Integrating Two Worlds: Relational and NoSQL

Dražena Gašpar, Mirela Mabić
Faculty of Economics, University of Mostar
Matice hrvatske bb, 88 000 Mostar, BiH
{drazena.gaspar, mirela.mabic}@gmail.com

Tihomir Krtalić
HERA d.o.o.
Kralja Petra Krešimira IV bb, 88000 Mostar, BiH
tihomir.krtalic@hera.ba

Abstract.*NoSQL databases emerged as a response to the Big Data challenges generated because of extensive use of Internet and Web applications. However, not all problems are the Big Data problems. Transaction-oriented problems still can be better-solved using relational databases, while NoSQL databases are better solution for the Big Data demands. Because the users want to analyze this data together, the integration of relational and NoSQL databases becomes necessity.*
The authors discuss two approaches to data integration between relational and NoSQL databases: native and hybrid solutions explained on the example of integration transactional data from Oracle databases with data stored in MongoDB.

Keywords. NoSQL, relational databases, native solution, hybrid solution, Oracle, MongoDB.

1 Introduction

Over the last ten years, the challenges related to the Big Data phenomenon have been the main driving forces behind NoSQL database development. There are many different and unclear definitions of the Big Data term. Usually, the definitions are related to domains and use different numbers of Vs to explain the Big Data. Firstly, the Vs refer to data volume, velocity and variety (Lanely, 2001). In 2016 IBM added veracity, while Microsoft added variability and visibility. However, these definitions are focused on data itself, not on the using the data as a tool for resolving business problems. That is the reason why Wu, Buyya, and Ramamohanarao (2016) in their definition of the Big Data started from the business view and use three aspects: data domain (searching for patterns), business intelligence domain (making predictions) and statistical domain (making assumptions). They understood that the purpose of data is to gain hindsight (i.e., metadata patterns from historical data), insight (i.e., a deep understanding of issues or problems), and foresight (i.e., more accurate predictions in the near future) in a cost-effective manner (Wu, Buyya, & Ramamohanarao, 2016). The development and practical implementation of both, NoSQL and relational databases, is showing that they were right. Today, after more than 10 years of development and use of NoSQL databases, it is clear that neither everything is good in NoSQL nor everything is so bad in relational databases. Not all problems are the Big Data problems, so transaction-oriented problems still can be better solved by using relational databases, while NoSQL databases are better in resolving the Big Data demands. But, the real problem is the fact that when business perspective of data is in questions, there is no always clear boundary between transactional and the Big Data requirements. Namely, database users are not interested in technical details about their data (about type of database, storage methods, database server, etc.), they just want to analyse their data together, regardless of whether the data are stored in relational or NoSQL databases. Business users are looking for patterns in their data, they need better insight of data in order to make better decisions and ensure further development and survival of their organizations in globalized, dynamic and turbulent world. Users' requirements related to data analysis through business intelligence (BI)tools have forced vendors of both, relational and NoSQL databases, to look for solutions that will enable them to integrate data stored in relational with data stored in NoSQL databases. Most of NoSQL vendors have recognized the necessity to provide support for Structured Query Language - SQL (Hive Query Language – HQL, Cassandra Query Language – CQL, Cypher, etc.) in order to attract more experienced developers and users, to enhance usability and programmer efficiency, and to provide easier integration with BI and analytical tools. On the other hand, most of relational databases vendors offer support for JSON (JavaScript Object Notation). The JSON is primarily used as data interchange format between a server and web application. It enables storing of JavaScript objects as text, avoiding complicated parsing and translations (JSON, 2017). When applications use JSON data stored in NoSQL database, it is their responsibility to ensure data integrity, but native support for JSON by relational databases means that these databases provide use of JSON together will all of the benefits of relational databases (transactions, indexing, declarative queries, etc.).

In this paper, the authors discuss two approaches to data integration between relational and NoSQL databases: native and hybrid solutions. These solutions are explained on the example of integration transactional data from Oracle databases with data stored in MongoDB.

