| 5000 | 76 | 537.62 | 192 | 494.56 | 88.67 | 602.28 |
| 10000 | 159 | 1075.24 | 374 | 989.12 | 177.08 | 1125.93 |

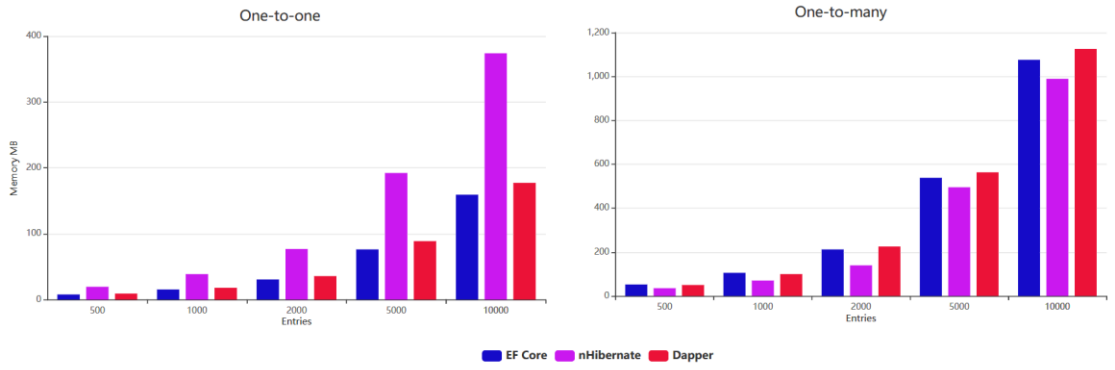


Fig. 7. Memory used – Update.

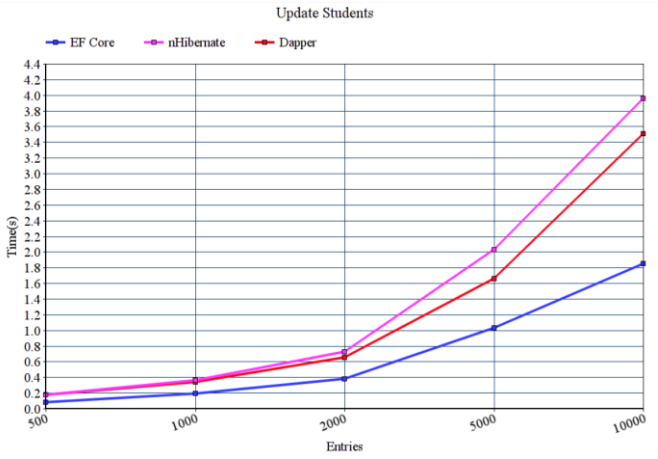


Fig. 8. Update Students (One-to-One).

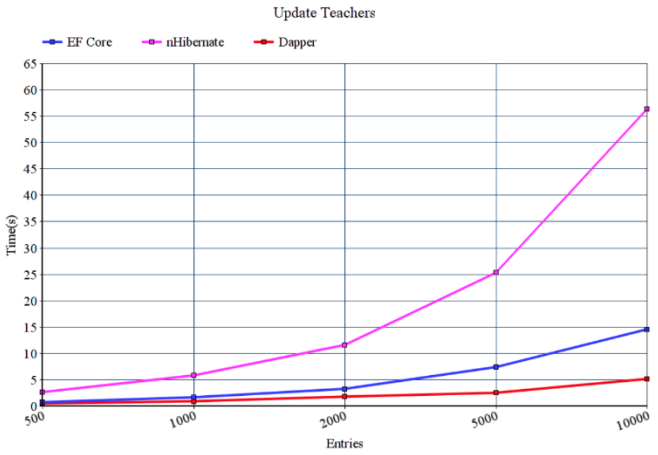


Fig. 9. Update Teachers with Corresponding Courses (One-to-Many).

