MongoDB Data Modeling

Focus on data usage and better design schemas with the help of MongoDB

Wilson da Rocha França
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Querying Documents'
- A synopsis of the book’s content
- More information on **MongoDB Data Modeling**
Wilson da Rocha França is a system architect at the leading online retail company in Latin America. An IT professional, passionate about computer science, and an open source enthusiast, he graduated with a university degree from Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, Rio de Janeiro, Brazil, in 2005 and holds a master's degree in Business Administration from Universidade Federal de Rio de Janeiro, gained in 2010.

Passionate about e-commerce and the Web, he has had the opportunity to work not only in online retail but in other markets such as comparison shopping and online classifieds. He has dedicated most of his time to being a Java web developer.

He worked as a reviewer on Instant Varnish Cache How-to and Arduino Development Cookbook, both by Packt Publishing.
Preface

Even today, it is still quite common to say that computer science is a young and new field. However, this statement becomes somewhat contradictory when we observe other fields. Unlike other fields, computer science is a discipline that is continually evolving above the normal speed. I dare say that computer science has now set the path of evolution for other fields such as medicine and engineering. In this context, database systems as an area of the computer science discipline has not only contributed to the growth of other fields, but has also taken advantage itself of the evolution and progress of many areas of technology such as computer networks and computer storage.

Formally, database systems have been an active research topic since the 1960s. Since then, we have gone through a few generations, and big names in the IT industry have emerged and started to dictate the market's tendencies.

In the 2000s, driven by the world's Internet access growth, which created a new network traffic profile with the social web boom, the term NoSQL became common. Considered by many to be a paradoxical and polemic subject, it is seen by some as a new technology generation that has been developed in response to all changes we have experienced in the last decade.

MongoDB is one of these technologies. Born in the early 2000s, it became the most popular NoSQL database in the world. Not only the most popular database in the world, since February 2015, MongoDB became the fourth most popular database system according to the DB-Engines ranking (http://db-engines.com/en/), surpassing the well-known PostgreSQL database.

Nevertheless, popularity should not be confused with adoption. Although the DB-Engines ranking shows us that MongoDB is responsible for some traffic on search engines such as Google, has job search activity, and has substantial social media activity, we can not state how many applications are using MongoDB as a data source. Indeed, this is not exclusive to MongoDB, but is true of every NoSQL technology.
Preface

The good news is that adopting MongoDB has not been a very tough decision to make. It's open source, so you can download it free of charge from MongoDB Inc. (https://www.mongodb.com), where you can find extensive documentation. You also can count on a big and growing community, who, like you, are always looking for new stuff on books, blogs, and forums; sharing knowledge and discoveries; and collaborating to add to the MongoDB evolution.

MongoDB Data Modeling was written with the aim of being another research and reference source for you. In it, we will cover the techniques and patterns used to create scalable data models with MongoDB. We will go through basic database modeling concepts, and provide a general overview focused on modeling in MongoDB. Lastly, you will see a practical step-by-step example of modeling a real-life problem.

Primarily, database administrators with some MongoDB background will take advantage of MongoDB Data Modeling. However, everyone from developers to all the curious people that downloaded MongoDB will make good use of it.

This book focuses on the 3.0 version of MongoDB. MongoDB 3.0, which was long awaited by the community, is considered by MongoDB Inc. as its most significant release to date. This is because, in this release, we were introduced to the new and highly flexible storage architecture, WiredTiger. Performance and scalability enhancements intend to strengthen MongoDB's emphasis among database systems technologies, and position it as the standard database for modern applications.

What this book covers

Chapter 1, Introducing Data Modeling, introduces you to basic data modeling concepts and the NoSQL universe.

Chapter 2, Data Modeling with MongoDB, gives you an overview of MongoDB's document-oriented architecture and presents you with the document, its characteristics, and how to build it.

Chapter 3, Querying Documents, guides you through MongoDB APIs to query documents and shows you how the query affects our data modeling process.

Chapter 4, Indexing, explains how you can improve the execution of your queries and consequently change the way we model our data by making use of indexes.

Chapter 5, Optimizing Queries, helps you to use MongoDB's native tools to optimize your queries.
Chapter 6, Managing the Data, focuses on the maintenance of data. This will teach you how important it is to look at your data operations and administration before beginning the modeling of data.

Chapter 7, Scaling, shows you how powerful the autosharing characteristic of MongoDB can be, and how we think our data model is distributed.

