CRUD Operations in MongoDB

Ciprian Octavian Truică, Alexandru Boicea, Ionut Trifan
University "Politehnica" of Bucharest, Romania
ciprian.truica@cti.pub.ro, alexandru.boicea@cs.pub.ro, ionut.trifan1@gmail.com

Abstract - In this paper we will examine the key features of the database management system MongoDB. We will focus on the basic operations of CRUD and indexes. For our example we will create two databases one using MySQL and one in MongoDB. We will also compare the way that data will be created, selected, inserted and deleted in both databases. For the index part we will talk about the different types used in MongoDB comparing them with the indexes used in a relational database.

Index Terms - MongoDB, NoSQL, BSON, CRUD, index.

I. Introduction

MongoDB is a high performance and very scalable document-oriented database developed in C++ that stores data in a BSON format, a dynamic schema document structured like JSON. Mongo hits a sweet spot between the powerful query ability of a relational database and the distributed nature of other databases like Riak or HBase [1]. MongoDB was developed keeping in mind the use of the database in a distributed architecture, more specifically a Shared Nothing Architecture, and that is way it can horizontally scale and it supports Master-Slave replication and Sharding.

MongoDB uses BSON to store its documents. BSON keep documents in an ordered list of elements, every elements has three components: a field name, a data type and a value. BSON was designed to be efficient in storage space and scan speed which is done for large elements in a BSON document by a prefixed with a length field. All documents must be serialized to BSON before being sent to MongoDB; they're later deserialized from BSON by the driver into the language's native document representation [2].

II. CRUD operations In MongoDB

In this chapter we will analyze the CRUD (Create, Read, Update, and Delete) operations in MongoDB. For a simple illustration of these operations we will also create a MySQL database and use the equivalent SQL queries.

The database will model an article with tags and comments. The article must have author and can have many comments and tags. In other words, the relational design of the database will have the following tables:

1. Users – the table where we will keep information about the article's author.
2. Articles- the table where we will store the article data
3. Tags – table for tags.
4. Comments – table for comments.
5. Link_article_tags – an article can have more than one tag.

For the MongoDB database, all data will be stored in one collection that will have documents that will look like:

```
{ user_id: "1",
first_name: "Ciprian",
last_name: "Truica",
article:{
date_created: "2013-05-01",
text: "This is my first article",
tags: ["MongoDB", "CRUD"],
comments: [
{author_name: "Alexandru",
date_created: "2013-05-02",
text: "good article"},
{author_name: "Andrei",
date_created: "2013-05-03",
text: "interesting article"}]}});
```

We will use the MongoDB nested property for nesting inside a document other documents. In other words document will keep tags in an object array and the comments in a BSON document array.

A. Create operation

In MongoDB the create operation is used to create a collection and add new documents to a collection. In SQL the equivalent operations are CREATE and INSERT.

A collection is implicitly created at the first insert: db.articles.insert(<document>). To explicitly create a new collection we can use: db.createCollection("articles").

So, for creating our collection and inserting our first article we can simple use the command:

```
db.articles.insert ({
user_id: "1",
first_name: "Ciprian",
last_name: "Truica",
article :{
date_created: "2013-05-01",
title: "Overview of MongoDB",
text: "This is my first article",
tags: ["MongoDB", "CRUD"],
comments: [
{author_name: "Alexandru",
date_created: "2013-05-02",
text: "good article"},
{author_name: "Andrei",
date_created: "2013-05-03",
text: "interesting article"}]}});
```

To do the same thing using a relational database we first need to create a schema, then to create the tables and lastly to do the insert operations for each table. We will only create the schema and the users table and then insert some date in this table.

```
CREATE SCHEMA `BlogDB`;
CREATE TABLE IF NOT EXISTS `BlogDB`.`users`
( `id` INT NOT NULL,
`first_name` VARCHAR(64) NULL ,
`last_name` VARCHAR(45) NULL ,
 PRIMARY KEY (`id`) );
```

INSERT INTO `BlogDB`.`users`(id, first_name, last_name) VALUES(1, "Ciprian" , "Truica");

