Procedia Computer Science

Volume 51, 2015, Pages 2593–2602

ICCS 2015 International Conference On Computational Science

A Framework for Migrating Relational Datasets to NoSQL*

Leonardo Rocha, Fernando Vale, Elder Cirilo, Dárlinton Barbosa, and Fernando Mourão

Universidade Federal de São João del Rei, São João del Rei, Minas Gerais, Brasil
{lcrocha,fvale,darlinton,elder,fhmourao}@ufsj.edu.br

Abstract

In software development, migration from a Data Base Management System (DBMS) to another, especially with distinct characteristics, is a challenge for programmers and database administrators. Changes in the application code in order to comply with new DBMS are usually vast, causing migrations infeasible. In order to tackle this problem, we present NoSQLayer, a framework capable to support conveniently migrating from relational (i.e., MySQL) to NoSQL DBMS (i.e., MongoDB). This framework is presented in two parts: (1) migration module; and, (2) mapping module. The first one is a set of methods enabling seamless migration between DBMSs (i.e. MySQL to MongoDB). The latter provides a persistence layer to process database requests, being capable to translate and execute these requests in any DBMS, returning the data in a suitable format as well. Experiments show NoSQLayer as a handful solution suitable to handle large volume of data (e.g., Web scale) in which traditional relational DBMS might be inept in the duty.

Keywords: Framework, Big Data Migration, Relational, NoSQL

1 Introduction

A modern generation of software applications, designed to meet demands from small groups of users or large organizations, has to deal with a huge growth in the volume of data to be processed and stored. Applications related to Social Media [13] data illustrates this scenario, in which the decentralized creation and exchange of user-generated content are fostered and leads to a awe-inspiring worthy amount of data. For many decades, the Relational Database Management Systems (RDBMS), based on the relational model, satisfactorily fulfilled the requirements of the most diverse software applications. Its success is due to the offering of simplicity to developers as well as robustness, high performance and compatibility. However, more recently, the traditional RDBMSs have been shown not able to handle efficiently the demands of a new generation of software applications, which are especially focused on unstructured data and massive storage.

*This work was partially supported by CNPq, CAPES, Fapemig, and INWEB.

doi:10.1016/j.procs.2015.05.367

In order to meet the needs for efficient storage and accesses of large amounts of data, the Database Management Systems "Not only SQL", or simply NoSQL [12], were proposed. We can observe today that some companies have been joined to this new way of massive data computing, such as Google, Facebook, Twitter and Amazon. In most cases, they are trying to adhere to demands for scalability, high availability and storage of huge amount of unstructured data. The main characteristics that distinguish the NoSQL model from the traditional RDBMS one are partitioning of data and data replication. Despite these benefits, most of existing large and medium scale software applications are still based on RDBMS since there are many challenges associated with the migration process. The first one is the volume of data to be migrated. Organizations, in general, decide to migrate their database when the volume of stored data is huge and the RDBMS are no longer able to satisfy the scalability and high availability expectations. Another challenge is related to the relational database model's manner of avoiding data redundancy, which is part of the NoSQL models. In this case, it is required to maintain the new data model semantically identical to the original one. Thus, all existing relationships have to be adequately represented without loss or distortion of data. Finally, in addition to all data and model migration, there is a cost associated with adapting software applications to communicate properly with the new database model.

We can find in the literature some solutions that offer an (semi-)automatic migration process or adjustments in relational models so they become able to offer better performance and scalability when dealing with large volume of data [15]. However, the cost of adapting the source code of applications is usually neglected. In this paper, we present *NoSQLayer*, a framework to perform automatically and transparently data and model migration from relational (i.e., the MySQL) to NoSQL databases, more specifically MongoDB. The main idea behind the *NoSQLayer* solution is to keep the semantics of the original database, not only in the manner data is modeled, but also w.r.t. how programmers write the source code to query the database. Our framework offers an abstract layer that allows software applications to access data in the NoSQL model transparently, without the need of changing the existing queries in the applications. *NoSQLayer* grants this feature by performing all data migration, the entire structure of the original database is maintained and being all data stored as a NoSQL model. Each SQL operation (i.e., select, insert, update and delete) requested by the original application is captured by our framework and converted into requests to the NoSQL database. Once the SQL operations are performed on the NoSQL database, its result is converted to the SQL standard format and forwarded to the application, so that any modification in the original application is need. [Some information about the structure of the original database is maintained to assist the *NoSQLayer*.

