WT1 Webapplications
NoSQL Databases

HTW BERLIN

INTERNATIONALE MEDININFORMATIK

JAKUB MUELLER / DAVID SCHMOECKER / JONAS HEINIG
544832 / 544655 / 550169
WS18/19

Contents

1 Brief look into History

In 1998 the term NoSQL was first used by Carlo Strozzi while naming his open-source "relational" database that did not use SQL. In this context the meaning of the acronym NoSQL was "no SQL". In early 2009 Johan Oskarsson reintroduced the term NoSQL when he organized an event to discuss "open source distributed, non relational databases". The acronym NoSQL becomes a term to classify the movement of non-relational, distributed data stores. So the meaning changed from "no SQL" to "not only SQL". These non-relational systems were called "Document-oriented databases" before. Most of the early NoSQL systems did not attempt to provide atomicity, consistency, isolation and durability guarantees, contrary to the prevailing practice among relational database systems (ACID). NoSQL systems are more popular than ever before but relational databases are still dominating the market.

2 Terminology

The terminology in NoSQL databases are slightly different compared to relational models. Below are some changed terms with a short description:

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Join	Embedding & Linking

Figure 1: Terminology

A table or relation is defined as a set of tuples (rows) that have the same attributes (columns). A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints.

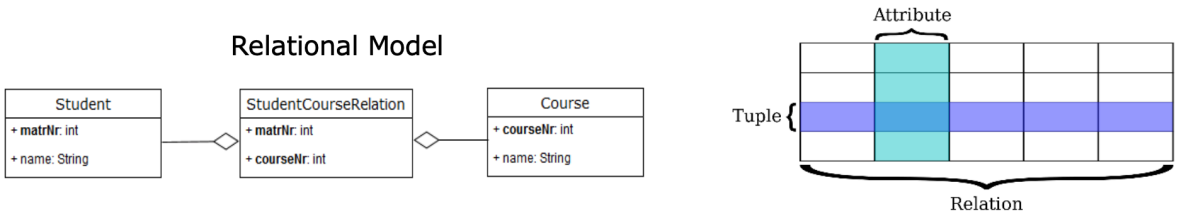

Figure 2: Relational Model

NoSQL databases like MongoDB using collections instead of tables. A collection stores different objects with the same attributes. A collection is a set of documents, so these objects are represented as a single document in the collection.

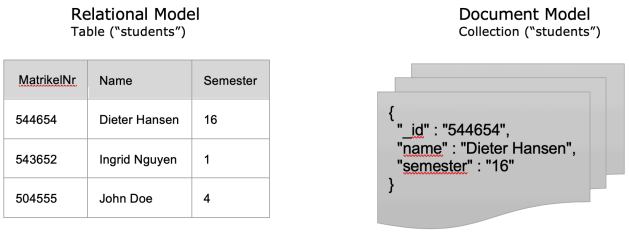

Figure 3: Table vs. Collection

These documents representing one row in the table. So one document is one object containing the information, split in columns which are the attributes. In relational models it is common to combine different tables with a "join", to get the needed information. Instead of joining different tables, MongoDB provides the possibility to embed or link documents to create a relation between documents in different collections.

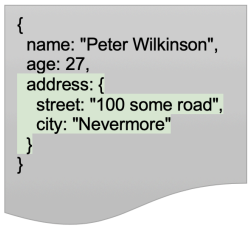

Figure 4: Embedding

You can embed an object into another object, If you add a field to your object, which contains another object. Here you can see a document of "user", where another object was added to store and access the address of the user.

Another way to store and access the address of a specific user, is to link both documents with a foreign key. You have just to write the key of the user as foreign key into the document of the address.

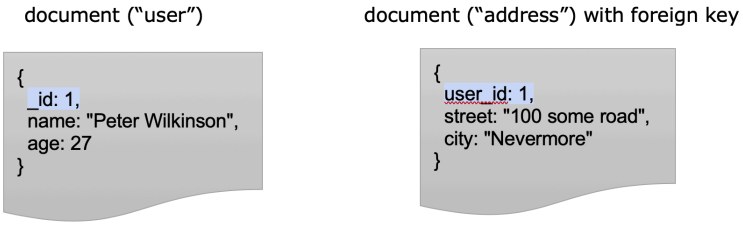

Figure 5: Linking

3 Types of NoSQL databases

Just as the term "Not only SQL" is pretty broad, there are a broad variety of different database-systems to which it applies. Many of these different systems can be categorised in one of the following four general types. [1]

3.1 Key-Value pair based databases

Key-Value pair based databases are the simplest type of NoSQL databases. As the name suggests they store key-value pairs. The key is a unique string which is also the only way to query the database.[2] The value on the other hand can be anything: a string, a number, a JSON document or even an entire new set of key-value pairs encapsulated in an object. Known key-value pair based databases are Amazon's DynamoDB and the open source database Riak.

