NoSQL and SQL Databases for Mobile Applications. Case Study: MongoDB versus PostgreSQL

Marin FOTACHE, Dragos COGEAN
Al. I. Cuza University of Iasi, Romania
fotache@uaic.ro, dragos.cogean@gmail.com

Compared with "classical" web, multi-tier applications, mobile applications have common and specific requirements concerning data persistence and processing. In mobile apps, database features can be distinctly analyzed for the client (minimalistic, isolated, memory-only) and the server (data rich, centralized, distributed, synchronized and disk-based) layers. Currently, a few lite relational database products reign the persistence for client platforms of mobile applications. There are two main objectives of this paper. First is to investigate storage options for major mobile platforms. Second is to point out some major differences between SQL and NoSQL datastores in terms of deployment, data model, schema design, data definition and manipulation. As NoSQL movement lacks standardization, from NoSQL products family MongoDB was chosen as reference, due to its strengths and popularity among developers. PostgreSQL serves the position of SQL DBMSs representative due to its popularity and conformity with SQL standards.
Keywords: *Mobile Applications, NoSQL, SQL, PostgreSQL, MongoDB*

1 Introduction

"Mobile world" is one of the most dynamic areas of Information Technology today. Smartphones and tablets explosion have created a huge market for mobile applications. Consequently there is an increasing demand for mobile application developers so that, even when taking into account just the case of Romania, many books [1] [2] [3] and articles (e.g. [4], [5], [6], etc.) have been published. Also many Business Information Systems/Business Informatics undergraduate and master programs introduced in their curriculum courses related to mobile devices and applications (e.g. http://dice.ase.ro/?page_id=31, http://ism.ase.ro/ curriculum_2012_2014.jsp, http://www.feaa.uaic.ro/ro/main/page/335, http://www.econ.ubb cluj.ro/planuri_invatamant/master/planuri_master_2012_2013/E-Business.pdf).

Almost all of the mobile applications require a persistent data layer, including options for queries. So the interest of database professionals, academics and researchers for mobile technologies is increasing. Mahmoud et al. [7] even suggest including mobile storage topics, devices and applications into database courses.

This paper presents technologies and tools for deployment of the data layer in mobile applications on some major platform focusing on a NoSQL datastore, MongoDB. Section 2 deals with some specific requirements of mobile software in terms of storage and databases. Section 3 deepens the discussion of data storage features for major mobile platforms. The arguments for a NoSQL solution, including a basic description of mobile cloud solution implemented in MongoDB are subjects of section 4. Sections 5, 6 and 7 analyse some of the main differences in terms of database schema, Data Definition Language (DDL), Data Manipulation Language (DML) and database query between SQL (PostgreSQL dialect) and MongoDB.

2 Common Issues of Mobile Applications Storage Layer

Mobile applications share many features of "classical" client/server or web multi-layered applications architecture. Figure 1 depicts a classical six-layered (Security layer covers all the other five layers) framework [8] for enterprise mobile application development.

DOI: 10.12948/issn14531305/17.2.2013.04

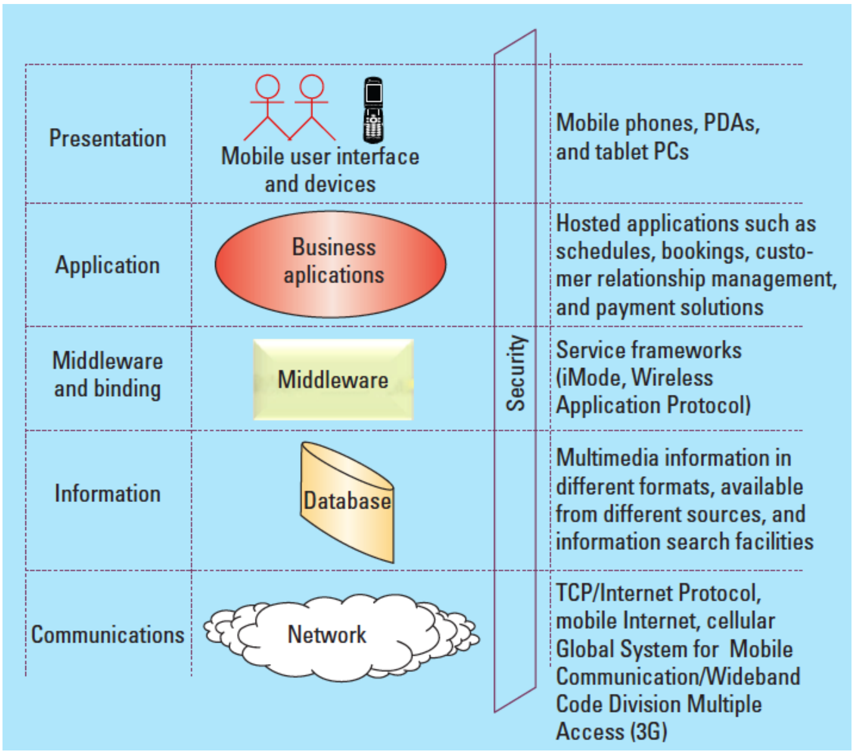
Presentation
Mobile user interface and devices
Mobile phones, PDAs, and tablet PCs
Application
Business aplications
Hosted applications such as schedules, bookings, customer relationship management, and payment solutions
Middleware and binding
Middleware
Service frameworks (iMode, Wireless Application Protocol)
Information
Database
Multimedia information in different formats, available from different sources, and information search facilities
Communications
Network
TCP/Internet Protocol, mobile Internet, cellular Global System for Mobile Communication/Wideband Code Division Multiple Access (3G)
Security

Fig. 1. Layers of the enterprise Mobile Applications Development Framework [8]

Persistence layer have similar requirements in terms of storage, DDL, DML and database query. Also the data accessibility mechanisms from application layers are similar to other web applications.
The most striking difference between mobile and web application databases concerns is that while server persistence layer is similar, the client platforms of mobile apps (running mainly on smartphones and tablets) have a smaller scale of available resources (memory, processing power). Many mobile applications require replication and synchronization mechanisms for data persistent on smartphone with a centralized large scaled database available on a server [9] [10]. Data storage for client must be processed without hard disk techniques [11] [12].
Nori [13] identifies 11 characteristics that have to be provided, partially or fully, by the mobile and embedded DBMSs:

a. Embeddable in applications – sometimes requiring no administration.
b. Small footprint – in order to be downloadable in broader range of mobile devices
c. Run on mobile devices and operates in conditions of small amount of processor power, RAM and permanent memory.
d. Componentized DBMS – supports from all DBMS functions only the ones required by the specific application.
e. Self-managed DBMS, with no hope for the user to be a DBA who can ne able to restore a crashed database.
f. In-memory DBMS – requiring special query processing and indexing technics which are optimized for main memory usage.
g. Portable databases with very simple deployment.
h. No code in the database that protects against viruses and malware.
i. Synchronize with back-end data sources.
j. Remote management – especially in the case of enterprise-wide applications.
k. Custom programming interfaces for specialized data-centric applications.

Mahmoud et al. [7] point out the most common software options for storage in mobile applications:

- HTML5 (*localStorage* API which stores objects as key-value pairs and *IndexDB* which implements relational technology);

- SQLLite – a over-simplified relational database server;
- Cloud storage (Apple iCloud, Dropbox, Google Drive, etc.);
- Device specific storage (APIs, tools, frameworks such as WebWorks, Shared Preferences, Network IO, WebView, Core Data.