2 Relational and NoSQL databases integration

The integration of relational and NoSQL databases is one of the possible solutions that can ensure that users get the best of the both worlds, high scalability and availability of NoSQL databases and transactions, integrity and standardization (SQL) of relational databases. During last few years different approaches, all with the same goal to ensure data integration and/or migration from relational and NoSQL databases, have arisen. Developing of Bridge Query Language (BQL) as a tool for interface standardization (Curé, Hecht, Le Duc & Lamolle, 2011) was one of the approaches. Atzeni, Bugiotti, and Rossi (2012) proposed the *"SOS (Save Our Systems) platform"* as a common API for interface standardization. Some authors (Rocha et al., 2015; Stanescu, Brezovan & Burdescu, 2016, Nikam et al., 2016) proposed frameworks for mapping relational data to NoSQL databases, while others (Liao et al., 2016) developed data adapters for querying and transformation between relational and NoSQL databases. The different approaches related to relational and NoSQL databases integration can be summarized as:

1. Native solution,
2. Hybrid solution.

A native solution is based on the standard database drivers and ways how the business layer communicates with a specific database. Since data is stored in both, relational and NoSQL, databases in order to use it together it is necessary to integrate it somehow. This integration is implemented on the business layer. When data is extracted from the databases (relational and NoSQL), it is linked and converted into format adequate for its use on the user (business) layer. If data is storing in a database (either relational or NoSQL), it is the responsibility of the business layer to prepare it to storage and to store it in the specific (relational or NoSQL) database.

A hybrid solution is based on development of additional layer that enables SQL communication between business and data layer. It means that developers can use familiar SQL patterns on the business layer, but they must employ new layer for translating these patterns into the NoSQL programming interface in order to enable communication with the NoSQL database.

It is easier to explain and understand these two solutions if concrete examples are presented. In preparing the examples the authors started with requirement for integration of data about university's teaching staff stored in relational (Oracle) database with data about their publications (papers, chapters, books, etc.) stored in NoSQL database (MongoDB). Namely, in this example, data about teaching staff is part of university's information system based on transactional (relational) database, in this case Oracle database. Storing data about teachers' publications is new application, developed lately, and it requires storing a lot of pdf documents and images. That is the reason why MongoDB is used for storing this data. Namely, MongoDB performs extremely well in working with such types of data. MongoDB uses GridFS based on two collections within the database: one for storing metadata about the file, and another for the file itself where the file is broken down into small pieces called chunks. Thanks to this file system organization, neither the number of files nor their size is limited. Also, it is easier to access specific parts of large files by extracting respective chunks, which makes MongoDB very fast in storing and extracting data of any type. Figure 1 shows user's requirement and how data used in the examples is stored.

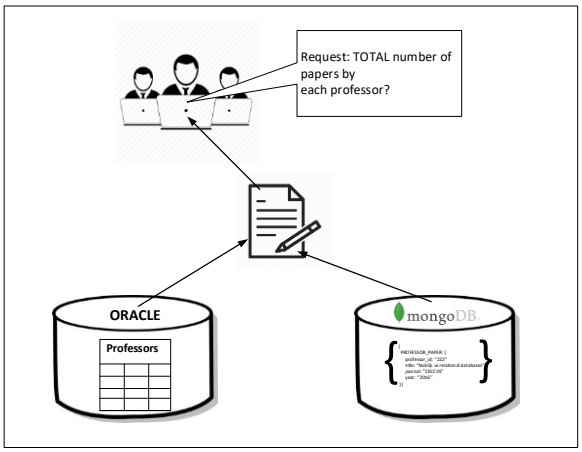

Figure 1. User's requirement and data storage

2.1 Native solution

The native solution for the requirement presented in Figure 1 involved the use of native drivers for communication with Oracle and MongoDB. In order to answer on question how many papers was published by each teacher, it is necessary to integrate data about teaching staff from Oracle database with data about publications from MongoDB. In native solution this integration is implemented on the business layer. In the presented example, REpresentational State Ttransfer(REST) Web services, the Java programming language and the Java EE 7 framework were used. Although it is common to use JPA (Java Persistence API) for work with relational data when using this technology, in this case that was not possible because JPA lack the functionality of working with multiple data sources.

