An efficient uniform model to integrate SQL and NoSQL databases

R. Zaki
Faculty of computers and information systems, I.S dep.
Mansoura University, Egypt
eng.rzaki@scc.mans.edu.eg

A. Rezk
Faculty of computers and information systems, I.S dep.
Mansoura University, Egypt
amira_rezk@mans.edu.eg

Sh. I. Barakat
Faculty of computers and information systems, I.S dep.
Mansoura University, Egypt
sherifiib@yahoo.com

ABSTRACT

In recent years, systems become more complex due to massive growth and variety of data which stored in different formats (structured and unstructured). In many cases the organizations need to use both SQL and NoSQL databases to store their data and get the advantages of both systems. A big challenge for organizations is how to integrate these data and retrieve it in a uniform format. The difficulty of integration is not only because the heterogeneity of query language and the data format but also the heterogeneity of semantic and structure. Although many researches were done to overcome these challenges, it introduced a specific solution for a special case or limited feature rather than a general model. This paper introduces a standard solution to integrate SQL with NoSQL databases and retrieve data from them in a unified form. The proposed model allows developers to write and execute queries against different data sources easily at the same time. This model solves the problems of heterogeneous data integration along three diagonal structure, semantic, and syntax heterogeneity. A web-based application is developed to ensure the usefulness of the proposed solution.

General Terms
Database System, Data Integration

Keywords
NoSQL, SQL, Integration, Uniform Database Access

1. INTRODUCTION

Recently, Database domain becomes more heterogeneous than ever due to the rapid growth in digital data generated by real-time web applications and social networking websites which is very huge and unstructured. [1, 2]

Although Relational databases provide the easiest way to store and retrieve data as they follow a predefined schema with a well-known structured relation and a standard query language called SQL (Structured Query Language), they can’t handle such huge amount of digital unstructured and semi-structured data generated by current applications. As a result, new database model is gaining significant attention in the enterprise called NoSQL (Not Only SQL). [2, 3] NoSQL emerged to serve as the core system of Big Data applications because it can handle massive volumes of unstructured data and it is characterized by its high availability, scalability, and performance. [1, 4] NoSQL databases fall into four models (key-value, columnar, document, and graph) each is convenient for specific scenario. [1]

The “one size fits all” thinking about database systems has been questioned because new current applications started to require unprecedented needs that can’t be fulfilled easily by relational databases. These needs are not only managing huge amounts of data but also providing more flexible schemas to handle rapidly changing needs of organizations, high throughput, and quick scaling up or down. [1, 3] Relational databases can handle large volumes of data but with some weaknesses; unstructured and semi-structured are hard to be handled, it is very complicated in scaling up as well as higher cost, and performance is affected when data is distributed over geographically distant sites. [1]

There are many situations where NoSQL databases are the right tool for the job, but many others are well suited for traditional relational data storage. For example, a software application requires data storage where a part of the data is perfectly suitable for traditional relational databases, whereas the other part of data is stored ideally in NoSQL databases. This raises a problem regarding which type of data storage is the perfect choice for the application. Since, different parts of data are suitable for different types of data storage, then choosing one type of database means that a part of the data is stored in a less convenient way. [5]

A compromise solution is to store structured data in SQL database and unstructured data in NoSQL database. [5] By this way, it will be possible to take advantage of both relational and NoSQL databases, but this solution raises a problem of how to retrieve and provide users with a unified output from these different data sources. The best solution to this problem is to integrate data from different sources and retrieve it as if stored in one place. As a result, ERP (Enterprise Resource Planning) systems emerged to support the integration of different systems in organization and handle communication between them. Over time, ERP systems become more painful to implement due to a serious drawback which is the incompatibility between ERP standards and organization’s business model. [6, 7]

So, organization starts to seek for new approaches to integrate their disparate systems. A new class called EAI (Enterprise Application Integration) spotted in the community of software integration to merge different systems within organization. [8] EAI is handled through four levels: Data Level, Application
The rest of the paper is organized as follows. In section 2, the proposed framework is explained from a theoretical perspective as well as presenting the application that implements the proposed framework. Section 3 handles the evaluation of the proposed framework. Finally, section 4 concludes the paper.

2. The Proposed Model

This paper proposes a generic framework to integrate data from different SQL and NoSQL data sources into a unified format without the burden of moving data between different stores. To fulfill this goal, the paper aims to achieve the following objectives:

- Integrate data from different databases in uniform format.
- Provide a standard access method to query data from different sources without moving data between data stores or any conversion between NoSQL and SQL data.
- Solve problems of syntax, semantic, and structure heterogeneities of data by developing an intermediate model between different sources and users.