Next section deepens the discussion of data storage features for each of the major mobile platforms.

3 Data Layer Options on Some Major Mobile Platforms. From Local to the Cloud

Microsoft and Blackberry are struggling to get more users, by renewing their operations systems with Windows Phone 8 and Blackberry OS 10. But Android and iOS, produced by Google and Apple, are still ahead of the pack with a combined market share over 87% in 2012. When customer decide, not only the features provided by the operation system are significant, but also hardware characteristics, applications available on the market, social status, durability and so on.

Most of the applications developed for mobile devices need storage options for user's data, session data, life-cycle timestamps (especially for states la pause or suspend). Storage requirements can go from user specific options (simple values for dimensions, user accounts, etc.) to significant amounts of data gather by sensor, received from internet or internet by the user himself.

When using Android, there are several options for saving data. Among them, *shared preferences* are based on key-value pairs mechanism, only usable for primitive data types and limited by the current user session. A developer can use the SDK to retrieve, add, delete and modify preferences either related to the current activity or shared across multiple activities for the same user session. For both variants, a public interface is available, *android.con-tent.SharedPreferences* that contains two other nested classes, one as an editor for the current key-value pairs and another which acts as a listener for changing preferences and related callbacks. Other public methods are related to obtaining a map of names and values (different data types) along with getters and setters for primitive data types. Apart from the shared preferences, any application with correct permissions grated, can access the file system from the internal storage.

By default, Android limits the files access to the application which creates them (MODE_PRIVATE), but this behaviour can be modified to: MODE_WORLD_READABLE in order to allow any other application or user to read created or modified files (deprecated since API level 17), MODE_WORLD_WRITEABLE for open write access to others (also deprecated starting with API level 17) and finally MODE_APPEND which limits future users to add data at the end of the file. Classes for write and read are available in the API, as *FileOutputStream* and *FileInputStream,* associated to public methods as *openFileOutput*() and *openFileInput*(). Read, write and close methods are also available. In addition, the developer can make use of built-in functionality for getting the path to a certain file, manage directories, delete files and retrieve a list of files from a specific folder.

Also, methods exist for raw resources (added at design time usable at compile time) and for saving cache files (not recommended for sensitive data). An alternative for managing persistent data is the usage of external storage, fixed (internal non-removable partition) or removable (SD, microSD cards, devices connected through micro-USB ports). An essential feature of the external storage is the possibility of all application to access it. Also, special precautions are recommended, as the external device can be disconnected by the user at any moment (the developer has to check media availability through specific methods provided by the SDK). Almost the same file and directory related functions are used, to which dedicated locations as Music, Podcasts, Ringtones, Alarms, Pictures and others are pre-defined by the system, since

API level 8 (all contained by the public class Environment - *android.os.Environment*).
For the internal storage, the system maintains a policy of deleting all related files for an application when the application is uninstalled. The policy doesn't apply for the external storage unless the files are created by the application for private usage, by invoking *getExternalFilesDir()* method. Of course, all presented above can be used for structured data, as pair of values or even locally stored XML. Problems occur in means or querying data and obtaining aggregated reports.
Among other data layer options, the most popular SQLite, advertised by its producers as a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. In fact, SQLite is present on a variety of devices and operation systems, precompiled packages can be downloaded for Linux, Mac OS, Windows, Windows Phone 8, Windows .Net platform [7], [5] [3]. Its popularity can be explained by the following features [3]:

- easy to install and configure
- simplicity
- does not require a server (can run in client-only mode)
- compactness of database (all database resides in a single file for each application)
- it is an open source product.

Being so compact, SQLite can be ported to almost any device, the source code being available for the community. Android offers full support libraries for SQLite. As each database resides in its own file on the disk and it is directly managed by the application (no server process involved, all classes are able to use one database if the name is known), it cannot be shared directly between multiple applications. The small footprint library can handle both DML and DDL statements, offers cursors access to clients, indexes, multiple data types, primary and foreign keys and a lot more features, which makes it very easy to use and a good candidate for being used as a persistence layer.
Applications written for iOS devices such as iPhone or iPad have several storage options, some of them being similar to the above presented for Android, and others specific to the host OS. Among these, one can find files (either as text, comma separated values, XML, JSON), property lists, core data, user preferences, SQLite. As almost the same features are available for SQLite on both mobile OS, and the files are managed by a similar mechanism, we'll only briefly describe the other options for the Apple devices.
Property lists are used by applications on both iOS and MacOS in order to storing simple hierarchies of either primitive data types or containers as arrays or dictionaries. These data types can be nested in multiple levels, as needed by the developer. As mentioned in the official documentation by Apple, property lists are recommended for small amounts of data. Data stored in property lists can be serialized for future usage in either binary or XML formats. Eckermann [14] finds these lists easy to use, especially due to a well-defined Objective-C API, but on the other hand, states that they are not suitable for any type of relational data, can't handle large sets of objects, use an inefficient memory loading mechanism, and let the developer handle too much of the responsibilities for data integrity and manipulation.
In order to help the developers with the usage of mode view controller pattern, the Cocoa framework help the iOS developers to define a data model easier through a feature called *Core Data*. By using this mechanism, the developer has access to a data management solution able to handle the definition and integration of data models. Core Data offers a visual interface, intuitive and easy to use.
The data model communicates with the controller of the application and hides details as storage and SQL queries to the developer (although this leads to a lack of control over the data). This storage framework can serialize data on XML files, binary or directly to a SQLite database. Internally, it organizes data in an object graph and provides features for declaring relationships between different collections of data. One of the main advantages is the fact that it was specially built solution for iOS and it integrates well with other components or tools in the Apple ecosystem,

DOI: 10.12948/issn14531305/17.2.2013.04

reducing the code volume needed for persistence. The “user default” storage environment can be easily used for saving user settings and options related to applications or the system. It is not built to help the developers store large amount of data or data with any kind of structured organization.
The SQLite for iOS can be directly used as a C library or by having an Objective-C wrapper as FMDB, ArchDBObjects, SQLiteManager or SQLite persistent objects. In a similar way with Android, the developer has to handle the single file needed for each SQLite database, used as a seed.
Along with the option for locally storing data enumerated above for both Android and iOS, there are several other options as accessing database servers across the network with dedicated drivers, most of them created by the community. In the last couple of years, cloud databases had gained increase popularity among the developers and they seem to be important options for mobile devices. These kinds of services are offered by important players as Amazon with its S3 or Relational Services, Microsoft with Azure, Salesforce with Database.com, and the examples can continue.
Related to the subject of the next section, there are a few cloud database providers specialized in NoSQL services, having MongoDB as an underlying service. The most popular are MongoLab, MongoHQ, Rackspace Cloud or dotCloud. These kinds of services are first of all characterized by the lack of involving the client in phases like installing or configuring the software. The customer pays for all these services, for data access, storage space, network traffic, monitoring programs and specialized interventions when needed. The provider has to be complaint with Service Level Agreement regarding infrastructure maintaining, quality if the service, response time, etc.
Most of the database services in the cloud expose polyglot interfaces for the client, offering at the same time SQL and NoSQL data storage engines. For mobile applications developers, using database services in the cloud can be a big challenge on one hand, but can come with significant advantages on the other. It is worth mentioned that the device (with limited computing resources) can allocate more processing power to the application, which is more appreciated by the end user, replication problems are avoided, the need for expensive hardware (we have devices in 2013 running Android with 8 processor cores – see Samsung S4 and other new devices). Of course, issues like constant need for network connection (through Wi-Fi or data plans), battery consumption or prices for consumed database services arise. One can think about other advantages as the usage of the same persistence technology for both desktop and mobile apps, backup and restore plans and procedures already defined by the service provider, ability to scale up or down storage and access capacity based on the business needs, and others.

