DATABASE FINAL ASSIGNMENT

Project STIX Database

Group Members

Christopher Sulistiyo (4850025)
Line Amini Kaveh (4929284)
Robert Rachita (4859367)
Atanas Hristov (4785002)
Stefan Untura (4839161)
Victor Tromp (4922972)

Version 1.0

Result Matrix

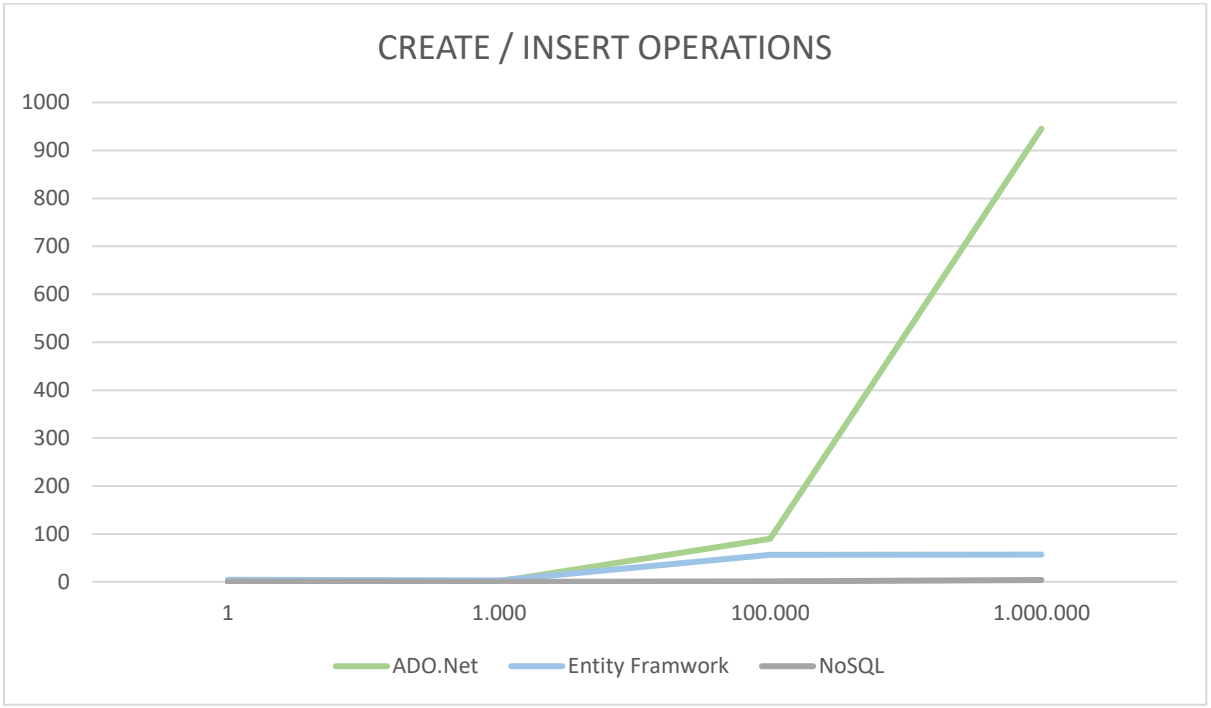
The table below, contains the results of the executed CRUD operations for each of the approaches. Each method is tested with 1, 1.000, 100.000 and 1.000.000 lines for each of the CRUD operations respectively.