We evaluate our proposal following two different perspectives: qualitative and quantitative. The qualitative evaluation is a proof of concept, in which a typical software application based on the relational database is implemented using *NoSQLayer* in order to show it working in practice. In this case, firstly, we performed the entire migration process, moving from the original RDBMS (MySql) to MongoDB. Later, various SQL operations were executed through *NoSQLayer* over both databases (MySQL and MongoDB) showing that in 100% of the cases the results were identical. In the quantitative evaluation, our objective was to evaluate the overhead presumably caused by the introduction of the *NoSQLayer* layer. To perform this assessment, we employed an experimental set-up with a huge volume of data. In this case, we evaluated the response times of various operations executed over both the original RDBMS and over the *NoSQLayer* layer, always varying the volume of data involved in the operation. In all scenarios we observed that the overhead presumably caused by NoSQLayer is only significant when the volume of data involved in the operation is small, that is, according to the growth of data applied in each operation, the perceived overhead was neutralized by the excellent performance of the NoSQL model. Based on the observed results, we can state that our solution, in addition to be economically viable, is also computationally efficient.

2 Grounding Research

Comparisons between relational data models and NoSQL models have already been presented in the literature. For instance, [12] identified some distinct characteristics between these two models, such as the method of storing and retrieving information through requests. In relational data models, tables are typically defined and stored by a rigid schema contrasting with NoSQL models that have no pre-defined schema, making the attributes of a record are not necessarily the same. The authors also performed a comparison between some of the major NoSQL database models, such as: Key-value [10], used by the management systems Riak, Redis and Project Voldemort; document oriented [8], used by the MongoDB, CouchDB and SimpleDB systems, and column oriented [8], implemented by Google Big Table and Cassandra systems. One of the main findings of this work is the difficulty of RDBMS to deal with databases on which the data volume is large.

Recently, we have noted works aimed at dealing with large volume of data in RDBMSs. In [14], the authors created an abstraction layer between SQL and NoSQL databases. According to the authors, certain data sets show better results when processed by relational databases, and others are better running on NoSQL databases. From there, two models are used in such a way that requests are analyzed in order to decide which model would be ideal for processing such request. Differently, our approach keeps the structure of the original database and all data are stored in the NoSQL database. Queries arising from applications are translated at runtime to NoSQL, without requiring any manual work on the part of administrators or programmers.

To overcome the problems associated with the storage of unstructured data, [9] authors presented a framework able to integrate, in the same application, data from different sources. In this proposal, the structured data are kept in a relational database while unstructured data are stored in several files from different formats such as CSV, XML, etc. This framework has an intermediate module that is connected to the application and all queries requested is treated by it. This module checks the various data sources and consolidates the result, which is forwarded back to the application. Another concept used in this framework, which is particularly interesting for our proposed framework, is the Interceptor/Controller/Mediator protocol. The Interceptor performs a sidetracking in the information flow between the application and the data source ensuring that the application is not modified when the data source changes. In our work, we propose the complete migration of relational databases to NoSQL, maintaining only one data source, eliminating the need to decide between a source or another. Moreover, this strategy keeps the application unchanged and considering the data as being stored in a relational model. In order to achieve this goal, the main challenge is to provide a complete abstraction of the relational model for NoSQL model, since they are completely different w.r.t. their structures, and the means to use it as an abstraction layer.

3 NoSQLayer

In this section, we present the implementation details of the *NoSQLayer* framework. The framework is divided into two main modules: **Data Migration Module** and **Data Mapping Module**. The data migration module is responsible for identifying automatically all elements from the original database (e.g. tables, attributes, relationships, indexes, etc.), creation of equivalent structure using NoSQL data model, and finally, completely migrate the data. The data-mapping module consists of the persistence layer, designed to be an interface between the application and the DBMS, which monitors all SQL transactions from the application, translates these operations and redirects to the NoSQL model created in the previous module. Finally, the result of each operation is treated and transformed to the standard expected by the SQL application. The following subsections describe each of these modules.

3.1 Data Migration Module

The data migration module starts with an analysis of the relational database in order to find out which metadata is required at the conversion process. The goal is to identify all elements belonging to the database. As mentioned in Section 1, the first release of *NoSQLayer* handles relational databases implemented using MySQL DBMS. Figure 1 illustrates the operation of this module, which we detailed as follows.