4 NoSQL Datastores for Mobile Applications. The Case of MongoDB

There has been huge interest about NoSQL data stores for the last three years. A good amount of literature has been dedicated to NoSQL movement, which varies from full excitement [15] to criticism [16][17] and conciliation with relational technologies [18][19]. For a brief discussion about the context of NoSQL emergence, NoSQL basic features see [20], [21], [22], [23]. In advocating NoSQL versus Relational Databases for Mobile Applications Asay [24] points out two weaknesses of relational technologies – the rigidity of schema and impossibility of handling all of the different use cases mobile applications call for. Due to their scalability and speed Selvadurai [25] recommends using NoSQL datastores when mobile applications manage huge amount of data on a central server. In this paper MongoDB was chosen as a NoSQL camp representative due to its features and popularity [21][26].
Next we will describe the usage of an external cloud database service (MongoDB) for Android. Connection requires a Java driver and the *mongo.jar* library added to the *classpath*. The driver was developed by the community and released for Android (com-

patibility with Dalvik virtual machine fixed) in august 2012.

When connecting to a MongoDB server from a Java application several classes are to be used, depending of the desired functionality. The packages to be considered are *com.mongodb* (main package having functionality for the connection, database server, MongoClient, query builder, Map Reduce commands, database cursors, etc.), *org.bson* that implements functionality for BSON object and encoding, and *org.bson.types* (package containing implemented functionality for data types which can be stored in BSON format, from primitive data types to arrays and nested structures). Also, packages for obtaining statistics on the connection pool (like *com.mongodb.tools*) and packages containing classes to implement utility functions (for MongoDB database connection and for JSON data types) can prove useful.

According with MongoDB architecture, multiple entry points can be used, all being part of replica sets (database instances forming a cluster in order to easily replicate data). As stated in the official documentation, the java MongoDB driver is thread safe and it is recommended to create a single object which can be accessed by multiple threads. Internally, this object creates a pool of connections, and for each operation it is able to find an available connection, use it and release the resources after it is done. The developer can enforce a certain consistent behaviour (usage of the same socket by the client) by calling the two specially designed functions, *db.requestStart()* and *db.RequestDone().*

The driver also allows the developer to use authentication by username and password. After the connection sequence is performed, collections (all, or a specific one) can be retrieved, documents can be inserted, simple find operations can be performed by calling *findOne*() method. An important functionality is the usage of cursors by assigning the result of a *collection.find()* call to a DBCursor object. Sequential operations can be performed using the obtained cursor object. In addition to the operations enumerated above, some administrative actions can be performed as creating indexes, dropping them and obtaining a list for the collections in the current database.

Some limitations might appear when connecting to a database service in the cloud. A very important functionality covered by the Java MongoDB driver is the ability to access the aggregation framework (see section 6). The wrapper around this functionality provided by the driver is *DBCollection.aggregate().* Generic *DBObject* instances can be used by the programmer to create the pipeline of operations needed in aggregation processes. Finally, and *AggregationOutput* object can be obtained after calling the aggregate method over a certain collection.

In order to better illustrate the above features, a popular service for MongoDB database in the cloud will be used as persistence layer for a demonstrative Android application. The service is provided by MongoLab and it offers physical storage in different data centers like Amazon, Joyent or Microsoft Azure. At the time of writing of this paper, the available storage options for shared plans are: 0.5, 1, 2 and 4 GB. The corresponding prices are between 10 to 40 USD each month (except the 0.5 GB instance which is free). The customer can also buy dedicated plans (details and prices varies from one storage provider to another. For Amazon hosted MongoDB databases in the cloud, available RAM resources start from 1.7 GB and go up to 68 GB, available processors core are between 1 and 8 and the user can choose one or more dedicated nodes. Default storage for each unit type varies from 40 to 160 GB and can be easily extended or moved to SSD disks. Just as a plan example, for a 34 GB of RAM, 2 dedicated nodes, 4 processor cores, 80 GB storage capacity (on SSD disks) costs about $3000 each month. Of course, the storage capacity doesn't come cheap, but includes additional services as MongoDB monitoring service activated, real-time access to created log files, 24/7 DBA assistance from 10Gen (creator of MongoDB), replica sets and dedicated virtual machines.

In order to create a database, some information needs to be provided: database name, data center provider (we choose Amazon, an EU Ireland instance based on proximity criteria), the instance type (depending on the storage capacity needed), a new username and password for connections. Also, MongoLab allows the users to clone any already created database using its interface or any other accessible in the cloud.

After creating the database, some additional information is provided to the user: the connection string for MongoDB shell (in our case, *# mongo ds061807.mongolab.com:61807/testdb -u <dbuser> -p <dbpassword>*), the standard URI for client connection (for our database: *mongodb://<dbuser>:<dbpassword>@ds061807.mongolab.com:61807/testdb*). Connections are available using database drivers, including the java driver for Android. Moreover, the web interface allows users to add collections, manage the existing ones, adding and removing users and privileges, perform back-ups or back-up scheduled plans, import and export collections in binary, comma separated value and JSON formats, or even the entire database (binary only). The same web interface allows consulting statistics about the database as number of collections, number of objects, indexes, total storage size, average object dimensions, files size for the database.

In the remaining of this section we will exemplify some basic operations in a mobile application running on Android or iOS. The persistence layer is assured by the previously described provider, MongoLab, and all three components connect directly to the database without the need for middleware software. The application is designed to be used by didactic personnel in a faculty in order to remotely register grades obtained by students during exams. The following operations are performed using the mobile application:

- the user connects to the cloud database, across the network;
- the user retrieves all students from a collection, based on their group number;
- adding a new grade (for a student);
- getting the average grade for each group of students.

