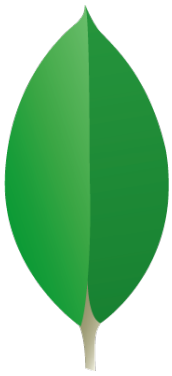
Web Technologies

WS 2015/16

NoSQL Databases with mongoDB



Author

Dr. Wolfgang Maaß

Mayur Bhamborae

Chair in Business Administration

esp. Information and Service Systems

Univ.-Prof. Dr.-Ing. Wolfgang Maaß

UNIVERSITÄT DES SAARLANDES

NoSQL Databases

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, objects and products, the frequency in which this data is accessed, and performance and processing needs. Relational databases, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today.

Types of NoSQL Databases

- **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.

- **Graph stores** are used to store information about networks, such as social connections. Graph stores include Neo4J and HyperGraphDB.

- **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. Examples of key-value stores are Riak and Voldemort. Some key-value stores, such as Redis, allow each value to have a type, such as "integer", which adds functionality.

- **Wide-column stores** such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.

The Benefits of NoSQL

When compared to relational databases, NoSQL databases are more scalable and provide superior performance and their data model addresses several issues that the relational model is not designed to address:

- 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

Dynamic Schemas

Relational databases require that schemas be defined before you can add data. For example, you might want to store data about your customers such as phone numbers, first and last name, address, city and state – a SQL database needs to know what you are storing in advance.

This fits poorly with agile development approaches, because each time you complete new features, the schema of your database often needs to change. So if you decide, a few iterations into development, that you'd like to store customers' favorite items in addition to their addresses and phone numbers, you'll need to add that column to the database, and then migrate the entire database to the new schema.

If the database is large, this is a very slow process that involves significant downtime. If you are frequently changing the data your application stores – because you are iterating rapidly – this downtime may also be frequent. There's also no way, using a relational database, to effectively address data that's completely unstructured or unknown in advance.

NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed.

Auto-sharding

Because of the way they are structured, relational databases usually scale vertically – a single server has to host the entire database to ensure reliability and continuous availability of data. This gets expensive quickly, places limits on scale, and creates a relatively small number of failure points for database infrastructure. The solution is to scale horizontally, by adding servers instead of concentrating more capacity in a single server.

"Sharding" a database across many server instances can be achieved with SQL databases, but usually is accomplished through SANs and other complex arrangements for making hardware act as a single server. Because the database does not provide this ability natively, development teams take on the work of deploying multiple relational databases across a number of machines. Data is stored in each database instance autonomously. Application code is developed to distribute the data, distribute queries, and aggregate the results of data across all of the database instances. Additional code must be developed to handle resource failures, to perform joins across the different databases, for data rebalancing, replication, and other requirements. Furthermore, many benefits of the relational database, such as transactional integrity, are compromised or eliminated when employing manual sharding.

NoSQL databases, on the other hand, usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool. Data and query load are automatically balanced across servers, and when a server goes down, it can be quickly and transparently replaced with no application disruption.

Cloud computing makes this significantly easier, with providers such as Amazon Web Services providing virtually unlimited capacity on demand, and taking care of all the necessary database administration tasks. Developers no longer need to construct complex, expensive platforms to support their applications, and can concentrate on writing application code. Commodity servers can provide the same processing and storage capabilities as a single high-end server for a fraction of the price.

Replication

Most NoSQL databases also support automatic replication, meaning that you get high availability and disaster recovery without involving separate applications to manage these tasks. The storage environment is essentially virtualized from the developer's perspective.

Integrated Caching

A number of products provide a caching tier for SQL database systems. These systems can improve read performance substantially, but they do not improve write performance, and they add complexity to system deployments. If your application is dominated by reads then a distributed cache should probably be considered, but if your application is dominated by writes or if you have a relatively even mix of reads and writes, then a distributed cache may not improve the overall experience of your end users.