For insertion of a new document we can write a simple JavaScript function. If we use PL/SQL or Transact-SQL this is a very difficult thing to do because we do not have a way to send arrays as input to a stored procedure or a function. A simple insertion function in MongoDB can be insertArticle, where articleText parameter is an object array and articleComments is a BSON document array:

```
Function  insertArticle (userId, firstName, lastName,
articleCreateionDate, articleText, articleTags,
articleComments){
db.articles.insert ({ user_id: userId,
first_name: firstName, last_name: lastName,
 article:{
 date_created: ISODate(articleCreateionDate),
text: articleText,
tags: tags,
comments: articleComments}})};
```

We can create a collection using the explicit method db.createCollection(<collection name>);. So if we want to explicitly create the collection articles we will use the next command in the mongo shell prompt:

```
> db.createCollection ("articles");
```

This command is equivalent to the CREATE command in SQL.

B. *Read Operations*

Read operations are used to retrieve data from the database. They can use aggregation functions like count, distinct or aggregate.

MongoDB offers two functions for this find() and findOne(). Their syntax is similar:

```
>db.collection.find( <query>, <projection>)
>db.collection.findOne(<query>, <projection>)
```

The db.collection specifies the collection, for our example we can use db.articles. To see all the collections contain by a database we can use the show collections command in the shell prompt. The <query> argument describes the returned set and the <projection> argument describes the returned field we want. For example if we want only the comments for all the articles for user with user_id=1 we can use the next query: db.articles.find({user_id:"1"}, { "article.comments" : 1});

To do the same query on a relational database we need two JOIN operations:

```
SELECT c.author_name, c.comment
FROM `BlogDB`.`articles` AS a
INNER JOIN `BlogDB`.`users` AS u
ON u.id = a.id_user
INNER JOIN `BlogDB`.`comments` AS c
ON c.id_article = a.id
WHERE u.id = 1;
```

As already mentioned before, JOIN operations are very costly. In MongoDB we stored comments in an array of BSON documents, and keeping in mind that the relation between articles and comments is one to many, we actually do one projection operation using the query:

```
>db.articles.find( {user_id:"1"}, { "article.comments" : 1});
```

If we wanted only one record for the user we could have used the findOne() function.

C. *Update operation*

In MongoDB we can use the update function for modifying a record. This function modifies a document if the record exists; otherwise an insert operation is done. For example we will change the title of the article:

```
>db.articles.update (
{_id: "1"},
{$set: { "article.title": "MongoDB" }},
{upsert: true});
```

To do this simple query in SQL we will use the next statement:

UPDATE `BlogDB`.`articles` SET title="MongoDB" WHERE id = 1;

Let's do another example, this time we will add a new comment to the existing article:

```
>db.articles.update  (
{_id: "1"},
{ $addToSet:
{"article.comments":
{author_name: "Aurel", date_created: "2013-02-02",
text: "MongoDB is so simple"}}},
{upsert: true});
```

MongoDB supports update operation on nested documents using $addToSet operator. The upset parameter is a Boolean. When true the update() operation will update an existing document that matches the query selection criteria or if no document matches the criteria a new document will be inserted. Another parameter that can be used with the update statement is the Boolean multi that, if the query criteria matches multiple documents and its value is set to true, then all the documents will be updated. If it value is set to false it will update only one document.

An update can be done by using the save() method which is analog with an update that has upsert: true.

D. *Remove operation*

To delete a document from a collection, in MongoDB we will use the remove() function that is analog with the DELETE operation in SQL:

```
>db.collection.remove(<query>, <limit>);
```

The arguments are:

1. <query> is the corresponding WHERE statement in SQL
2. <limit> is an Boolean argument that has the same effect as LIMIT 1 in SQL

Let do a delete on our database where the record's_id is 1. The statement looks like this:

```
>db.articles.remove ({_id: "1"});
```

By using this statement we will not only delete the article, but also the comment and the tags for the article.

In SQL this statement is a composed statement. First we will need to delete the information stored in the link_article_tags:

```
DELETE `BlogDB`.`link_article_tags` WHERE id_article=1;
```

Then we need to delete the comments:

```
DELETE `BlogDB`.`comments`
WHERE id_article=1;
```

Only after these operations we can delete the article, because all the constraints have been deleted.

```
DELETE `BlogDB`.`article`
 WHERE id = 1;
```

To delete all the data from o collection we use remove() function without any where clause as is done in SQL.

