UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

ULB

mongoDB

{ name: mongo, type: DB }

ADVANCED DATABASES PROJECT

Juan Manuel Benítez V. - 000425944

Gledys Sulbaran - 000423426

TABLE OF CONTENTS

Contents

Introduction

The database systems can be classified into different types, some examples are: relational, object-oriented, relational and object-oriented. However, in practice, most database engines are based on the relational architecture and they use the SQL query language to work with the data. Due its use over the years SQL has become a standard "de facto" language for relational databases. Relational databases require a higher degree of Normalization i.e. decomposition of relations to eliminate redundancy [1] and furthermore, they require scalability in order for them to be expanded dependent upon varying demands.

Interactive applications have changed dramatically through the years and web companies have emerged with dramatic increases in their demands on data storage. As soon as Web companies and successful startups arrived with millions of users so did the requirement of high scalability.

Each time multiple JOINs are managed in SQL the process becomes increasingly complex; the amount of data collected and processed increases faster and it becomes more valuable to capture all types of data. When demand on data storage first started to increase relational database technology was behind and could not keep up with the requirement to process faster and capture more and more data. At that time the technology had limitations, which led to the requirement for high scalability becoming a problem.

The problem was solved with “NoSQL”, or non-relational databases systems by building flexible data models with higher scalability and higher performance. Web companies and other organisations recognised that to operate at scale it is more effective to run on cluster and schema-less data models.

For this project we chose to research one of the most famous NoSQL databases; MongoDB. Within this project we will explain how this NoSQL Database works, the benefits of using NoSQL, we will compare SQL vs. Mongo DB query operations and we will show some representatives queries in MongoDB.

What is NoSQL?

According to MongoDB 3.0 org, NoSQL encompasses a wide variety of different database technologies that were developed in response to; a rise in the volume of data stored about users, the amount of objects and products, the frequency in which this data is accessed, and increasing needs in performance and processing. Relational databases on the other hand, were not designed to cope with the scale and agility challenges that modern applications face, nor were they built to take advantage of the cheap storage and processing power available today.

NoSQL Databases Types

The primary difference between relational and non-relational databases is the way data is stored. Relational data is tabular by nature (it is stored in Tables with rows and columns). Non-relational data on the other hand, does not fit in tables of rows and columns; non-relational data is often stored as Collections, like in **documents, key-value pairs**, **graphs** or **Wide-column stores**.

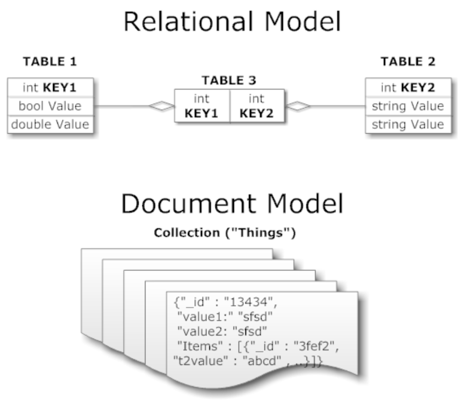
- Document databases pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents e.g. MongoDB and CouchDB.
- Graph stores are used to store information about networks, such as social connections e.g. Neo4j and Giraph.
- Key-value stores are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value e.g. Riak and Redis
- Wide-column stores are optimized for queries over large datasets and store columns of data together, instead of in rows e.g. HBase and Cassandra.

Why NoSQL?

Relational databases were not designed to deal with: [2]

- Large volumes of structured, semi-structured, and unstructured data.
- Agile sprints, quick iteration, and frequent code pushes.
- Object-oriented programming that is easy to use and flexible.
- Efficient, scale-out architecture instead of expensive, monolithic architecture.

Google, Amazon, Facebook and LinkedIn were among the first companies to discover the serious limitations of relational database technology for supporting their new application requirements. A RDBMS is not always the best solution as it cannot meet the increasing growth of unstructured data. As data processing requirements have grown exponentially, NoSQL has become a dynamic and cloud friendly approach to dynamically process unstructured data with ease.



