International Journal of Computer Science and Applications,
©Technomathematics Research Foundation
Vol. 16, No. 1, pp. 38 – 54, 2019

MONGODB VS. SQL SERVER IN MEDICAL APPLICATIONS

LIANA STANESCU

Computer Science and Information Technology, University of Craiova, Bvd. Decebal 106, Craiova, 200440, Romania
stanescu@software.ucv.ro

MARIUS BREZOVAN

Computer Science and Information Technology, University of Craiova, Bvd. Decebal 106, Craiova, 200440, Romania
mbrezovan@software.ucv.ro

COSMIN STOICA SPAHIU

Computer Science and Information Technology, University of Craiova, Bvd. Decebal 106, Craiova, 200440, Romania
stoica.cosmin@software.ucv.ro

The paper presents the technical details of the implementation for a medical application that can be used to handle a large volume of information of medical records. A prototype of a complete application was designed to allow using two types of database servers handling: A SQL server database and a non-relational database server (MongoDB). The aim of the experiments performed to measure the scalability and performances of the two types of servers showed that the MongoDB server behaves better on large-scale databases and this could be a better choice for the final version of this application.

Keywords: Relational databases; document databases; MongoDb.

1. Introduction

The paper presents the technical details of the implementation for a medical application that can be used to handle a large volume of information of medical records.

In order to reach the objectives of the business, large volume of information needs to be analyzed and processed. The application which was implemented is designed to process hospitalization medical records from hospitals in Romania. The created project is a prototype of a complete application. The medical records have a number of information to be filled in. For this prototype, about 25-30% of the data found in the forms provided by the Emergency County Hospital, Buzau, has been used.

The application can handle two types of database servers: a classical SQL server and a non-relational server, namely Mongo DB.

The benefit of the application is to give the doctors the possibility to keep track of the treatments and diagnosis provided for each patient in part.

The application contains only basic functionality for the following medical type of records:

- Cardiology;
- ORL;
- Urology.

Processing data from medical records is one of the possible operations within this application. In addition to this operation, the following roles have been designed:

- Administrators;
- Doctors;
- Patients;
- Drugs;
- Diagnoses;
- Images related to medical records.

The application was designed to be compatible with on all devices running Windows 10, the layout being used for desktop and tablet devices. With UWP technology, this is possible without writing specific lines of code for each platform. Due to the use of the .NET Core package, the application can run on other Microsoft platforms, but is not recommended due to the implemented features that require a certain type of device for quick operations.

The paper is structured as following: Section 2 present general consideration regarding similar solutions, Section 3 presents the architecture of the application, Section 4 presents experiments and results and the Section 5 presents the conclusions of the paper.

2. Application Overview

In order to reach the objectives of the business, large volume of information needs to be analyzed and processed daily. The hospitals need to handle nowadays huge volume of information, and the size of the database is increasing more and more.

HealthFirst is an application designed to process hospitalization medical records from hospitals in Romania. The created project is a prototype of a complete application. The medical records have a number of information to be filled in. For this prototype, about 25-30% of the data found in the forms provided by the *Emergency County Hospital, Buzau*, has been used.

3. Architecture

The application was designed to be compatible with all devices running Windows 10, the layout being used for desktop and tablet devices. The technology used is UWP, which ensure the compatibility, without having different implementation for each platform. The application is also compliant with other Microsoft platforms due to the use of the .NET Core package, but this was not tested yet.

The application is based on web services to process part of the functionalities. While it can be owned by any user, web services are available only by the person holding the web server. The first step for the application to be functional is to be configured to connect to the web services. Web API 2 and MVC technologies are part of existing deployments to create a web service. In the implementation of this system, has been decided to use Web API 2 services because of the simplicity and speed of their understanding.

Due to the large number of medical records data needed to be handled, was preferred a non-relational database. This supports embedded data records that lead to their fast processing. For comparing reasons, support for a relational Database server was also implemented (Celko, 2014) (DuBois, 2014).

MongoDB was the non-relational database server which was selected to be used. Technologies have evolved, and writing code to access or process data from databases is no longer a feature in the field of quickly application development. Also, writing queries into the code leads to lost time in understanding and debugging them. MongoDB has a MongoDB driver library, through which all these operations can be done in a language close to the developer. The following chapters will describe the MongoDB collections and theSQL Server database with equivalent tables resulted by applying the normalization principles on the MongoDB collections(Krisciunas, 2014) (Kvalheim, 2015).

The combination of application and web services improves system security. However, performance issues related to the security level still exists. The main issue t needed to be handled was the URL copying. Anyone who would have copied the URL would have access to all of the functionality that he had and could have made requests at any time to the web services.

To enhance system security, the OAUTH2 security service has been integrated, which, based on login information provided by the user, generates an authentication token. The expiration time was set at 7 minutes, after which a new token would be needed to continue using the application. It is recommended that a new token to be generated faster than its expiration time. For this, a separate service has been implemented in the application to get a new token at every 5 minutes.