For nHibernate in particular, as it is using lazy loading by default when realising an Update, if the targeted entity is not yet loaded, it will make a Select statement first, the will update the returned entity's values with the new ones, then will realise the UPDATE on the database. All these operations are very costly from the execution time point of view.

When looking at memory usage, a considerable difference when updating entities with one-to-one relationship versus one-to-many can be observed. This could be explained by the underlying logic which translates the .NET code into SQL statements because, when checking with SQL Profiler, in all cases, simple SQL *Update* statements are being made.

### C. Delete

Delete operation follows the same approach by taking into consideration two scenarios: delete students and delete teachers and corresponding courses. Delete students implies deleting entries from student, person and address tables involving only one-to-one relationships; delete teachers implies deleting entries from teacher, person and address tables including corresponding courses (both on-to-one and one-to-many relationships). As in the previous methods that were analysed, a Select statement to get top 500/1000/2000/5000/10000 entries from Student and Teacher tables with all related entries will be previously run. The select queries will be then measured and extracted from the total results, after a delete process, in order to obtain only delete operation values. For the first scenario of deleting all students, the same delete statement is used for all three targeted technologies:

*get Students and related entities;*
*for each Student*
*delete Address; delete Person; delete Students;*
*end;*

Just as for *Insert* and *Update* cases, also *Delete* operation has some particularities depending each targeted technology:

- EF Core: because of the foreign key AddressId present in Person table and PersonId present in Student table, the Address table is seen as the parent, so deleting the Address will automatically cascade delete also to the corresponding Person and Student
- NHibernate: because of the mapping classes, entities for cascade deletion can be directly marked and consequently, by simply deleting the teacher entity, all other related entities will be deleted; this behaviour is different when compared to EF Core and Dapper where in order to trigger the cascade deletion it is needed to delete the parent entity;
- Dapper: the simple Delete method from SimpleCRUD Dapper library was used to remove the parent Address entity and the BulkDelete method from Dapper-PLUS library was used to remove all related Teacher Course entities just as for EF Core case.

Before making the deletion statements, a *Select* query will be run to return a top 500/1000/2000/5000/10000 teachers from the database whose execution time was measured so that only the delete statement time for each used technology could be computed. From Fig. 11 and Table V it is obvious that EF Core is having the best timing results, close to Dapper, nHibernate having the worst time results in this case. NHibernate in this case is affected by the lazy loading default setting by having to load all yet unloaded entities before deleting them. Dapper and EF Core are having very close timing results, as the simplest delete option from each of them has been used, the difference between them could be explained by the logic behind the scenes which is transforming the code to SQL statements.

The second scenario used for deletion is running the Delete queries with the purpose of removing a given number of teachers with their corresponding courses. This will require a deletion of two one-to-one relationship (Teacher-Person-Address) and one one-to-many relationship (Teacher-Courses):

*get Teachers and related entities;*
*foreach Teacher*
*delete Address; delete Person;*
*delete Teacher; delete Courses (if any);*
*end;*

In this scenario, Dapper and EF Core have close results for delete statements and much higher than nHibernate for the case of deleting one-to-many entities, where nHibernate is

having the best results in comparison with the case when is deleting only one-to-one relationship (Fig. 12 and Table V). This could be explained mainly because of the underlying logic for Dapper and EF Core that translates the Remove (EF Core) / BulkDelete (Dapper) methods into explicit delete SQL statement that could have an impact on execution time.

Dapper and EF Core are having better deleting timing results than nHibernate until reaching the level of 1000 deleted entries. By analysing the resulted SQL statements after a deletion command, the SQLs are very much the same for all three technologies, with the exception for EF Core that is running also a Select @@Rowcount to check return the number of effected entries. The delete methods memory allocation for each technology used is presented in Fig.10 and Table VI.

According to all these results, nHibernate has the highest memory usage when deleting entities in both one-to-one and one-to-many relationship. An explanation for this massive memory usage in comparison with EF Core and Dapper could be the fact that nHibernate is using lazy loading by default.

Consequently, instead of the real objects, it has some proxys that will be replaced with real object upon accessing their values.