Chapter 8, Logging and Real-time Analytics with MongoDB, takes you through an schema design of a real-life problem example.
Querying Documents

In a NoSQL database, such as MongoDB, planning queries is a very important task, and depending on the query you want to perform, your document can vary greatly.

As you saw in Chapter 2, Data Modeling with MongoDB, the decision to refer or include documents in a collection is, in a large part, the result of our planning. It is essential to determine whether we will give a preference to reading or writing in a collection.

Here, we will see how planning queries can help us create documents in a more efficient and effective way, and we will also consider more sensible questions such as atomicity and transactions.

This chapter will focus on the following subjects:

• Read operations
• Write operations
• Write concerns
• Bulk writing documents

Understanding the read operations

Read is the most common and fundamental operation in a database. It's very hard to imagine a database that is used only to write information, where this information is never read. By the way, I have never heard of such an approach.

In MongoDB, we can execute queries through the find interface. The find interface can accept queries as criteria and projections as parameters. This will result in a cursor. Cursors have methods that can be used as modifiers of the executed query, such as limit, map, skip, and sort. For example, take a look at the following query:

db.customers.find({"username": "johnclay"})
Querying Documents

This would return the following document:

```
{
   "_id" : ObjectId("54835d0ff059b08503e200d4"),
   "username" : "johnclay",
   "email" : "johnclay@crgv.com",
   "password" : "bf383e8469e98b44895d61b821748ae1",
   "details" : {
      "firstName" : "John",
      "lastName" : "Clay",
      "gender" : "male",
      "age" : 25
   },
   "billingAddress" : [
      {
         "street" : "Address 1, 111",
         "city" : "City One",
         "state" : "State One"
      }
   ],
   "shippingAddress" : [
      {
         "street" : "Address 2, 222",
         "city" : "City Two",
         "state" : "State Two"
      },
      {
         "street" : "Address 3, 333",
         "city" : "City Three",
         "state" : "State Three"
      }
   ]
}
```

We can use the `find` interface to execute a query in MongoDB. The `find` interface will select the documents in a collection and return a cursor for the selected documents.
Compared with the SQL language, the find interface should be seen as a select statement. And, similar to a select statement where we can determine clauses with expressions and predicates, the find interface allows us to use criteria and projections as parameters.

As mentioned before, we will use JSON documents in these find interface parameters. We can use the find interface in the following way:

```
db.collection.find(
  {criteria},
  {projection}
)
```

In this example:

- **criteria** is a JSON document that will specify the criteria for the selection of documents inside a collection by using some operators
- **projection** is a JSON document that will specify which document's fields in a collection will be returned as the query result

Both are optional parameters, and we will go into more detail regarding these later.

Let's execute the following example:

```
db.customers.find(
  {"username": "johnclay"},
  {_id: 1, username: 1, details: 1}
)
```

In this example:

- `{"username": "johnclay"}` is the criteria
- `{_id: 1, username: 1, details: 1}` is the projection

This query will result in this document:

```
{
  "_id" : ObjectId("54835d0ff059b08503e200d4"),
  "username" : "johnclay",
  "details" : {
    "firstName" : "John",
    "lastName" : "Clay",
    "gender" : "male",
    "age" : 25
  }
}
```
Selecting all documents

As mentioned in the previous section, in the find interface, both the criteria and projection parameters are optional. To use the find interface without any parameters means selecting all the documents in a collection.

Note that the query result is a cursor with all the selected documents.

So, a query in the products collection executes in this way:

```javascript
db.products.find()
```

It will return:

```json
{
   "_id" : ObjectId("54837b61f059b08503e200db"),
   "name" : "Product 1",
   "description" : "Product 1 description",
   "price" : 10,
   "supplier" : {
      "name" : "Supplier 1",
      "telephone" : "+552199998888"
   }
}
{
   "_id" : ObjectId("54837b65f059b08503e200dc"),
   "name" : "Product 2",
   "description" : "Product 2 description",
   "price" : 20,
   "supplier" : {
      "name" : "Supplier 2",
      "telephone" : "+552188887777"
   }
}
...
```
Selecting documents using criteria

Despite the convenience, selecting all the documents in a collection can turn out to be a bad idea due to a given collection's length. If we take as an example a collection with hundreds, thousands, or millions of records, it is essential to create a criterion in order to select only the documents we want.

However, nothing prevents the query result from being huge. In this case, depending on the chosen drive that is executing the query, we must iterate the returned cursor.

Note that in the mongo shell, the default value of returned records is 20.