Times are noted down in seconds, with a max. of 4 decimal places after the comma

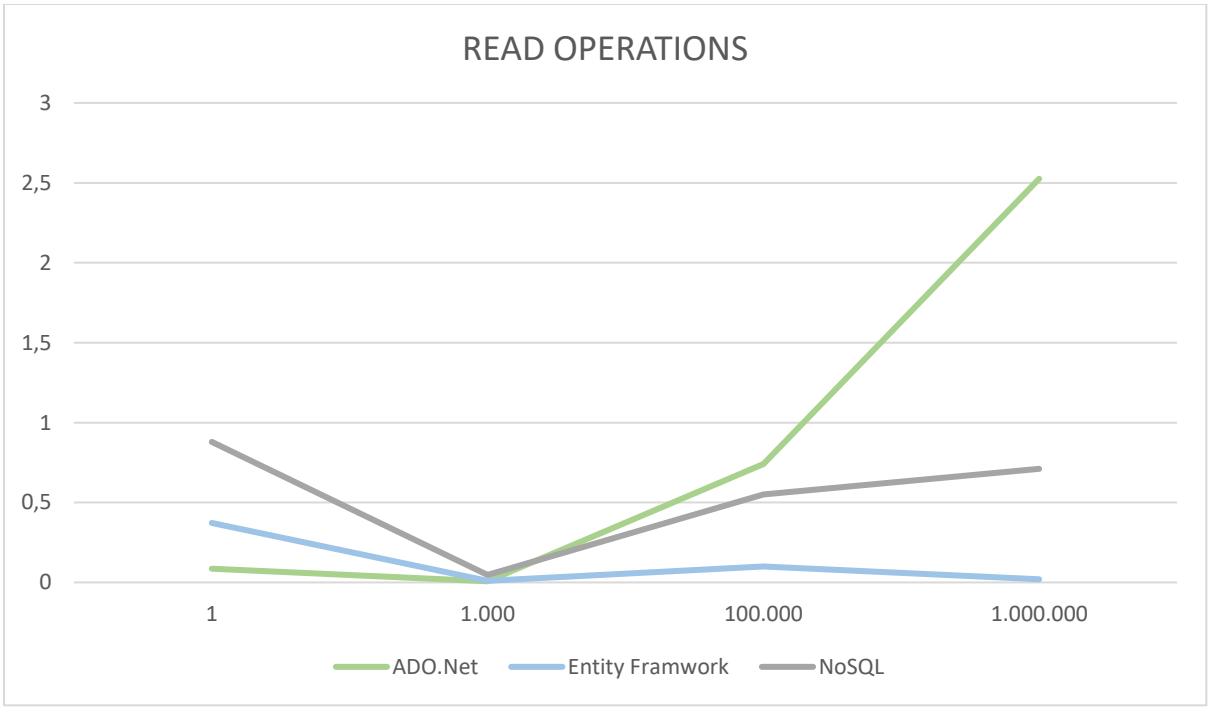
	ADO.Net	Entity Framework	NoSQL
Create / Insert			
1 line	0,4242	4,5482	0,0640
1000 lines	1,012	2,9797	0,0089
100.000 lines	89,5362	56,4023	1,0389
1.000.000 lines	945,2524	56,8158	3,8229
Read			
1 line	0,0859	0,3720	0,8793
1000 lines	0,007	0,0100	0,0469
100.000 lines	0,7408	0,0999	0,5498
1.000.000 lines	2,5255	0,0200	0,7103
Update			
1 line	0,0046	0.0200	0,1393
1000 lines	0,009	0.1700	0,1652
100.000 lines	0,23	0,0019	0,0036
1.000.000 lines	2,4847	0,0020	0,0023
Delete			
1 line	0,002	0,0699	0,0628
1000 lines	0,005	0,0299	0,0823
100.000 lines	0,233	0,0100	0,0995
1.000.000 lines	2,7824	0,0200	0,0086

Graphs

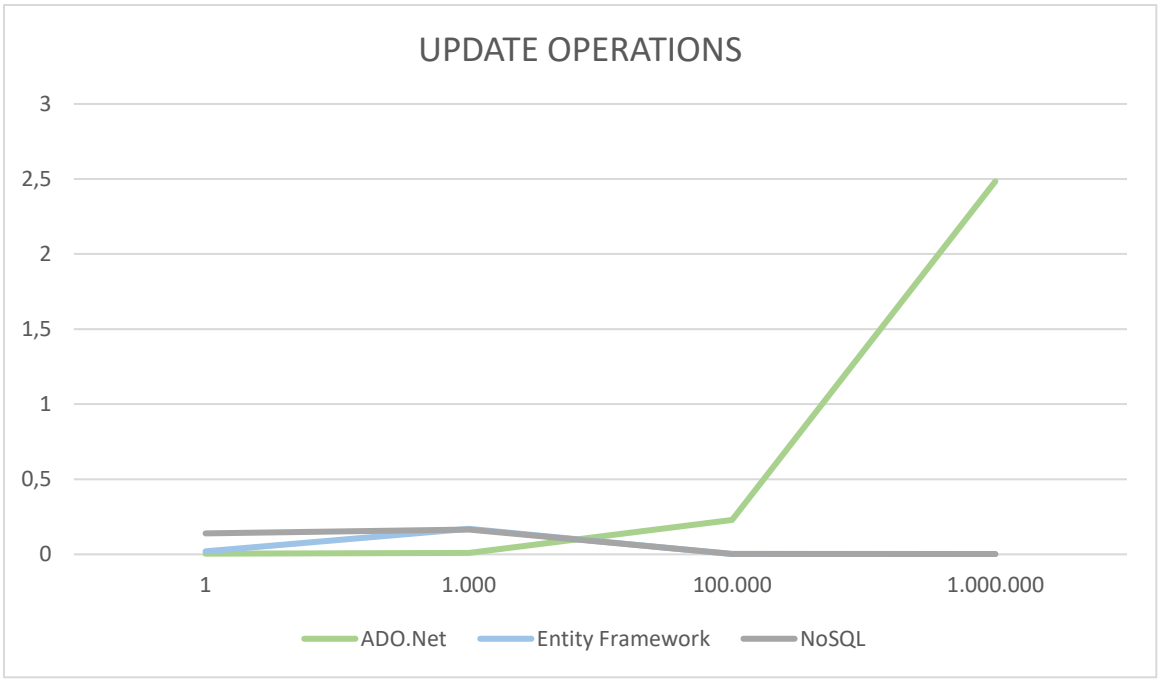
The following graphs depict the execution speed of the different CRUD operations for 1, 1000, 100.000 and 1.000.000 lines in seconds. Each graph has three lines representing the three different approaches, with each operation being given their own diagram.



