The NoSQL movement is growing in relevance, attracting more and more developers. The MongoDB database is a well-recognized rising star in the NoSQL world. It is a document database that allows data persistence and enables you to query data in a nested state without any schema constraints and complex joins between documents.

This book provides all the knowledge you need to make MongoDB fit into your application schema. It starts with a basic introduction to the driver that can be used to perform some low-level interaction with the storage. Then it moves on to using different patterns to abstract the persistence layer into your applications, starting with the flexible Google JSON library to the Hibernate OGM framework and finally landing on the Spring Data framework.

By the end of this book, you will know everything you need to integrate MongoDB in your Java applications.

Who this book is written for
This book is for Java developers and architects who want to learn how to develop Java applications using the most popular NoSQL solution and the use cases of it.

What you will learn from this book
- Install MongoDB and its client tools
- Set up a basic project that uses a MongoDB driver and performs CRUD operations with it
- Explore simple strategies for mapping MongoDB documents with Java classes
- Use bulk operations to speed up the creation of massive documents
- Design and develop a Java Enterprise application that uses MongoDB as storage
- Develop and deploy an application that uses Hibernate OGM as a persistence layer for your entities
- Use Spring Data and Spring Boot to leverage micro-services using MongoDB as storage
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'MongoDB CRUD Beyond the Basics'
- A synopsis of the book’s content
- More information on MongoDB for Java Developers
About the Author

Francesco Marchioni is a Red Hat Certified JBoss Administrator (RHCJA) and a Sun Certified enterprise architect working as a freelancer in Rome, Italy. He started learning Java in 1997, and since then, he has followed the path to the newest application program interfaces released by Sun. In 2000, he joined the JBoss community, when the application server was running the release 2.X.

He has spent many years as a software consultant, wherein he envisioned many successful software migrations from vendor platforms to open source products such as JBoss AS, fulfilling the tight budget requirements of current times.

Over the past 5 years, he has been authoring technical articles for OReilly Media and running an IT portal focused on JBoss products (http://www.mastertheboss.com).


In December 2011, he published yet another title, JBoss AS 7 Configuration, Deployment, and Administration, which covers all the aspects of the newest application server release (http://www.packtpub.com/jboss-as-7-configuration-deployment-administration/book).

In June 2013, he authored a new title, JBoss AS 7 Development, which focuses on developing Java EE 6 API applications on JBoss AS 7 (https://www.packtpub.com/application-development/jboss-7-development).
The NoSQL movement is growing in relevance, and it is attracting more and more developers. The MongoDB database is a well-recognized rising star in the NoSQL world. It is a document database that allows data to persist and query data in a nested state without any schema constraint and complex joins between documents. Understanding when it is appropriate to use MongoDB against a relational database and the interfaces to be used to interact with it requires some degree of experience.

This book provides all the knowledge to make MongoDB fit into your application schema, at the best of its capabilities. It starts from a basic introduction to the driver that can be used to perform some low level interaction with the storage. Then it moves on to use different patterns for abstracting the persistence layer into your applications, starting from the flexible Google JSON library, to the Hibernate OGM framework, and finally landing on the Spring Data framework.

**What this book covers**

*Chapter 1, Introduction to MongoDB*, covers the installation steps of MongoDB and its client tools and how to use the Mongo shell to perform basic database operations.

*Chapter 2, Getting Started with Java Driver for MongoDB*, introduces the Java Driver for MongoDB using a simple Java project developed with the NetBeans development environment.

*Chapter 3, MongoDB CRUD Beyond the Basics*, covers the advanced usage of the MongoDB Java driver such as data mapping, index creation, and bulk operations.

*Chapter 4, MongoDB in the Java EE 7 Enterprise Environment*, demonstrates how to create and deploy a Java Enterprise application that uses MongoDB as the storage.
Chapter 5, *Managing Data Persistence with MongoDB and JPA*, covers the development of a Java Enterprise application using Hibernate Object/Grid Mapper (OGM), which provides Java Persistence API (JPA) support for NoSQL databases.