NoSQL vs. SQL

To make a clearly understandable comparison we found it useful to look at the general differences between SQL and NoSQL in the following table [3, 4]:

	SQL Databases	**NoSQL Databases**
Types	One type – SQL databases	There are many different types. The main types are: Key-value stores, document databases, wide-column stores, graph databases
Schema and flexibility	*"Structure and data types are fixed in advance"* this means the columns must be decided and locked before data entry and each row must contain data for each column. In order to store new information the entire database must be altered, during which time the database must be taken offline.	Schemas are dynamic. Applications can add new fields on the fly, and each 'row' (or equivalent) doesn't have to contain data for each 'column'.
Scaling	Vertically, meaning that to store more data a bigger server is needed which can get very expensive. It is possible to scale an RDBMS across multiple servers, but this is a difficult and time-consuming process.	Horizontally, meaning across servers i.e. to add capacity, a database administrator can simply add more commodity servers or cloud instances.
ACID Compliancy	The majority of relational databases are ACID compliant.	It depends on product, but many NoSQL solutions sacrifice ACID compliancy for performance and scalability

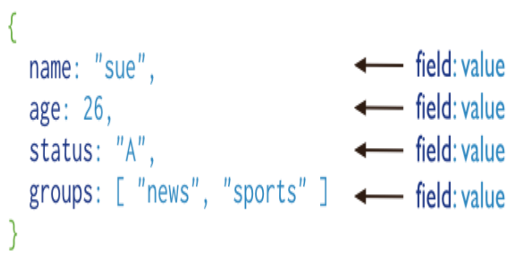
The advantages of using NoSQL databases are evident, now we will look in more detail at one of the most popular document based NoSQL databases, MongoDB. MongoDB is the most downloaded NoSQL database with over 10 million downloads and hundreds of thousands of deployments.

Mongo DB - Introduction

According to the official MongoDB website [3] it is an open-source document database that provides high performance, high availability and automatic scaling. MongoDB avoids the need for Object Relational Mapping (ORM) to facilitate development.

In other words, MongoDB is a flexible multiplatform data documents scheme oriented database system. This means that every registration scheme may have different data attributes or "columns" that need not be repeated from one record to another.

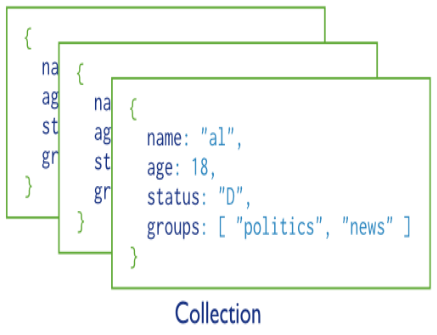
To understand how MondoDB works we can start with these simple concepts:



```
{
  name: "sue",                      ← field: value
  age: 26,                          ← field: value
  status: "A",                      ← field: value
  groups: [ "news", "sports" ]      ← field: value
}
```

Image Credit 1: ***MongoDB.org***

MongoDB documents are BSON documents. BSON is a binary representation of JSON with additional type information. In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents.



```
{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

Image Credit 2: MongoDB.org

MongoDB stores all documents in **collections**. A collection is a group of related documents that have a set of shared common indexes. Collections are analogous to a table in relational databases. In MongoDB, documents stored in a collection must have a unique **_id field** that acts as a primary key.

In summary, **MongoDB** is made up of databases which contain **collections**. A collection is made up of **documents**. Each document is made up of **fields**. Collections can be **indexed**, which improves lookup and sorting performance. Finally, when we get data from MongoDB we do so through a cursor whose actual execution is delayed until necessary.

Installation - Connection to MongoDB

For this project we installed MongoDB on a Windows system. However, you can install MongoDB in OS X and various Linux systems. Using the most popular Linux distributions such as Red Hat, SUDE, Amazon Linux, Ubuntu or Debian is recommended for the best installation experience.

Installing MongoDB

- Firstly, download MongoDB according to your Windows version (Refer to the link below):

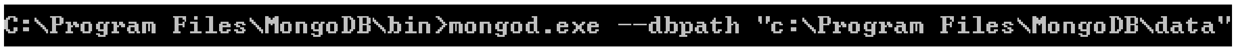
https://www.mongodb.org/downloads?_ga=1.205285456.174827563.1446822567#production

- Secondly, double-click the downloaded MongoDB .msi file to start the installation. If you want to specify the installation directory, then you have to select the "Custom" installation. MongoDB can be installed in any folder due to it being self-contained and not having any system dependencies.

Run MongoDB

1. Setting up the environment

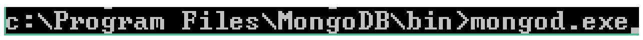
MongoDB requires a directory to store data. The default data directory is **c:\data\db** (this directory has to be manually created). The path for data storage can be specified using the --*dbpath* option to **mongodb.exe** from the **Command Prompt**:



```
C:\Program Files\MongoDB\bin>mongod.exe --dbpath "c:\Program Files\MongoDB\data"
```

2. Starting MongoDB

To start MongoDB all you need to do is run mongod.exe. For this you can use the **Command Prompt**:



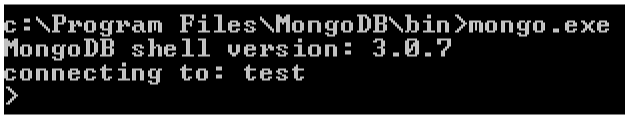
```
c:\Program Files\MongoDB\bin>mongod.exe_
```

If the message "waiting for connections" appears, it means the process is running successfully.

It is possible to get a **Security Alert** dialog box about network issues. For security documentation refer to https://docs.mongodb.org/manual/security/

3. Connecting to MongoDB

To connect to MongoDB you just have to run **mongo.exe**:



```
c:\Program Files\MongoDB\bin>mongo.exe
MongoDB shell version: 3.0.7
connecting to: test
>
```

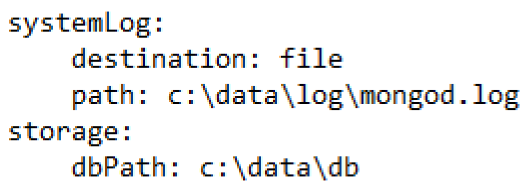
Configure a Windows Service for MongoDB

- Make sure the directories for your database and log files are created:

  c:\data\db

  c:\data\log

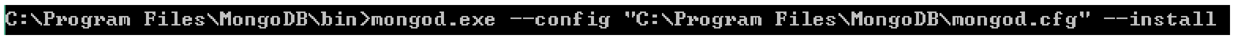
- **Create a configuration file**. The file must set *systemLog.path*. You can include additional information to this file when necessary. The configuration file should look like this:



```
systemLog:
    destination: file
    path: c:\data\log\mongod.log
storage:
    dbPath: c:\data\db
```

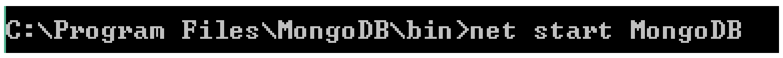
  This information is stored in the file C:\Program Files\MongoDB\mongod.cfg, which specifies both *systemLog.path* and *storage.dbPath*.

- **Install the MongoDB service** by starting **mongod.exe** with the --*install* option and the --*config* option so you can specify the previously mentioned configuration file:



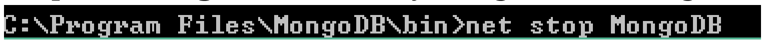
```
C:\Program Files\MongoDB\bin>mongod.exe --config "C:\Program Files\MongoDB\mongod.cfg" --install
```

- **Start the MongoDB service** by using the following command:



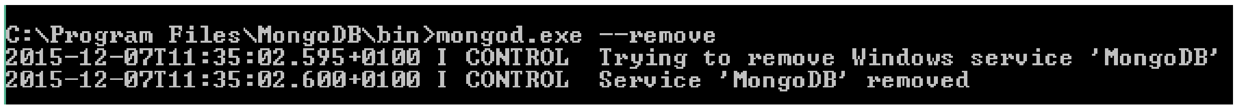
```
C:\Program Files\MongoDB\bin>net start MongoDB
```

- **Stop the MongoDB service** by using the following command:



```
C:\Program Files\MongoDB\bin>net stop MongoDB
```

- **Remove the MongoDB service** by using the following command:



```
C:\Program Files\MongoDB\bin>mongod.exe --remove
2015-12-07T11:35:02.595+0100 I CONTROL  Trying to remove Windows service 'MongoDB'
2015-12-07T11:35:02.600+0100 I CONTROL  Service 'MongoDB' removed
```

IMPORT EXAMPLE DATASET

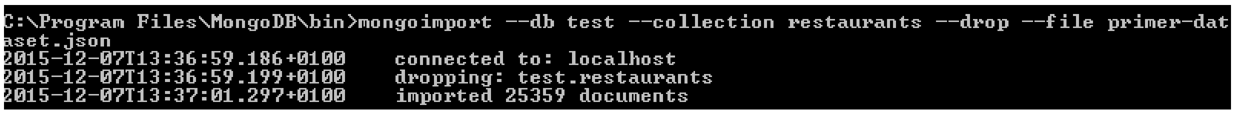
Before importing data into the database, it is necessary to have a running MongoDB instance.

Procedure:

1. Retrieve the sample dataset from: https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/dataset.json

   Save the data in a file named *primer-dataset.json*.

2. In order to import this data into the *test* database the **mongoimport** command can be used either in the **system shell** or the **command prompt**:



```
C:\Program Files\MongoDB\bin>mongoimport --db test --collection restaurants --drop --file primer-dat
aset.json
2015-12-07T13:36:59.186+0100    connected to: localhost
2015-12-07T13:36:59.199+0100    dropping: test.restaurants
2015-12-07T13:37:01.297+0100    imported 25359 documents
```

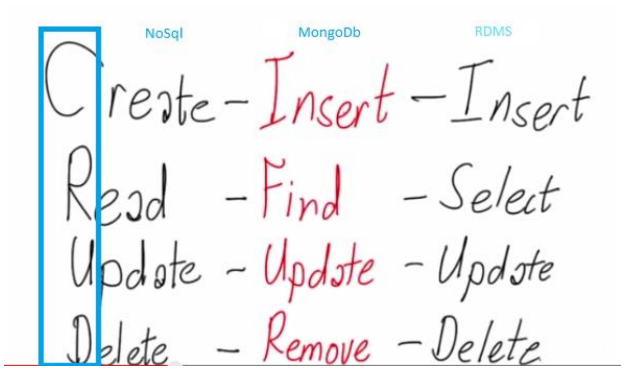
After executing this instruction, the system shows a summary of the action. In this specific case, the collection *restaurants* is dropped if a collection with the same name already exists.

The **mongoimport** connects to a **mongod** instance running on localhost on port number 27017.

In order to import data into a **mongod** instance running on a different host or port, specify the hostname or port by including the --host and the --port options in your **mongoimport** command.

SQL vs. Mongo DB CRUD operations

In MongoDB a query targets a specific collection of documents. Queries specify criteria or conditions that identify the documents that MongoDB returns to the clients. A query may include a projection that specifies the fields from the matching documents to return. You can optionally modify queries to impose limits, skips and sort orders.



The following table compares some SQL operations with Mongo DB operations; the SQL statements are presented alongside the corresponding MongoDB statements:

SQL Schema Statements	MongoDB Schema Statements
Create, Insert, Update: Table-level actions and the corresponding MongoDB statements.	

`CREATE TABLE users (`
`    id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,`
`    user_id Varchar(30),`
`    age INT,`
`    status char(1)`
`)`
 	
`INSERT INTO users(user_id,age,status)`
`VALUES ('abc123',55,'A')`
In this case, we specified **id** as primary key.
	`db.createCollection("users")`
	`db.users.insert( {`
	`    user_id: "abc123",`
	`    age: 55,`
	`    status: "A"`
	` } )`
	
	The primary key **_id** is automatically added if **_id** field is not specified.

`ALTER TABLE users`
`ADD join_date DATETIME`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	
	However, at the document level `update()` operations can add fields to existing documents using the $set operator.
	
	`db.users.update(`
	`    { },`
	`    { $set: { join_date: new Date() } },`
	`    { multi: true }`
	`)`

Index
`CREATE INDEX idx_user_id_asc`
`ON users(user_id)`
	`db.users.createIndex( { user_id: 1 } )`

`DROP TABLE users`	`db.users.drop()`
SELECT: Reading records from tables in SQL (`SELECT`) MongoDB statements (`db.collection.find()`).	

`SELECT *`
`FROM users`
	`db.users.find()`

`SELECT user_id, status`
`FROM users`
`WHERE status = 'A'`
	`db.users.find(`
	`{ status: "A" },`
	`{ user_id: 1, status: 1, _id: 0 })`

`SELECT *`
`FROM users`
`WHERE status = 'A'`
`OR age = 50`
	`db.users.find(`
	`{ $or: [ { status: "A" } ,`
	`{ age: 50 } ] })`

`SELECT *`
`FROM users`
`WHERE age > 25`
`AND age <= 50`
	`db.users.find(`
	`{ age: { $gt: 25, $lte: 50 } })`

`SELECT COUNT(user_id)`
`FROM users`
	`db.users.find( { user_id: { $exists: true } } ).count()`

`SELECT COUNT(*)`
`FROM users`
`WHERE age > 30`
	`db.users.find( { age: { $gt: 30 } } ).count()`

`SELECT *`
`FROM users`
`WHERE status = "A"`
`ORDER BY user_id DESC`
	`db.users.find( { status: "A" } ).sort( { user_id: -1 } )`

Update Records: Updating existing records in tables	

`UPDATE users`
`SET status = 'C'`
`WHERE age > 25`
	`db.users.update(`
	`{ age: { $gt: 25 } },`
	`{ $set: { status: "C" } },`
	`{ multi: true })`

Delete Records	

`DELETE FROM users`
`WHERE status = 'D'`
 	
`DELETE FROM users`
	`db.users.remove( { status: "D" } )`
	
	`db.users.remove({})`

Where to use MongoDB instead of RDBMS?

Any application that needs to store semi-structured data can use MongoDB. Organizations are adopting MongoDB due to it enabling them to build applications **faster**, handle highly **diverse data types** and manage applications more **efficiently at scale**.

- In order to facilitate **fast development**, MongoDB documents map naturally to object-oriented programming languages.
- When applications make significant changes in real time they must integrate seamlessly with **diverse data types,** MongoDB's flexible data model enables database schema to evolve with business requirements.
- When your deployments grow in terms of data volume and throughput, MongoDB scales easily with no downtime and without changing your application; this is simply unachievable with relational databases.

Some of the most common uses of MongoDB include: Single View, Internet of Things, Mobile, Real-Time Analytics, Personalization, Catalog, and Content Management.

Representative Queries

In this section a list of typical queries in MongoDB will be presented, the queries are written using **Java**, so a basic knowledge in this programming language is mandatory. The dataset used (*Restaurants* collection) is the same one presented in the **Installation section**. The next document is a sample in *Restaurants* collection:

```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

In order to test the following queries using Java, it is necessary to install the MongoDB Java driver. The jar file (library to be added) can be found in the following link. The name of the required file is **mongo-java-driver-3.0.4.jar**.

https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongo-java-driver/3.0.4/

Before executing the next code snippets, a connection to a MongoDB instance must be done. The way to connect is the following:

```
MongoClient mongoClient = new MongoClient();
MongoDatabase db = mongoClient.getDatabase("test");
```

In this case, **test** is the name of the used database. Here is assumed the MongoDB service is properly running.