Connection to the cloud based MongoDB instance using the Android SDK requires the provided standard URI, username (*professor1*) and password (*education*):

```
MongoClient mongoAndroid = new MongoClient(
    "mongodb://professor1:education>@ds061807.
mongolab.com:61807/testdb" );
```

Assuming that each group of students has the grades stored in a specific collection, for the group *InfoEc2*, the mechanism of retrieving the entire collection is:

```
DBCollection grades =
db.getCollection("InfoEc2");
```

For inserting a document, multiple options are available, e.g using document builder classes, generic database objects, parsing strings in JSON formats. The next three solutions insert a grade for a student:

```
//solution 1
BasicDBObject examination = new
BasicDBObject();
examination.put("Subject Examined", "Database
Fundamentals");
examination.put("Grade", 9);
examination.put("Student
ID","SL31040701445");
grades.insert(examination);
// solution 2
String cText = "{'Subject Examined':'Database
Fundamentals','Grade' : 9," +
        "'Student ID' : 'SL31040701445'}";
DBObject dbgenericObject =
(DBObject)JSON.parse(cText);
Grades.insert(dbgenericObject);
//solution 3
BasicDBObjectBuilder gradeBuilder =
BasicDBObjectBuilder.start()
.add("Subject Examined", "Database Fundamen-
tals");
.add("Grade", 9);
.add("Student ID","SL31040701445");
Grades.insert(gradeBuilder.get());
```

Assuming a collection containing all the students and the groups they belong to, here is the solution for getting all from group "1204":

```
BasicDBObject studentQ = new BasicDBObject();
studentQ.put("Group", "1204");
DBCursor ResultCursor = stu-
dents.find(studentQ);
```

As a final operation, the aggregation framework will be invoked from the client side. The goal is to obtain the average grade for each group of the students having a major in "Business Information Studies":

```
{  "_id" : ObjectId(<string>),
       "student ID" : <string>,
       "major" : <string>,
       "subject" : <string>,
       "grade" : <integer>,
       "group": <integer>
};
// aggregation code in Java
DBObject      filterCriteria      =      new
BasicDBObject("$match",
       new  BasicDBObject("major",  "Business
Information Studies") );
DBObject   groupCriteriaComponents   =   new
BasicDBObject("$group",
       new  BasicDBObject("major",  "Business
Information Studies") );
groupCriteriaComponents.put("average",    new
BasicDBObject("$avg", "$grade"))
DBObject      groupCriteria      =      new
BasicDBObject("$group",
groupCriteriaComponents);
AggregationOutput       aggResults        =
grades.aggregate(filterCriteria,
groupCriteria);
```

In the next section we will focus on some of the MongoDB-SQL differences in terms of basic data definition and manipulation.

5 Basic Differences between MongoDB and SQL in Terms of Data Definition and Manipulation

Compared with the data solution in previous section for the remaining of this paper MongoDB database is installed following a classical replica set which is required for the server side of a distributed mobile application.

The relational database schema which serves as case study is depicted in figure 2. *Invoices* is the main table which stores - along with *invoices_items* and *products* - data about sales. The buyers (*customers*) have one or more *contacts* - *people* with important positions (CIOs, financial and procurement managers, etc.). Postal addresses refer to present Romanian administrative organization - address, postal/zip code, location (city or village), and county - but can easily be understood by non-Romanians (also by Romanians!). Invoice payments are stored in *receipts* and *receipts_invs* tables.

The data model is rather different in MongoDB. In SQL databases the above (sub)schema is composed by tables (views, stored procedures, etc.), each table having a common structure for all of its rows. Equivalent to tables, MongoDB databases have collections. Each collection have is composed by documents which can have a completely different structure.

As the data model differs, there are some SQL option with no MongoDB equivalence and vice versa. For example in a typical relational (SQL) database server, the user (or application) is connected to a database (sub)schema. Depending on the user rights, from current subschema, user can access objects placed in other subschemas. In MongoDB the user can choose the current database (schema) using a command which reminds us of dBase or FoxPro (xBases):

```
use local
```

The storage objects in SQL databases - tables - can be displayed, as with every type of database object, by querying the data dictionary. In PostgreSQL the syntax is:

```
SELECT table_name FROM infor-
mation_schema.tables
WHERE table_schema = 'public' AND table_type
= 'BASE TABLE'
ORDER BY table_name
```

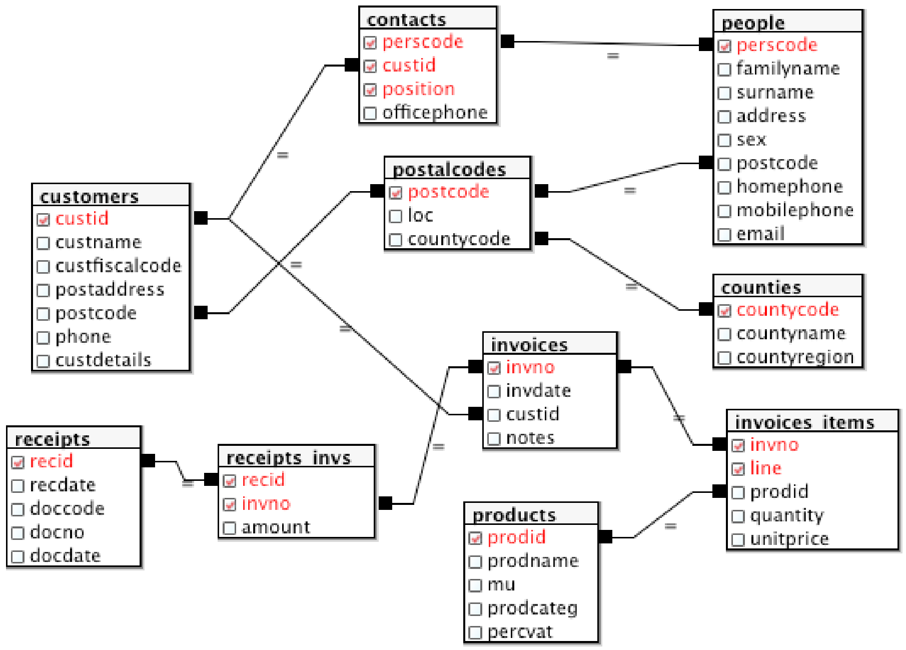

Fig. 2. Database schema for the case study

In MongoDB data is stored in collections which can be displayed as below:

```
show collections
```

Deleting a table in SQL requires DROP TABLE command (e.g. `DROP TABLE countie`) whereas in MongoDB the command (actually, it is a function) is *drop*:

```
db.counties.drop()
```

One of the main differences between SQL and NoSQL datastores concerns data objects creation and population. In SQL there is a clear distinction between DDL (Data Definition) commands, such as:

```
CREATE TABLE counties ( countyCode CHAR(2),
countyName VARCHAR(25),
    countyRegion VARCHAR(15) )  ;
```

and DML (Data Manipulation Language) commands like:

```
INSERT INTO counties VALUES ('IS', 'Iasi',
'Moldova');
```

On the contrary, this distinction does not exist in MongoDB and other NoSQL DBMSs. Collections are created on the fly, when a fist document is inserted using functions like *insert*, *save* or *update*.

```
db.counties.save ( { _id : 'IS', countyName :
'Iasi',countyRegion  : 'Moldova' });
```

All documents in a collection have an object *id* which can be generated automatically by the system or – as in above *save* statement – specified by the user.
As previously pointed out, another major difference is related to the record structure. In SQL database tables, every row has a similar (tabular) structure. When inserting a row without declaring values for all attributes, unspecified attributes will get by default null values:

```
INSERT INTO counties (countyCode, countyName)
VALUES ('B', 'Bucuresti');
```

But in MongoDB in the same collection, every document could have different number of attributes (it is not quite a brilliant idea to have completely different attributes/values in the same document):

```
db.counties.insert ( { _id : 'B', countyName
: 'Bucuresti'});
```

As generally acknowledged, SQL allow declarations of the following constraints (in in order to maintain a decent level of data integrity):

- primary key;

- alternate key (unique);
- not null;
- referential integrity;
- attribute and record level validation rules (check constraints).