Many NoSQL database technologies have excellent integrated caching capabilities, keeping frequently-used data in system memory as much as possible and removing the need for a separate caching layer that must be maintained.

NoSQL vs. SQL Summary

	SQL Databases	NoSQL Databases
Types	One type (SQL database) with minor variations	Many different types including key-value stores, document databases, wide-column stores, and graph databases
Development History	Developed in 1970s to deal with first wave of data storage applications	Developed in 2000s to deal with limitations of SQL databases, particularly concerning scale, replication and unstructured data storage
Examples	MySQL, Postgres, Oracle Database	MongoDB, Cassandra, HBase, Neo4j
Data Storage Model	Individual records (e.g., "employees") are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., "manager," "date hired," etc.), much like a spreadsheet. Separate data types are stored in separate tables, and then joined together when more complex queries are executed. For example, "offices" might be stored in one table, and "employees" in another. When a user wants to find the work address of an employee, the database engine joins the "employee" and "office" tables together to get all the information necessary.	Varies based on database type. For example, key-value stores function similarly to SQL databases, but have only two columns ("key" and "value"), with more complex information sometimes stored within the "value" columns. Document databases do away with the table-and-row model altogether, storing all relevant data together in single "document" in JSON, XML, or another format, which can nest values hierarchically.
Schemas	Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.	Typically dynamic. Records can add new information on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically.

Scaling	Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required.	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary.
Development Model	Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database)	Open-source
Supports Transactions	Yes, updates can be configured to complete entirely or not at all	In certain circumstances and at certain levels (e.g., document level vs. database level)
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE…	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product. Some provide strong consistency (e.g., MongoDB) whereas others offer eventual consistency (e.g., Cassandra)

Table 1: SQL vs NoSQL

Document Databases

Why do we prefer to use Document Databases?

One of the most popular ways of storing data is a document data model, where each record and its associated data is thought of as a "document". In a document database, such as MongoDB, everything related to a database object is encapsulated together. Storing data in this way has the following advantages:

- **Documents are independent units**, which makes performance better (related data is read contiguously off disk) and makes it easier to distribute data across multiple servers while preserving its locality.

- **Application logic is easier to write**. You don't have to translate between objects in your application and SQL queries, you can just turn the object model directly into a document.

- **Unstructured data can be stored easily**, since a document contains whatever keys and values the application logic requires. In addition, costly migrations are avoided since the database does not need to know its information schema in advance.

Document databases generally have very powerful query engines and indexing features that make it easy and fast to execute many different optimized queries. The strength of a document database's query language is an important differentiator between these databases.

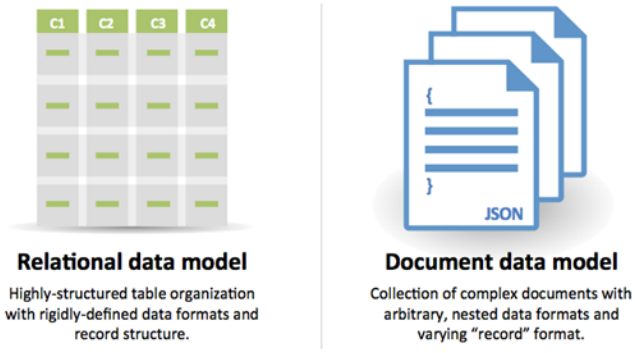


Figure 1: Relational Data Model vs. Document Data Model

MongoDB – The Most Popular Document Database

MongoDB's document data model makes it easy to build on, since it supports unstructured data natively and doesn't require costly and time-consuming migrations when application requirements change. MongoDB's documents are encoded in a JSON-like format, called BSON, which makes storage easy, is a natural fit for modern object-oriented programming methodologies, and is also lightweight, fast and traversable.

In addition, MongoDB supports rich queries and full indexes, distinguishing it from other document databases that make complex queries difficult or require a separate server layer to enable them. Its other features include automatic sharding, replication, and more.

Below are some of the features of MongoDB that make it an attractive choice.