Security is not only based on access to the functionalities offered by web services using an authentication token, but also through roles and permissions. Each user connected to the system has a role and through it receives a list of permissions for activating the specific functionality in the application. The web services methods have a tag describing the role with access rights. Some methods have multiple tags, since functionalities are common for many types of users. All users are saved in the database with a specific role excepting one of them. The Super Administrator is an integrated code user. The system will always have a user with this type, but the number of accessible operations is limited. This user's password has been stored in a web-based configuration file. The system was designed to have the following types of users with the following management capabilities available:

- SuperAdmin
    - Administrators
- Administrator

    - Hospitals
    - Drugs
    - Diagnoses
    - Doctors
    - Medical records
- Doctor
    - Patients
    - Medical records
- Patient
    - Medical records

As it can be seen in the hierarchical structure, Admin, Doctor and Patient users have access to information about medical records, but the permissions are different. To avoid creating a complex system of permissions, some of them have been returned by the service to unlock the above functionality, and another part has been integrated directly into the application depending on the role of the connected user. In this case, doctors can create and view medical records only for the type of specialization they are enrolled in. They also have management operations for images on medical records. Admin users can view medical records from all specializations, but do not have editing rights on them, and patients can view only the medical records associated with them.

One of the purposes of developing this system was to reuse as much as possible the written code. For the previous case, the same views are common to all user types for the management of the medical records.

In order to compare the processing time, a relational database was created in SQL Server equivalent to MongoDB collections. It was also implemented 2 data generators for the related databases. The insertion has been done using advanced technologies. About MongoDB, it was stated that its equivalent is the MongoDB driver. In turn, SQL Server owns a similar library for access to databases called EntityFramework. For each management system, 4 databases were created with a number of medical records equal to 1000, 10000, 100000 and 1000000. For each, the insertion times and the running times of certain queries were measured using the management system. The results of the experiments are presented in Chapter 4.

3.1. *HealtFirstApp*

The architecture of the application is organized on 3 levels:

- Model-View;
- ViewModel;
- Repository;

3.1.1. MVVM

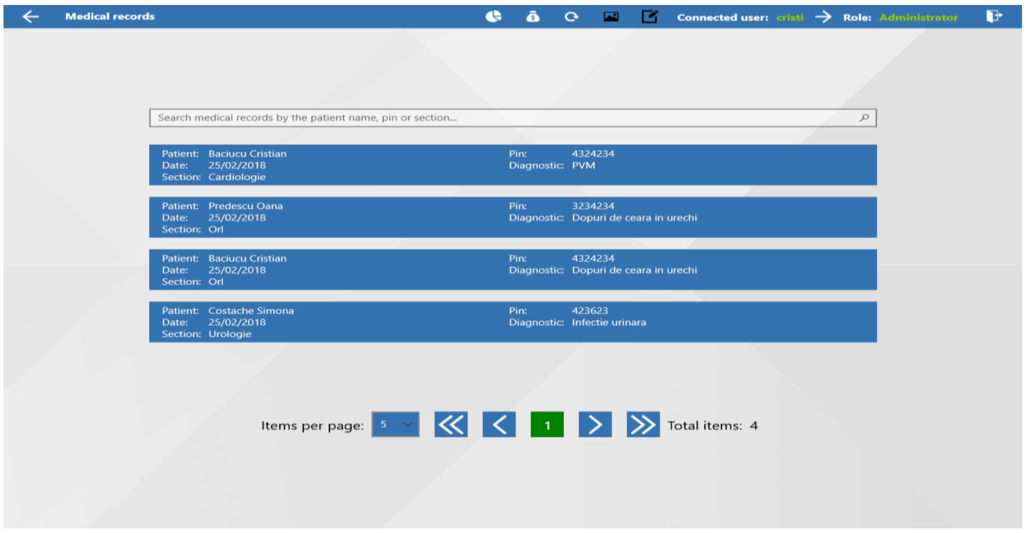
The first two form a group and follow the design of the MVVMpattern, but their implementation logic is different. The Model-View layer deals with drawing the user interface and the ViewModel layer with the operations behind them. Each Model- View is matched to a ViewModel. In this way, it ensures that the views do not have the logic of processing data available at the next level. However, the communication between them is required to transmit data to the controls or events launched. For the data transmission operation, UWP has an implemented technique called Bind. This specifies the name of the property from which the information will be read. Example Text = {Binding PropertyName}. This statement describes that the property Text on the controller will be assigned the value owned by PropertyName in ModelView. In addition to this method, a converter can also be added. The user can format their display mode or convert a data type of a ViewModel property into a format supported by the property on the controller. For example, the user wants to show or hide certain controls. In ViewModel it will use a Bool property, but the Visibility property on the controls is of another type. With a converter, the transition from the Bool type to the type accepted by the Visibility property can be made.