Let's check the following example query. We want to select the documents where the attribute name is **Product 1**:

```javascript
db.products.find({name: "Product 1"});
```

This will give us as a result:

```json
{
   "_id" : ObjectId("54837b61f059b08503e200db"),
   "name" : "Product 1",
   "description" : "Product 1 description",
   "price" : 10,
   "supplier" : {
      "name" : "Supplier 1",
      "telephone" : "+552199998888"
   }
}
```

The preceding query selects the documents through the equality `{name: "Product 1"}`. It's also possible to use operators on the criteria interface.

The following example demonstrates how it's possible to select all documents where the price is greater than 10:

```javascript
db.products.find({price: {$gt: 10}});
```
Querying Documents

This produces as a result:

```javascript
{
    "_id" : ObjectId("54837b65f059b08503e200dc"),
    "name" : "Product 2",
    "description" : "Product 2 description",
    "price" : 20,
    "supplier" : {
        "name" : "Supplier 2",
        "telephone" : "+552188887777"
    }
}
{
    "_id" : ObjectId("54837b69f059b08503e200dd"),
    "name" : "Product 3",
    "description" : "Product 3 description",
    "price" : 30,
    "supplier" : {
        "name" : "Supplier 3",
        "telephone" : "+552177776666"
    }
}
```

When we execute a query using the operator `$gt`, only documents that have an information price greater than 10 will be returned as a result in the cursor.

In addition, there are other operators such as comparison, logical, element, evaluation, geographical, and arrays.

Let's take, for example, the documents from the `products` collection, shown as follows:

```javascript
{
    "_id" : ObjectId("54837b61f059b08503e200db"),
    "name" : "Product 1",
    "description" : "Product 1 description",
    "price" : 10,
    "supplier" : {
        "name" : "Supplier 1",
        "telephone" : "+552166665555"
    }
}
```
"telephone" : "+552199998888",
"review" : [
    {
        "customer" : {
            "email" : "customer@customer.com"
        },
        "stars" : 5
    },
    {
        "customer" : {
            "email" : "customer2@customer.com"
        },
        "stars" : 6
    }
]
{
    "_id" : ObjectId("54837b65f059b08503e200dc"),
    "name" : "Product 2",
    "description" : "Product 2 description",
    "price" : 20,
    "supplier" : {
        "name" : "Supplier 2",
        "telephone" : "+552188887777"
    },
    "review" : [
        {
            "customer" : {
                "email" : "customer@customer.com"
            },
            "stars" : 10
        },
        {
            "customer" : {
                "email" : "customer2@customer.com"
            }
        }
    ]
}


Querying Documents

},
   "stars" : 2
}
]

{
   "_id" : ObjectId("54837b69f059b08503e200dd"),
   "name" : "Product 3",
   "description" : "Product 3 description",
   "price" : 30,
   "supplier" : {
      "name" : "Supplier 3",
      "telephone" : "+552177776666"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 9
      }
   ]
}
Comparison operators

MongoDB provides us with a way to define equality between values. With comparison operators, we can compare BSON type values. Let's look at these operators:

- The `$gte` operator is responsible for searching values that are equal or greater than the value specified in the query. If we execute the query `db.products.find({price: {$gte: 20}})` it will return:

```javascript
{
   
   "_id" : ObjectId("54837b65f059b08503e200dc"),
   
   "name" : "Product 2",
   
   "description" : "Product 2 description",
   
   "price" : 20,
   
   "supplier" : {
      
      "name" : "Supplier 2",
      
      "telephone" : "+552188887777"
   },
   
   "review" : [
      
      {
         
         "customer" : {
            
            "email" : "customer@customer.com"
         },
         
         "stars" : 10
      },
      
      {
         
         "customer" : {
            
            "email" : "customer2@customer.com"
         },
         
         "stars" : 2
      }
   ]

}
```

```javascript
{
   
   "_id" : ObjectId("54837b69f059b08503e200dd"),
   
   "name" : "Product 3",

```
With the \$lt operator, it's possible to search for values that are inferior to the requested value in the query. The query `db.products.find({price: \$lt: 20})` will return:

```json
{  
  "_id" : ObjectId("54837b61f059b08503e200db"),  
  "name" : "Product 1",  
  "description" : "Product 1 description",  
  "price" : 10,  
  "supplier" : {  
    "name" : "Supplier 1",  
    "telephone" : "+552199998888"  
  },  
  "review" : [  
    {  
      "customer" : {  
        "email" : "customer2@customer.com"  
      },  
      "stars" : 9  
    }  
  ]
}
```
"email" : "customer@customer.com"
},
  "stars" : 5
},
{
  "customer" : {
    "email" : "customer2@customer.com"
  },
  "stars" : 6
}
]
}