Chapter 6, *Building Applications for MongoDB with Spring Data*, teaches you how to use Spring Data and Spring Boot to leverage micro services using MongoDB as the storage.
The previous chapter of this book took you through the first green bar in connecting Java and MongoDB. You learned how to perform some basic create, read, update, and delete operations using simple Java classes. It is now time to address some advanced concerns, which are part of every real work application. Here is what we are going to discuss in this chapter in detail:

- How to map MongoDB documents in Java objects and vice versa
- How to apply indexes to your documents to speed up searches
- How to code bulk operations to improve the speed of your insert/updates

Seeing MongoDB through the Java lens

So far, we have had some interaction with the Java driver using `com.mongodb.DBCollection` as a simple translator between the Java objects and the MongoDB documents:

```java
DBCursor cursor = coll.find();

try {
    while(cursor.hasNext()) {
        DBCollection object = cursor.next();
        System.out.println(object.get("username"));
    }
} finally {
    cursor.close();
}
```
When you move from the basics to a more complex project, you will find that this approach requires writing lots of code and it is prone to runtime errors.

There are some solutions to this problem with different degrees of complexity. In this chapter, we will account for some simple ones that require a minimal learning curve. Later on, in Chapter 5, Managing Data Persistence with MongoDB and JPA, we will describe how to use some frameworks that can let you persist Java objects directly into MongoDB as documents, at the price of some enhanced complexity.

Here is what we are going to learn in the next section:

- Extending the MongoDB core classes to save custom objects into the database.
- Using a Java library to translate Mongo documents into Java objects (and vice versa) via JSON.

**Extending the MongoDB core classes**

The first approach requires that you either implement `com.mongodbDBObject` (and provide some default implementation of its core methods) or directly extend `com.mongodb.BasicDBObject`.

![Diagram](image)

As you can see from the preceding diagram, the first choice is more flexible; however, you need to provide some default implementation for the basic methods of the collections mapped by your POJO. (For the sake of simplicity, only the two most common methods, **put** and **get**, are indicated in the diagram.)
Let's see a minimal implementation of a Java class SimplePojo, which implements com.mongodbDBObject:

```java
package com.packtpub.mongo.chapter3;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.bson.BSONObject;
import com.mongodbDBObject;

public class SimplePojo implements DBObject {
    private Map<String, Object> data;
    private boolean partial;

    public SimplePojo() {
        data = new HashMap<>();
        partial = false;
    }

    @Override
    public Object put(String key, Object value) {
        return data.put(key, value);
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    @Override
    public void putAll(BSONObject o) {
        data.putAll(o.toMap());
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    @Override
    public void putAll(Map m) {
        data.putAll(m);
    }

    @Override
    public Object get(String key) {
```

[59]
return data.get(key);
}

@SuppressWarnings("rawtypes")
@Override
public Map toMap() {
    return data;
}

@Override
public Object removeField(String key) {
    return data.remove(key);
}

@Override
public boolean containsKey(String key) {
    return data.containsKey(key);
}

@Override
public boolean containsField(String key) {
    return data.containsKey(key);
}

@Override
public Set<String> keySet() {
    return data.keySet();
}

@Override
public void markAsPartialObject() {
    partial = true;
}

@Override
public boolean isPartialObject() {
    return partial;
}
As you can see, we had to provide a default implementation for the methods specified in the \texttt{com.mongodb.DBOBJECT} interface. Now we'll insert our \texttt{SimplePojo} class directly into our collection, as follows:

```java
DB db = mongoClient.getDB( "sampledb" );
DBCollection coll = db.getCollection("pojo");

SimplePojo obj = new SimplePojo();
obj.put("user", "user1");
obj.put("message", "message");
obj.put("date", new Date());

coll.insert(obj);
```

Retrieving the Java class from the database is straightforward as well. First you need to call \texttt{setObjectClass} on your collection to state that you are going to retrieve objects of that type. Then you can use the finder methods of the collection as usual:

```java
coll.setObjectClass(SimplePojo.class);

SimplePojo tw = (SimplePojo)coll.findOne();
System.out.println(tw.get("user"));
```

The major downside of this approach is that you have to provide some boilerplate code with a default implementation of the \texttt{com.mongodb.DBOBJECT} class. As an alternative, you can consider extending the class \texttt{com.mongodb.BasicDBObject}, which already contains a \texttt{com.mongodb.DBOBJECT} default implementation. This will avoid writing boilerplate code, at the price of lack of flexibility in your code. As a matter of fact, you will not be able to extend any other class from your code.