As stated, another way of communication between entities is given by the transmission of the events launched. UWP does not have such functionality. Just for buttons there is an implementation using the Command pattern. One method is to write many lines of code, which is not recommended. To solve this problem, the Caliburn.Micro library was integrated. With this library, it is necessary to write a single line of code that specifies the name of the event to be transmitted and the name of the ViewModel method that will be called upon to launch it.

Also at this level various services are created for:

- Authentication Token Synchronization;
- Translation;
- Storing permanent data for future use;
- Notification system with the status of execution of various operations.

The MongoDB driver imposes limits on the documents that work with, which is equal with 16 MB. Transmitting all data brings great performance problems and that's why the MongoDBdriver's developers choose this limit. The main technique applied by developers is the paging of the results. UWP controls do not have this functionality, so it would have been impossible to access all the data available in the database. The first step was to improve the functionality of these controls. First of all, UWP does not have a way to display the data in a table view. We chose ListView control for their presentation. There are other libraries that come to solve these problems, but they have errors, access to various members is difficult or there is not offered for free. We have created a pagination control and integrated with ListView control functionality.



Search medical records by the patient name, pin or section...

Patient: Baciucu Cristian
Date: 25/02/2018
Section: Cardiologie
Pin: 4324234
Diagnostic: PVM

Patient: Predescu Oana
Date: 25/02/2018
Section: Orl
Pin: 3234234
Diagnostic: Dopuri de ceara in urechi

Patient: Baciucu Cristian
Date: 25/02/2018
Section: Orl
Pin: 4324234
Diagnostic: Dopuri de ceara in urechi

Patient: Costache Simona
Date: 25/02/2018
Section: Urologie
Pin: 423623
Diagnostic: Infectie urinara

Items per page: 5 1 Total items: 4

Fig. 1. Medical records general presentation

The user can select the number of items listed on the screen from the options: 5, 10 and 15. It can also be selected:

- The first page available;
- The last page;
- The next page;
- The previous page;
- One of the pages next to the current page.

Initially, the control was created only for this purpose, but during the development of the application it was integrated into the functionality of other controls such as AutoSuggestBox. Also, the application owns an image viewer for medical records, and navigating among them is also based on the functionality of this pagination control, but the number of items returned was reduced to 1 because of the large images size.

Filtering data is another way to get desired information. The functionality of the AutoSuggestBox control has been changed and integrated with pagination control. Placeholder specifies the members after which the filtering will be done. Each time Enter key is pressed, the control will reset the current page to the first page available in the system, and the user will then be able to navigate to the other pages with the applied filter.

One way of displaying the data is shown in Figure 1. This is a simplified data listing model. From toolbar, you can navigate to the detailed information of such an item, provided it is selected first.

Each such control is based on an MVVM system and is inherited from other ModelViews so that the reuse of the code is valid. Including the base class specific to the Caliburn.Micro library, the inheritance tree reaches up to 6 levels. The organization of this tree is shown in Figure 2.

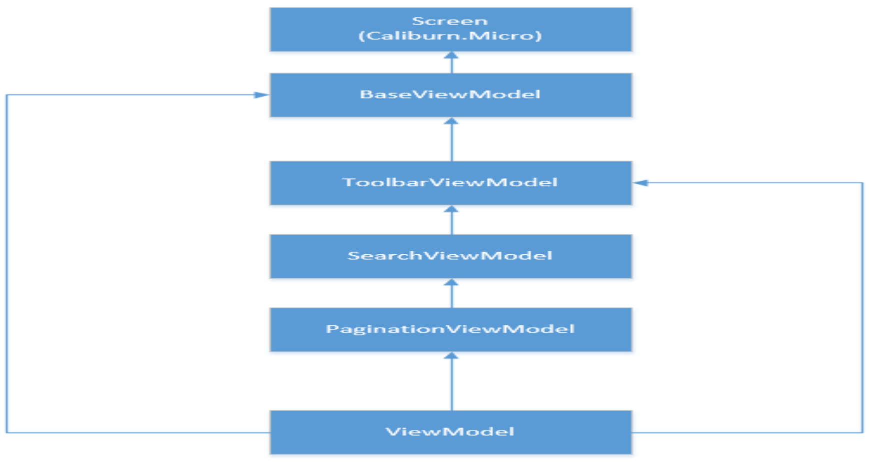


Fig. 2. Application's inheritance tree

3.1.2. Repository

The last logical level of the application contains all classes of communication with web services. Functionality is simple and has operations to organize different data to be transmitted on various contents: header, body, and URL.

3.2. ***MasterDataService***

MaterDataService is the starting point for receiving customer requests and delivering results. Here, the requests are authenticated based on authentication token. Practically, MasterDataService owns the entire security system and the application responding to its permissions.