Because of that, access to Oracle database was ensured through the JDBC (Java DatabaseConnectivity) driver, while the Java MongoDB driverwas used to access the MongoDB database. Since it is not possible to map the results of queries from the MongoDB database with Java objects directly, the authors decided to use the open-source library Gson. Gson Java library performs conversion in two directions: converting Java objects into their JSON representation or converting JSON string to an equivalent Java object.

In presented example, relational table of professors (teaching staff) and MongoDB collections of publications need to be linkedin order to get the required data. A native driver was used as the connector for an individual database. Therefore, the use of native drivers eliminates the possibility to make queries that would be performed on Oracle and MongoDB databases at the same time. The problem of linking databases and extracting the data was solved through the following steps:

- Data about teaching staff (professors) were stored in relational database (Oracle) and for access was used Oracle JDBC driver. The method *List<Professor>getProfessorList()*was used for retrieving data about teaching staff (Figure 2).
- Java MongoDB driver was used for access to MongoDB where publications were stored. All publications were retrieved by the method *List<ProfessorPaper>getProfessorPaperList()* (Figure 3).
- The method *ProfessorPaper getProfessor PaperById(String id)* returns all publications by professor (for a submitted professor's id) (Figure 4).
- Upon retrieval, the data are linked by a key (id).
- Linking results in an object of the class *ReportModel* (Figure 5). It contains data about professors and the total number of publications by each professor while the result is a Java object that is passed to the report engine that uses it to generate the expected report.

```
public List<Professor> getProfessorList()throws BusinessException, SQLException {
    Connection conn =null;
    PreparedStatement preparedStatement =null;
    String selectSQL ="SELECT ID,TITLE, FIRST_NAME, LAST_NAME, UNIVERSITY_ID,
FACULTY_ID FROM PROFESSOR";
    List<Professor> professorList =new ArrayList();
try{
      conn = dataSource.getConnection();
      preparedStatement = conn.prepareStatement(selectSQL);
      ResultSet rs = preparedStatement.executeQuery();
while(rs.next()){
        Professor professor =null;
        professor =new Professor();
        professor.setId(rs.getBigDecimal(1));
        professor.setTitle(rs.getString(2));
        professor.setFirstName(rs.getString(3));
       professor.setLastName(rs.getString(4));
        University university = universityService.getUniversityById(rs.getBigDecimal(5));
        BigDecimal fac = rs.getBigDecimal(6);
        Faculty faculty = facultyService.getFacultyById(fac);
        professor.setUniversity(university);
        professor.setFaculty(faculty);
        professorList.add(professor);}
}catch(SQLException ex){
thrownew BusinessException("Error fetching professor!");
}finally{
if(conn !=null){
        conn.close();
}
if(preparedStatement !=null){
        preparedStatement.close();}}
return professorList;}
```

Figure 2. The method for retrieving data about teaching staff

```
public List<ProfessorPaper> getProfessorPaperList(){
      MongoCollection collection = mongoClientProvider.getMongoCollection("professorPaper");
      MongoCursor<Document> cursor = collection.find().iterator();

if(cursor !=null){
         String cursorString = util.cursorToString(cursor);
         GsonBuilder gsonBuilder =new GsonBuilder();
         Gson gson = gsonBuilder.create();

         List<ProfessorPaper> list = gson.fromJson(cursorString,new TypeToken<List<ProfessorPaper>>(){
}.getType());
return list;
}else{
returnnull;}}
```

Figure 3. The method for retrieving all publications by professor

```
public ProfessorPaper getProfessorPaperById(String id){
      MongoCollection collection = mongoClientProvider.getMongoCollection("professorPaper");
      Bson filter = eq("_id", id);
      Document document =(Document) collection.find(filter).first();

if(document !=null){
         String json = document.toJson();
         Gson gson =new Gson();
         ProfessorPaper ecoTest = gson.fromJson(json,new TypeToken<ProfessorPaper>(){
}.getType());

return ecoTest;
}else{
returnnull;}}
```