Here is a rewritten version of \texttt{SimplePojo} that extends \texttt{com.mongodb.BasicDBObject} and merely contains a business method to return an uppercased version of a key requested:

```java
package com.packtpub.mongo.chapter3;

import com.mongodb.BasicDBObject;

public class SimplePojo extends BasicDBObject {

public String getUpperCaseKey(String key) {
    String value = (String) super.get(key);
    if (value != null)
        return value.toUpperCase();

```
In terms of implementation, nothing will change, and you can pass your Java classes to the `insert` method of your collection, as you already know:

```java
SimplePojo pojo = new SimplePojo();
pojo.put("user", "user2");
pojo.put("message", "msg");
pojo.put("date", new Date());

coll.insert(pojo);
```

### Using the Gson API with MongoDB

Using JSON as a medium between Java and external system is a well-tested integration pattern. There are several libraries available to serialize and deserialize Java classes in JSON, the most popular one being Google's Gson (https://code.google.com/p/google-gson/). This API provides two simple `toJson()` and `fromJson()` constructs to convert Java objects to JSON and vice versa; besides this, Gson supports converting arbitrarily complex objects including deep inheritance hierarchies and makes extensive use of Java's generic types.

#### Downloading the Gson API

You can download the latest release of Gson from the Maven central repository at http://search.maven.org/#browse%7C472424538.

Include the JAR driver of Gson in the libraries of your project, as displayed in the following screenshot:
If you are using Maven, then you can include the following dependency in your pom.xml file (see the next chapter for more details about using Maven in your projects):

```xml
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.3.1</version>
</dependency>
```

**Using Gson to map a MongoDB document**

In the first example, we will map a MongoDB document with a Java class composed of a set of fields. So first, let's create a document that contains some basic keys:

```java
MongoClient mongoClient = new MongoClient("localhost", 27017);

DB db = mongoClient.getDB("sampledb");

DBCollection coll = db.getCollection("javastuff");

DBObject doc = new BasicDBObject("name", "owen")
    .append("age", 25)
    .append("email", "owen@gmail.com")
    .append("phone", "321-456-778");

coll.insert(doc);
```

As evident from the following document query, a document has been created in the javastuff collection:

```shell
> db.javastuff.find().pretty()
{
    "_id" : ObjectId("55266979d3d5368080f97f92"),
    "name" : "owen",
    "age" : 25,
    "email" : "owen@gmail.com",
    "phone" : "321-456-778"
}
```

Now let's create a simple Java bean that will be able to contain this document structure:

```java
public class Customer {
    String name;
```
int age;
String email;
String phone;

public Customer(String name, int age, String email, String phone) {
  super();
  this.name = name;
  this.age = age;
  this.email = email;
  this.phone = phone;
}

@Override
public String toString() {
  return "Customer [name=" + name + ", age=" + age + ", email=" + email + ", phone=" + phone + "]";
}

Please note that it is not necessary to use any annotations to indicate that a class field is to be included for serialization and deserialization. By default, all fields contained in the class (and its superclasses) will be serialized into Java objects. In the Custom field names in your Java classes section, we will show you how to use annotations to map the field with custom names.

Being based on simple Java types such as string and int, mapping the MongoDB document to the Customer Java class is a piece of cake. Let’s see how to do it:

```java
Gson gson = new Gson();
DBasicDBObject doc = new BasicDBObject("name", "owen");

DBObject obj = coll.findOne(doc);
Customer c = gson.fromJson(obj.toString(), Customer.class);
System.out.println("Found customer " + c);
```

The expected output will be the toString() method of the Customer class that dumps the fields contained in the class:

```
Customer [name=owen, age=25, email=owen@gmail.com, phone=321-456-778]
```
Inserting Java objects as a document

The Google's Gson API can also leverage the reverse process, that is, inserting a Java class into MongoDB via JSON. The trick is done by the `toJson` method that serializes a Java class into the JSON format:

```java
DB db = mongoClient.getDB("sampledb");

DBCollection coll = db.getCollection("javastuff");

System.out.println("Collection created successfully");

Customer c = new Customer("john", 22, "john@gmail.com", "777-666-555");

Gson gson = new Gson();
String json = gson.toJson(c);

DBObject dbObject = (DBObject) JSON.parse(json);

coll.insert(dbObject);
```

In the above example, the JSON string mapping the `Customer` class is stored in the string `JSON`. You can then use the static `parse` method of the `com.mongodb.util.JSON` class to convert the JSON string into a `DBObject` type.