In MongoDB only first two can be implemented by creating indexes. Equivalent to PostgreSQL command (an index will be created automatically):

```
ALTER TABLE counties ADD PRIMARY KEY
(countyCode);
```

is Mongo function:

```
db.counties.ensureIndex({_id : 1}, {unique:
true}) ;
```

An alternate key can be declared in SQL/PostgreSQL as follows:

```
ALTER TABLE counties ADD UNIQUE (countyName);
```

In MongoDB there is no such thing as alternate key, but the similar result can be achieved creating another index using UNIQUE option:

```
db.counties.ensureIndex({countyName:      1},
{unique: true});
```

6 Queries in MongoDB and SQL/PostgresSQL

The core SQL strength is its almighty SELECT command used for expressing queries with various degree of complexity. In Mongo, there are many techniques for database query, some of them being shown below. For basic queries, the best equivalent of SQL SELECT is MongoDB *find* function. Whereas

```
SELECT * FROM counties ORDER BY countyName
```

extracts all rows (ordered) in a SQL table, in MongoDB all documents (ordered) in a collection can be extract as follows:

```
db.counties.find().sort({ countyName : 1 } )
```

First argument - which lacks in above query - is, by default, predicate for filtering documents. To order the documents in the result, *sort* clause is needed:

```
db.counties.find( { countyRegion : "Moldova"
} ).sort({ countyName : 1 })
```

SQL language is set oriented. Current row, first row which fulfills a predicate do not exists in SQL base vocabulary of SELECT statement, but only in procedural extensions of SQL (cursors). So, for example, in order to extract first county in Moldova region, ranking options are needed (e.g. TOP, LIMIT, RANK, DENSE_RANK):

```
SELECT * FROM counties WHERE countyRegion =
'Moldova' ORDER BY countyName LIMIT 1;
```

On the contrary, in MongoDB *findOne* function returns and displays just first record from all the records which fulfills a predicate (again, older database guys will remember LOCATE command in dBase or FoxPro):

```
db.counties.findOne ( { countyRegion : "Mol-
dova" } )
```

Both SQL and MongoDB have options for nesting queries. Taking the problem of extracting last two records matching a predicate – let's say, display the last two counties in Moldova region. In SQL the above solution with and additional query nesting in FROM clause is functioning:

```
SELECT * FROM (SELECT * FROM counties WHERE
countyRegion = 'Moldova'
        ORDER BY countyName DESC LIMIT 2) t
ORDER BY countyName
```

In MongoDB we can use a JavaScript variable (see next section), but also we can nest a *count* function in *find* statement:

```
db.counties.find( { countyRegion : "Moldova"
} ).sort({ countyName : 1 }).
        skip(db.counties.count({  countyRegion
: "Moldova" } )-2) .limit(2)
```

Since every document in a collection could have a different schema, MongoDB permits finding documents with no value declared for an attribute. So, in order to extract counties with no region declared (documents in which attribute *countyRegion* was not declared) the following query uses *$exists*:

```
db.counties.find( { countyRegion :  { $exists
: false } } ). sort({ countyName : 1 })
```

In SQL there is no equivalent query, since all the table rows share the same structure. So all records will have *countyRegion* attribute, but in some of the rows its value are NULL:

```
SELECT * FROM counties WHERE countyRegion IS
NULL ORDER BY countyName ;
```

Mongo is not able to make the difference between "not declared" attributes and "declared but null values" ones. Both criteria will be met by the following query:

```
db.counties.find( { countyRegion : { $exists
: true }, countyRegion : null }  )
```

In both SQL and MongoDB NULL value can appear in a list. Taking the following example: extract counties with no region (*countyRegion* attribute) plus counties from Moldova region. In SQL the query that answers the problem is:

```
SELECT * FROM counties WHERE countyRegion IS
NULL OR countyRegion = 'Moldova' ORDER BY
countyName;
```

The two following MongoDB solutions use *$or* and *$in* operators:

```
// 1 ("$or")
db.counties.find( { $or : [ { countyRegion :
"Moldova"}, { countyRegion : null} ] }
).sort({ countyName : 1 })
// 2 ("$in")
db.counties.find( { countyRegion : { $in : [
"Moldova", null] } }). sort({ countyName : 1
})
```

Things get slightly more complicated when interested in extracting the *distinct* values of an attribute. Whereas in SQL there is a simple DISTINCT clause used in SELECT command:

```
SELECT DISTINCT countyRegion FROM counties
ORDER BY countyRegion ;
```

in MongoDB extracting distinct values of an attribute (countyRegion in collection "counties") can be achieved with (at least) three types of queries:

```
db.runCommand({"distinct" : "counties", "key"
: "countyRegion"})
//or:
db.counties.distinct( 'countyRegion')
//or using aggregation framework:
db.counties.aggregate ( [
        {    $project    :    {countyRegion    :
"$countyRegion", _id:0 } },
        { $group : {_id : "$countyRegion" }} ,
        { $sort : { _id : 1} } ] )
```

or using another aggregation framework solution, based on *$addToSet* option:

```
db.counties.aggregate(
        { $project : { countyRegion : 1} },
        { $group : { _id : null ,regions :
       { $addToSet : "$countyRegion" } } } );
```

The basic task of filtering records - e.g. how many counties are there in Moldova region? - can be achieved by COUNT function in SQL:

```
SELECT COUNT(*) FROM counties WHERE
countyRegion = 'Moldova'
and a similar one in MongoDB.
db.counties.count( { countyRegion : "Moldo-
va"} )
```

One of the most important feature in SQL queries concerns grouping set of rows/records. Finding in SQL how many counties are in each region requires:

```
SELECT countyRegion, COUNT(*) FROM counties
GROUP BY countyRegion ORDER BY countyRegion
```

Corresponding MongoDB query can use either *group* method (which reminds map-reduce queries):

```
db.counties.group( {
          key : { countyRegion : true },
          initial : { n_of_counties : 0 },
          reduce : function (doc, aggregator
) {
              aggregator.n_of_counties += 1;
}})
```

or the *aggregation framework*:

```
db.counties.aggregate (
          {    $group    :    {    _id    :
"$countyRegion", n_of_counties : { $sum : 1 }
}},
          { $sort : { _id : 1 } } )
```

To exemplify how to filter groups of records, we'll try to answer the problem of finding how many regions have more than three counties. In SQL GROUP BY option must be combined with HAVING:

```
SELECT countyRegion, COUNT(*)
FROM counties
GROUP BY countyRegion
HAVING COUNT(*) > 3
```

In MongoDB aggregate function combines *$group* and *$match*:

```
db.counties.aggregate (
        {    $group    :    {    _id    :
"$countyRegion", n_of_counties : { $sum : 1 }
}},
        { $match : { n_of_counties : { $gt
: 3} }})
```

7 More Advanced Options for Data Definition, Manipulation and Query

We start this section with a problem from the previous one: display the last two counties in Moldova region. The next MongoDB solution shows another difference from SQL – the extensive use of variables in some queries:

```
var    countMoldova    =    db.counties.count({
countyRegion : "Moldova" } )
db.counties.find( { countyRegion : "Moldova"
} ).
        sort({       countyName       :       1
}).skip(countMoldova-2).limit(2)
```

Remember that actually MongoDB shell is a JavaScript shell. The above solution uses *countMoldova* JavaScript variable that stores the number of counties in Modova region. This is the argument for the second command, which displays the last two counties in that region.