3.2.1. BusinessLogic

Within this system, this level may be missing. Its implementation mode makes data from the master data service to be forwarded to the last level and vice versa. However, in an application in production, depending on the type of customers, it may be required to save the data in different storage environments. This level has a set of interfaces that must be implemented by the last level classes. If the last level would be separate depending on the type of storage, then the BusinessLogic level would expect that regardless of the type of the last level, receive the same set of data. However, you need to specify which storage environment can be used at a time. Obviously this requires that the system to be recreated for each storage environment but with minimal impact on its choice.

Both solutions follow SOLID principles, and one of those applied is represented by the dependency injection. Using this principle makes classes indicate the dependencies that they use on their constructor. The new operator is no longer required because a dedicated dependency injection library, Autofac, has been used. The library allows only the recording of classes or adjacent to the interfaces, separately or by specifying anamespace, for each occurrence on the constructor to use a new instance or a common one for all (Singleton). Assuming that the data acquisition mode is implemented for each type, selecting the desired environment is done by changing the namespace of the classes that will be registered. This is the real reason for having a BusinessLogic level.

3.2.2. Repository

The last level available on the service side has directly access to the database. All access or insertion operations are processed at this level. The pagination control also has a logic at this level by setting the next page to be returned to the user. The user may have selected the Next Page option, but the current page can be also the last one in the system. In this case, the current page will be returned once there is no new data page. Getting data is done in two steps. The first step is to get the total number of items in the list with a particular filter. By doing so, the number for the new page will be calculated. The second step is to access the data on the selected page by applying the same search filters.

MongoDB driver has the functionality to insert or update data using the same instruction. Because one of the purposes was to reuse the existing code, a single common method for insertion or update operations was created. Functionality was allowed by adding theIsUpsert = true parameter to the ReplaceOne method. Based on this parameter, the method will search for an item with the object to be processed in the specified collection. If an object in the collection with the same key was found, it will be overwritten; otherwise a new item will be generated. All newly created items in the application do not have a generated key. The service takes into account this case and generates such a key.

As has been said before, the MongoDB driver is limited to a 16 MB document size. Being a medical application, the image clarity is important. Their clarity also requires storing files with dimensions above the specified limit. For this issue, MongoDB's developers have created a separate library, GridFS, to process large-sized files.

For the medical records images, some information was wanted to be stored in

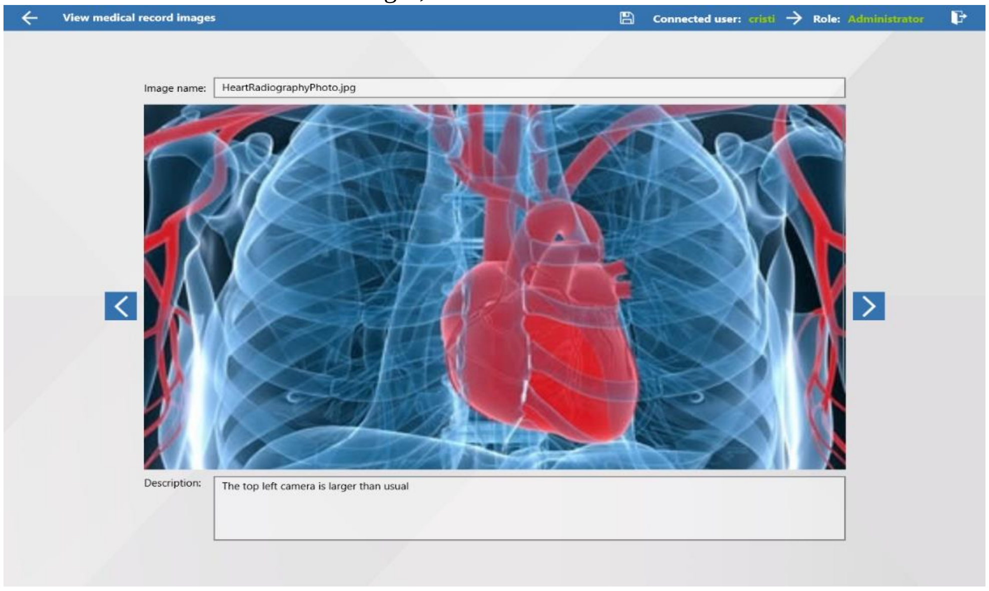


Image name: HeartRadiographyPhoto.jpg

Description: The top left camera is larger than usual

Fig. 3. Image viewer for medical records

addition to: image type, medical record Id, and a short description about it. MongoDB files have only one property, Metadata, that can be used to store additional data, but in this case 3 properties were required. The NewtonSoftlibrary allows serializing objects in a JSON format string. Thus, the whole object was written as a string in the only available property. The GridFSfacility allows you to filter these documents as long as the filter is applied for the members of the first level of the serialized object. Filtering was required to find images specific to a particular medical record. An example of image processing was listed in Figure 3. One of the best improvements will be to edit this kind of images, but in the UWP technology, these operations require a lot of time for research.