- MongoDB supports Map and Reduce Aggregation Tools.
- JavaScripts are used instead of procedures.
- MongoDB is a schema less database.
- MongoDB supports secondary indexes and geospatial indexes.
- Simple to Administer the Mongo DB in cases of failures.
- MongoDB is designed to provide high performance.
- MongoDB stores files of any size without complicating your stack.

That's enough theory for now. Let's learn some actual MongoDB!

MongoDB Basics

To get started, there are six simple concepts we need to understand.

1. MongoDB has the same concept of a `database` with which you are likely already familiar (or a schema for you Oracle folks). Within a MongoDB instance you can have zero or more databases, each acting as high-level containers for everything else.

2. A database can have zero or more `collections`. A collection shares enough in common with a traditional `table` that you can safely think of the two as the same thing.

3. Collections are made up of zero or more `documents`. Again, a document can safely be thought of as a `row`.

4. A document is made up of one or more `fields`, which you can probably guess are a lot like `columns`.

5. `Indexes` in MongoDB function mostly like their RDBMS counterparts.

6. `Cursors` are different than the other five concepts but they are important enough, and often overlooked, that I think they are worthy of their own discussion. The important thing to understand about cursors is that when you ask MongoDB for data, it returns a pointer to the result set called a cursor, which we can do things to, such as counting or skipping ahead, before actually pulling down data.

To recap, 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.

Please refer to the E-Book which is linked in our Moodle to understand these concepts better.
(Page 7 – 11 : MongoDB – The Definitive Guide by Kristina Chodrow)

Starting MongoDB

After installing MongoDB on your machines, we can start the MongoDB Server by giving the following command.

```
mongod --smallfiles --dbpath /home/user/mongodb/data/db
```

```
user@webtech:~$ mongod --smallfiles --dbpath /home/user/mongodb/data/db
2015-11-11T21:13:51.525+0100 I JOURNAL  [initandlisten] journal dir=/home/user/mongodb/data/db/journal
2015-11-11T21:13:51.526+0100 I JOURNAL  [initandlisten] recover : no journal files present, no recovery needed
2015-11-11T21:13:51.600+0100 I JOURNAL  [durability] Durability thread started
2015-11-11T21:13:51.601+0100 I JOURNAL  [journal writer] Journal writer thread started
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] MongoDB starting : pid=6864 port=27017 dbpath=/home/user/mongodb/data/db 64-bit host=webtech
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten]
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten]
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten]
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] db version v3.0.7
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] git version: 6ce7cbe8c6b899552dadd907604559806aa2e9bd
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] build info: Linux ip-10-229-88-125 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86_64 BOOST
_LIB_VERSION=1_49
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] allocator: tcmalloc
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] options: { storage: { dbPath: "/home/user/mongodb/data/db", mmapv1: { smallFiles: true } } }
2015-11-11T21:13:51.603+0100 I NETWORK  [initandlisten] waiting for connections on port 27017
```

Then open a new terminal and give the `mongo` command.

```
user@webtech:~$ mongo
MongoDB shell version: 3.0.7
connecting to: test
Server has startup warnings:
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten]
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten]
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2015-11-11T21:13:51.601+0100 I CONTROL  [initandlisten]
>
```

Now you should have the Mongo Shell up and running.

Because this is a JavaScript shell, if you execute a method and omit the parentheses `()`, you'll see the method body rather than executing the method.

Creating an empty database

First we'll use the global use helper to switch databases, so go ahead and enter `use demoDatabase`. It doesn't matter that the database doesn't really exist yet.

```
> use demoDatabase
switched to db demoDatabase
>
```

Now we have an empty database called "demoDatabase". The `db` object refers to this database from now on.

Let's see information about this "empty" database.

```
> db.stats()
{
        "db" : "demoDatabase",
        "collections" : 0,
        "objects" : 0,
        "avgObjSize" : 0,
        "dataSize" : 0,
        "storageSize" : 0,
        "numExtents" : 0,
        "indexes" : 0,
        "indexSize" : 0,
        "fileSize" : 0,
        "ok" : 1
}
>
```

Creating an empty collection

Let us create a collection in which we will start storing our documents.