The inserted structure will be as follows:

```json
> db.javastuff.find({"name":"john"}).pretty()
{
   
   "_id" : ObjectId("55268359d3d51c80b0b231b5"),
   "name" : "john",
   "age" : 22,
   "email" : "john@gmail.com",
   "phone" : "777-666-555"

}
```
Mapping embedded documents

So far, we have mapped very simple basic structures with MongoDB. In real world cases, you will have to deal with Java classes having references to other objects. For example, we could think of a Customer class, which contains some information in a separate class named Info:

```java
public class CustomerInfo {

    String name;
    Info info;

    public CustomerInfo(String name, int age, String email, String phone) {
        this.name = name;
        this.info = new Info(age, email, phone);
    }

    public Info getInfo() {
        return info;
    }

    public void setInfo(Info info) {
        this.info = info;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "CustomerInfo [name=" + name + ", info=" + this.info + "]";
    }
}

class Info {
```
public Info(int age, String email, String phone) {
    super();
    this.email = email;
    this.phone = phone;
    this.age = age;
}

public Info() {
}

String email;
String phone;
int age;

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getPhone() {
    return phone;
}

public void setPhone(String phone) {
    this.phone = phone;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

@Override
public String toString() {
    return "Info [email=" + email + ", phone=" + phone + ",
    age=" + age + "]";
}
}
In the new class called CustomerInfo, we have highlighted the fields that will be mapped by MongoDB keys.

As you are aware, creating embedded documents in MongoDB can be done by setting a key to a DDBObject structure containing the embedded document. In our case, we will structure the info key to contain the embedded document's information:

```java
BasicDBObject doc = new BasicDBObject("name", "owen").append("info", new BasicDBObject("age", 25).append("email", "owen@gmail.com").append("phone", "321-456-778"));

coll.insert(doc);

DBObject obj = coll.findOne(doc);

CustomerInfo c = gson.fromJson(obj.toString(), CustomerInfo.class);

System.out.println("Found customer " + c);
```

The expected output should match the following query executed through the mongo shell:

```bash
> db.javastuff.find({"name":"owen"}).pretty()
{
   "_id" : ObjectId("5526888bd3d56a86cea8ea12"),
   "name" : "owen",
   "info" : {
      "age" : 25,
      "email" : "owen@gmail.com",
      "phone" : "321-456-778"
   }
}
```

**Custom field names in your Java classes**

The Customer class contains a set of fields that are exactly equivalent to the key names to be found in the MongoDB collection. The schema of a MongoDB document is, however, quite flexible compared to a standard database table. One simple strategy could be choosing custom names for your class fields and mapping the corresponding MongoDB keys with the com.google.gson.annotations.SerializedName annotation. See the following class as an example:
import com.google.gson.annotations.SerializedName;

public class Customer {

    @SerializedName("name")
    String userName;

    @SerializedName("age")
    int userAge;

    @SerializedName("email")
    String userEmail;

    @SerializedName("phone")
    String userPhone;

}

In the next section, we will deal with a more complex concern, that is, mapping complex BSON types used by MongoDB to store entries.

Mapping complex BSON types

The preceding examples used simple Java types such as String and Integers. Sometimes, however, you might need to use a custom serialization/deserialization strategy for your classes.

For example, consider the following document structure:

```json
{
    "_id" : ObjectId("5527b0bbd53064aac7c991"),
    "name" : "john",
    "age" : 22,
    "email" : "john@gmail.com",
    "phone" : "777-666-555"
}
```

You might think that adding the _id field to the Customer class will do the job of mapping Mongo's _id key:

```java
public Customer(Object _id, String name, int age, String email, String phone) {
    super();
    this._id = _id;
    this.name = name;
    this.age = age;
}
```
MongoDB CRUD Beyond the Basics

```java
this.email = email;
this.phone = phone;
}
public String toString() {
    return "Customer{" + ",  _id=", name=" + name + "}, age=" + age + ", email=" + email + "}, phone=" + phone + "}
}
```

Let's see what happens if we try to deserialize the preceding document by using the `fromJson` method:

```java
Customer c = gson.fromJson(obj.toString(), Customer.class);
System.out.println(c);
```

What we are trying to achieve is the following representation of the `Customer` class:

```
{ _id=558c1007578ef44c4cbb0eb8, name=owen, age=25,
  email=owen@gmail.com, phone=321-456-778
```

However, as you can see from the following output, the `_id` object was not correctly parsed as we expected:

```
_id={$oid=5527b117d3d511091e1735e2}, name=owen, age=22,
email=john@gmail.com, phone=777-666-555
```

Many other examples exist, for example, if you are dealing with date and time libraries.