The main source of the need for variables is the lack of joins in MongoDB. When a collection is queried based on the documents extracted from another collection, generally it is necessary to split the solution into many queries and to store intermediate results in variables. As pointed out in section 4, there are no constraints to be declared in MongoDB. There are no foreign keys and the database is not normalized [26]. There is a special datatype – *DBRef* - (*http://docs.mongodb.org/manual/applications/database-references/*) but even the mongo official documentation does not recommend using it. Collection *postalCodes* has a foreign key-like - *countyCode* – but its values will be handled "manually":

```
db.postalCodes.save ( { _id : '700505', loc :
'Iasi', countyCode : 'IS' });
```

In order to display postal codes in Iasi county, the SQL query is:

```
SELECT postalCodes.*
FROM postalCodes INNER JOIN counties
        ON   postalCodes.countyCode   =   coun-
ties.countyCode
WHERE countyName = 'Iasi'
ORDER BY postCode
```

Answering the same problem in MongoDB requires some JavaScript programming. Both solutions presented below use cursors. The first one is based on *hasNext()* for navigation among documents:

```
var myCursor = db.counties.find ({ countyName
: 'Iasi'}) ;
var     myRow     =     myCursor.hasNext()     ?
myCursor.next() :null ;
if (myRow) {var myCountyCode = myRow._id ; }
db.postalCodes.find({       countyCode       :
myCountyCode }) ;
```

The second is similar but based on *forEach*():

```
var myCursor = db.counties.find ({ countyName
: 'Iasi'}) ;
var myCountyCode ;
myCursor.forEach(function(x) {
        myCountyCode = x._id ;
        } ) ;
db.postalCodes.find({       countyCode       :
myCountyCode }) ;
```

Many join-based SQL solutions have equivalents in sub-queries based ones. Taking a similar problem to the last one: extract all postal codes in Moldova region. A subquery-based SELECT is:

```
SELECT postalCodes.*
FROM postalCodes
WHERE countyCode IN (SELECT countyCode FROM
counties
WHERE countyRegion = 'Moldova')
ORDER BY postCode
```

In MongoDB there are no subqueries, but the rows selected by a first query (sub-query equivalent) can be transferred to an array searched with *$in* operator:

```
var arrayCountyCodes = [] ;
var    myCursor    =    db.counties.find    ({
countyRegion : 'Moldova'}) ;
myCursor.forEach(function(x) {
        arrayCountyCodes.push(x._id) ;
        } ) ;
db.postalCodes.find({ countyCode : { $in :
arrayCountyCodes } }) ;
```

Instead of array, the following MongoDB solution uses a regular expression:

```
var myRegExp = "";
var    myCursor    =    db.counties.find    ({
countyRegion : 'Moldova'}) ;
myCursor.forEach(function(x) {
        var myCountyCode = x._id ;
        if (myRegExp == "") {
                myRegExp = "^" + myCountyCode
; }
        else {
                myRegExp = myRegExp + "|^" +
myCountyCode ; }
        print (myRegExp) ;
        } ) ;
        db.postalCodes.find({'countyCode'    :
{"$regex" : myRegExp  } }) ;
} ) ;
```

As previously pointed out the "relaxed" structure of collection records is one of the most trumped advantage of NoSQL datastores. In this section we will present a classic way of dealing with denormalization of the database. Three tables in figure 2 - *customers*, *people* and *contacts* – that were briefly discussed in section 5 store people who hold important positions in our company's customers (customers that are, on their turn, companies, not individuals). This is a classic example of normalization. In MongoDB (and most of the NoSQL datastores) there are no explicit relationships among collections, so no join is possible. Composite documents can deal very well with this type of problems. Equivalent to those three tables a single collection will be used in MongoDB, but each document in this collection contains both the person and the position she/he holds in that customer:

```
db.customers.save ( { custName : 'Client 2
SA', custFiscalCode: 'R1002',
        postCode    :    '700505',    phone    :
'0232212121',
        contacts : [
{ person : { persCode : 'CNP2', familyName :
'Vasile', surName : 'Ion', sex : 'B',
         postCode  :  '700505',  homePhone  :
'0234234567', officePhone : '0234876543',
         mobilePhone  :  '0794222223',  email  :
'Ion@a.ro'},
                position :  'Director general'
        },
{ person : { persCode : 'CNP3', familyName :
'Popovici', surName : 'Ioana',
        address  :  'V.Micle,  Bl.I,  Sc.B,Ap.2',
sex : 'F', postCode : '701150',
        homePhone  :  '0233534568',  mobilePhone
: '0744222224'},
                position      :              'Sef
        aprovizionare' }] }) ;
```

For customer *Client 2 SA* two contacts were introduced, *Vasile Ion* who is *Director general* (general manager) and *Popovici Ioana* who is *Sef aprovizionare* (procurement management). In terms on MongoDB the document that describes the above customer contains a number of key-values pairs, from which the values of the attribute *contacts* is an array containing two elements. Each element is, at its turn, a (sub) document composed of (sub) document *person* and attribute (and its values) *position*.
We do not dive into details above the strengths and weaknesses of this schema, but develop some differences in terms of data definition, manipulation and query that occur. What in SQL is the operations of updating and querying rows in tables can be translated in MongoDB not only in update/query documents in a collection, but also, update/query a subdocument or an array (included into another array…) in a document.
Taking the simple example of updating an attribute value on a record, i.e. update phone number for customer 'Client 1 SRL'. In SQL that is possible by an UPDATE command:

```
UPDATE   customers   SET   phone   =   '0232217001'
WHERE custName = 'Client 1 SRL'
```

In MongoDB the query is pretty similar:

```
db.customers.update({  custName  :  'Client  1
SRL'},{$set : { phone : '0232217001'}})
```

Now the problem is finding customers for which position *Sef aprovizionare* (Procurement Manager) is in contacts. This requires a join in SQL:

```
SELECT *
FROM customers
        INNER    JOIN    contacts    ON    custom-
ers.custID = contacts.custID
        INNER JOIN people ON contacts.persCode
= people.persCode
WHERE contacts.position = 'Sef aprovizionare'
```

But in MongoDB due to the schema of records in *contact* collection this problem requires querying defining a predicate that involves an attribute within an array:

```
db.customers.find ({ "contacts.position" :
"Sef aprovizionare" } ).pretty()
```

If in SQL adding as customer contact a person who already exists in the database translates into a simple insert in *contacts* table:

```
INSERT INTO contacts VALUES ('CNP7', (SELECT
custId FROM customers
    WHERE  custName ='Client  6  SA'),
'Consultant aprovizionare')  ;
```

In MongoDB the problem requires inserting an element into array *customers* which is possible either by *$push* or *$addToSet*:

```
// 1: $push
db.customers.update ( {custName : "Client 6
SA"}, {$push : { contacts :
       { person : { persCode : 'CNP7',
familyName : 'Popa', surName : 'Ioanid',
              address : 'I.Ion, Bl.H2, Sc.C,
Ap.45', sex : 'B', postCode : '701900',
              homesPhone  :  '0238789012',
officePhone : '0238321098'},
       position :  'Consultant aprovizionare'
}}}) ;
// 2. $addToSet is better when need to be
sure the array will not contain duplicate el-
ements
db.customers.update ( {custName : "Client 6
SA"}, {$addToSet : { contacts :
       { person : { persCode : 'CNP7',
familyName : 'Popa', surName : 'Ioanid',
              address : 'I.Ion, Bl.H2, Sc.C,
Ap.45', sex : 'B', postCode : '701900',
                     homesPhone           :
'0238789012', officePhone : '0238321098'},
       position :  'Consultant aprovizionare'
}}}) ;
```