Figure 4. The method for returning the total number of publications by professor

```
public List<ReportModel> getNumberOfPaperByProfessor()throws BusinessException, SQLException {
      List<Professor> professorList;
      professorList=professorService.getProfessorList();
      List<ReportModel> reportList =new ArrayList();

for(Professor professor : professorList){
         Long count = professorPaperMongoService.getCountByProfessorId(professor.getId());
         reportList.add(new ReportModel(professor, count));
}
return reportList;}
```

Figure 5. The class ReportModel

2.2 Hybrid solution

A hybrid solution involves development of additional layer that enables SQL communication between business and data layer. Namely, work with data stored in relational databases is based on the unique standard of the SQL. SQL implementations vary slightly among relational database vendors. Acceptance of the standard by different relational vendors has allowed integration of various databases, and brought the developer community to a situation that it is almost the immaterial which relational database product they use. As opposite, development of the NoSQL market has brought a huge number of solutions that offer various data organization models and data management APIs. This diversity of programming languages and interfaces for work with data in the NoSQL databases makes the huge challenge in finding appropriate solution for integration relational and NoSQL databases. That is the reason why the SQL has been seen as powerful

solution for the problem of integration relational and NoSQL world (Lawrence, 2014).

In their paper, Vilaça et al. (2012) present the DQE (Distributed Query Engine) as a platform to execute SQL queries on NoSQL databases, while keeping their data-model scalability and flexibility. DQE represents a strong concept that will combine said scalability and flexibility of NoSQL schemas with the expressiveness of SQL. Further, they show how simple key-value operations and the associated data model can be mapped into SQL structures, and describe a full implementation of the model on the HBase database using Apache Derby's query engine.

Lawrence (2014) considered DQE-like solutions to connect the relational and NoSQL worlds, and found a solution in SQL, which provided a theoretical foundation for the system called Unity. Unity is a generalizable integration and virtualization system based on SQL interface. According to Lawrence (2014), the key features of the Unity are:

- It uses a SQL query processor and optimizer that include support for push-down filters and cross-source hash joins.
- It has a SQL dialect translator that brings different SQL dialects of relational-database vendors to the same form.
- It has a SQL-form to NoSQL APIs translator, and operators not supported by native NoSQL database APIs are performed using the Unity virtualization engine.
- It offers mapping of SQL functions into appropriate forms supported by various vendors of relational and NoSQL databases.
- It provides data virtualization allowing queries and joins across relational and NoSQL data sources.

Unity offers compatibility with existing SQL solutions and new ones being developed by the developer community, based on the SQL standard. At the same time, it offers the possibility of using the appropriate dataorganization philosophy that meets the needs and requirements that applications bring to developer teams. Unity offers an answer to growing needs of web applications for simultaneous use of NoSQL and relational systems.

Unity also offers a kind of extension of the NoSQL system, by implementing unsupported operators through its virtualization engine. It seems that this architecture allows use of the best of the two philosophies, using the interface most appropriate for most of the developer community.

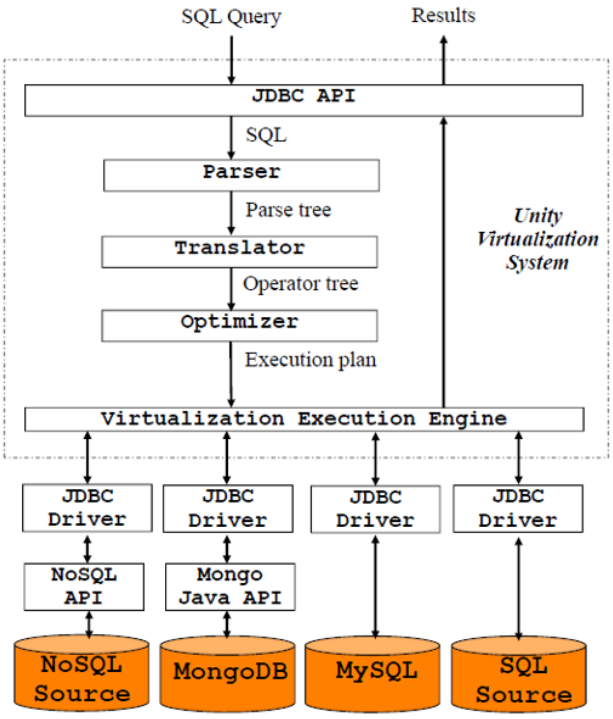

Figure 6. Unity architecture (Lawrence, 2014)