Both EF Core and Dapper have good memory usage results, Dapper being more efficient than EF Core when deleting entities in one-to-many relationship, maybe because, behind the scenes EF Core and Dapper are having a better logic implemented than nHibernate and possible because EF Core and Dapper are using eager loading when returning entities without the need of the proxy objects that nHibernate is using to replace the objects not yet returned from the database.

### *D. Get*

Get queries have been already run in the scenarios presented before to obtain a specific number of entities on which update or delete queries where run afterwards. Two scenarios were considered also here: getting a number of students and getting all students participating to a teacher's courses to return entities with one-to-one and one-to-many relationships respectively. These methods were run on a higher number of entities. Consequently, first the Get calls were made upon a database with 10.000 entries on Student table, then cleared the database and its cache and re-entered 10.000 Teacher entities alongside with 40.000 Courses (each teacher will have 4 courses). The is the following:

get Students (including Person and Address);

TABLE. V. DELETE METHOD EXECUTION TIMES

| Entries /Time(s) | Delete methods Time(s) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** |
| 500 | 0.15 | 2.61 | 0.38 | 3.59 | 0.19 | 2.84 |
| 1000 | 0.28 | 8.43 | 0.72 | 6.36 | 0.31 | 9.34 |
| 2000 | 0.64 | 20.07 | 1.54 | 13.13 | 0.68 | 27.05 |
| 5000 | 1.49 | 39.12 | 2.82 | 26.72 | 1.74 | 53.75 |
| 10000 | 2.81 | 75.16 | 5.78 | 53.44 | 3.56 | 108.43 |

TABLE. VI. DELETE METHOD USED MEMORY

| Entries /MB | Delete methods used memory (MB) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** | ***Students*** | ***Teachers*** |
| 500 | 12.78 | 4.21 | 27.98 | 87.31 | 3.98 | 8.53 |
| 1000 | 26.02 | 8.56 | 55.98 | 144 | 7.77 | 42.96 |
| 2000 | 52.26 | 17.14 | 112.98 | 289 | 15.56 | 85.93 |
| 5000 | 131.26 | 34.8 | 279.9 | 720 | 38.85 | 214.8 |
| 10000 | 262.52 | 69.6 | 559.9 | 1440 | 77.7 | 429.6 |

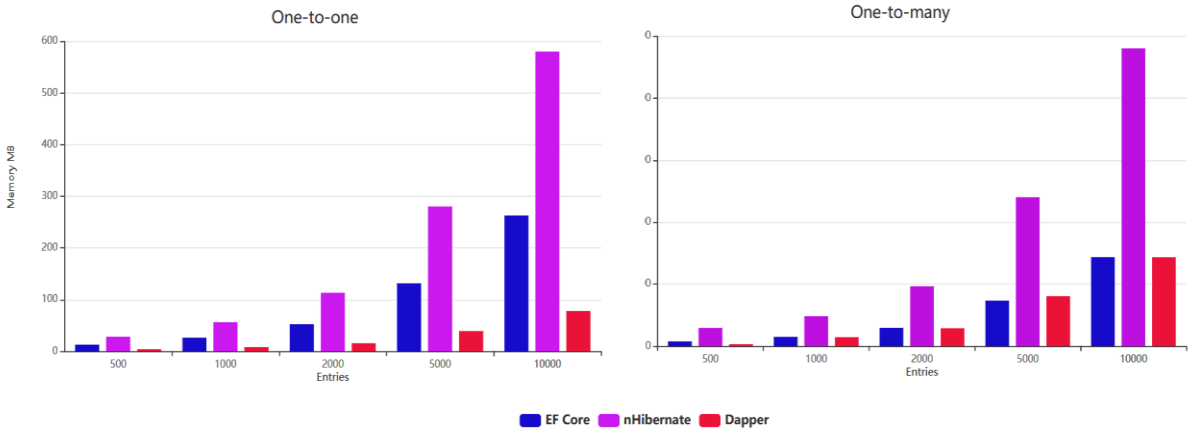


Fig. 10. Memory used – Delete.