1. Adding a new document to the *Restaurants* collection:

```
DateFormat    format    =    new    SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'Z'",
Locale.ENGLISH);
              db.getCollection("restaurants").insertOne(
                      new Document("address",
                              new Document()
                                      .append("street", "2 Avenue")
                                      .append("zipcode", "10075")
                                      .append("building", "1480")
                                      .append("coord",     asList(-73.9557413,
40.7720266)))
                              .append("borough", "Manhattan")
                              .append("cuisine", "Italian")
                              .append("grades", asList(
                                      new Document()
                                              .append("date",
format.parse("2014-10-01T00:00:00Z"))
                                              .append("grade", "A")
                                              .append("score", 11),
                                      new Document()
                                              .append("date",
format.parse("2014-01-16T00:00:00Z"))
                                              .append("grade", "B")
                                              .append("score", 17)))
                              .append("name", "Vella")
                              .append("restaurant_id", "41704620"));
```

It is possible to use the method **insertMany** instead of the method **insertOne**. In such a case, one or more documents will be inserted, and a call to the **insertMany** method would be equivalent to a bulk write operation.

2. Return all documents in the *Restaurants* collection:

```
// Retrieve all documents without using a specific criteria:
FindIterable<Document> iterable = db.getCollection("restaurants").find();

// Iterate the results and apply an operation to each resulting document
iterable.forEach(new Block<Document>() {
       @Override
              public void apply(final Document document) {
                     System.out.println(document);
              }
       });
```

The block applied to each document can contain multiple kinds of operations, for example; one could print the name of each restaurant instead of the document itself. This could be done in the following way: `document.getString("name")`

3. Query by a top-level field, for example; return all documents in the *Restaurants* collection whose borough is *Brooklyn*.

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(
            new Document("borough", "Brooklyn"));
```

The obtained results can be iterated as shown in the previous point.

In this case the **find** method receives a document as a parameter indicating the criteria. In general, to implement equality conditions the following code is used: `new Document ( <field>, <value> )`. The dot notation must be used whenever the <field> is an embedded document or an array.

Note: The Java driver provides the Filters class to help specify the query condition. It contains different static methods to simplify building the query predicate, for example, the **eq** method can be used in the previous example as follows (the result will be the same):

```
db.getCollection("restaurants").find(Filters.eq("borough", "Brooklyn"));
```

4. Query by a field in an embedded document, for example; return all documents in the *Restaurants* collection whose zipcode is *10075*.

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(
                  new Document("address.zipcode", "10075"));
```

Similarly, the Filters class can be used as follows:

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(
             Filters.eq("address.zipcode", "10075"));
```

It is important to notice the dot notation (“address.zipcode”) must be used because it has an embedded document in the <field>.

5. Query by a field in an Array, for example; return all documents in the *Restaurants* collection whose grades array contains an embedded document with a field grade equal to “A”.

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(
                        new Document("grades.grade", "A"));
```

Similarly, the Filters class can be used as follows:

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(
                              Filters.eq("grades.grade", "A"));
```

6. Specify conditions with operators, for example; return all documents whose grades array contains an embedded document with a field score greater than 35.

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(
                new Document("grades.score", new Document("$gt", 35)));
```

In this case, the **greater than operator** is used **($gt)**, query conditions using operators generally have the following form:

```
new Document ( <field>, new Document( <operator>, <value> ) )
```

However, it is important to keep in mind there are some exceptions, such as the $or and $and conditional operators.

Similarly, the Filters class can be used as follows:

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(
                         Filters.gt("grades.score", 35));
```

The **less than operator ($lt)** can be used in the same way as the **greater than operator** was presented.

7. Combine conditions, for example; return all documents whose cuisine is "Mexican" AND zipcode is "10016".

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(

new Document("cuisine","Mexican").append("address.zipcode", "10016"));
```

In this example a logical conjunction (AND) is specified for multiple query conditions by appending conditions to the query document.

Similarly, the Filters class can be used as follows:

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(

Filters.and(Filters.eq("cuisine", Mexican"),
Filters.eq("address.zipcode", "10016")));
```

8. Combine conditions, for example; return all documents whose cuisine is "Mexican" OR zipcode is "10016".