• The $lte operator searches for values that are less than or equal to the requested value in the query. If we execute the query `db.products.find({price: {$lte: 20}})`, it will return:

{  
  "_id" : ObjectId("54837b61f059b08503e200db"),  
  "name" : "Product 1",  
  "description" : "Product 1 description",  
  "price" : 10,  
  "supplier" : {  
    "name" : "Supplier 1",  
    "telephone" : "+5521999988888"  
  },  
  "review" : [
    {  
      "customer" : {
        "email" : "customer@customer.com"
      },  
      "stars" : 5
    },
    {  
      "customer" : {
        "email" : "customer2@customer.com"
      },  
      "stars" : 6
    }
  ]
}
The \$in\ operator is able to search any document where the value of a field equals a value that is specified in the requested array in the query. The execution of the query `db.products.find({price:{$in: [5, 10, 15]}})` will return:

```json
{
  "_id" : ObjectId("54837b61f059b08503e200db"),
  "name" : "Product 1",
  "description" : "Product 1 description",
  "price" : 10,
  "supplier" : {
    "name" : "Supplier 2",
    "telephone" : "+552188887777"
  },
  "review" : [ ]
}
```

- The \$in\ operator is able to search any document where the value of a field equals a value that is specified in the requested array in the query. The execution of the query `db.products.find({price:{$in: [5, 10, 15]}})` will return:
"supplier": {
  "name": "Supplier 1",
  "telephone": "+552199998888"
},
"review": [
  {
    "customer": {
      "email": "customer@customer.com"
    },
    "stars": 5
  },
  {
    "customer": {
      "email": "customer2@customer.com"
    },
    "stars": 6
  }
]

• The $nin operator will match values that are not included in the specified array. The execution of the `db.products.find({price:{$nin: [10, 20]}})` query will produce:

```json
{  
  "_id": ObjectId("54837b69f059b08503e200dd"),
  "name": "Product 3",
  "description": "Product 3 description",
  "price": 30,
  "supplier": {
    "name": "Supplier 3",
    "telephone": "+552177776666"
  },
  "review": [
    {
      "customer": {
        "email": "customer@customer.com"
      }
    }
  ]
}
```
Querying Documents

```javascript
{
  "stars" : 5,
}
{
  "customer" : {
    "email" : "customer2@customer.com"
  },
  "stars" : 9
}
```

- The `$ne` operator will match any values that are not equal to the specified value in the query. The execution of the `db.products.find({name: {$ne: "Product 1"}})` query will produce:

```javascript
{
  "_id" : ObjectId("54837b65f059b08503e200dc"),
  "name" : "Product 2",
  "description" : "Product 2 description",
  "price" : 20,
  "supplier" : {
    "name" : "Supplier 2",
    "telephone" : "+552188887777"
  },
  "review" : [
    {
      "customer" : {
        "email" : "customer@customer.com"
      },
      "stars" : 10
    },
    {
      "customer" : {
        "email" : "customer2@customer.com"
      },
      "stars" : 2
    }
  ]
}
```
Logical operators

Logical operators are how we define the logic between values in MongoDB. These are derived from Boolean algebra, and the truth value of a Boolean value can be either true or false. Let's look at the logical operators in MongoDB:

- The $and operator will make a logical AND operation in an expressions array, and will return the values that match all the specified criteria. The execution of the `db.products.find({$and: [{price: {$lt: 30}}, {name: "Product 2"}]}))` query will produce:

```json
{
   "_id" : ObjectId("54837b65f059b08503e200dc"),
   "name" : "Product 2",
   "description" : "Product 2 description",
   "price" : 30,
   "supplier" : {
      "name" : "Supplier 3",
      "telephone" : "+55217776666"
   },
   "review" : [;
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 9
      }
   ]
}
```
### Querying Documents

```

{  
  "description" : "Product 2 description",
  "price" : 20,
  "supplier" : {  
    "name" : "Supplier 2",
    "telephone" : "+552188887777"
  },
  "review" : [
    {
      "customer" : {
        "email" : "customer@customer.com"
      },
      "stars" : 10
    },
    {
      "customer" : {
        "email" : "customer2@customer.com"
      },
      "stars" : 2
    }
  ]
}
```

- The `$or` operator will make a logical OR operation in an expressions array, and will return all the values that match either of the specified criteria. The execution of the `db.products.find({$or: [{price: {$gt: 50}}, {name: "Product 3"}]})` query will produce:

```

{  
  "_id" : ObjectId("54837b69f059b08503e200dd"),
  "name" : "Product 3",
  "description" : "Product 3 description",
  "price" : 30,
  "supplier" : {  
    "name" : "Supplier 3",
    "telephone" : "+552177776666"
  },
  "review" : [
    {
    
```