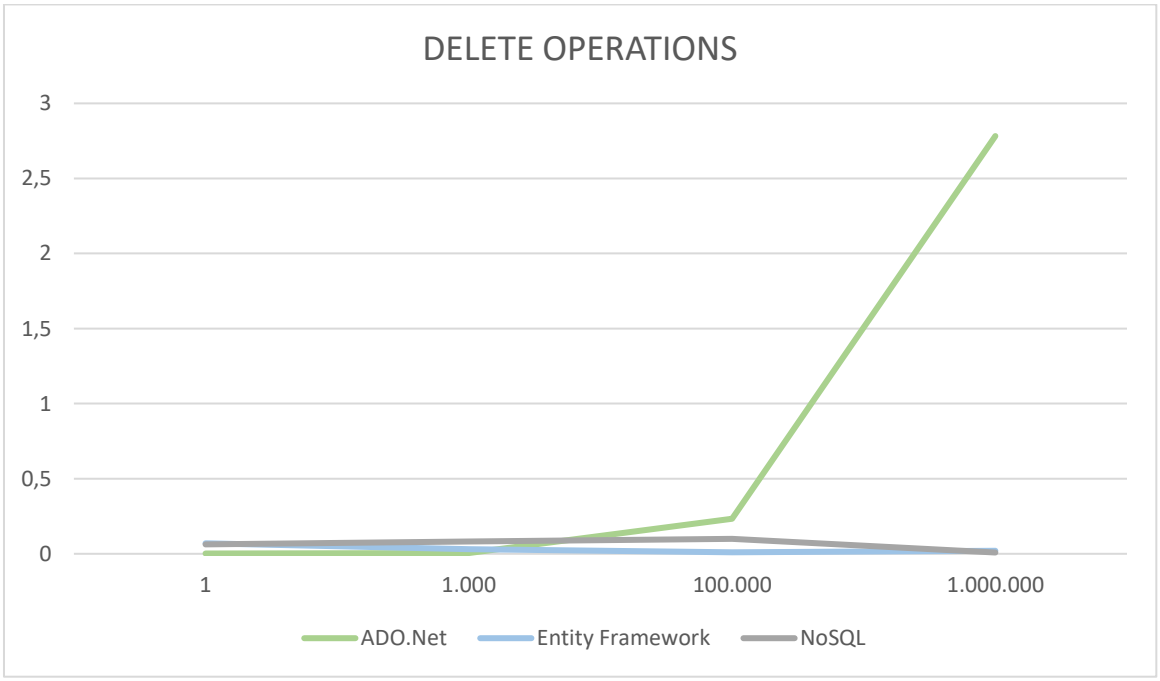
.



.



.



Approach Comparisons and Remarks

Each of the test approaches were making use of the User table in Netflix database to run the CRUD operations. A SQL Server Express was installed on the device that the test was run on. In case of the NoSQL approach, a local MongoDB database was created, in which the User table was also being implemented with all of its foreign keys.

Insertion (Create)

ADO.NET

For this approach, all the insertions for 4 different amounts of rows were executed with a single method, which is manually inserting them with the Transact-SQL statement within their respective For loops. This explains the increased amount of time needed to execute the test for this specific approach.

This technique makes use of SQL Client to connect to the database based on a string path, and SqlCommand to execute whatever command string the tester wants to make us of. After telling the command on which client and what command it needs to execute, a method called ExecuteNonQuery() was called to execute all the non-query commands.

As expected, this approach runs significantly slower at ~945 seconds, than the other two approaches.

Entity Framework

This approach makes use of creating classes in which the properties within the classes represent columns of the table it is targeting. A class named DatabaseContext was created, in which it is connecting to the local SQL Server database, and it functions as a bridge between the database to the code part. In the DatabaseContext class, a list of all entities is contained, and the entities can be stored in both memory and in the database from there.

For the insertion of 100,000 and 1,000,000 rows the test using for loops with individual inserts was taking too much time, so a different approach was taken into consideration. Instead of manually inserting the data in the For loop for 100,000 times, a list of the data collection is created, and the data is stored for 100,000 times in the For loop, and then AddRange() method is called to insert the bulk of the data along with SaveChanges() method to save it to the database.