4. Experiments and Results

After the system was implemented, based on the MongoDB collections, the equivalent structure of a relational database was created(Sadalage, 2013) (Trivedi, 2014). Excluding the collection of images that had not been projected in the relational database, the system had the following collections:

- Persons;
- Hospitals;
- Drugs;

- Diagnoses;
- MedicalRecords.

Following the application of the principles of normalization of databases to MongoDB collections, four times more tables were created. The relational schema is shown in Figure 4.

It can be observed that storing information in the form of embedded documents reduces the number of entities needed for working. The big advantage brought by MongoDB is that the data access is made much faster and easier than in SQL Server,

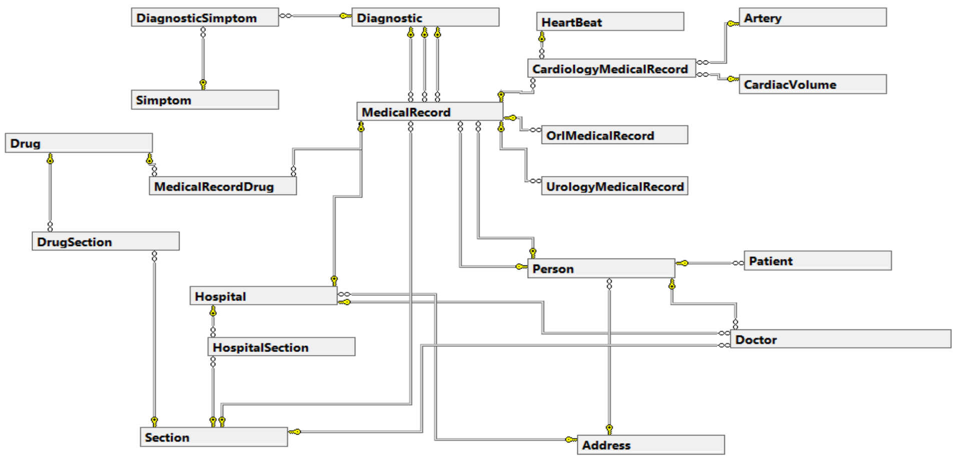


Fig. 4. SQL Server database schema

without linking the collections. There is also a disadvantage: data duplication since there are no separate collections to contain them.

4.1. *Experiment 1 – Insert data*

With both database structures, they were multiplied to hold a different number of medical records: 1000, 10000, 100000, and 1000000. The test data needed to be generated, so generators were created for them. Since working with advanced technologies, MongoDB driver and EntityFramework is fast for a developer, the generators have been built by implementing these technologies.

MongoDB has its own document processing limit, and EntityFramework has high performance issues when processing a large number of records. To avoid this problem, the data has been synchronized in batches. This technique requires the processing of a limited number of records at a time. It has been chosen that the total number of processed items to be equal to 100. During this experiment the times for the insertion of the medical records have been obtained. Because EntityFramework has performance issues, there are extensions that bring benefits. A new EntityFramework experiment was handled using the Z.EntityFramework.Extensions library. All results were expressed in seconds. In Table 1 and Figure 1, the results for this experiment can be viewed. It is noted that the insertion of 1000000 medical records from the MongoDB driver lasted about a minute and a half, and in EntityFramework about 55 minutes. The installed extension also brought better results, but far from those offered by the MongoDBdriver.

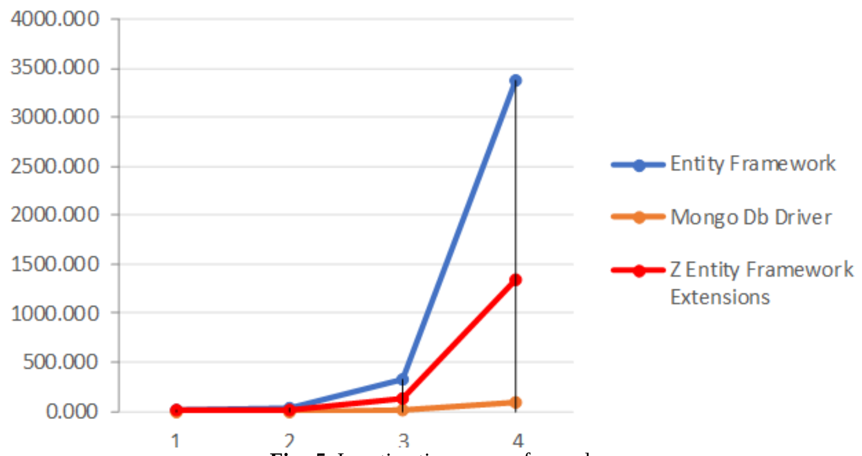


Fig. 5. Insertion time vs no of records

Table 1. Insertion time

Records / Framework	1000	10000	100000	1000000
Entity Framework	3,242	31,688	332,340	3374,387
MongoDB Driver	0,145	0,855	7,663	80,790
Z Entity Framework Extensions	2,384	14,415	128,269	1340,831

4.2. *Experiment 2–Database Query Type 1*

The following two experiments contain MongoDB / SQL Server equivalent queries, respectively the MongoDB driver / EntityFramework queries.