Sometimes operations that are simple in SQL raise serious problems in MongoDB. For example, setting an attribute value from another attribute value in the same (sub)record. We discover that the mobilePhone number of *Iurea Simion* (personal code *CNP5*) is actually his *officePhone* number. Two updates (that could be fusioned into a single one) solve the problem in SQL:

```
UPDATE people SET officePhone = mobilePhone
WHERE persCode = 'CNP5' ;
UPDATE people SET mobilePhone = NULL WHERE
persCode = 'CNP5' ;
```

In MongoDB this type of updates is not yet possible, so setting an attribute value from another attribute in the same document requires some basic programming/scripting. The idea is to store *mobilePhone* value into a variable and then update the office phone:

```
// extract Client 3 SRL record
var cust = db.customers.findOne ( {custName :
'Client 3 SRL', "contacts.person.persCode" :
"CNP5"}) ;
// gather mobilePhone
var              mobPh                      =
cust.contacts[1].person.mobilePhone ;
// update officePhone
db.customers.update ( {custName : 'Client 3
SRL',
       "contacts.person.persCode" : "CNP5"},
              {     $set     :      {"con-
tacts.$.person.officePhone" : mobPh } } ) ;
// delete mobilePhone
db.customers.update ( {custName : 'Client 3
SRL',
       "contacts.person.persCode" : "CNP5"},
              {    $unset     :     {"con-
tacts.$.person.mobilePhone" : 1 } } ) ;
```

Database denormalization has other updating consequences too. If person *Popa Ioanid* (whose personal code is CNP7) changes his *homePhone* number into *0232789012*, because of normalization in SQL databases a single row will be modified:

```
UPDATE people SET homePhone = '0232789012'
WHERE persCode = 'CNP7'
```

Since the document structure in *contacts* collection is denormalized, *Popa Ioanid* appears in three documents, as contact for three different customers. This is an opportunity to point out another difference SQL-MongoDB. Whereas in SQL the default scope of *update* command is all the set of rows that satisfy the predicate in WHERE clause, in MongoDB the default scope is the only first document satisfying the predicate. So in this case update function must presents a third parameter – *multi* – that sets the scope to all documents that meet the criteria:

```
db.customers.update        (         {"con-
tacts.person.persCode" : "CNP7"},
       {      $set      :       {"con-
tacts.$.person.homePhone" : "0232789012" } },
       { multi : true } ) ;
```

Another difference resides in the dynamic of the language. SQL is an established lan-

DOI: 10.12948/issn14531305/17.2.2013.04

guage. Even versions of the standard have been published every now and then (1989, 1992, 1999, 2003, 2008, 2011), the core of the language remains unchanged since its inceptions. On the contrary, in MongoDB, DDL, DML and query options improve at a rapid pace with every new version, even the product seems to have reached a certain level of maturity.

If, for example, attribute address must be renamed into *postAdress*, in SQL that has been possible through ALTER TABLE (PostgreSQL syntax):

```
ALTER TABLE customers RENAME COLUMN address to postAddress;
```

In MongoDB, prior to version 1.7.2, renaming operations had to be done "manually" for each document in collection:

```
db.customers.find().forEach(
  function (x) {
    // declare new property which gather the value of an existing property
    x.postAddress = x.address ;
    // remove old property
    delete x.address;
    // save the updated document
    db.customers.save(x);
  } )
```

Since Mongo 1.7.2 *$rename (predicate, fields_for_renaming, upsert, multiple_documents)* method is available, so the operation is much simplified:

```
db.customers.update( {},{ $rename : { 'address' : 'postAddress' }}, false, true ) ;
```

As already exemplified, due to different data structure, the some problem translates into different operations in SQL and MongoDB. Continuing with a correction. In both schemas, *officePhone* was placed in *person* table or sub-document. This is wrong, since *officePhone* is related simultaneously to both person and customer. So we need to move *officePhone* attribute (along with its values) from *person* to *contact* (mobilePhone is personal, but officePhone belongs to employer). In SQL (PostgreSQL syntax) this id done by following succession:

```
// ALTER TABLE for adding contact.officePhone
ALTER TABLE contacts ADD officePhone VARCHAR(10) ;
// UPDATE contact.officePhone from people.officePhone
UPDATE contacts SET officePhone = (SELECT officePhone FROM people
       WHERE persCode = contacts.persCode) ;
// ALTER TABLE for deleting people.officePhone
ALTER TABLE people DROP COLUMN officePhone ;
```

Currently, in MongoDB the problem required renaming an array field which is possible only by the following code:

```
db.customers.find().forEach( function (x) {
       for ( var idx = 0; idx < x.contacts.length; idx++) {
              y = x.contacts[idx].person ;
              if ('officePhone' in y) { x.contacts[idx].officePhone = y.officePhone;
                     delete x.contacts[idx].person.officePhone;   }
       }
       // save the updated document
       db.customers.save(x);
} )
```

Upsert (update combined in a single statement with insert) is among operations that are easier to perform in Mongo than in SQL/ PostgreSQL. In SQL standard and some dialects there is a special command – MERGE. Unfortunately, it is not yet implemented in PostgreSQL so the following MongoDB *upsert* has not an equivalent:

```
db.products.update ( {_id : 1},
       {_id: 1, prodName : 'Produs 1', mu : 'buc', prodCateg : 'Cosmetice',
              percVAT  : .24 },
       {upsert : true} ) ;
```

If in collection *products* there is already a document with _id = 1, then the above statement is an *update*. Otherwise, it is an *insert*.

As usual, things get easier with deletions, but, as with inserts and updates, sometimes a SQL DELETE operation translates into MongoDB into deleting documents, or subdocuments or array elements. But we will start with deletion of a data storage object – table or collection – which is quite similar:

```
// PostgreSQL:
DROP TABLE invoices
// MongoDB:
db.invoices.drop()
```

Also in many situations deleting rows in a table translates simply in deleting documents in a collection:

DOI: 10.12948/issn14531305/17.2.2013.04

```
// PostgreSQL: delete products in Cosmetice
(cosmetics) category
DELETE FROM products WHERE prodCateg =
'Cosmetice' ;
// same operation in MongoDB
db.products.remove({ prodCateg : 'Cosmetice'}
)
```

Supposing that customer *Client 6 SA* ends consulting contract ("Consultant aprovizionare") of *Popa Ioanid* (CNP7). This is a case when a row deletion in SQL:

```
DELETE FROM contacts WHERE custId = 1006 AND
persCode = 'CNP7' AND
        position = 'Consultant aprovizionare'
;
```

means in MongoDB a remove operation in an array – *update ... $pull* option:

```
db.customers.update( { custName : "Client 6
SA" },
        { $pull : { contacts : { position :
"Consultant aprovizionare" } }} )
```

We store here with the parallel between SQL and MongoDB options in terms of DDL, DML and database query. Even if we covered with examples most of these commands and functions, there are also many MongoDB features which deserve deeper investigations such as map-reduce and the aggregation framework.