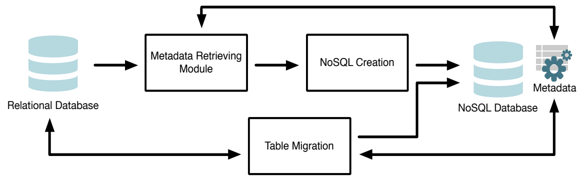


Figure 1: Data Migration Module Working Diagram

MySQL, like most DBMSs, has a data dictionary containing all the information required by the data migration module. However, the manner each DBMS access such information varies significantly. In order to make *NoSQLayer* extensible to different DBMSs, this implementation takes advantage of the *Java DatabaseMetaData* API [1], which consists of several classes and methods that facilitate the recovery of metadata. This API can be instantiated for different DBMSs, using their respective drivers. In the MySQL case, we used the MySQL Connector [2] to establish the connection. Table 1 lists the main API methods used for this stage of the module.

Method	Description
getTables()	Method that returns all tables contained in the database. This method returns a list that is then traversed in order to obtain information about each table.
getColumns()	Method that retrieves the names of all the attributes of a given table defined by the parameter, and their characteristics (name, type, etc.). Furthermore, this method also allows the identification of the primary keys.
getIndexInfo()	Method that retrieves all indexes created on the relational database.
getMetaData()	Other information of the relational database are obtained by this method, such as integrity constraints and data types.

Table 1: Description of the main methods of DatabaseMetaData API

After identifying the elements of the relational database, the next step is to create a new scheme suitable for NoSQL database. Currently, *NoSQLayer* supports MongoDB [5], which is a document-oriented NoSQL DBMS. MongoDB stores data as documents, allowing the representation of complex data structures. Furthermore, data may be distributed among different machines, increasing the system availability and performance as a whole. The construction of this new schema follows the mapping model presented in [6], which creates into MongoDB a table for each one existing in the relational database. The records of each table are retrieved and mapped as documents, where each attribute of the tables are represented by fields in these documents. Also, the relationships existing among tables are represented by document references, to facilitate the mapping module in transactions involving table joins. Finally, for each identified index, a function provided by MongoDB (*ensureIndex(index, unique)* is used to set the names of the indexes in the collections.

NoSQLayer also creates a specific collection in MongoDB to store all information collected in the previous step, such as table names, names of attributes, types of attributes and integrity constraints, which is called *Metadata*. This Metadata collection plays an important role in our framework, since the mapping module uses this collection to check whether the integrity constraints are being met, as well as the types of data used. The method of mapping operations also uses the Metadata collection to retrieve information of the attributes involved in the SQL operations that restore the response forwarded to the application.

Finally, the last step performs the data migration from a relational database to the NoSQL one. This step consists of mapping the complete requests on each table of the relational database stored in MySQL (*select * from table*) onto data insertions in the corresponding collections of MongoDB NoSQL database.

3.2 Data Mapping Module

This module provides an abstraction layer (i.e., persistence layer) between the application and the DBMS (i.e. MongoDB). The goal of this module is to allow a seamless database migration, preventing any change in the application code before changing the data model used. Therefore, application developers will continue creating queries in the relational model, but the data will be fetched in a NoSQL database, benefiting from the advantages of performance and scalability offered by this management system. Figure 2 illustrates the working operation of this module.

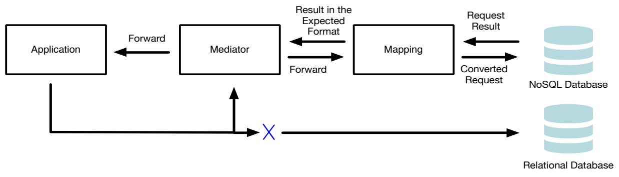


Figure 2: Data Mapping Module Working Diagram

The first task of this module is to intercept queries issued by the application to the DBMS, in order to redirect them to the suitable NoSQL DBMS. In this sense, we created a sub-module called Mediator. We built Mediator using MySQL Proxy [7], an open source tool that performs the communication between MySQL server and client application. This proxy uses the LUA language [11] for performing data manipulation. It also comes with predefined functionalities in order to intercept queries, refine the results and send signals regarding queries performed successfully or with errors. We changed the MySQL Proxy using LUA, so these operations are properly intercepted and forwarded to a second sub-module, which is responsible for query conversion. We implemented the communication between the Mediator and the request conversion sub-module using the Using LuaSoap library [4], which uses SOAP (*Simple Object Access Protocol*) protocol and XML files containing all the information related to the operations.