The first query consisted in calculating the costs for each diagnosis from medical records of the Cardiology type, with certain parameters and created within a certain interval. In Table 3 we have the equivalent queries, and in Table 2, Figure 2, the results obtained. Some additional joins have been added to have a SQL Server query model as the same as the MongoDB models.

At first sight, the MongoDB query appears larger, but most of the lines contain only the JSON file character. The SQL Server achieved better times than MongoDB. It seems that the “join” operations does not affect significantly the performances.

Table 2. Query response time dependence of database size

Technology \ Records	1000	10000	100000	1000000
Entity Framework	3,242	31,688	332,340	3374,387
MongoDB Driver	0,145	0,855	7,663	80,790
Z Entity Framework Extensions	2,384	14,415	128,269	1340,831

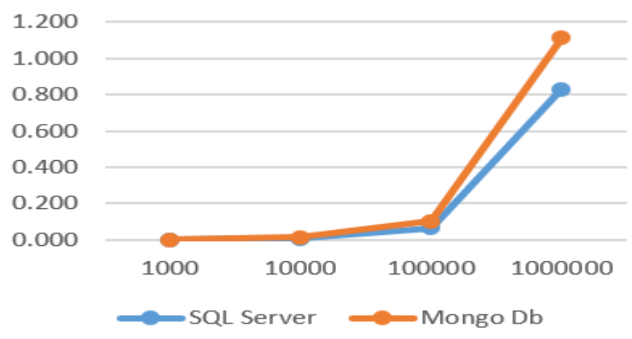


Fig. 6. Execution time vs no. of records

Table 3. Query equivalence (MongoDB vs. SQL)

MongoDB	**SQL Server**

db.getCollection('MedicalRecord').
aggregate([{
 "$match": {
  "FileType": "Cardiologie",
  "AdmissionDate": {
   "$gte": ISODate("2017-12-31T22:00:00Z"),
   "$lte": ISODate("2018-01-09T22:00:00Z")
  },
  "Arteries": "Arteries1",
  "HeartBeat": "HeartBeat2"
 }
},
{
 "$group": {
  "_id": "$SentDiagnostic",
  "__agg0": {
   "$first": "$SentDiagnostic"
  },
  "__agg1": {
   "$sum": "$HospitalizationCost"
  }
 }
},
{
 "$project": {
  "Diagnostic": "$__agg0",
  "HospitalizationCost": "$__agg1",
  "_id": 0
 }
},
{
 "$sort": {
  "Diagnostic": 1,
  "HospitalizationCost": 1
 }
}
])
	SELECT SD.Name as DiagnosticName, sum(HospitalizationCost) As TotalCost from MedicalRecord
	left join CardiologyMedicalRecord on MedicalRecord.Id = CardiologyMedicalRecord.MedicalRecordId
	left join UrologyMedicalRecord on MedicalRecord.Id = UrologyMedicalRecord.MedicalRecordId
	left join OrlMedicalRecord on MedicalRecord.Id = OrlMedicalRecord.MedicalRecordId
	join Diagnostic SD on MedicalRecord.SentDiagnosticId = SD.Id
	join Diagnostic AD on MedicalRecord.AdmissionDiagnosticId = AD.Id
	join Diagnostic LD on MedicalRecord.DiagnosticAfter72HoursId = LD.Id
	join Section on MedicalRecord.SectionId = Section.Id
	left join Artery on CardiologyMedicalRecord.ArteriesId = Artery.Id
	left join HeartBeat on CardiologyMedicalRecord.HeartBeatId = HeartBeat.Id
	where Section.Name = 'Cardiologie'
	 and AdmissionDate between '2018-01-01' and '2018-01-10'
	 and Artery.Name = 'Artery1'
	 and HeartBeat.Name = 'HeartBeat2'
	group by SD.Name
	order by TotalCost, DiagnosticName

4.3. *Experiment 3 – C# Query Type 1*

The size and the view of the queries using the technologies is different. MongoDB driver is the best for creating such queries. In the Table 5 the EntityFramework query is almost double as size.

Table 4 and Figure7 show the results obtained. In each case, EntityFramework takes more than 2 seconds, and the MongoDB driver is under this interval. However, if wconsidering the difference between the results with the lowest number of records and the last valid database, there is a difference of only about 0.35 seconds for EntityFramework and 1.25 seconds for the MongoDB driver, which is 3 times higher. Most likely, the MongoDB driver performances will increase and get close to EntityFramework on larger databases.