Refer to : https://docs.mongodb.org/v3.0/reference/method/db.createCollection/

```
> db.createCollection("bands")
{ "ok" : 1 }
>
```

Now we should have an empty collection called “bands”.

Inserting documents into the collection

Let us now insert a document into this collection.

```
> db.bands.insert({
...  name : 'Pink Floyd',
...  genre : 'progressive rock',
...  country : 'england',
...  members: 4
...  })
WriteResult({ "nInserted" : 1 })
>
```

Let’s check some stuff.

```
> show collections
bands
system.indexes
> db.getCollectionNames()
[ "bands", "system.indexes" ]
>
```

The collection `system.indexes` is created once per database and contains the information on our database's indexes.

So far we have created a database, then created a collection and added a document into our collection. All of this can be done in one shot too.

```
> db.cars.insert({
... brand : 'Toyota',
... model : 'Land Cruiser',
... year : 2015,
... })
WriteResult({ "nInserted" : 1 })
```

```
> db.getCollectionNames()
[ "bands", "cars", "system.indexes" ]
```

`db.stats()`

```
> db.stats()
{
        "db" : "demoDatabase",
        "collections" : 4,
        "objects" : 9,
        "avgObjSize" : 76.44444444444444,
        "dataSize" : 688,
        "storageSize" : 28672,
        "numExtents" : 4,
        "indexes" : 2,
        "indexSize" : 16352,
        "fileSize" : 16777216,
        "nsSizeMB" : 16,
        "extentFreeList" : {
                "num" : 0,
                "totalSize" : 0
        },
        "dataFileVersion" : {
                "major" : 4,
                "minor" : 22
        },
        "ok" : 1
}
```

`system.indexes` : contains the information on our database's indexes
`system.users` : user authentication and authorization.

Getting a list of documents in your collection

```
db.COLLECTION_NAME.find()
```

```
> db.bands.find()
{ "_id" : ObjectId("5643a9260bc6c3233700743b"), "name" : "Pink Floyd", "genre" : "progressive rock", "country" : "england", "members" : 4 }
>
```

```
> db.cars.find()
{ "_id" : ObjectId("5643abae0bc6c3233700743c"), "brand" : "Toyota", "model" : "Land Cruiser", "year" : 2015 }
>
```

```
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "demoDatabase.bands" }
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "demoDatabase.cars" }
>
```

Schema-less collections in action

Previously we had inserted this..

```
> db.bands.insert({
... name : 'Pink Floyd',
... genre : 'progressive rock',
... country : 'england',
... members: 4
... })
WriteResult({ "nInserted" : 1 })
>
```

Now let us insert a document which has a different schema

```
> db.bands.insert({
... name : 'Lynyrd Skynyrd',
... genre : 'southern rock',
... place : 'jacksonville, florida',
... members : {
... vocals : 'singer',
... lead_guitar : 'leadG',
... rhythm_guitar : 'rhythmG',
... drums : 'drummer',
... bass : 'bassman'
... }
... })
WriteResult({ "nInserted" : 1 })
```

Removing all documents from a collection

```
> db.bands.remove({})
WriteResult({ "nRemoved" : 2 })
>
```

Dropping a collection

```
> db.bands.drop()
true
```

```
> db.stats()
{
        "db" : "demoDatabase",
        "collections" : 3,
        "objects" : 5,
        "avgObjSize" : 73.6,
        "dataSize" : 368,
        "storageSize" : 20480,
        "numExtents" : 3,
        "indexes" : 1,
        "indexSize" : 8176,
        "fileSize" : 16777216,
        "nsSizeMB" : 16,
        "extentFreeList" : {
                "num" : 2,
                "totalSize" : 139264
        },
        "dataFileVersion" : {
                "major" : 4,
                "minor" : 22
        },
        "ok" : 1
}
```

Dropping an entire database

First, let's see the databases available on our system.

```
> show dbs
demoDatabase  0.031GB
local         0.031GB
```

Let's drop our demo Database.

```
> db.dropDatabase()
{ "dropped" : "demoDatabase", "ok" : 1 }
```

Do note that dropping the database has to happen via the db object of the database, which you want to drop.