The conversion sub-module, called Convert, receives from Mediator all requests forwarded to the relational database and converts them to queries supported by MongoDB. This sub-module was developed as a *WebService* using Java language. It waits for new requests, which are sent by Mediator as XML files. An example of XML files used in the request communication is presented below. In this example, the first attribute of the *xmlns* represents which class of the WebService should be used (i.e., *queryInterceptor*). The second attribute represents which method should be executed (i.e., *Intercepta*). Furthermore, we highlight two required parameters: *query* and *queryType*, which represent, respectively, the intercepted request and its type.

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
        xmlns:ns1='http://queryInterceptor/'>
        <soap:Body>
                <ns1:Intercepta>
                        <query>SELECT id, name FROM users
WHERE status='A' AND (id>10 OR name='guest')
                        </query>
                        <queryType>SELECT</queryType>
                </ns1:Intercepta>
        </soap:Body>
</soap:Envelope>
```

Convert evaluates the parameter *queryType* in order to identify what should be the next procedure to be executed, according to the operation type (i.e. Select, Insert, Update or Delete). For each operation, a corresponding method is responsible for performing the operation onto the NoSQL database and returning the result of this operation. All methods perform the same steps described below, with some peculiarities related to each transaction:

Information extraction about the query: The method evaluates the *query* parameter in order to gather details about the SQL operations, such as tables, attributes, etc., in addition to the criteria used in *where* clauses. We implemented this method using Java library JSQLParser [3]. What differentiates each method in this step is the information that each one exploits. For example, while the corresponding operation to the SQL *Select* method needs to treat the involved tables, the existing *joins*, as well as functions for sorting and grouping. In the other hand, an insert operation must address only the attributes and tables involved in the operations.

Generation and implementation of equivalent operation in NoSQL: This step corresponds to translate the SQL operations onto its equivalent NoSQL ones. The translation process is based on the official MongoDB form to SQL operations mappings [6]. The collection *Metadata*, which stores the relationship of the original database with its correspondence in NoSQL database, is very important. At the end, this new operation is executed on MongoDB and the results are processed in the next step.

Mapping return results: The results returned by MongoDB is then sent to Mediator, which is responsible to forward the result to the application. First, the result header is built, which contains the correct identifications of tables, attributes, and other elements related to the result. Then, each record returned by MongoDB is mapped, following the header built. Based on this information, an XML is built and sent by Convert to Mediator. We present below an example of this XML.

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
      <S:Body>
          <ns2:interceptaResponse xmlns:ns2="http://queryInterceptor/">
                  <return>header = {'id','name'}</return>
                  <return>
                        result_set = {{'12', 'John'}, {'15', 'Steve'},
                        {'2','guest'}}
                  </return>
            </ns2:interceptaResponse>
      </S:Body>
</S:Envelope>
```

NoSQLayer also supports SQL nested transactions, such as a *Delete*, in which the items to be removed rely on a *Select*. Initially, NoSQLayer triggers the corresponding method related to the most external operating. During the parser execution, as soon as the nested transaction is identified, the corresponding method for that operation is triggered in order to completely resolve it. Then, it returns the required information to execute the external operation. Considering again the *Delete* operation, the respective method to this operation, mapped by the parser execution, identifies the *Select* operation and invokes the method responsible to deal with it. The result of the *Select* execution is then returned to the method *Delete*. Each nested operation of the corresponding method is invoked. This scheme supports even the use of recursive calls (e.g., several nested queries), in which the limitation of these calls is only subject to the hardware resources availability.