---

[52]
The $not operator inverts the query effect and returns the values that do not match the specified operator expression. It is used to negate any operation. The execution of the `db.products.find({price: {$not: {$gt: 10}}})` query will produce:

```javascript
{  
  "_id" : ObjectId("54837b61f059b08503e200db"),  
  "name" : "Product 1",  
  "description" : "Product 1 description",  
  "price" : 10,  
  "supplier" : {  
    "name" : "Supplier 1",  
    "telephone" : "+552199998888"  
  },  
  "review" : [  
    {  
      "customer" : {  
        "email" : "customer@customer.com"  
      },  
      "stars" : 5  
    },  
    {  
      "customer" : {  
        "email" : "customer2@customer.com"  
      }  
    },  
    {  
      "customer" : {  
        "email" : "customer@customer.com"  
      },  
      "stars" : 5  
    }  
  ]
}
```
• The $nor operator will make a logical NOR operation in an expressions array, and will return all the values that fail to match all the specified expressions in the array. The execution of the `db.products.find({$nor:[{price:{$gt: 35}}, {price:{lte: 20}}]})` query will produce:

```javascript
{
   "_id" : ObjectId("54837b69f059b08503e200dd"),
   "name" : "Product 3",
   "description" : "Product 3 description",
   "price" : 30,
   "supplier" : {
      "name" : "Supplier 3",
      "telephone" : "+552177776666"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 9
      }
   ]
}
```
Element operators

To query a collection about our documents fields, we can use element operators.

The $exists operator will return all documents that have the specified field in the query. The execution of `db.products.find({sku: {$exists: true}})` will not return any document, because none of them have the field `sku`.

Evaluation operators

Evaluation operators are how we perform an assessment of an expression in MongoDB. We must take care with this kind of operator, especially if there is no index for the field we are using on the criteria. Let’s consider the evaluation operator:

- The $regex operator will return all values that match a regular expression. The execution of `db.products.find({name: {$regex: /2/}})` will return:

```json
{
   "_id" : ObjectId("54837b65f059b08503e200dc"),
   "name" : "Product 2",
   "description" : "Product 2 description",
   "price" : 20,
   "supplier" : {
      "name" : "Supplier 2",
      "telephone" : "+552188887777"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 10
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 2
      }
   ]
}
```
Array operators

When we are working with arrays on a query, we should use array operators. Let's consider the array operator:

- The `$elemMatch` operator will return all documents where the specified array field values have at least one element that match the query criteria conditions.

The `db.products.find({review: {elemMatch: {stars: {$gt: 5},
customer: {email: "customer@customer.com"}}}})` query will look at all the collection documents where the `review` field has documents, the `stars` field value is greater than 5, and customer email is `customer@customer.com`:

```
{
    "_id" : ObjectId("54837b65f059b08503e200dc"),
    "name" : "Product 2",
    "description" : "Product 2 description",
    "price" : 20,
    "supplier" : {
        "name" : "Supplier 2",
        "telephone" : "+552188887777"
    },
    "review" : [
        {
            "customer" : {
                "email" : "customer@customer.com"
            },
            "stars" : 10
        },
        {
            "customer" : {
                "email" : "customer2@customer.com"
            },
            "stars" : 2
        }
    ]
}
```
Besides the presented operators, we have: $mod, $text, $where, $all, $geoIntersects, $geoWithin, $nearSphere, $near, $size, and $comment. You can find more information regarding this in the MongoDB manual reference at http://docs.mongodb.org/manual/reference/operator/query/.

### Projections

Until now, we have executed queries where the presented result is the document as it is persisted in MongoDB. But, in order to optimize the network overhead between MongoDB and its clients, we should use projections.

As you saw at the beginning of the chapter, the `find` interface allows us to use two parameters. The second parameter is projections.

By using the same sample collection we used in the previous session, an example of a query with projection would be:

```javascript
db.products.find({price: {$not: {$gt: 10}}}, {name: 1, description: 1})
```

This query produces:

```
{ 
   "_id" : ObjectId("54837b61f059b08503e200db"),
   "name" : "Product 1",
   "description" : "Product 1 description"
}
```

The projection is a JSON document with all the fields we would like to present or hide, followed by 0 or 1, depending on what we want.