Importing a bigger dataset into a collection

We will now import a JSON dataset, which we will be using for demo purposes.
Please download the JSON from here.

Use this command to import the data into a dataset. Observe how the name of the database and the collection is a part of the command.

```
user@rfc1918:~/Desktop$ mongoimport --db demoDatabase --collection unicorns --drop --file testData.json
2015-11-12T16:23:05.229+0100    connected to: localhost
2015-11-12T16:23:05.229+0100    dropping: demoDatabase.unicorns
2015-11-12T16:23:05.230+0100    imported 14 documents
user@rfc1918:~/Desktop$
```

Now that we have some test data in our database, we can see how to make queries and use selectors. Take a look into the JSON file just to get an idea of what kind of data we have within the database.

Querying the Database and Using Selectors

We have previously used the `db.COLLECTION_NAME.find()` to view all the contents of the collection.

```
> db.unicorns.find().pretty()
{
        "_id" : ObjectId("5644b45200448646aa5048c0"),
        "name" : "Starbust",
        "dob" : "01-02-1990",
        "loves" : [
                "carrot",
                "papaya"
        ],
        "weight" : 600,
        "gender" : "m",
        "score" : 63
}
{
        "_id" : ObjectId("5644b45200448646aa5048c1"),
        "name" : "Hothoof",
        "dob" : "08-12-1987",
        "family" : [
                {
                        "brother" : "Starbust"
                },
                {
                        "sister" : "Solnara"
```

{field: value} is used to find any documents where field is equal to value. {field1: value1, field2: value2} is how we do an and statement. The special $lt, $lte, $gt, $gte and$ne are used for less than, less than or equal, greater than, greater than or equal and not equal operations. For example, to get all male unicorns that weigh more than 700 pounds, we could do:

```
> db.unicorns.find({name : 'Kenny'}).pretty()
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa0"),
        "name" : "Kenny",
        "dob" : "17-03-1988",
        "loves" : [
                "grape",
                "lemon"
        ],
        "weight" : 690,
        "gender" : "m",
        "score" : 39
}
> db.unicorns.find({name : 'Kenny'})
{ "_id" : ObjectId("5644cdf0f577c483d7f34aa0"), "name" : "Kenny", "dob" : "17-03-1988", "loves" : [ "grape", "lemon" ], "weight" : 690, "gender" : "m",
"score" : 39 }
>
```

Special Selectors

- `$lt, $lte, $gt, $gte and $ne`

```
> db.unicorns.find({gender : 'm', weight : {$gt : 700}}).pretty()
{
        "_id" : ObjectId("5644cdf0f577c483d7f34a9c"),
        "name" : "Unicrom",
        "dob" : "11-09-2001",
        "loves" : [
                "energon",
                "redbull"
        ],
        "weight" : 984,
        "gender" : "m",
        "score" : 182
}
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa5"),
        "name" : "Dunx",
        "dob" : "17-08-1983",
        "loves" : [
                "grape",
                "watermelon"
        ],
        "weight" : 704,
        "gender" : "m",
        "score" : 165
}
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa6"),
        "name" : "Dunx",
        "dob" : "23-03-1988",
        "loves" : [
                "grape",
                "watermelon"
        ],
        "weight" : 704,
        "gender" : "m",
        "score" : 165
}
```

- `$exists` (whether a field exists or not)

```
> db.unicorns.find({score : {$exists : false} }).pretty()
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa4"),
        "name" : "Nimue",
        "dob" : "17-07-1981",
        "loves" : [
                "grape",
                "carrot"
        ],
        "weight" : 540,
        "gender" : "f"
}
```