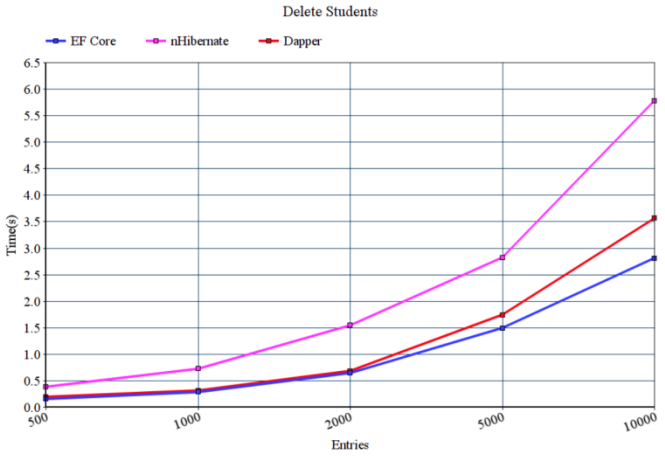


Fig. 11. Delete Students (One-to-One).

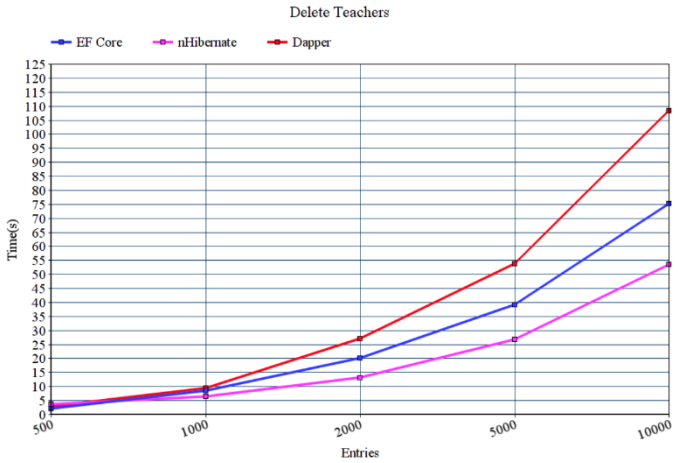


Fig. 12. Delete Teachers with Corresponding Courses (One-to-Many).

The Select (Get) operation has some particularities depending each targeted technology, that implies few differences as follows:

- EF Core: for data retrieval, the Include and ThenInclude functions were used, which will return multiple levels of related data also using the method Take() in order to select a given top entities from database. Eager loading is used (by default) when returning data, as lazy loading needs to be specifically turned on as from EF Core 2.1;
- Nhibernate: QueryOver<EntityType> generic method is used to retrieve all data regarding the given entity as parameter and also the Take() method to select a top from database. Lazy loading is used as it is the default behaviour for nHibernate.
- Dapper: the same SQL codes which EF Core is generating when running a Get Students was replicated used by injecting the SQL with Query<*EntityType*> method, to test the speed of Dapper when running and retrieving the result entities. Also, eager loading is used here, as Dapper being a direct-SQL library.

The obtained results are presented in Fig. 13 and Table VII. Dapper has the best timing results when returning entities on one-to-one relationship because of the missing entity tracking logic which is present behind the scenes for EF Core and nHibernate.

The second scenario used for *Get* method implies the retrieval of teachers alongside with their corresponding courses. A simple pseudocode to describe this can be seen below:

*get Teachers(including Person and Address);*
*get Courses;*

The obtained results after the execution time benchmark analysis are presented in Fig. 14 and Table VII. Dapper has registered very inefficient results, having the worst results from all three technologies when it comes to returning a higher number of entities. One possible cause for this could be the use of the generated EF Core SQL into Dapper database call, indeed, and not using a Dapper built-in method to retrieve one-to-many entities. The reason for that is that, after testing Dapper's abilities to manage the same SQL code EF Core is generating behind the scenes, it was considered preferable not to use predefined Dapper get method.