When a field is followed by a 0, then this field will not be shown in the resulting document. On the other hand, if the field is followed by a 1, then this means that it will be shown in the resulting document.

By default, the `_id` field has the value 1.

The `db.products.find({price: {$not: {$gt: 10}}}, { _id: 0, name: 1, "supplier.name": 1 })` query will show the following document:

```
{ "name" : "Product 1", "supplier" : { "name" : "Supplier 1" } }
```
Querying Documents

In fields that have an array as a value, we can use operators such as $elemMatch, $split, $slice, and $.

The db.products.find({price: {$gt: 20}}, {review: {$elemMatch: {stars: 5}}}) query will produce:

```json
{
   "_id" : ObjectId("54837b69f059b08503e200dd"),
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      }
   ]
}
```

Introducing the write operations

In MongoDB, we have three kinds of write operations: insert, update, and remove. To run these operations, MongoDB provides three interfaces: `db.document.insert`, `db.document.update`, and `db.document.remove`. The write operations in MongoDB are targeted to a specific collection and are atomic on the level of a single document.

The write operations are as important as the read operations when we are modeling documents in MongoDB. The atomicity in a level of a single document can determine whether we embed documents or not. We will go into this in a little more detail in Chapter 7, *Scaling*, but the activity of choosing a shard key will be decisive in whether we write an operation's performance because, depending on the key choice, we will write in one or many shards.

Also, another determining factor in a writing operations' performance is related to the MongoDB physical model. There are many recommendations given by 10gen but let's focus on those that have the greatest impact on our development. Due to MongoDB's update model, which is based on random I/O operations, it is recommended that you use solid state discs, or SSD. The solid state disk has superior performance compared to spinning disks, in terms of random I/O operations. Even though spinning disks are cheaper, and the cost to scale an infrastructure based on this kind of hardware is not that expensive either, the use of SSDs or increasing the RAM is still more effective. Studies on this subject show us that SSDs outperform spinning disks by 100 times for random I/O operations.
Another important thing to understand about write operations is how the documents are actually written on disk by MongoDB. MongoDB uses a journaling mechanism to write operations, and this mechanism uses a journal to write the change operation before we write it in the data files. This is very useful, especially when we have a dirty shutdown. MongoDB will use the journal files to recover the database state to a consistent state when the `mongod` process is restarted.

As stated in Chapter 2, *Data Modeling with MongoDB*, the BSON specification allows us to have a document with the maximum size of 16 MB. Since its 2.6 version, MongoDB uses a space allocation strategy for a record, or document, named "power of two sized allocation." As its name suggests, MongoDB will allocate to each document a size in bytes that is its size to the power of two (for example, 32, 64, 128, 256, 512, …), considering that the minimum size of a document is 32 bytes. This strategy allocates more space than the document really needs, giving it more space to grow.

**Inserts**

The `insert` interface is one of the possible ways of creating a new document in MongoDB. The `insert` interface has the following syntax:

```
db.collection.insert(
    <document or array of documents>,
    {
        writeConcern: <document>,
        ordered: <boolean>
    }
)
```

Here:

- `document or array of documents` is either a document or an array with one or many documents that should be created in the targeted collection.
- `writeConcern` is a document expressing the write concern.
- `ordered` should be a Boolean value, which if true will carry out an ordered process on the documents of the array, and if there is an error in a document, MongoDB will stop processing it. Otherwise, if the value is false, it will carry out an unordered process and it will not stop if an error occurs. By default, the value is `true`. 
Querying Documents

In the following example, we can see how an `insert` operation can be used:

```javascript
db.customers.insert({
    username: "customer1",
    email: "customer1@customer.com",
    password: hex_md5("customer1paswd")
});
```

As we did not specify a value for the `_id` field, it will be automatically generated with a unique `ObjectId` value. The document created by this `insert` operation is:

```
{
    "_id" : ObjectId("5487ada1db4ff374fd6ae6f5"),
    "username" : "customer1",
    "email" : "customer1@customer.com",
    "password" : "b1c5098d0c6074db325b0b9dddb068e1"
}
```

As you observed in the first paragraph of this section, the `insert` interface is not the only way to create new documents in MongoDB. By using the `upsert` option on updates, we could also create new documents. Let's go into more detail regarding this now.

# Updates

The `update` interface is used to modify previous existing documents in MongoDB, or even to create new ones. To select which document we would like to change, we will use a criterion. An update can modify the field values of a document or an entire document.

An update operation will modify only one document at a time. If the criterion matches more than one document, then it is necessary to pass a document with a `multi` parameter with the `true` value to the update interface. If the criteria matches no document and the `upsert` parameter is `true`, a new document will be created, or else it will update the matching document.