```
 FindIterable<Document> iterable = db.getCollection("restaurants").find(

new Document("$or", asList(new Document("cuisine", "Mexican"),

        new Document("address.zipcode", "10016"))));
```

In this example a logical disjunction (OR) is specified by using the **$or** query operator.

Similarly, the Filters class can be used as follows:

```
FindIterable<Document> iterable = db.getCollection("restaurants").find(

Filters.or(Filters.eq("cuisine", "Mexican"),
Filters.eq("address.zipcode", "10016")));
```

9. Sort query results, for example; return all documents sorted first by borough in ascending order, and then sorted in ascending order by zip code within each borough.

```
FindIterable<Document> iterable = db.getCollection("restaurants").find()
.sort(new Document("borough", 1).append("address.zipcode", 1));
```

This example shows the **sort** method for sorting the results set. This method receives a document as a parameter which contains the fields to sort by and the respective sort type, e.g. 1 for ascending and -1 for descending.

In respect of sorting, the Java driver provides the Sorts class, which contains static methods to simplify the query.

Therefore, the Sorts class can be used as follows:

```
FindIterable<Document> iterable = db.getCollection("restaurants").find()
.sort(Sorts.ascending("borough", "address.zipcode"));
```

10. Update a top-level field, for example; update the **cuisine** field and the **lastModified** field of the first document whose name is “Juni”.

```
UpdateResult result =
db.getCollection("restaurants").updateOne(new Document("name", "Juni"),
 new Document("$set", new Document("cuisine", "American (New)"))
                  .append("$currentDate", new Document("lastModified",
true)));
```

In this example the **updateOne** method is used. The first parameter is the matching criteria of the document to modify (name is equal to “Juni”). In the second parameter the **cuisine** field is modified to “American (New)” by using the **$set** operator, then the **lastModified** field is updated with the current date by using the **$currentDate** operator. Some update operators such as $set will create the field, if the field does not exist. The **updateOne** method returns an **UpdateResult** which contains information about the operation.

11. Update an embedded field, for example; update the **street** field in the embedded **address** document.

```
UpdateResult result =

db.getCollection("restaurants").updateOne(new Document("restaurant_id",
"41156888"),

new Document("$set", new Document("address.street", "East 31st Street")));
```

Once again, when embedded documents are manipulated it is necessary to use the dot notation.

12. Update multiple documents, for example; for all documents whose **zipcode** is “10016” and **cuisine** is “Other”, set the **cuisine** field to “Category To Be Determined” and the **lastModified** field to the current date.

```
UpdateResult result =
db.getCollection("restaurants").updateMany(new
Document("address.zipcode","10016")
       .append("cuisine", "Other"),
       new   Document("$set",   new   Document("cuisine",   "Category   To   Be
Determined"))
                    .append("$currentDate",   new   Document("lastModified",
true)));
```

In order to update all the documents matching the specified criteria the **updateMany** method is used. When necessary the **getModifiedCount** method can be used to obtain the number of documents modified.

13. Replace an entire document except for the **_id** field.

```
UpdateResult result =
db.getCollection("restaurants").replaceOne(new Document("restaurant_id",
"41704620"),
                    new Document("address",
                            new Document()
                                   .append("street", "2 Avenue")
                                   .append("zipcode", "10075")
                                   .append("building", "1480")
                                  .append("coord", asList(-73.9557413,
40.7720266)))
                            .append("name", "Vella 2"));
```

To do the replacement it is necessary to pass a new document as the second parameter of the **replaceOne** method. The new document can contain different fields from the original one. However, keep in mind that the new document will not contain any of the old fields and it will only keep the information passed to the **replaceOne** method. On the other hand, it is not mandatory to specify the **_id** field since it cannot be changed. In the case that it is included, then it has to be the same one as in the old document.

14. Remove all documents that match a condition, for example; delete the documents whose **cuisine** field is "Brazilian".