Luckily, Gson allows registering custom serializers/deserializers so that you can convert these objects into the type that is needed by your application.

This is done in two steps:

1. At first you need to code a serializer (if you are inserting custom types in MongoDB) or a deserializer (if you are going to parse custom entries contained in MongoDB).
2. Then, you need to register this custom adapter.

Let's see both steps, in case you want to parse the `_id` unique identifier of MongoDB documents.

A custom deserializer needs to implement the `com.google.gson.JsonDeserializer` class as follows:

```java
import com.google.gson.JsonDeserializationContext;
import com.google.gson.JsonDeserializer;
```
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParseException;

public class CustomAdapter implements JsonDeserializer<Customer> {

    public Customer deserialize(JsonElement json,
            java.lang.reflect.Type typeOfT, JsonDeserializationContext
            context) throws JsonParseException {
        JsonObject jObj = json.getAsJsonObject();

        String id =
                jObj.get("_id").toString().replaceAll(".*"/(\w+)/"},
                        "$1");

        String name = jObj.get("name") != null ?
                jObj.get("name").getAsString() : "";

        String email = jObj.get("email")!= null ?
                jObj.get("email").getAsString() : "";

        String phone = jObj.get("phone")!= null ?
                jObj.get("phone").getAsString() : "";

        int age = jObj.get("age")!= null ? jObj.get("age").getAsInt()
                        : 0;

        return new Customer(id,name,age,email,phone);
    }
}

As you can see, this class contains the deserialization logic in the method
deserialize, where each field is parsed according to your custom parsing rules.
In this case, the value of the _id field is extracted using a regular expression, which
scans for the identifier contained in the parentheses.

Please note that using a schemaless database implies a lack of
control over the data contained in your collection. As you can
see, we had to check against null on each field of our document.

Some changes will also be required in your main Java class, so that you register the
adapter on the Gson class, by means of the registerTypeAdapter method contained
in the com.google.gson.GsonBuilder class:

    GsonBuilder builder=new GsonBuilder();

    Gson gson = new GsonBuilder().registerTypeAdapter(Customer.class, new
            CustomerAdapter()).create();
BasicDBObject doc = new BasicDBObject("name", "owen");
DBObject obj = coll.findOne(doc);

Customer c = gson.fromJson(obj.toString(), Customer.class);
System.out.println("Found customer " + c);

Now the toString output of the Customer class reveals that you have been able to parse the $id field correctly:

_id=5527b117d3d511091e1735e2, name=owen, age=22,
    email=john@gmail.com, phone=777-666-555

Using indexes in your applications

The concept of an index in a database is pretty equivalent to the index contained in a book. So, instead of searching for a section across all the pages of the book onwards, you search for the relevant section in the index and then open the book on that page.

This concept has been adopted by all relational databases and it works quite the same on MongoDB, that is, by creating a special data structure that is able to store a small part of the collection's dataset in such a way that is easy to traverse from.

Without using indexes, MongoDB must perform an expensive collection scan, which means to scan every document in a collection, in order to find those documents that match the query string. Indexes can improve the efficiency of your queries by limiting the number of documents they must inspect on each query.

This is not true in every case. As a matter of fact, a built-in index already exists on every collection on the _id field. This index is unique and prevents duplicate insertions using the _id field in a collection.

Let's see a practical example:

MongoClient mongoClient = new MongoClient("localhost", 27017);

DB db = mongoClient.getDB("sampledb");

DBCollection coll = db.getCollection("indextest");

for (int ii=0;ii<100000;ii++) {

    DBObjec t doc = new BasicDBObject("userid", ii);
}
Here, we are inserting 1,00,000 documents in one go. Once the insertion completes, we can move to the mongo shell and execute the explain function to see what happens behind the scenes when mongo performs a query:

```javascript
> db.indextest.find({userid: 50000}).explain("executionStats")
{
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "sampledb.indextest",
        "indexFilterSet": false,
        "parsedQuery": {
            "userid": {
                "$eq": "1111"
            }
        },
        "winningPlan": {
            "stage": "COLLSCAN",
            "filter": {
                "userid": {
                    "$eq": "1111"
                }
            },
            "direction": "forward"
        },
        "rejectedPlans": []
    },
    "executionStats": {
        "executionSuccess": true,
        "nReturned": 0,
        "executionTimeMillis": 6,
        "totalKeysExamined": 0,
        "totalDocsExamined": 100000,
        "executionStages": {
```
The `totalDocsExamined` attribute reveals the number of documents MongoDB has looked through; as you can see, every single document has been searched. Although the time needed to scan the collection might look insignificant (6 ms) this is due to the fact that we are dealing with a minimal document and that of course MongoDB is quite fast!