2.1 System Architecture

The architecture of the proposed solution consists of five major components namely, Controller, View, EF Layer, ADO.Net Providers, and data stores represented as SQL Data Store, Cassandra Data Store, and MongoDB Data Store as seen in Figure (1).

![Architecture of proposed model](image-url)
programming against a conceptual model instead of directly programming against a data storage schema, and it supports both relational and NoSQL databases. [13, 14] EF layer consists of a Model layer and an EDM (Entity Data Model) layer.

Model is a layer responsible for application data management routines to handle database operations. [11] It contains a set of domain classes to represent each element in the database in the form of entity within the application regardless of real database structure and type. This facilitates the communication between application and data source.

EDM is a mapping layer responsible for creating a relationship between application data and data stored in database. It describes the structure of data despite how it is stored. [13, 15] It consists of three basic components:

- Conceptual Schema is responsible for representing the structure of data in the form of entities and relationships using a domain-specific language called Conceptual Schema Definition Language (CSDL). [15]
- Mapping Schema contains information about how entities and relationships from conceptual layer are mapped to actual tables at logical layer. Mapping information is represented using an XML-based language called Mapping Specification Language (MSL). [16]
- Storage Schema contains the entire database schema (tables, relations, views, and keys) represented using an XML-based language called Store Schema Definition Language (SSSDL). [16]

ADO.Net providers are responsible for communicating with specific data store. Each database has its own provider that allows easier and faster access to data. [17]

Data Stores represent databases where data is stored in (SQL data store, Cassandra data store, and MongoDB data store). Choices of data stores are made mainly because of their wide popularity and the fact that they are supported by EF architecture, so they implement their own database providers.

2.2 Resolving data integration problems
The proposed solution to resolve problems of semantic, structural, and syntactic heterogeneities of source systems is introduced in following sections.

2.2.1 EF Layer: Solve semantic and structural heterogeneity problem
The basic idea is to create a middle layer between data source and user to retrieve and execute queries easily. Then, merge results and display them in a unified form to the user.

EF layer represents the middle layer in the proposed architecture. This layer is built automatically through EF graphical user interface or manually by writing codes of domain model classes and EDM schemas. In case that data in data stores are identical (e.g. the same table in SQL server refers to the same collection in MongoDB) then the developer can generate EF layer automatically. If data is not identical in data stores, then the developer will create the EF layer manually to handle semantic differences easily. Figure (2) shows how EF layer works.

Domain Classes allow the developer to focus on domain object instead of database object. Each object in database (e.g. table in SQL, collection in MongoDB) has a domain model class that contains all its metadata information (e.g. table name, column name, and column datatype) in form of properties to facilitate developer’s work.

EDM is responsible for generating general and independent descriptions of the data models of source systems called SS (Storage Schema), CS (Conceptual Schema), and MS (Mapping Schema).

EDM schemas contain all metadata about source system that will help in resolving semantic and structural heterogeneities. The data model descriptions are based on XML elements as mentioned in section (2.1) because XML is a text based markup language with self descriptive tags that provide easy and understandable structure of the data of schemas.

SS allows the developer to overcome the problem of differences in the data model of source databases by representing the real information of the source database as XML tags regardless of its structure. For example, in SQL database, it stores database name and type, table name, table columns metadata (name, datatype, length… etc.), and table keys. The same SS schema with the same structure can created for MongoDB and Cassandra DB.

Domain Classes and CS help the developer solve the semantic differences between source databases by defining aliases for elements that have the same meaning in different databases as properties in Domain Classes. These alias properties guarantee consistent name conversion for database elements. CS contains a description for each domain class by storing all its properties as XML tags (e.g. property name, datatype, keys, and relationships).

MS contains information about linking both SS and CS together to guarantee consistent semantic mapping of database elements. It consists of XML entities where each entity contains SS element and its matched CS element.

As it can be seen, EF layer is a flexible intermediate layer that describes schemas of source databases independently from the implemented data model’s type. The proposed solution proved that the principle of generating general and independent schemas of source systems is applicable for different realational databases (SQL), NoSQL document-oriented data stores (MongoDB), and NoSQL column-oriented data stores (Cassandra DB). It can be possible to describe schemas of other NoSQL databases easily with EDM general schemas.
2.2.2 Database Providers & LINQ Query: Solve syntactical heterogeneity problem

Syntactic heterogeneities can be resolved by developing database providers which can translate different queries to different database query languages. In the proposed solution, database queries are integrated into the programming language of the application, written as a set-based queries, and are called LINQ (Language Integrated Query).

LINQ queries help the developer handle data as objects when retrieve, as well as generating queries in one format without the need for separate query language for each database. Providers connect to the database, translate LINQ queries into the appropriate database query language, execute queries against data source, and retrieve results.