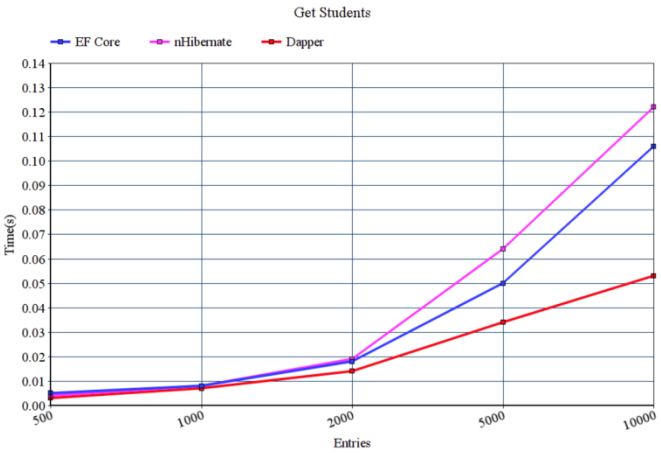


Fig. 13. Get Students (One-to-One).

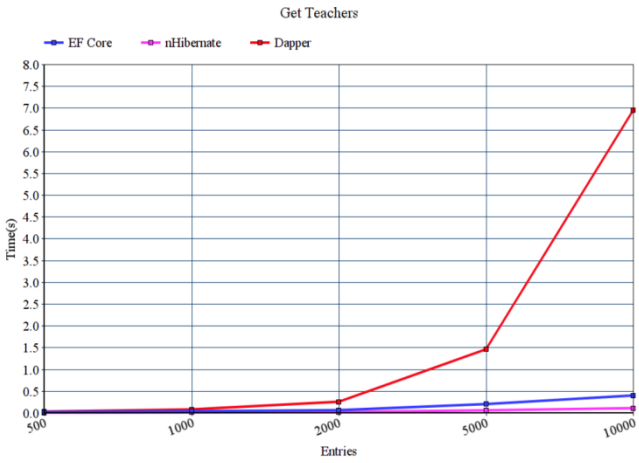


Fig. 14. Get Teachers with Corresponding Courses (One-to-Many).

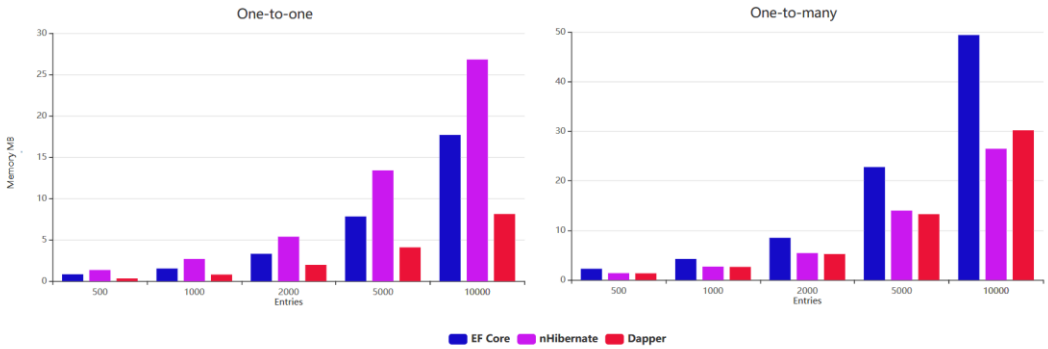


Fig. 15. Get Memory usage.