Also, the attribute `totalKeysExamined` set to 0 indicates that no index key was scanned during the search.
Besides this, you might instruct the Mongo cursor to stop looking through other documents once an occurrence is found using the `limit(1)` operator (which is also available through the Java driver). That could be helpful but may not be exactly what you are looking for in your search.

In the next section, we will see how to use an index to limit the number of documents to be scanned.

### Defining an index in your Java classes

Adding an index to your collection is quite easy. In order to do that, you need to specify which fields of a document need to be indexed and state whether the index ordering is going to be in ascending (1) or descending (-1) order. For example, the following creates an ascending index on the `userid` field:

```java
coll.createIndex(new BasicDBObject("userid",1));
```

If you have been using the MongoDB Java driver in its earlier version, you might have used the `ensureIndex` method to create an index if that is not available. This method is now deprecated and you have to use `createIndex` as shown.

Now, let’s execute the `explain` plan query again and examine the result (we will show you just the relevant part of the statistics):

```bash
> db.indextest.find({userid: 5000}).explain("executionStats")
{
   ...
   "executionStats":{
      "executionSuccess":true,
      "nReturned":0,
      "executionTimeMillis":0,
      "totalKeysExamined":1,
      "totalDocsExamined":1,
      "executionStages":{
         "stage":"FETCH",
         "nReturned":0,
         "executionTimeMillisEstimate":0,
         "works":1,
```
MongoDB CRUD Beyond the Basics

"advanced":0,
"needTime":0,
"needFetch":0,
"saveState":0,
"restoreState":0,
"isEOF":1,
"invalidates":0,
"docsExamined":0,
"alreadyHasObj":0,
"inputStage":{
  "stage":"IXSCAN",
  "nReturned":0,
  "executionTimeMillisEstimate":0,
  "works":1,
  "advanced":0,
  "needTime":0,
  "needFetch":0,
  "saveState":0,
  "restoreState":0,
  "isEOF":1,
  "invalidates":0,
  "keyPattern":{
    "userid":1
  },
  "indexName":"userid_1",
  "isMultiKey":false,
  "direction":"forward",
  "indexBounds":{
    "userid":[
      "["1111", "1111"]"
    ]
  },
  "keysExamined":0,
  "dupsTested":0,
  "dupsDropped":0,
The explain() output is now a bit more complex; let's focus on the fields we are interested in. The number of totalDocsExamined documents is just one and the query is now instantaneous as the index named userid_1 has been used. However, everything has its flip side. In this case, we will have super-fast queries at the price of slower inserts/updates as indexes have to be rewritten too. More storage has to be planned also since indexes will need it. However, that is now a peculiarity of MongoDB, but it is a clear assumption that is true on every database.

For the sake of completeness, we will mention that the explain function is also available on the Java side, by calling the explain method directly on a search string:

```java
BasicDBObject doc = new BasicDBObject("userid", "1111");
DBObject explainObject = coll.find(doc).explain();
System.out.println(explainObject) ;
```

### Using compound indexes

The preceding examples are making the assumption that our indexed field is a prefix in the query. For example, consider the following search:

```java
BasicDBObject doc = new BasicDBObject("code", "1111").append("userid",5000);
DBObject explainObject = coll.find(doc);
```

In this case, having defined the index on the userid field, this is not helping our query too much, as the index will come into play only after scanning the first key, that is, code. A solution, in this case, could be to create a compound index that is a handy solution if your search contains multiple criteria.
The following sample diagram illustrates a compound index using two fields, such as `userid` and `code`:

![Compound Index Diagram]

As you can see from the preceding figure, in a **Compound Index**, a single index structure holds references to multiple fields (userid and code) within a collection's documents.