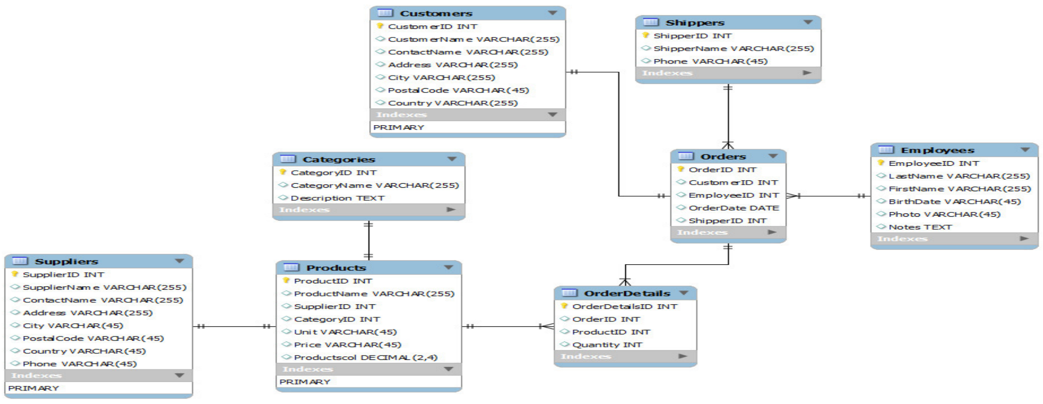
Finally, Mediator handles the XML sent by Convert. Mediator is responsible for transforming the XML into the MySQL format expected by the *MySQL Resource*. In order to perform this, Mediator relies on the support of MySQLProxy. The source code of NoSQLayer is available at https://www.github.com/nosqlayer/code.

4 Evaluation

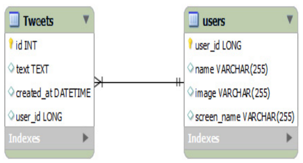
In this section, we present an assessment of NoSQLayer focusing on two types of analysis: qualitative and quantitative. While in the qualitative evaluation our goal is to present a proof of concept by showing the NoSQLayer execution in practice, in the quantitative one we aim to verify whether the use of NoSQL, with our framework, leverages the system performance.

4.1 Experimental Setup

In our analysis, we used two relational databases, as presented in Figure 3. The first one comes from W3Schools (http://www.w3schools.com/), a well-known web developer information website. Although presenting simple structure, this database may capture characteristics quite different from those typically used in most of the applications, such as integrity constraints, relationships or various types of data. The second database has a simple database model regarding an application that collects and stores user posts from Twitter. This database has a high volume of stored data that may help us, mainly, in the quantitative assessment of NoSQLayer.



(a) W3schools



(b) Twitter

Figure 3: **Database models used in the evaluation.**

Both databases were implemented using MySQL. For each database, an application, written in Java, was created with the purpose of executing several SQL operations. We call this configuration *MySQL Scenario*. For each arrangement application/database, we created another version that used NoSQLayer to perform the entire migration of the MySQL database to MongoDB, including the structure and the data themselves. For this latter configuration, which we call *NoSQLayer Scenario*, the NoSQLayer was also used to capture and map the operations requested by the applications and MongoDB. Thus, we performed quantitative and qualitative comparisons of the results achieved by each scenario. Although several SQL operations were tested during the development process, due the lack of space, in this paper we focus only on the operations presented in Table 2. These operations used various operators such as set operators, MAX functions, ORDER BY, different Joins, among others.

Table 2: Operations performed in the evaluations.

Identifier	Dataset	Operation
Select1	Twitter App.	SELECT created_at,user_id FROM collections WHERE user_id IN (831178813,14152035,341925705)
Select2	Twitter App.	SELECT MAX(user_id) AS HighestID, MIN(user_id) AS SmallestUserID FROM collections
Select3	Twitter App.	SELECT COUNT(*) AS NumberOfTweets FROM collections
Select4	W3Schools App.	SELECT Customers.CustomerName, Orders.OrderID FROM Customers LEFT JOIN Orders ON Customers.CustomerID=Orders.CustomerID
Select5	W3Schools App.	SELECT * FROM Customers WHERE Country='Germany' AND (City='Berlin' OR City='München')
Select6	W3Schools App.	SELECT * FROM Customers ORDER BY Country DESC
Select7	W3Schools App.	SELECT * FROM Employees WHERE FirstName LIKE '%a'
Select8	W3Schools App.	SELECT Orders.OrderID, Employees.FirstName FROM Orders RIGHT JOIN Employees ON Orders.EmployeeID=Employees.EmployeeID
Delete1	W3Schools App.	DELETE FROM Categories WHERE CategoryID >= 1 AND CategoryID <= end_value
Delete2	W3Schools App.	DELETE FROM Customers WHERE CustomerID >=1 AND CustomerID <=end_value
Insert1	W3Schools App.	INSERT INTO OrderDetails
Insert2	W3Schools App.	INSERT INTO Customers
Update1	W3Schools App.	UPDATE Categories SET CategoryName='NewName', Description='NewDescrip' WHERE CategoryID >= 1 AND CategoryID <= end_value
Update2	W3Schools App.	UPDATE Customers SET CustomerName='NewName', ContactName='NewContact', Address='NewAdd.', City='NewCity', PostalCode='NewPostal', Country='NewCountry' WHERE CustomerID >=1 AND CustomerID <= end_value