TABLE. VII. GET METHOD EXECUTION TIMES

| Entries /Time(s) | Get methods Time(s) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | *Students* | *Teachers* | *Students* | *Teachers* | *Students* | *Teachers* |
| 500 | 0.005 | 0.024 | 0.004 | 0.004 | 0.003 | 0.024 |
| 1000 | 0.008 | 0.037 | 0.008 | 0.011 | 0.007 | 0.042 |
| 2000 | 0.018 | 0.077 | 0.019 | 0.018 | 0.014 | 0.080 |
| 5000 | 0.050 | 0.201 | 0.064 | 0.058 | 0.034 | 0.185 |
| 10000 | 0.106 | 0.444 | 0.122 | 0.110 | 0.053 | 0.343 |

NHibernate is using by default lazy loading and thus having low timing results when returning one-to-many results, but this will be compensated by the calls that will have to be realised when accessing entities that have not yet been loaded. When returning one-to-one results, it is close to EF Core, both being overcome by Dapper, thing that could be easily explained by the simple fact that Dapper is in fact a micro-ORM and thus not having all the backside logic a fully-featured ORM has behind the scene. However, EF Core using eager loading has very close results to nHibernate in comparison to Dapper, but still 3 times slower than nHibernate.

By analysing the memory results in the one-to-one approach, it can be seen that nHibernate using proxy objects when returning entries using lazy loading (Fig. 15 and Table VIII).

TABLE. VIII. GET METHOD USED MEMORY

| Entries /MB | Get methods used memory (MB) | | | | | |
|---|---|---|---|---|---|---|
| | *Tab Entity Framework Core* | | *nHibernate* | | *Dapper* | |
| | *Students* | *Teachers* | *Students* | *Teachers* | *Students* | *Teachers* |
| 500 | 0.83 | 2.61 | 1.34 | 1.35 | 0.31 | 2.67 |
| 1000 | 1.51 | 4.98 | 2.68 | 2.69 | 0.78 | 5.32 |
| 2000 | 3.29 | 9.95 | 5.36 | 5.39 | 1.96 | 10.64 |
| 5000 | 7.81 | 26.43 | 13.4 | 13.97 | 4.07 | 26.8 |
| 10000 | 17.69 | 53.07 | 26.8 | 26.44 | 8.12 | 53.59 |

But, when selecting more complex entities (one-to many approach), EF Core and Dapper have higher memory usage for returning one-to-many entities, in comparison with the case of returning one-to-one entities, because of the impact of the logic used there to map all returned courses to each teacher while nHibernate is using proxy objects.

## V. RESULTS AND DISCUSSION

For Insert operations, nHibernate has the best results, followed by EF Core and then by Dapper. For Update operations EF Core has the best results when considering one-to-one relationships; but when considering also one-to-many relationships, Dapper has the best results.

A similar situation was observed for the Delete operations, where, when considering one-to-one relationships, EF Core has best execution time results followed relatively closely by Dapper and then by nHibernate; but, when considering also one-to-many relationships, nHibernate has on average the best results in terms of execution time for high number of entries (over 1000) followed closely by EF Core and then by Dapper. This performance is obtained to the detriment of memory usage.

When realizing Get calls, nHibernate and EF Core execution times are very close; for one-to-one relationships, Dapper is having better results as number of entries increases, followed by EF Core and then by nHibernate; but, when also one-to-many relationships are involved, nHibernate has, on average, the best results in terms of execution time followed by Dapper, EF Core having the worst performance as the number of entries increases. Significant differences could be observed for Get operations from an overall perspective.

If using nHibernate's default settings, it will make use of lazy loading and thus showing great timing results and memory when one-to-many relationships are involved; but, for only one-to-one relationships its performance is the lowest one. EF Core with lazy loading being disabled exhibits good results when returning entities in one-to-one relationship but memory and time consuming when returning complex double one-to-many entities.

In a real-life project, all technologies could be used just with the statements on which are showing the best results, such as: nHibernate for Insert, nHibernate (using default lazy loading) or EF Core for Get, EF Core or Dapper for Update and Delete.

## VI. Conclusions

After analyzing the results, it can be concluded that none of the three technologies targeted for benchmarking analysis is having the very best results both from the time point of view and memory usage. The decision to use one of another is dependent of the most used type of operation.