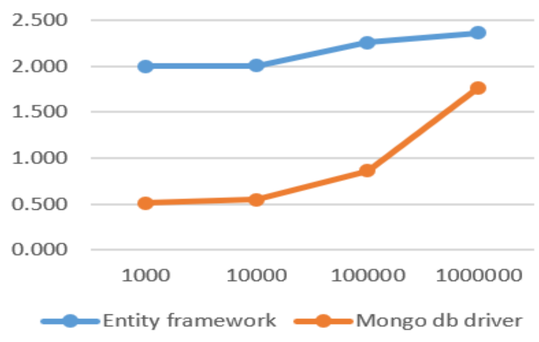


Fig. 7. Execution time vs no of records

Table 4. Query response time dependence of database size

Records / Technology	1000	10000	100000	1000000
EntityFramework	2,003	2,010	2,261	2,367
MongoDB driver	0,508	0,548	0,858	1,767

Table 5. Query equivalence (MongoDB vs. SQL)

MongoDB driver	**EntityFramework**
mongoDbContext.MedicalRecords .AsQueryable() .Where(x =>x.FileType == "Cardiologie" &&x.AdmissionDate>= new DateTime(2018, 1, 1) &&x.AdmissionDate<= new DateTime(2018, 1, 10) &&x.Arteries == "Arteries1" &&x.HeartBeat == "HeartBeat2") .GroupBy(x =>x.SentDiagnostic)	(from mr in appContext.MedicalRecords join cmr in appContext.CardiologyMedicalRecords on mr.Id equals cmr.MedicalRecordId into tcmr from cmr in tcmr.DefaultIfEmpty() join omr in appContext.OrlMedicalRecords on mr.Id equals omr.MedicalRecordId into tomr from omr in tomr.DefaultIfEmpty() join umr in appContext.UrologyMedicalRecords on mr.Id equals umr.MedicalRecordId into tumr

.Select(x => new { Diagnostic = x.First().SentDiagnostic, HospitalizationCost = x.Select(y => .HospitalizationCost).Sum() }) .OrderBy(x =>x.Diagnostic).ThenBy(x =>x.HospitalizationCost).ToList();	from umr in tumr.DefaultIfEmpty() join sd in appContext.Diagnostics on mr.SentDiagnosticId equals sd.Id join ad in appContext.Diagnostics on mr.SentDiagnosticId equals ad.Id join ld in appContext.Diagnostics on mr.SentDiagnosticId equals ld.Id join s in appContext.Sections on mr.SectionId equals s.Id join a in appContext.Arteries on cmr.ArteriesId equals a.Id into ta from a in ta.DefaultIfEmpty() join hb in appContext.HeartBeats on cmr.HeartBeatId equals hb.Id into thb from hb in thb.DefaultIfEmpty() where s.Name == "Cardiologie" &&mr.AdmissionDate>= startDate &&mr.AdmissionDate<= endDate &&a.Name == "Artery1" &&hb.Name == "HeartBeat2" group mr by sd.Name into g select new {DiagnosticName = g.Key, TotalCost = g.Select(y =>y.HospitalizationCost).Sum()}) .OrderBy(x =>x.DiagnosticName) .ThenBy(x =>x.TotalCost) .ToList();

4.4. *Experiment 4 –Query Type 2*

The last query is simple. It sorts out all medical records. The SQL Server query has been trimmed because EntityFramework did not generate a complete MongoDB file model in memory. If the MongoDB driver has created all the documents in memory with a 2.5GB RAM space, EntityFramework has reached 9GB of RAM without completely creating all the documents. The query from MongoDB needed an additional setting allowDiskUse = true because the limit is set to 100 MB of data that can be sorted into memory. In each case, MongoDB has achieved better results. Table 6 and Figure8 show the results and the Table 7 the equivalent queries

Table 6. Query execution time

Technology \ Records	1000	10000	100000	1000000
SQL Server	0,234	0,758	6,411	65,621
MongoDB	0,016	0,141	2,140	20,400

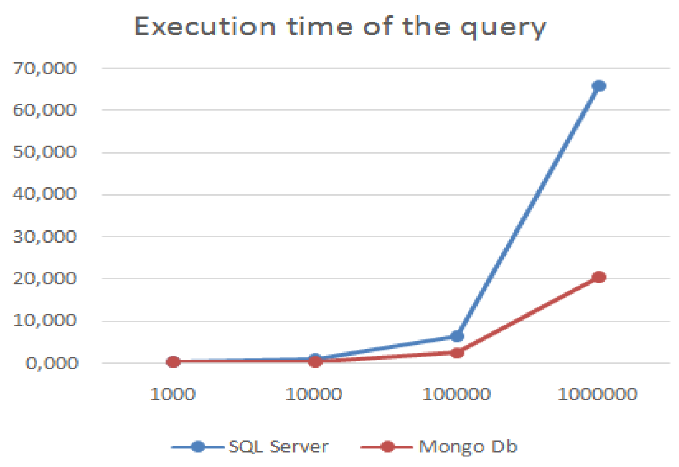


Fig. 8 Execution time vs no of records

Table 7. Query equivalence (MongoDB vs. SQL)