4.2 Qualitative Evaluation

The purpose of this analysis is to assess whether the use of NoSQLayer affect the functionality of the applications w.r.t. performing various SQL operations. In our experiments, we consider two scenarios, which are listed below. In order to accomplish a complete assessment, we vary the amount of records that were selected or affected by the operations.

Scenario 1: Query operations: For every operation performed, the results were stored in plain text files. Then, an application that identifies differences between files was executed and the result was stored in another file. For all queries shown in Table 2, the resulting files with the differences were empty, meaning that no anomaly had occurred and, consequently, NoSQLayer had worked properly.

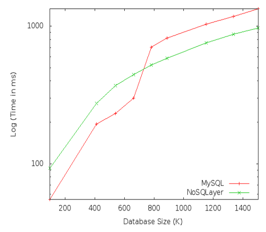
Scenario 2: Operations that change the database state: For each execution of an operation that changes the database state, we conducted a dump of the tables (or collections in the case of MongoDB) and stored the results in text files, named according to the operation performed. We also evaluated differences in this result files, and again, no difference was found, thereby demonstrating the proper functioning of NoSQLayer.

4.3 Quantitative Evaluation

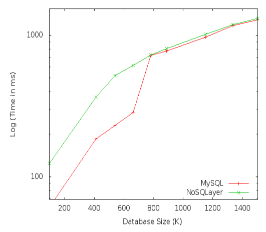
In this evaluation, we checked the efficiency of the proposal. The focus is to assess whether, despite the overhead of defining a layer between the application and NoSQL database, the use of NoSQLayer is still an advantageous option in terms of performance compared to the RDBMS. We conducted a set of experiments to measure the runtime of several SQL transactions, considering each of the aforementioned scenarios.

The SQL operations considered were those presented in Table 2. We vary the amount of data processed by each operation. For query operations, before each query, we changed the total of records in the database that should be returned. In order to enable this setup, before each query we perform the insertion of the data that should be retrieved. We used this same strategy for the Update and Delete operations, in this case, changing the amount of data affected by these operations. For the insert operation, we varied the amount of data that was sent for insertion.

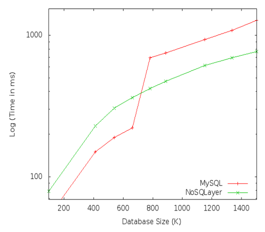
In all these operations, the execution time was measured from the instant that the application performed the request up to the instant the results were sent back to the application. These measurements were performed considering both MySQL Scenario (the time was measured from the instant of the application request until the response from MySQL) and the NoSQLayer Scenario (the time was measured until the response from NoSQLayer). We performed each operation 10 times and the final time was given according to the average of these 10 executions. All experiments were performed using a computer equipped with an Intel Core i3 2.53GHz, 4GB RAM, 500GB hard processor system, using operating system Ubuntu 04.13 64-bit. We present these results in Figure 4. For all plots, we adopt a log scale in y-axis (execution time) aiming at a better visualization. Due to lack of space, we present the results related to four Select operations and one for each changing operations.