Figure (3) shows an example of a LINQ query. In this example, the developer wants to query all students’ information that is recorded in a specific subject. The query consists of database context object that contains all database connection information and allows the provider to identify in which data store information is stored. The domain model class contains all information about the student object identical to its counterpart in the database as explained in section (2.2.1). The condition method allows the developer to apply specific conditions on the domain class and accepts the conditional parameters in form of Lambda Expression.

The LINQ query is expressed as a set of standard declarative operators. These operators are translated by EF to command trees representation and with the help of database providers the CS and SS descriptions map entities of domain model class to their counterparts in the data store, generate native query expressions as presented in Figure (4), and execute them in the appropriate data store.

2.2.3 System Flow Chart

Figure (5) presents a flow chart of the proposed solution and illustrates how different queries are executed and merged into single output.

3. The Proposed Model Evaluation

The proposed model was evaluated for its performance and types of queries supported using hardware and software configurations displayed in Table (1).
Software | Microsoft SQL Server 2014 x64, MongoDB 3.4.4, Cassandra 3.0.9, and Microsoft Visual Studio 2015 x64.
---|---
Data Providers | Cassandra CData.Cassandra v18.0.6719
| MongoDB CData.MongoDB v18.0.6705
Implementation | C# Language.

This section proceeds as follows. Section 3.1 analyzes the performance of the developed system. Section 3.2 discusses different types of queries supported by the developed system.

### 3.1 Performance Evaluation

Performance tests were based on Training Center databases. Same database schema was used for SQL, MongoDB, and Cassandra databases. Tables of Student, Subjects, and Cert were used during the tests. Tests were performed on two databases with different sizes. All tests were performed 15 times to avoid result’s skewing. Different database queries were performed on these datasets.

Each query contained only a few selected attributes. Two queries performed select statement that affected only one object in the source system, one without a condition and the other one with a condition. Two queries performed join statement; the first join statement was performed on two objects of the source systems while the second join statement was performed on three objects of the source systems. Data rows resulting from executing previous queries on both Dataset 1 and Dataset 2 are listed as follows in Table (2).

<p>| Table 2: Data rows of test queries |</p>
<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>87401</td>
</tr>
<tr>
<td>Query 2</td>
<td>8771</td>
</tr>
<tr>
<td>Query 3</td>
<td>87401</td>
</tr>
<tr>
<td>Query 4</td>
<td>8051</td>
</tr>
</tbody>
</table>

Each query was evaluated regarding to the application’s entire running time and data retrieval time. Entire running time starts when the user sends a request to the system and ends when the queried data is available in HTML table format. It includes execution time on the application and execution time on the database as illustrated in Equation (1).

\[
ERT = AET + DET \tag{1}
\]

Where:

- \( ERT \) refers to Entire Running Time.
- \( AET \) refers to Application Execution Time.
- \( DET \) refers to Database Execution Time.

Equation (1.1) illustrates the execution time on the application which includes sending query request to the controller, identifying type of query and translating it to data source’s appropriate query language, transforming native query result into objects, and merging results in single output. Execution time on the database includes executing translated queries on the data source and sending native query results to the application.

\[
AET = QTN + QTO + QMT \tag{1.1}
\]

Where:

- \( AET \) refers to Application Execution Time.
- \( QTN \) refers to Query Translation Time to Native Language.
- \( QTO \) refers to Query Result Transformation Time to Object.
- \( QMT \) refers to Query Merge Time.

Data retrieval time starts when a user request is sent from the system to the data source and ends when the queried data is available in object format. The data retrieval time includes executing translated queries on original data source, retrieving native query results, and transforming them to objects of the system as illustrated in Equation (2).

\[
DRT = QET + QTO \tag{2}
\]

Where:

- \( DRT \) refers to Data Retrieval Time.
- \( QET \) refers to Query Execution Time on database.
- \( QTO \) refers to Query Result Transformation Time to Object.

The performance of the tested queries according to previously mentioned time measures is presented in Figure (7) and Figure (8).

**Fig 7: Time measures of Dataset 1**

The time measures of tested queries performed on the first dataset of the test are shows in Figure (7). In Dataset 1, the highest average values of the entire running time were in third and fourth query with values 0.719 and 0.663 seconds respectively. These values were affected by many factors the number of records returned from the queries about 87401 and 8051 records respectively, the complexity of join operation (query 3 affects 2 objects of source systems and query 4 affected 3 objects of source systems), and the execution time on each data source separately.