MongoDB	SQL Server

db.getCollection('MedicalRecord').
aggregate([{
 "$sort": {
  "CreateDate": 1
 }
}],
{
allowDiskUse: true
})
	SELECT * from MedicalRecord
	left join CardiologyMedicalRecord on MedicalRecord.Id =
	CardiologyMedicalRecord.MedicalRecordId
	left join UrologyMedicalRecord on MedicalRecord.Id =
	UrologyMedicalRecord.MedicalRecordId
	left join OrlMedicalRecord on MedicalRecord.Id =
	OrlMedicalRecord.MedicalRecordId
	order by CreateDate

4.5. *Experiment 4 – C# Query Type 2*

Only links to the secondary medical records tables have been made. No connection for diagnoses, drugs or other important tables due to the performance problem. For the MongoDB driver, the same parameter was added to be able to sort out more than 100 MB of data in memory. Again the MongoDB driver came out higher.Table 8 and Figure 9show the results and the Table 9 the equivalent queries. In the completion mode, the SQL server query was much bigger.

Table 8. C# Query execution time

Records / Technology	1000	10000	100000	1000000
EntityFramework	2,494	3,811	17,027	203,590
MongoDB driver	0,656	1,489	11,305	108,354

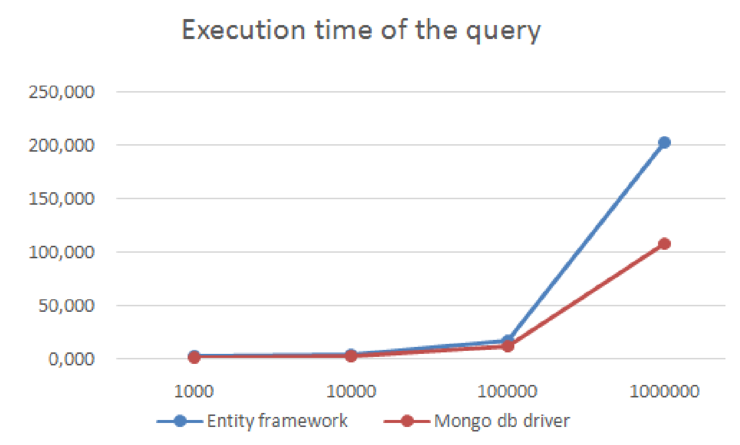


Fig. 9. Execution time or for C# query vs. no of records

Table 9. Query equivalence (MongoDB vs. SQL)

MongoDB driver	**EntityFramework**

var aggregateArgs = new AggregateOptions() {
AllowDiskUse = true};
mongoDbContext.MedicalRecords
.AsQueryable(aggregateArgs).OrderBy(x
=>x.CreateDate).ToList();
	appContext.MedicalRecords
	.Include(x =>x.CardiologyMedicalRecords)
	.Include(x =>x.OrlMedicalRecords)
	.Include(x =>x.UrologyMedicalRecords)
	.OrderBy(x =>x.CreateDate)
	.ToList();

5. Conclusions

The paper presents the technical details of the implementation for a medical application that can be used to handle a large volume of information of medical records.

A prototype of a complete application was designed that allows 2 types of Database servers handling: A SQL server database and a non-reational database server (MongoDB).

The information used for tests was based on medical records examples provided by the Emergency County Hospital, Buzau.

The aim of the experiments was to measure the scalability and performances of the two type of servers for different types of queries and different number of records.

The results obtained with this application shows that the MongoDB server behaves better on large-scale databases which could be a better choise for the final version of this application.

References

Celko, J. (2014). Joe Celko'sComplete Guide to NoSQL: What Every SQL Professional Needs toKnowabout Non-Relational Databases, 1st Edition. Elsevier, USA

DuBois, P. (2014). MySQL Cookbook: Solutions for Database Developers and Administrators. 3rd Edition. O'Reilly Media, USA

Krisciunas, A. (2014).Benefitsof NoSQL: https://www.devbridge.com/articles/benefits-of-nosql/

Kvalheim, C. (2015). The Little Mongo DB Schema Design Book. The Blue Print SeriesMembrey, P., Hows. D., Plugge, E. (2014). MongoDB Basics. Apress

Sadalage, P.J.; Fowler, M. (2013). NoSQL Distilled: A Brief Guide tothe Emerging World ofPolyglotPersistence, 1st Edition, Addison Wesley, USA

Trivedi, A. (2014). Mapping Relational Databases and SQL to MongoDB: http://code.tutsplus.com/articles/mapping-relational-databases-and-sql-to-mongodb--net-35650

Strathweb, Dealingwith large files in ASP.NET Web API https://www.strathweb.com/2012/09/dealing-with-large-files-in-asp-net-web-api/