Presently, Unity is available as a commercial product on the market of JDBC drivers. It supports Oracle, MySQL, SQL Server, and any other relational JDBC data source. The main features of the MongoDB drivers of Unity are (UNITYJDBC, 2017):

- Access to MongoDB collections using SQL including WHERE filters and ORDER BY clause.

- Data manipulation using standard SQL functions that are not natively supported in the MongoDB.
- Use of JOIN clause between MongoDB collections and relational tables of databases that are supported by Unity.
- Full support for nested documents andstrings including functionalities of filters and regular expressions (RegEx).

All that lead to decision that the Unity model should be used in testing hybrid solution. The same example, making the report that shows professors with total number of paper by each professor, was used for presenting hybrid solution based on Unity model. Figure 7shows the example of method that retrieves data from Oralce and MongoDB for the required report.

```
public List<ReportModel> getReport()throws UnsupportedEncodingException {
      List<ReportModel> reportModelList =new ArrayList();
      Connection con =null;
      Statement stmt =null;
      ResultSet rst;
      String url ="jdbc:unity://"+ getWebInfPath()+"/OracleMongoSources.xml";
try{
        Class.forName("unity.jdbc.UnityDriver");
        con = DriverManager.getConnection(url);
        stmt = con.createStatement();
        String sql ="  SELECT PR.ID,\n"
+"       PR.TITLE,\n"
+"       PR.FIRST_NAME,\n"
+"       PR.LAST_NAME,\n"
+"       PR.UNIVERSITY_ID,\n"
+"       PR.FACULTY_ID,\n"
+"       COUNT (1)\n"
+"   FROM Mongo.PROFESSOR_PAPER PP, Oracle.PROFESSOR PR\n"
+"  WHERE PP.doc.professor.id = PR.id\n"
+"GROUP BY PR.ID,\n"
+"       TITLE,\n"
+"       PR.FIRST_NAME,\n"
+"       PR.LAST_NAME,\n”
+"       PR.UNIVERSITY_ID,\n"
+"       PR.FACULTY_ID";
        rst = stmt.executeQuery(sql);
while(rst.next()){
           ReportModel report =new ReportModel();
           Professor professor =new Professor();
           professor.setId(rst.getBigDecimal(1));
           professor.setTitle(rst.getString(2));
           professor.setFirstName(rst.getString(3));
           professor.setLastName(rst.getString(4));
           report.setValue(rst.getLong(5));
           report.setProfessor(professor);
           reportModelList.add(report);}
}catch(Exception ex){
        System.out.println("Exception: "+ ex);
}finally{
if(con !=null){
try{
              con.close();
}catch(SQLException ex){ System.out.println("SQLException: "+ ex);}}}
return reportModelList;}
```

Figure 7. The method that retrieves data from Oracle and MongoDB

The query is executed on two data sources: Mongo.Professor_Paper and Oracle.Professor. These sources are described in the UnityJDBC configuration file OracleMongoSources.xml. The WHERE clause of a SQL query reveals the possibility of linking data from different sources. The query result is a Java object that can be forwarded to the report engine for the formatting and presentation of the retrieved data.

3 Conclusion

From the viewpoint of users, the real value of data, stored either in relational or NoSQL databases, lays in the possibility to use it for better understanding of their businesses, customers and suppliers. In globalized, volatile and dynamic world, data is stored everywhere, in transactional systems, social networks, web sites, and users do not want to be limited in data analysis to the only some data storages. Overall data analysis can result with discovering of knowledge hidden in huge amounts of data and empower the users and their organizations to efficiently respond to present and future business challenges. In that sense, the users see the database technology as a powerful tool that has the task to provide access and use of data wherever is stored. Today, it necessary leads to integration of data stored in different databases, relational and NoSQL.