Creating a **Compound Index** is not very different from creating a single index field. Using the fluent API provided by the `BasicDBObject`, you can append the keys and then create the index with that object:

```java
DBObject obj = new BasicDBObject();
obj.put("userid", 1);
obj.put("code", 1);
coll.createIndex(obj);
```

You can run the **Compound Index** creation and verify that the search cursor is using the **Compound Index** and scanning only one document:

```bash
> db.indextest.find({userid: 5000, code:5000}).explain("executionStats")
```

```json
{
   ...
   "executionStats":{
   ```
"executionSuccess":true,
"nReturned":0,
"executionTimeMillis":0,
"totalKeysExamined":1,
"totalDocsExamined":1,
"executionStages":{
  "stage":"FETCH",
  "nReturned":0,
  "executionTimeMillisEstimate":0,
  "works":1,
  "advanced":0,
  "needTime":0,
  "needFetch":0,
  "saveState":0,
  "restoreState":0,
  "isEOF":1,
  "invalidates":0,
  "docsExamined":0,
  "alreadyHasObj":0,
  "inputStage":{
    "stage":"IXSCAN",
    "nReturned":0,
    "executionTimeMillisEstimate":0,
    "works":1,
    "advanced":0,
    "needTime":0,
    "needFetch":0,
    "saveState":0,
    "restoreState":0,
    "isEOF":1,
    "invalidates":0,
    "keyPattern":{
      "userid":1,
      "code":1
Using text indexes in your documents

MongoDB has support for text indexes that can be used to search strings of text contained in documents of a collection.

Since version 2.6 of MongoDB, the text search feature is enabled by default, so you don't need to do anything in order to activate it.

In order to perform queries using the text index, you need to use the `$text` query operator. In the following example, we are creating a text index on the `content` key:

```java
MongoClient mongoClient = new MongoClient("localhost", 27017);
DB db = mongoClient.getDB("sampledb");
```
DBCollection coll = db.getCollection("textitems");

coll.createIndex(new BasicDBObject("content", "text"));

coll.insert(new BasicDBObject().append("content", "mytext other content"));

DBObject search = new BasicDBObject("$search", "mytext");

DBObject textSearch = new BasicDBObject("$text", search);

int count = coll.find(textSearch).count();
System.out.println("Found text search matches: " + count);

Once the index has been created, we will use the $text operator to perform a text search on the collection, using the string of words contained in the $search operator:

MongoClient mongoClient = new MongoClient("localhost", 27017);

DB db = mongoClient.getDB("sampledb");

DBCollection coll = db.getCollection("textitems");

coll.insert(new BasicDBObject("_id", 1).append("text", "mytext"));

List<DBObject> list = coll.getIndexInfo();

for (DBObject obj:list)
    System.out.println(obj);
}

The method getIndexInfo returns a list of the indexes for this collection as DBOBJECT. This information is printed on the console, which in our case, outputs the following:

{ "v" : 1 , "key" : { "_id" : 1} , "name" : "_id_" , "ns" : "sampledb.textitems"}

**Searching for text by language**

Text search can be done using additional options such as language search, which enables restricting the text search to a particular language. The list of languages supported in this text search is contained in the driver documentation at http://docs.mongodb.org/manual/reference/text-search-languages/.
MongoDB CRUD Beyond the Basics

Here is a full example that shows how to restrict your searches only to English words by means of the $language operator:

```
MongoClient mongoClient = new MongoClient("localhost", 27017);

DB db = mongoClient.getDB("sampledb");

DBCollection coll = db.getCollection("textitems");

coll.createIndex(new BasicDBObject("textcontent", "text"));

coll.insert(new BasicDBObject("_id", 0).append("textcontent", "Some data"));
coll.insert(new BasicDBObject("_id", 1).append("textcontent", "Other data"));
coll.insert(new BasicDBObject("_id", 2).append("textcontent", "Not important"));

BasicDBObject search = new BasicDBObject("$search", "data");

DBObject textSearch = new BasicDBObject("$text",
  search.append("$language", "english"));

int matchCount = coll.find(textSearch).count();
System.out.println("Found language matches: " + matchCount);
```

The expected output, from the last line of code, is to print:

```
Found language matches: 2
```

**Searching for text by score**

A common requirement for a text search engine is to provide a score, for example, in the case of searches including a complex set of words. Score search can be done by setting the textScore parameter in the $meta projection operator.

```
DBObject scoreSearch = new BasicDBObject("score", new
  DDBObject("$meta", "textScore"));
```

The score represents the relevance of a document to a given text search query.