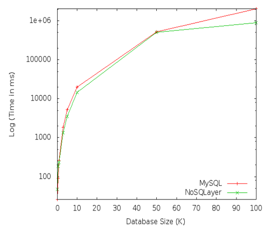
(a) Select1



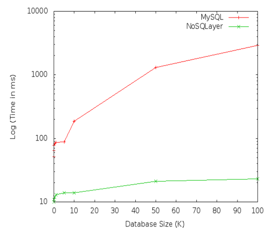
(b) Select2



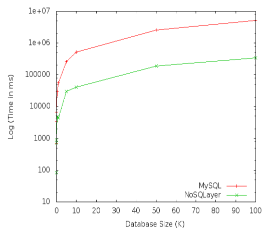
(c) Select3



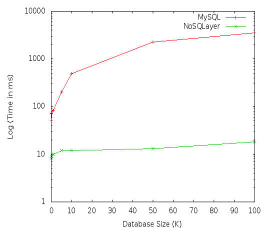
(d) Select4



(e) Delete 1



(f) Insert 1



(g) Update 1

Figure 4: **Execution time of different operations**

Observing the results related to the Select operations, MySQL has a lower response time for lower data volume, compared to NoSQLayer. As the total number of records to be retrieved increases, the response time also increases. We also observe a peak in the response time of MySQL. We explain this behavior by an excessive use of the CPU to deal with large amount of data. In this case, MySQL starts using a new processing unit (i.e. new CPU core), which in turn, must address issues related to synchronization among distinct CPUs, increasing substantially the response time. This result shows that, even with the overhead generated by the database abstraction layer, NoSQLayer is more efficient than a relational database as the data volume grows. Considering the distributed processing features of MongoDB, not explored in this study, these overtakes might be even more significant. Finally, analyzing the plots related to changing the database state, all of them have the runtime in NoSQLayer much lower than the execution in MySQL. The main reason for this behavior may be the absence of integrity constraints of NoSQL databases. An improvement in NoSQLayer would be to treat and internally verify these integrity constraints related to change of state of the database process. Currently, through the collection *Metadata*, NoSQLayer supports only primary key constraint.

5 Conclusions

In this paper we present NoSQLayer, a framework to support developers at migrating automatically from relational databases (MySQL) to NoSQL ones (MongoDB) while preserving the semantics of the original database. NoSQLayer has a database persistence layer that allows applications to access data in NoSQL model seamlessly, without the need of modifying queries and application code. NoSQLayer was implemented in Java, employing concepts of object-oriented programming, in order to make it easily extensible to other DBMS.

We evaluate our proposal by qualitative and quantitative perspectives. The qualitative assessment compared results of various operations applied directly to SQL and MySQL to MongoDB using our framework. In all evaluated queries the results were identical, showing the effectiveness of our proposal. The quantitative assessment compared the runtime of different queries, varying the total number of records involved in the operations. Despite the overhead generated by the persistence layer, which makes the framework a less efficient for small volumes of data, as the volume of data grew, our framework showed to be more efficient than using only MySQL. These experiments showed that NoSQLayer is a solution suitable to handle large volume of data. As future work, we aim to extend NoSQLayer to other management systems and technologies.

References

[1] Databasemetadata. http://docs.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html. Accessed:2014-09-24.

[2] Jdbc. http://dev.mysql.com/doc/refman/5.6/en/connector-j-reference.html. Accessed:2014-09-24.

[3] Jsql parser. http://jsqlparser.sourceforge.net/. Accessed:2014-09-24.

[4] Luasoap. http://tomasguisasola.github.io/luasoap/. Accessed:2014-09-24.

[5] Mongodb. http://docs.mongodb.org/manual. Accessed:2014-09-24.

[6] Mongodb mapping chart. http://docs.mongodb.org/manual/reference/sql-comparison. Accessed:2014-09-24.

[7] Mysql proxy. http://dev.mysql.com/refman/5.6/en/mysql-proxy.html. Accessed:2014-09-24.

[8] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[9] R. GAIOSO, F. LUCENA, and J. SILVA. Integrate: Infra-estrutura para integração de fontes de dados heterogêneas. *Master Thesis, Federal University of Goiás. Informatic Institute*, 2007.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua—an: Extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652, June 1996.

[12] R. P. Padhy, M. R. Patra, and S. C. Satapathy. Rdbms to nosql: Reviewing some next-generation non-relational database's. *International Journal of Advanced Engineering Science and Technologies*, 11(1), September 2011.

[13] Jaroslav Pokorny. Nosql databases: A step to database scalability in web environment. In *Proc. of the 13th iiWAS*, pages 278–283, New York, NY, USA, 2011. ACM.

[14] John Roijackers. Bridging sql and nosql. *Master Thesis, Eindhoven University of Technology. Department of Mathematics and Computer Science*, 2012.

[15] Aaron Schram and Kenneth M. Anderson. Mysql to nosql: Data modeling challenges in supporting scalability. In *ACM SPLASH '12*, pages 191–202, 2012.