NoSQL

NoSQL appears to be the fastest one of all three. This approach starts by installing a MongoDB .NET driver in C#, and from there the application can make use of its stored-in MongoDB drivers. A class called UserMongo was created in Model folder to represent the collection from the MongoDB database, along with its properties to represent the object notation (column). The application also makes use of 3 C# MongoDB drivers, which are MongoClient, IMongoDatabase, and IMongoCollection. MongoClient and IMongoDatabase are both an interface for connecting to the database host and getting which database the tester wants to use. The UserMongo can be stored in IMongoCollection, which functions as a way to connect to which collection of the database the tester wants, as well as storing its type of document. The possible document types are UserMongo and BsonDocuments. For inserting just one row of data in the collection, a method called InsertOne() is called from the IMongoCollection to insert that one specific dataset which can be either BsonDocument or UserMongo class. For the rest, a method called InsertMany() can be called which can insert a list of UserMongo

Select (Read)

ADO.NET

The reading of rows from the database in this approach was done in a very similar fashion as the insertion of rows. A Transact-SQL query command was created, but a method called ExecuteReader() was called in place of ExecuteNonQuery().

As with the Insert operations, ADO.Net is the worst performing out of the three approaches when it comes to increased amounts of operations executed at once, which may also be due to the approach that was explained in the Create/Insert section.

Entity Framework

For the read operations, Entity Framework came out to be best, beating ADO.NET and came out close to NoSQL because it makes use of the LINQ commands, which also because of its built-in method, only takes 1 line of code for all of the querying of different number of rows.

NoSQL
NoSQL is unsurprisingly better than ADO.NET but got beaten by Entity Framework which was surprising for the tester

Update

ADO.NET
Just like the previous create operation, a SQL Update statement is created and then it will be set to the SqlCommand to be executed. Because in Transact-SQL, there is already a TOP query clause to select how many rows the tester wants to update, no For loop is used. Because of this, the speed of updating rows in ADO.NET is faster than the previous inserting and reading.

Entity Framework

In Entity Framework, once again a LINQ command is used to query the rows in which the results are stored in a variable, and a foreach loop was run to update each of the rows present in the storing variable. It turns to be quite fast, beating both ADO.NET and NoSQL in 1,000,000 rows.

NoSQL

For updating in NoSQL, the program just finds, and limits data based on how many rows need to be tested in the collection in the MongoDB, then a foreach loop is called to iterate over every single one of these datasets queries, and it is being updated from the foreach loop. It once again came close to the Entity Framework in terms of speed.

Delete

ADO.NET

For deletion in ADO.NET, a SQL command is used and then executed from SqlCommand, from where it is saved to the database. This time, it went out pretty fast and beats the Entity Framework and NoSQL except for the 1,000,000 rows execution. It didn't need to use a for loop to manually delete all of the desired rows from the database.

Entity Framework

In Entity Framework, a similar method called RemoveRange() is called from the Users collection to delete the amount of data desired from the database. The method SaveChanges() was also called after this to ensure the deletion of data from both device memory and the database. It beats both ADO.NET and NoSQL.

NoSQL

NoSQL deletion beats ADO.NET and Entity Framework for deleting 1,000,000 rows of data but got beaten by Entity Framework in 1,000 and 100,000 rows, which indicates that NoSQL is well suited to process large amounts of data at a high speed.

Reliability

For testing the speed for each one of the different approaches on the individual rows, a stopwatch was created inside of the program itself. This will add more accuracy and reliability to the test data as opposed to the tester manually using a stopwatch from the real-world. Basically, before the querying starts, a DateTime variable called before was created and it stores a DateTime value type, and after the queries are being run another DateTime variable called after was created. The subtraction of both variables will create a TimeSpan in which the tester can extract a total millisecond from the time that has passed. Additionally, C# also have a built-in Stopwatch object in which also takes a TimeSpan as a result between the time between the stopwatch was started and stopped.

Device and Programs

The following data shows on which device specifications to execute the test and what kind of software along with their versions were used

Device Specs

Below are the specifications of the device on which the test was run:

Processor : Intel® Core™ i5-10300H CPU @ 2.50GHz

Installed RAM : 8.00GB DDR4

System Type : 64-bit operating system, x64-based processor

Graphic Cards : NVIDIA® GeForce® GTX 1650 4GB

Storage Device : 512GB SSD

Windows specifications

Edition : Windows 11 Home

Version : 21H2

OS Build : 22000.739

Software

In order to make the program and run the test, an Integrated Development Environment was used to run the C# Console Application. The environment in which the test was run was in the following:

Microsoft Visual Studio Community 2022 (64-bit)

Version 17.24

NuGet Packages

In addition for using an IDE, a couple of C# frameworks were also used to develop the test application, those frameworks were:

- Microsoft SqlClient version 4.8.3
- Microsoft Odbc version 6.0.0
- Microsoft Entity Framework version 6.4.4
- MongoDB Driver version 2.16.1