The following example shows how to return the score in a search by means of the metadata associated with the query:

```
DBObject scoreSearch = new BasicDBObject("score", new
  DDBObject("$meta", "textScore"));
```

[82]
Chapter 3

DBObject doc = coll.findOne(textSearch, scoreSearch);
System.out.println("Highest scoring document: " + doc);

Coding bulk operations

One of the highlights available since MongoDB 2.6 is the new bulk write operations. Bulk operations allow building a list of write operations to be executed in bulk for a single collection. The Java driver for MongoDB includes a new bulk API as well, which allows your applications to leverage these new operations using a fluent-style API.

First of all, let’s explore this API that can be executed using two main styles:

- **Ordered bulk operations**: Every operation will be executed in the order they are added to the bulk operation, halting when there's an error
- **Unordered bulk operations**: These operations are executed in parallel and neither guarantee order of execution, nor do they stop when an error occurs

First, let's see an example of OrderedBulkOperation:

```
BulkWriteOperation builder =
  collection.initializeOrderedBulkOperation();

builder.insert(new BasicDBObject("item", "A1"));
builder.insert(new BasicDBObject("item", "A2"));
builder.insert(new BasicDBObject("item", "A3"));

builder.find(new BasicDBObject("item", "A1")).updateOne(new
  BasicDBObject("$set", new BasicDBObject("A1", "AX")));

BulkWriteResult result = builder.execute();

System.out.println("Bulk Completed: Inserted documents " +
  result.getInsertedCount());
System.out.println("Bulk Completed: Modified documents " +
  result.getModifiedCount());
```

As you can see, an instance of the BulkWriteOperation class is created using the initializeOrderedBulkOperation method of the collection class. Operations are added using the fluent API available in the BulkWriteOperation.
The expected output of the preceding execution will be as follows:

Bulk Completed: Inserted documents 3
Bulk Completed: Modified documents 1

Finally, the BulkWriteResult is used as a wrapper that contains the results of the Bulk.execute() method.

The same code written using an unordered bulk operation can be coded this way:

```java
BulkWriteOperation builder =
    collection.initializeUnorderedBulkOperation();

builder.insert(new BasicDBObject("item", "A1"));
builder.insert(new BasicDBObject("item", "A2"));
builder.insert(new BasicDBObject("item", "A3"));

builder.find(new BasicDBObject("item", "A1"))
    .updateOne(new BasicDBObject("$set", new BasicDBObject("A1", "AX")));

BulkWriteResult result = builder.execute();
```

**Comparing plain inserts with BulkWriteOperations**

Bulk operations are most useful when you have a batch of inserts/updates which need to be executed in one single shot. The advantage in terms of performance is notable. As a proof of evidence, we will compare the execution of a batch of 10,000 documents using the default approach:

```java
long l1 = System.currentTimeMillis();

for (int ii=0; ii<10000; ii++) {
    DDBObject doc = new BasicDBObject("name", "frank")
        .append("age", 31)
        .append("info", new BasicDBObject("email",
            "frank@mail.com")
            .append("phone", "222-111-444");

    coll.insert(doc);
}

long l2 = System.currentTimeMillis();

System.out.println(l2-l1);
```
By running the above example, 10,000 inserts are performed in 7421 ms. Let's reengineer the code to use bulk operations:

```java
long l1 = System.currentTimeMillis();

BulkWriteOperation builder = coll.initializeOrderedBulkOperation();

for (int ii=0; ii<10000; ii++) {
    DDBObject doc = new BasicDBObject("name", "frank")
        .append("age", 31)
        .append("info", new BasicDBObject("email", "frank@mail.com")
            .append("phone", "222-111-444"));

    builder.insert(doc);
}

BulkWriteResult result = builder.execute();

long l2 = System.currentTimeMillis();

System.out.println(l2-l1);
```

The second execution was completed in as little as 1535 ms.

You can further reduce the execution by switching to an unordered bulk execution:

```java
BulkWriteOperation builder = coll.initializeUnorderedBulkOperation();
```

This will bring down the time to even less, that is, 1446 ms to insert 10,000 records.

Summary

In this chapter, we have gone through some advanced features available in the MongoDB Java driver, and we examined some strategies for mapping Java objects to MongoDB documents. The solutions we have covered will fit into many practical scenarios. We now need to change the perspective of our examples a bit as we move into an enterprise context. In the next chapter, we will show you how to use the knowledge we have built so far to create a Java Enterprise application using MongoDB as the storage.
Where to buy this book

You can buy MongoDB for Java Developers from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.