If we want to delete a collection from the database we will use the mongo shell prompt command db.collection.drop(). For example if we want to drop the articles collections we will use the fallowing command:

```
>db.articles.drop();
```

This command is equivalent to the SQL DROP command. So the corresponding query for db.articles.drop(); that will delete the articles table in SQL is:

```
DROP TABLE `BlogDB`.`articles`;
```

III. Indexes in MongoDB

Indexes offer enhanced performance for read operations. These are useful when documents are larger than the amount of RAM available.

MongoDB indexes are explicitly defined using an ensureIndex call, and any existing indices are automatically used for query processing [2].

The most important index is _id, if it's not introduces at the creation of a document it is automatically created. This unique index main purpose is to be the primary key.

The remaining indexes in MongoDB are known as secondary indexes. Secondary indexes can be created using ensureIndex (). If a field is a vector we can create separate indexes for each value in the array, the index is known as the multi-key index.

Let's add an index to the title field for our articles:

```
>db.articles.ensureIndex ({"article.title" : 1}, {unique :
 true});
```

In SQL we would write:

```
CREATE UNIQUE INDEX index_name ON `BlogDB`.`articles` (title);
```

Let's create a compound index for first_name and last_name for ensuring that the select operations are done faster:

```
>db.articles.ensureIndex ({first_name : 1, last_name:
        1}, {unique: true});
```

```
CREATE UNIQUE INDEX compound_index ON `BlogDB`.`users` (first_name, last_name);
```

And a last example is for a multi-key index, we will add an unique index for each separate tag:

```
>db.articles.ensureIndex({"article.tags" : 1},
        {unique : true});
```

Another type of index is the sparse index. This type of index is useful for documents that have a particular field, but that field is not in all documents in the collection, hence name. An example could be the comments field, we are not sure that all of our articles have comments but we want to add an index to this particular field. To do this we will use a sparse index:

```
>db.articles.ensureIndex({"article.comments" : 1},
        {sparse : true});
```

MongoDB provides geospatial indexes that are constrained by the location in a two-dimensional system and are useful for search documents that are similar to a given pair of coordinates. To declare a geospatial index a document must have stored inside a field with a two dimensional array or an embedded document, for example: loc: [x, y] or loc: {x: 1, y: 2}. To add a geospatial index one can use the fallowing command: db.collection.ensureIndex({ loc : "2d" }).

Let's add a geospatial index to our article. First we add the new information to the document and then we add the index on the new field location:

```
>db.articles.update (
{_id: "1"},
{ $addToSet: {location: {x: 1, y:2}}},
{upsert: true});
>db.ensureIndex ({location: "2d"});
```

To find a document based on a distance to a give point we can use a proximity query. For example, to find all the indexes near location x: 0, y: 0, one can use the fallowing query:

```
>db.articles.find({location: {$near: [0, 0]}});
```

To select documents that have coordinates that are in a specific geometric area one can use a bounded query. MongoDB supports the fallowing shapes: circles, rectangles and polygons. These queries perform faster than proximity queries.

For each shape, we present the next examples:

1. Circle with the center in (0,0) and radius=1: >db.articles. find ({ "location": { "$within": { "$center": [[0, 0], 2]}}})
2. Rectangle bound by the point (0,0) and (2,2)> >db.articles. find ({ "location": { "$within": { "$box": [[0, 0] , [2, 2]]}}})
3. Polygon defined by three points (0,0), (22), (2,0): >db.articles.find({"location": {"$within": {"$polygon": [[0,0], [2,2], [2,0]]}}})

IV. MySQL vs. MongoDB methods

To present the MongoDB methods we have installed and configured the latest database from the developers' site. The installation and configuration was done with easily in Ubuntu

X6412.4LST, the operation system we have chosen to use for both databases.

All SQL queries were done on the relational database management system MySQL 5.5. For a comprehensive overview we present the CRUD operations and the indexes' creation summarized in table 1.