The `update` interface is represented as:

```javascript
db.collection.update(
    <query>,
    <update>,
    {
```
Here:

• query is the criteria
• update is the document containing the modification to be applied
• upsert is a Boolean value that, if true, creates a new document if the criteria does not match any document in the collection
• multi is a Boolean value that, if true, updates every document that meets the criteria
• writeConcern is a document expressing the write concern

Using the document created in the previous session, a sample update would be:

db.customers.update(
    {username: "customer1"},
    {$set: {email: "customer1@customer1.com"}}
)

The modified document is:

{
    "_id" : ObjectId("5487ada1db4ff374fd6ae6f5"),
    "username" : "customer1",
    "email" : "customer1@customer1.com",
    "password" : "b1c5098d0c6074db325b0b9dddb068e1"
}

The $set operator allows us to update only the email field of the matched documents.

Otherwise, you may have this update:

```
db.customers.update(
    {username: "customer1"},
    {email: "customer1@customer1.com"}
)
```

upsert: <boolean>,
multi: <boolean>,
writeConcern: <document>
In this case, the modified document would be:

```json
{
    "_id" : ObjectId("5487ad1db4ff374fd6ae6f5"),
    "email" : "customer1@customer1.com"
}
```

That is, without the `$set` operator, we modify the old document with the one passed as a parameter on the update. Besides the `$set` operator, we also have other important update operators:

- **$inc** increments the value of a field with the specified value:
  ```javascript
db.customers.update(
    {username: "johnclay"},
    {$inc: {"details.age": 1}}
  )
```
  This update will increment the field `details.age` by 1 in the matched documents.

- **$rename** will rename the specified field:
  ```javascript
db.customers.update(
    {email: "customer1@customer1.com"},
    {$rename: {username: "login"}}
  )
```
  This update will rename the field `username` to `login` in the matched documents.

- **$unset** will remove the field from the matched document:
  ```javascript
db.customers.update(
    {email: "customer1@customer1.com"},
    {$unset: {login: ""}}
  )
```
  This update will remove the `login` field from the matched documents.
As the write operations are atomic at the level of a single document, we can afford to be careless with the use of the preceding operators. All of them can be safely used.

**Write concerns**

Many of the discussions surrounding non-relational databases are related to the ACID concept. We, as database professionals, software engineers, architects, and developers, are fairly accustomed to the relational universe, and we spend a lot of time developing without caring about ACID matters.

Nevertheless, we should understand by now why we really have to take this matter into consideration, and how these simple four letters are essential in the non-relational world. In this section, we will discuss the letter *D*, which means durability, in MongoDB.

Durability in database systems is a property that tells us whether a write operation was successful, whether the transaction was committed, and whether the data was written on non-volatile memory in a durable medium, such as a hard disk.

Unlike relational database systems, the response to a write operation in NoSQL databases is determined by the client. Once again, we have the possibility to make a choice on our data modeling, addressing the specific needs of a client.

In MongoDB, the response of a successful write operation can have many levels of guarantee. This is what we call a write concern. The levels vary from weak to strong, and the client determines the strength of guarantee. It is possible for us to have, in the same collection, both a client that needs a strong write concern and another that needs a weak one.

The write concern levels that MongoDB offers us are:

- Unacknowledged
- Acknowledged
- Journaled
- Replica acknowledged
Unacknowledged
As its name suggests, with an unacknowledged write concern, the client will not attempt to respond to a write operation. If this is possible, only network errors will be captured. The following diagram shows that drivers will not wait that MongoDB acknowledge the receipt of write operations:

![Diagram showing the process of unacknowledged write concern]

In the following example, we have an `insert` operation in the `customers` collection with an unacknowledged write concern:

```javascript
db.customers.insert(
    {username: "customer1", email: "customer1@customer.com", password: hex_md5("customer1paswd")},
    {writeConcern: {w: 0}}
)
```

Acknowledged
With this write concern, the client will have an acknowledgement of the write operation, and see that it was written on the in-memory view of MongoDB. In this mode, the client can catch, among other things, network errors and duplicate keys. Since the 2.6 version of MongoDB, this is the default write concern.
As you saw earlier, we can’t guarantee that a write on the in-memory view of MongoDB will be persisted on the disk. In the event of a failure of MongoDB, the data in the in-memory view will be lost. The following diagram shows that drivers wait MongoDB acknowledge the receipt of write operations and applied the change to the in-memory view of data:

![Diagram showing write concern process]