- `$in` (match one of several values passed as an array)

```
> db.unicorns.find({name : {$in : ['Jack','Leia','Pilot']} }).pretty()
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa2"),
        "name" : "Leia",
        "dob" : "17-07-1978",
        "loves" : [
                "apple",
                "watermelon"
        ],
        "weight" : 601,
        "gender" : "f",
        "score" : 33
}
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa3"),
        "name" : "Pilot",
        "dob" : "17-03-1988",
        "loves" : [
                "apple",
                "watermelon"
        ],
        "weight" : 650,
        "gender" : "m",
        "score" : 54
}
>
```

The $or operator

```
> db.unicorns.find({ $or : [ {score : 33}, {loves : 'sugar'} ] }).pretty()
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa1"),
        "name" : "Raleigh",
        "dob" : "17-03-1988",
        "loves" : [
                "apple",
                "sugar"
        ],
        "weight" : 421,
        "gender" : "m",
        "score" : 2
}
{
        "_id" : ObjectId("5644cdf0f577c483d7f34aa2"),
        "name" : "Leia",
        "dob" : "17-07-1978",
        "loves" : [
                "apple",
                "watermelon"
        ],
        "weight" : 601,
        "gender" : "f",
        "score" : 33
}
>
```

MongoDB supports arrays as first class objects. This is an incredibly handy feature!

Updating

Update takes two parameters: the selector (where) to use and what updates to apply to fields.

Let us update the Document for the Unicorn Leia and set a correct date of birth.

```
> db.unicorns.find({name : 'Leia'}).pretty()
{
        "_id" : ObjectId("56450ae5ab989d2056868dc4"),
        "name" : "Leia",
        "dob" : "01-01-1990",
        "loves" : [
                "apple",
                "watermelon"
        ],
        "weight" : 601,
        "gender" : "f",
        "score" : 33
}
> db.unicorns.update( {name : 'Leia'}, {dob : '31-12-2010'} )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.unicorns.find({name : 'Leia'}).pretty()
> db.unicorns.find()
{ "_id" : ObjectId("56450ae5ab989d2056868dbb"), "name" : "Starbust", "dob" : "01-01-1990", "loves" : [ "carrot", "papaya"
{ "_id" : ObjectId("56450ae5ab989d2056868dbc"), "name" : "Hothoof", "dob" : "01-01-1990", "family" : [ { "brother" : "Sta
", "grape" ], "gender" : "m", "score" : 77 }
{ "_id" : ObjectId("56450ae5ab989d2056868dbd"), "name" : "Aurora", "dob" : "01-01-1990", "loves" : [ "carrot", "grape" ],
{ "_id" : ObjectId("56450ae5ab989d2056868dbe"), "name" : "Unicrom", "dob" : "01-01-1990", "loves" : [ "energon", "redbull
{ "_id" : ObjectId("56450ae5ab989d2056868dbf"), "name" : "Roooooodles", "dob" : "01-01-1990", "loves" : [ "apple" ], "wei
{ "_id" : ObjectId("56450ae5ab989d2056868dc0"), "name" : "Solnara", "dob" : "01-01-1990", "loves" : [ "apple", "carrot",
 80 }
{ "_id" : ObjectId("56450ae5ab989d2056868dc1"), "name" : "Ayna", "dob" : "01-01-1990", "loves" : [ "strawberry", "lemon"
{ "_id" : ObjectId("56450ae5ab989d2056868dc2"), "name" : "Kenny", "dob" : "01-01-1990", "loves" : [ "grape", "lemon" ], "
{ "_id" : ObjectId("56450ae5ab989d2056868dc3"), "name" : "Raleigh", "dob" : "01-01-1990", "loves" : [ "apple", "sugar" ],
{ "_id" : ObjectId("56450ae5ab989d2056868dc4"), "dob" : "31-12-2010" }
{ "_id" : ObjectId("56450ae5ab989d2056868dc5"), "name" : "Pilot", "dob" : "01-01-1990", "loves" : [ "apple", "watermelon"
{ "_id" : ObjectId("56450ae5ab989d2056868dc6"), "name" : "Nimue", "dob" : "01-01-1990", "loves" : [ "grape", "carrot" ],
{ "_id" : ObjectId("56450ae5ab989d2056868dc7"), "name" : "Dunx", "dob" : "01-01-1990", "loves" : [ "grape", "watermelon"
{ "_id" : ObjectId("56450ae5ab989d2056868dc8"), "name" : "Dirtball", "dob" : "01-01-1990", "loves" : [ "passionfruit", "b
89 }
>
```