TABLE 1 CRUD Operations

Operation	**MySQL**	**MongoDB**
Schema creation	CREATE SCHEMA `BlogDB`	mongo BlogDB
Table creation	CREATE TABLE IF NOT EXISTS `BlogDB`.`users` (`id` INT NOT NULL, `first_name` VARCHAR(64) NULL , `last_name` VARCHAR(45) NULL , PRIMARY KEY (`id`));	Creation at first insert:db.articles.insert ({ user_id: "1", first_name: "Ciprian", last_name: "Truica" }) Explicit: db.createCollection("articles")
Add a new column	ALTER TABLE USERS ADD JOIN_DATE DATETIME	Does not exist
Delete a column	ALTER TABLE USERS DROP COLUMN JOIN_DATE	Does not exist
Create an index	CREATE INDEX IDX1 ON USERS(first_name)	db.articles.ensureIndex({dirst_name : 1})
Create a compound index	CREATE INDEX IDX2 ON USERS(first_name, last_name)	db.articles.ensureIndex({ first_name : 1, last_name : -1 })
Drop a table	DROP TABLE USERS	db.articles.drop()
Insert	INSERT INTO USERS(id, first_name, last_name) VALUES (1, "Ciprian", "Truica")	db.articles.insert({ _id: "1", age: 45, status: "A" })
Select	SELECT * FROM USERS	db.articles.find()
Select fields	SELECT frist_name, last_name STATUS FROM USERS	db.articles.find({ }, { first_name: 1, last_name: 1 })
Select with where	SELECT u.first_nameFROM `BlogDB`.`users` AS uWHERE u.id = 1;	db.articles.find({user_id:"1"}, { "first_name" : 1});
Ordered Select ASC	SELECT * FROM USERS ORDER BY USER_ID ASC	db.articles.find({}).sort({user_id : 1})
Ordered Select DESC	SELECT * FROM USERS ORDER BY USER_IDDESC	db.articles.find({}).sort({user_id: -1 })
Select with count	SELECT COUNT(*) FROM USERS	db.articles.count()
Explain	EXPLAIN SELECT * FROM USERS	db.articles.find().explain()
Update	UPDATE `BlogDB`.`articles` SET title="MongoDB" WHERE id = 1;	db.articles.update({_id: "1"}, $set : { "article.title": "MongoDB" }}, {upsert: true});
Delete	DELETE FROM USERS	db.articles.remove( )
Delete using where	DELETE FROM USERS WHERE id="1"	db.articles.remove({ _id: "1" })
Delete a table from dictionary	DROP TABLE `BlogDB`.`articles`	db.articles.drop()

V. Conclusions

MongoDB is a very flexible, schema-less database that that can be implemented in a distributed architecture. "MongoDB was build for the cloud" developers boast. MongoDB can scale horizontally using Sharding. Data in a collection can split across multiple shards. Also MongoDB provides build in load balancing; data is duplicated to keep the system up and running in case of a failure. From the point of CRUD operations this fact is not seen, data will be manipulated the same and to interrogate a distributed MongoDB system will not need any other query methods. Indexes full potential is seen in a distributed system.

Their main role is to help read queries perform fast. Although adding secondary indexes build more overhead in storing documents their B-tree structure is very helpful of keeping track of data that is split and stored on different servers.

MongoDB supports master-slave replication. From the point of view of the CRUD operations they are not influenced in any way by the number of slaves servers a master server has. In MongoDB there is no use of a JOIN operation. Documents can be nested inside other documents. Using the no normalization encourages data redundancy, an idea not shared by most developers due to the fact that this can create confusion in a database regarding the way records are store. But using a schema-less design comes in handy when using CRUD operations; they are more natural to write and they are easier to understand at a first glance.

MongoDB is a more rapid database management system. If you want a simple database that will respond very fast, this is the choice you should make[3]. To achieve scalability and mush higher performance, MongoDB gave up ACID(Atomicity, Consistency, Isolation, Durability) transaction, having a weaker concurrency model implemented known as BASE (Basically Available, Soft state, Eventual consistency). This means that updates are eventually propagated to all nodes in due time.

In conclusion, if a developer wants to build a web application that is fast and flexible, than MongoDB is the right choice. If the application designer's main concern is the relation between data and to have a normalized database that uses ACID transactions, then the right choice is a classic relational database.

VI . Acknowledgments.

The research presented in this paper was partially performed under the European FP7 project ERRIC (http://www.erric.eu).

References

[1] E. Redmond and J. R. Wilson, "Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement", 2012

[2] K. Banker, "MongoDB in action", 2011

[3] A.Boicea, F. Rădulescu, and L.I. Agapin, "MongoDB vs. Oracle - database comparison", *The 3-rd Conference EIDWT*, Bucharest, 2012

[4] http://www.mongodb.org/