The next step after data was retrieved from each data source was to merge them in single output. The concatenation process is very fast. It required about 0.005 seconds merging 29151 records from SQL with 29099 records from Cassandra with 29151 records from MongoDB which was highest average value according to Figure (7), and in case of smaller number of records the merge took about 0.000 seconds.
The time measures of tested queries performed on the second dataset of the test are shown in Figure (8). In Dataset 2, the entire running time increased dramatically in case of query 3 and query 4 with values 2.020 seconds and 2.004 seconds respectively. The reason for this increase was the complexity of join operation which in turn increased the execution time on each data source. Another reason was the big number of records retrieved, about 182208 records were retrieved from query 3 whereas about 39622 records were retrieved from query 4 which was smaller than records retrieved from query 3; however, joining three objects on three different data sources is a very complicated and exhausted process. Concatenating records retrieved from each data source is a very fast process. To merge about 92148 records from each data source together, it needed about 0.014 seconds and with smaller number of records it needed about 0.004 seconds to perform the merge operation as shown in Figure (8).

Overall, it can be stated that the entire running time of the solution depends primarily on the speed of executing native queries on different databases and on the transfer time between application and databases. The transfer time can be enhanced by increasing the capabilities of the server which hosts the application (e.g. higher processor and bigger RAM).

3.2 Supported Queries

The purpose of the application of the proposed solution is to retrieve data from heterogenous data sources (SQL, MongoDB, and Cassandra) regardless of how it is stored. This means that the application depends primarily on the SELECT statement in its multiple forms, which depends on the nature of the user's requirements. In the proposed solution, the application uses LINQ as a standard query language for different data sources with the help of data providers and EF layer. LINQ allows developers to execute almost all kinds of a select statement on different databases easily. Table (3) presents a list of some forms of a select statement which performed on the proposed solution.

Table 3: Supported Queries List.

<table>
<thead>
<tr>
<th>Select Where</th>
<th>SQL DB</th>
<th>MongoDB DB</th>
<th>Cassandra DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select All</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

According to Table (3) Cassandra allows filtering data on a select statement but with a restriction that the filtering column is a primary key or an indexed column. Unfortunately, the LINQ query couldn’t solve this issue.

The join operation is not supported by MongoDB (until v. 2.3) and Cassandra (until v. 3.10). In the proposed solution, LINQ query succeeded in executing the join operation on both databases with the help of their data providers.

The group by operation can’t be executed on Cassandra (until v. 3.10 with restriction that the group by column is a Partition Key or Partition Key and Clustering Key), but it was executed easily using LINQ query.

The order by operation is supported on Cassandra but only if the select statement has a where clause and the ordering column is a clustered column, this was solved by a LINQ query and data could be sorted easily.

As it can be seen in Table (3), the proposed solution focused on the most common used forms of the select statement and provided an efficient way to execute them despite of the restrictions mentioned above.

3.3 The Proposed Model Advantages and Disadvantages

Based on the model evaluation and the performance analysis, the proposed model advantages and disadvantages can be declared as following:

Advantages of the proposed model:

1. Only one query language LINQ.
2. Add any type of database if it has the appropriate ado.net provider.
3. Allow developers of any organization build their own integrated application easily using the same structure of the proposed model.
4. Overcome heterogeneity problems (syntax, structure, semantic).
5. Handle almost all common clauses used for retrieving data easily even if they are not supported by native database.
6. No data conversion from SQL to NoSQL or from NoSQL to SQL because data is executed on each data source and retrieved in uniform format.

Disadvantages of the proposed model:
4. Conclusion & Future Work

This paper focuses on solving problems which result from integrating and retrieving data stored in different data sources. A uniform model to integrate SQL and NoSQL databases is introduced. The proposed solution depended on an EF layer to represent meta-data information about different data sources in EDM schemas and map them to original database objects in order to execute queries. The solution used LINQ as a standard query language to write different database queries in single format, and then translated by database providers to native queries. The results were concatenated and displayed in single tabular form.

The proposed solution provides the developer with an easy and effective solution to write queries against different data sources using one query language. The solution succeeded in performing operations that are not supported by Cassandra and MongoDB. Also, it provided an efficient way to retrieve data in uniform format.

4.1 Future Work

In the future, supporting all other forms of the select statement is an important issue to be considered. Also, the proposed system was built specifically for SQL, MongoDB, and Cassandra databases so an area to be researched will be to add other NoSQL databases to the system.

5. ACKNOWLEDGMENTS

I would like to thank Dr. A. Rezk and Dr. Sh. I. Barakat for their help and guidance when supervising this research. Thanks and gratitude goes to my family and friends for their support in my study.

6. REFERENCES

[1] Bc. Ondrej Pánek. Integration of Heterogeneous Data Sources Based on a Catalog of Master Entities (May 2015). Czech Technical University in Prague - Faculty of Electrical Engineering - Department of Computer Science and Engineering. Pages (1-4)