In the following example, we have an `insert` operation in the `customers` collection with an acknowledged write concern:

```javascript
db.customers.insert(
    {username: "customer1", email: "customer1@customer.com", password: hex_md5("customer1paswd")},
    {writeConcern: {w: 1}}
)
```

**Journaled**

With a journaled write concern, the client will receive confirmation that the write operation was committed in the journal. Thus, the client will have a guarantee that the data will be persisted on the disk, even if something happens to MongoDB.
To reduce the latency when we use a journaled write concern, MongoDB will reduce the frequency in which it commits operations to the journal from the default value of 100 milliseconds to 30 milliseconds. The following diagram shows that drivers will wait MongoDB acknowledge the receipt of write operations only after committing the data to the journal:

In the following example, we have an insert in the customers collection with a journaled write concern:

```javascript
db.customers.insert(
  {username: "customer1", email: "customer1@customer.com", password: hex_md5("customer1passwd")},
  {writeConcern: {w: 1, j: true}}
)
```

## Replica acknowledged

When we are working with replica sets, it is important to be sure that a write operation was successful not only in the primary node, but also that it was propagated to members of the replica set. For this purpose, we use a replica acknowledged write concern.

By changing the default write concern to replica acknowledged, we can determine the number of members of the replica set from which we want the write operation confirmation. The following diagram shows that drivers will wait that MongoDB acknowledge the receipt of write operations on a specified number of the replica set members:
In the following example, we will wait until the write operation propagates to the primary and at least two secondary nodes:

```javascript
db.customers.insert(
    {username: "customer1", email: "customer1@customer.com", password: hex_md5("customer1passw")},
    {writeConcern: {w: 3}}
)
```

We should include a timeout property in milliseconds to avoid that a write operation remains blocked in a case of a node failure.

In the following example, we will wait until the write operation propagates to the primary and at least two secondary nodes, with a timeout of three seconds. If one of the two secondary nodes from which we are expecting a response fails, then the method times out after three seconds:

```javascript
db.customers.insert(
    {username: "customer1", email: "customer1@customer.com", password: hex_md5("customer1passw")},
    {writeConcern: {w: 3, wtimeout: 3000}}
)
```
Bulk writing documents

Sometimes it is quite useful to insert, update, or delete more than one record of your collection. MongoDB provides us with the capability to perform bulk write operations. A bulk operation works in a single collection, and can be either ordered or unordered.

As with the insert method, the behavior of an ordered bulk operation is to process records serially, and if an error occurs, MongoDB will return without processing any of the remaining operations.

The behavior of an unordered operation is to process in parallel, so if an error occurs, MongoDB will still process the remaining operations.

We also can determine the level of acknowledgement required for bulk write operations. Since its 2.6 version, MongoDB has introduced new bulk methods with which we can insert, update, or delete documents. However, we can make a bulk insert only by passing an array of documents on the insert method.

In the following example, we make a bulk insert using the insert method:

```javascript
db.customers.insert(

  [
    {
      username: "customer3", email: "customer3@customer.com", password: hex_md5("customer3paswd")},
    {
      username: "customer2", email: "customer2@customer.com", password: hex_md5("customer2paswd")},
    {
      username: "customer1", email: "customer1@customer.com", password: hex_md5("customer1paswd")
    }
  ]
)
```

In the following example, we make an unordered bulk insert using the new bulk methods:

```javascript
var bulk = db.customers.initializeUnorderedBulkOp();
bulk.insert({username: "customer1", email: "customer1@customer.com", password: hex_md5("customer1paswd")});
bulk.insert({username: "customer2", email: "customer2@customer.com", password: hex_md5("customer2paswd")});
bulk.insert({username: "customer3", email: "customer3@customer.com", password: hex_md5("customer3paswd")});
bulk.execute({w: "majority", wtimeout: 3000});
```
We should use all the power tools MongoDB provides us with, but not without paying all our possible attention. MongoDB has a limit of executing a maximum of 1,000 bulk operations at a time. So, if this limit is exceeded, MongoDB will divide the operations into groups of a maximum of 1,000 bulk operations.

**Summary**

In this chapter, you were hopefully able to better understand the read and write operations in MongoDB. Moreover, now, you should also understand why it is important that you already know the queries you need to execute even before the document modeling process. Finally, you learned how to use the MongoDB properties, such as atomicity, at the document level and saw how it can help us to produce better queries.

In the next chapter, you will see how a special data structure known as index can improve the execution of our queries.
Where to buy this book

You can buy MongoDB Data Modeling 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.