The paper presents two approaches to integration relational and NoSQL databases: native and hybrid solution. The use of native drivers eliminates the possibility to make queries that canbe performed on both, relational and NoSQL database at the same time, so the programmers are responsible for developing the code that will enable such queries. A hybrid solution means development of additional layer that enables to developers to use familiar SQL on the business layer. In this case, the query is executed on two data sources, relational and NoSQL, but the additional layer is used to enable data integration.

Theseapproachesare also generating additional challenges, including the following:

- Increased responsibility of developers in the optimization of database processes, which means that additional programming and time are necessary for completing the integration of data.
- Difficulties in finding developers with good knowledge of NoSQL databases, while finding developers with knowledge about both worlds, relational and NoSQL, is still equal to a jackpot.
- Finding an appropriate driver or tool for a particular problem is a time-consuming and exhaustive process because the market is full of many different drivers and tools offered by third vendors.

The need for the integration of relational and NoSQL databases can lead to databases evolution in sense that they will provide support for different and often opposite users requirements, by enabling combinations of both approach and tunable and configurable capabilities that will give the users the possibility to use databases on the way that best suits their needs.

References

Atzeni, P., Bugiotti, F., Rossi, L. (2012). Uniform access to non-relational database systems: the SOS platform, 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29. Retrieved from http://www.inf.uniroma3.it/~atzeni/psfiles/CAiSE2012Atzeni.pdf.

Curé, O., Hecht R., Le Duc C., Lamolle M. (2011). Data Integration over NoSQL Stores Using Access Path Based Mappings. DEXA, August, 481-495, Lecture Notes in Computer Science

JSON (2017). Introducing JSON. Retrieved from http://www.json.org/

Laney, D. (2001). 3D Data Management: Controlling Data Volume, Velocity, and Variety. META Group. Retrieved from http://blogs.gartner.com/douglaney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

Lawrence, R. (2014).Integration and Virtualization of Relational SQL and NoSQL Systems including MySQL and MongoDB, International Conference on Computational Science and Computational Intelligence. 285-290.

Liao, Y., Zhou, J., Lu, C-H., Chen, S-H., Hsu, C-H., Chen, W., Jiang, M-F., Chung, Y-C. (2016). Data adapter for querying and transformation between SQL and NoSQL database. *Future Generation Computer Systems 65*, 111–121.

Nikam, P., Patil, T., Hungund, G., Pagar, A., Talegaonkar, A., Pawar, S. (2016). Migrate and Map: A Framework to Access Data from Mysql, Mongodb or Hbase Using Mysql Queries. *IOSR Journal of Computer Engineering (IOSR-JCE).* Volume 18, Issue 3, Ver. IV (May-Jun), 13-17.

Rocha,L., Vale, F., Cirilo, E., Barbosa, D., Mourao, F. (2015). A Framework for Migrating Relational Datasets to NoSQL. *ICCS 2015 International Conference On Computational Science*. *Procedia Computer Science*. Volume 51, 2593–2602.

Stanescu, L., Brezovan, M., Burdescu, D.D. (2016). Automatic Mapping of MySQL Databases to NoSQL MongoDB. *Proceedings of the Federated Conference on Computer Science and Information Systems*. ACSIS, Vol. 8. 837–840.

UNITYJDBC (2017). JDBC Driver for MongoDB. Retrieved from http://www.unityjdbc.com/mongojdbc/mongo_jdbc.php

Vilaça R., Cruz F., Pereira J., Oliveira R. (2013). An effective scalable SQL engine for NoSQL databases. HASLab - High-Assurance Software Laboratory. INESC TEC and Universidade do Minho Braga, Portugal. doi: 10.1007/978-3-642-38541-4_12

Wu, C., Buyya, R., Ramamohanarao, K. (2016). BDA=ML +CC. In book ed. Buyya,R., Calheiros, R.N.,Dastjerdi, A.V. *Big Data Principles and Paradigms*. Elsevier Inc. USA.