Observed something strange?

So the update function REPLACED the old document!

To set the fields, we must use the $set operator!

We will use the update function and set operator to put Leia back into the collection.

```
> db.unicorns.update(
... {dob : '31-12-2010'},
... {$set :
... {
...     name: 'Leia',
...     dob: '01-01-1990',
...     loves: ['apple', 'watermelon'],
...     weight: 601,
...     gender: 'f',
...     score: 33
...
... }
... }
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.unicorns.find()
{ "_id" : ObjectId("56450ae5ab989d2056868dbb"), "name" : "Starbust", "dob" : "01-01-1990", "loves" : [ "carro
{ "_id" : ObjectId("56450ae5ab989d2056868dbc"), "name" : "Hothoof", "dob" : "01-01-1990", "family" : [ { "bro
", "grape" ], "gender" : "m", "score" : 77 }
{ "_id" : ObjectId("56450ae5ab989d2056868dbd"), "name" : "Aurora", "dob" : "01-01-1990", "loves" : [ "carrot"
{ "_id" : ObjectId("56450ae5ab989d2056868dbe"), "name" : "Unicrom", "dob" : "01-01-1990", "loves" : [ "energo
{ "_id" : ObjectId("56450ae5ab989d2056868dbf"), "name" : "Roooooodles", "dob" : "01-01-1990", "loves" : [ "ap
{ "_id" : ObjectId("56450ae5ab989d2056868dc0"), "name" : "Solnara", "dob" : "01-01-1990", "loves" : [ "apple"
 80 }
{ "_id" : ObjectId("56450ae5ab989d2056868dc1"), "name" : "Ayna", "dob" : "01-01-1990", "loves" : [ "strawberr
{ "_id" : ObjectId("56450ae5ab989d2056868dc2"), "name" : "Kenny", "dob" : "01-01-1990", "loves" : [ "grape",
{ "_id" : ObjectId("56450ae5ab989d2056868dc3"), "name" : "Raleigh", "dob" : "01-01-1990", "loves" : [ "apple"
{ "_id" : ObjectId("56450ae5ab989d2056868dc4"), "dob" : "01-01-1990", "name" : "Leia", "loves" : [ "apple", "
{ "_id" : ObjectId("56450ae5ab989d2056868dc5"), "name" : "Pilot", "dob" : "01-01-1990", "loves" : [ "apple",
{ "_id" : ObjectId("56450ae5ab989d2056868dc6"), "name" : "Nimue", "dob" : "01-01-1990", "loves" : [ "grape",
{ "_id" : ObjectId("56450ae5ab989d2056868dc7"), "name" : "Dunx", "dob" : "01-01-1990", "loves" : [ "grape", "
{ "_id" : ObjectId("56450ae5ab989d2056868dc8"), "name" : "Dirtball", "dob" : "01-01-1990", "loves" : [ "passi
89 }
>
```

The correct way to update Leia's DOB is

```
> db.unicorns.update(
...     {name: 'Leia'},
...     {$set: {dob: '21-12-2010'}}
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.unicorns.find({name :'Leia'})
{ "_id" : ObjectId("56450ae5ab989d2056868dc4"), "dob" : "21-12-2010", "name" : "Leia", "loves" : [ "apple
>
```

MongoDB is open source, so please go to the MongoDB official guide/website to get an idea of how to use the selectors. We cannot cover everything in detail here!