```
DeleteResult result =
db.getCollection("restaurants").deleteMany(new Document("cuisine",
"Brazilian"));
```

In order to remove documents the **deleteMany** method can be used. This method returns an **UpdateResult** which contains information about the operation. When necessary the **getDeletedCount** method can be used to obtain the number of documents deleted.

However, the **deleteOne** method will only remove the first document that matches the condition.

15. Remove all documents in a collection.

```
DeleteResult result = db.getCollection("restaurants").deleteMany(new
Document());
```

In order to remove all documents the **deleteMany** method must receive an empty conditions document as a parameter.

16. Drop a collection.

```
db.getCollection("restaurants").drop();
```

Dropping a collection may be more efficient than deleting all the documents in it. It is important to take into account that the **drop** method will not only remove the collection itself, but also all the indexes for the collection, which is not the same result as when deleting all the documents in a collection.

17. Aggregate documents, for example; calculate the amount of documents by **cuisine** field.

```
// Retrieve the amount of restaurants by cuisine
AggregateIterable<Document>                 iterable              =
db.getCollection("restaurants").aggregate(asList(
       new Document("$group", new Document("_id", "$cuisine")
       .append("count", new Document("$sum", 1)))));

// Iterate the result set and apply a bock to each document
iterable.forEach(new Block<Document>() {
    @Override
    public void apply(final Document document) {
        System.out.println(document.toJson());
    }
});
```

The **$group** stage is used to group by a specific key, the group by key is specified by the **_id** field. The **$group** accesses fields by the field path, which is the field name prefixed by a dollar sign **$**. Like the aggregation functions in SQL, in MongoDB the **$group** stage can use **accumulators** in order to perform calculations for each group. In this example the **$sum** accumulator is used to count the documents for each group of cuisine.

18. For those documents whose **borough** is “Manhattan” and **cuisine** is “Italian”, calculate the amount of documents by **zipcode**.

```
 // Retrieve the amount of restaurants by zipcode,
 // only for the documents whose borough is Manhattan and cuisine is Italian
              AggregateIterable<Document>             iterable             =
db.getCollection("restaurants").aggregate(asList(
       new Document("$match", new Document("borough","Manhattan")
              .append("cuisine", "Italian")),
       new Document("$group", new Document("_id", "$address.zipcode")
              .append("count", new Document("$sum", 1)))));

// Iterate the result set and apply a block to each document
              iterable.forEach(new Block<Document>() {
                  @Override
                  public void apply(final Document document) {
                      System.out.println(document.toJson());
                  }
              });
```

Unlike the previous example, a filter must be done before grouping the documents. To filter the documents the **$match** stage can be used. The **$match** uses the MongoDB query syntax, and the grouping is done in the same way as mentioned in example 17.

19. Create a single field index.

```
db.getCollection("restaurants").createIndex(new Document("cuisine", 1));
```

An index can be created by using the **createIndex** method. This method receives an index key specification document as a parameter, which contains the fields to index and the index type for each field (**1** is ascending index type and **-1** is desceding index type). In this example the only field to index is **cuisine,** and its type is ascending (1). The **createIndex** method only creates an index when an index does not exist.

20. Create a compound index.

```
db.getCollection("restaurants").createIndex(new         Document("cuisine",1)
       .append("address.zipcode", -1));
```

As shown in this example, MongoDB allows the creation of indexes in multiple fields. The order of the fields determines how the index stores its keys. In this example the index stores its entries by ascending **cuisine** values. Inside each cuisine group the entries are stored by descending **address.zipcode** values.

References

[1] U. W. Garcia-Molina, DATABASE SYSTEMS The Complete Book, New Jersey: Pearson Prentice Hall, 2009.

[2] MongoDB, "Top 5 Considerations When Evaluating," *A MongoDB White Paper,* p. 5, 2015.

[3] "Official Mongo Website," MongoDB, 2015. [Online]. Available: www.mongodb.com.

[4] S. Edlich, "The NoSQL eMag," *InfoQ.com,* 2013, May.

[5] K. Seguin, The little MongoDB Book 2.6, www.MongoDB.org.