This paper's work could be further developed by testing all three technologies over an Azure stored database. This could bring another live scenario to test, when there is a need to measure time responses of queries that target a remote database. Nevertheless, this approach will definitely have its downsides, for example, the internet connection stability, bandwidth, database type chosen from Azure, location of the Azure Storage in accordance with the server location on which the application is running, all will have a major role in realizing measurements.

## References

[1] K. Roebuck, Object-Relational Mapping: High-impact Strategies - What You Need to Know. Samford, Australia: Lightning Source Publisher, 633p., 2011.

[2] P. Van Zyl, D.G. Kourie, A. Boake, "Comparing the performance of object databases and ORM tools", Proceedings of the 2006 Annual research conference of the SAICSIT on IT research in developing countries, SAICSIT 2006, Somerset West, South Africa, DOI: 10.1145/1216262.1216263, pp. 1–11, 2006.

[3] A. Haug, J. Arlbjorn, F. Zachariassen, J. Schlichter, "Master data quality barriers: an empirical investigation". Industrial Management & Data Systems, Emerald Group Publishing Limited, DOI: 10.1108/02635571311303550, 113 vol. 2, pp. 234-249, 2013.

[4] J.P. Smith, Entity Framework Core in Action. New York: Manning Publications Company, 520p., 2018.

[5] G. Liljas, A. Zaytsev, J. Dentler J., NHibernate 4.x Cookbook - Second Edition. UK: Packt Publishing, 448p., 2017.

[6] Khan O.M.A., Benchmarking .NET Core 2.0 applications. In: C# 7 and .NET Core 2.0 High Performance: Build highly performant, multi-threaded, and concurrent applications using C# 7 and .NET Core 2.0. UK: Packt Publishing Ltd., 281p., 2018.

[7] Basheleishvili I., A. Bardavelidze, and K. Bardavelidze. "Study And Analysis Of The .Net Platform-Based Technologies For Working with the Databases, " Proceedings of the 33rd International Conference on Information Technologies (InfoTech-2019), Bulgaria, pp.1-8, 2019.

[8] V.J. Mehta, ORM Patterns and Domain-driven Design. In: Pro LINQ Object Relational Mapping with C#. New York: Apress Publisher, pp.17-43, 2008.

[9] S. Cvetković, D. Janković, "A Comparative Study of the Features and Performance of ORM Tools in a .NET Environment", in: A. Dearle, R.V.Zicari, Objects and Databases, ICOODB 2010. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, DOI: 10.1007/978-3-642-16092-9_14, vol. 6348, pp.147-158, 2010.

[10] A. Joshi, S. Kukreti, "Object Relational Mapping in Comparison to Traditional Data Access Techniques", International Journal of Scientific & Engineering Research Vol. 5, Issue 6, pp.540-543, 2014.

[11] A. Gruca, P. Podsiadło, "Performance Analysis of .NET Based Object–Relational Mapping Frameworks", 10th International Conference, BDAS Ustron, Poland, Springer International Publishing, DOI: 10.1007/978-3-319-06932-6, pp.40-49, 2014.

[12] G. Procaccianti, P. Lago, W. Diesveld, "Energy Efficiency of ORM Approaches: An Empirical Evaluation", Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement ESEM, Ciudad Real, Spain, ACM, DOI: 10.1145/2961111.2962586, pp.185–198, 2016.

[13] Suhas Chatekar, Learning NHibernate 4: Explore the full potential of NHibernate to build robust data access code. 402p. UK: Packt Publishing, 2015.

[14] T. H. Chen, W. Shang, Z. M. Jiang, A.E. Hassan, M. Nasser, P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping.", Proceedings of the 36th International Conference on Software Engineering. ACM, pp.1001–1012, DOI:10.1145/2568225.25682592014, 2014.

[15] W. Wiphusitphunpol, T. Lertrusdachakul. "Fetch performance comparison of object relational mapper in. NET platform." In 2017 IEEE 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology ECTI-CON, DOI: 10.1109/ECTICon.2017.8096264, pp. 423-426, 2017.