8 Conclusions

Mobile applications development is one of the most dynamic areas in IT industry. As mobile devices (smartphones, tablets, sensors) surpasses desktop PCs and laptops in terms of number of users, there is a growing interest in providing appropriate technologies for data storage in mobile applications.
The main challenge for database layer in many mobile applications is to mix the small scale of client resources (although in recent years tablets and smartphones increased their processing power and storage capacity) with huge amounts of data collected on the server side which need to be stored and processed, including proper replication and synchronization mechanisms between clients and servers.
Currently the data layer in mobile apps, especially on the client side is provided by proprietary non-database technologies or some special (lite) versions of relational DBMSs, NoSQL datastores are serious competitors due mainly to their schema flexibility and scalability. This paper tried to prove that MongoDB, as one the most praised NoSQL product, can be successfully used for deploying mobile applications for both client and server layers and also pointed out some common data definition, manipulation and query features and differences between MongoDB and SQL (PostgreSQL) DBMSs.

References

[1] P. Pocatilu, *Programarea dispozitivelor mobile*, Bucharest: ASE Publishing House, 2012

[2] N. Tomai, G. C. Silaghi (coords), A. Costişor, A.M. Ghiran, I. Petri, S. Presecan, C. Ştefanache, *Tehnologii şi aplicaţii mobile*. Cluj-Napoca: Risoprint Publishing House, 2012

[3] O. Dospinescu, M. Percă, *Aplicaţii mobile pe platforma Android*. Iaşi: Tehnopress Publishing House, 2013

[4] O. Dospinescu, D. Fotache, A. Munteanu, “Architecture For Enterprise Mobile Services,” in *Proc. of the 9th IBIMA International Business Information Management Conference*, Marrakech, Morocco, 2008, pp. 985-994

[5] P. Pocatilu (2012). Building Database-Powered Mobile Applications. *Informatica Economică Journal* [Online]. 16 (48), pp. 132-142. Available: http://revistaie.ase.ro/content /61/12%20-%20Pocatilu.pdf

[6] A. Zamfiroiu (2012). Integrability and Interoperability of Mobile Applications. *Informatica Economică* [Online] 16 (4), pp. 150-158. Available: http://www.revistaie.ase.ro/content/ 64/17%20-%20Zamfiroiu.pdf

[7] Q. H. Mahmoud, S. Zanin, T. Ngo, “Integrating Mobile Storage into Database Systems Courses”, in *Proc. of the 13th annual conference on Information technology education - SIGITE '12*, 2012, pp. 165-170

[8] B. Unhelkar, S. Murugesan, “The Enterprise Mobile Applications Development

Framework," *IT Professional*, 12 (3), May/June 2010, pp.33-39

[9] P. Padmanabhan, L. Gruenwald, A. Vallur, M. Atiquzzaman, "A survey of data replication techniques for mobile ad hoc network databases," *The VLDB Journal*, 17, pp. 1143–1164, 2008

[10] V.T.K. Tran, R.K. Wong, W.K. Cheung, J.Liu, "Mobile Information Exchange and Integration: From Query to Application Layer", in *Proc. of the 20th Australasian Database Conference (ADC 2009)*, 2009, pp.115-124

[11] S.W. Lee, G. J. Na, J.M. Kim, J.H. Oh, and S.W. Kim, "Research issues in next generation DBMS for mobile platforms", in *Proc of the 9th international conference on Human computer interaction with mobile devices and services (MobileHCI '07), 2007*, pp. 457-461

[12] H. Kim, N. Agrawal, C. Ungureanu, "Revisiting Storage for Smartphones," *ACM Transactions on Storage*, 8(4), Nov. 2012, pp. 14:1-14:25

[13] A. K. Nori (2007). Mobile and Embedded Databases. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. Available: ftp://ftp.research.microsoft.com/pub/debull/A07sept/nori.pdf (accessed March 2013)

[14] A. Eckermann (2011). Beginning iOS Development: Data Persistence. Available: http://mobile.tutsplus.com/tutorials/iphone/iphone-sdk_store-data/ (accessed March 2013)

[15] A. Floratou, N. Teletia, D.J. DeWitt, J.M. Patel, "Can the Elephants Handle the NoSQL Onslaught?," *Proceedings of the VLDB Endowment*, 5 (12), 2012, pp.1712-1723

[16]***, "Stonebraker on NoSQL and Enterprises," *Communications of the ACM*, 54 (8), August 2011 , pp.10-11

[17] C. Mohan, "History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla", on *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*, Genoa, Italy, 2013, pp. 11-16

[18]J. Pokorny, "NoSQL Databases: a step to database scalability in Web environment", in *Proc. of the 13th International Conference on Information Integration and Web-based Applications and Services (iiWAS '11)*, 2011, pp.278-283

[19] C. Nance, T. Losser, R. Iype, G. Harmon, "NoSQL vs RDBMS - Why There is Room for Both," in *Proceedings of the Southern Association for Information Systems Conference*, Savannah, GA, USA, March 2013, pp.111-116

[20] P. Helland, "If You Have Too Much Data, then 'Good Enough' Is Good Enough," *Communications of the ACM*, vol. 54, no. 6, pp.40-47, June 2011

[21] D. Cogean, M. Fotache, V. Şerban-Greavu, "NoSQL for Higher Education. A Case Sudy", in *Proc. of the 12th international conference on Informatics in Economy*, Bucharest, 2013, pp. 352-360

[22] R. Cattell, "Scalable SQL and NoSQL Data Stores," *ACM SIGMOD Record*, vol. 39, no. 4, December 2010, pp. 12-27

[23] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, D. Gosain, "A Survey and Comparison of Relational and Non-Relational Database," *International Journal of Engineering Research & Technology (IJERT)*, 1(6), August 2012, pp.1-5

[24] M. Asay (2013). Why NoSQL Trumps Relational Databases for Mobile Applications. Available: http://www.techopedia.com/2/29256/development/mobile-development/why-nosql-trumps-relational-databases-for-mobile-applications, March 5, 2013 (accessed April 2013)

[25] J. Selvadurai, "A Mobile Commerce Architecture Based on Location Based Services and Social Media Monitoring," *International Journal of Scientific & Engineering Research*, 3(9), September 2012, pp.1-4

[26] R. Copeland, *MongoDB Applied Design Patterns*. Sebastopol, California: O'Reilly, 2013.

Marin FOTACHE has graduated (long time ago) the Faculty of Economics at Alexandru Ioan Cuza University of Iasi, Romania. He holds a PhD diploma in Business Informationn Systems (Business Informatics) from 2000 and he had gone through all didactic positions since 1990 when he joined the staff of Alexandru Ioan Cuza University, from teaching assistant in 1990, to full professor in 2002. Currently he is professor within the Department of Accounting, Business Informatics and Statistics in the Faculty of Economics and Business Administration at Alexandru Ioan Cuza University. He is the (co)author of books and journal articles in the fields of SQL, database design, NoSQL, and knowledge management.

Dragos COGEAN has graduated the Faculty of Economics and Business Administration from "Alexandru Ioan Cuza" University of Iasi in 2008. He attended Master Studies in the fields of Business Information Systems at the same Faculty between 2008 and 2010. As Ph.D. student and employee of a Dutch software development company (Project Manager) he is interested in new technologies like NoSQL databases, big data software and analytics.

DOI: 10.12948/issn14531305/17.2.2013.04