3.2 Document databases

Document databases also called document store collections of documents that include the data to be saved. Depending on the specific database these documents have different encodings like for example XML, YAML or JSON. They are similar to Key-Value pair based databases. One key difference is that the content of the documents are used for queries. Next to MongoDB another well known document database is Apache CouchDB.

3.3 Graph databases

Graph databases are based on Graphs which consist of Nodes and Edges. A node represents and entity and edges represent relationships between those entities. This makes them suitable to model highly interconnected data, which other No-SQL databases struggle with.
Some graph databases use native graph storage that is optimised and designed for storing and managing graphs, others serialize the graph data into a relational, object-oriented database, or some other general-purpose data store. [3]

3.4 Column based databases

Column databases are based on tables just as relational databases. They use a different approach that is column oriented. They store each column separately to allow for quicker scans when only a small number of columns are involved. [4]

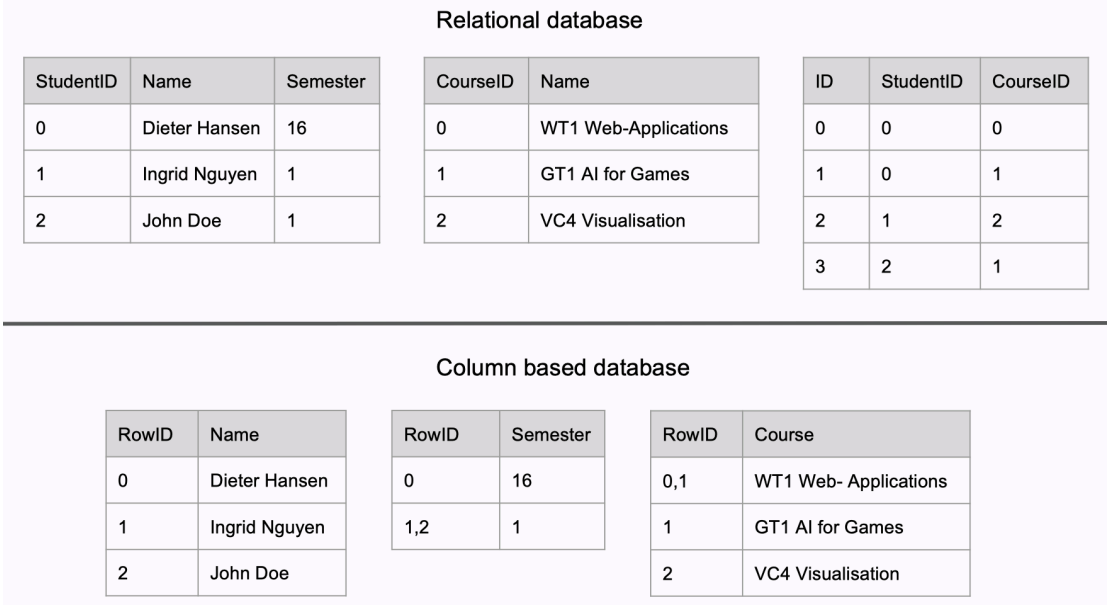

Figure 6: Standard relational and column based layout of tables that hold the same data

Examples for this type of databases are Apache Cassandra (which was developed by Facebook) and Google's Big Tables.

4 MongoDB

MongoDB is a very poplar and well supported open source No-SQL database. The main difference between traditional relational databases is that No-SQL databases do not store their data in tables, but rather in collections that are stored in JSON. Since JSON is the default object notation for JavaScript it works great with that language and makes development fast, efficient and easy. In general No-SQL databases claim to be highly scalable, efficient on storage, easily manageable and easy in their initial setup. On the contrary they often offer less functionality, are not so well-supported and there are not as many developers out there who can call themselves experts on the topic. In general it can be said that NO-SQL databases are a good choice when dealing with modern Web 2.0 applications, while they might not be the way to go when developing on different ground. [5] The following chart gives a basic overview about some differences and similarities between SQL and No-SQL databases.

	SQL	NO-SQL
Both have a	Database	Database
Structure of Database	Table	Collection
Entitiy	1 Row inside the table	1 Document inside the collection
Entity Structure	Table contains the same properties (columns) for all entities	Properties may vary between documents in the collection

Table 1: SQL and No-SQL Databases

4.1 MongoDB _id

The _id property is part of every document that is stored inside MongoDB. It serves as a primary key used to identify the document. In many SQL databases an auto-incrementing integer value is used as a primary key, MongoDB on the other hand uses a generated 12 byte value such as 507f1f77bcf86cd799439011. The _id is generated in the following way:

- First 4 bytes: Timestamp referring to moment in time the document was created
- Next 3 bytes: Machine-Identifiers. These make sure that 2 machines can insert at the exact same time without conflicts
- Next 2 bytes: Process-ID serves as another way to create unique identifiers
- Last 3 bytes: Counter, similar to what MYSQL would do

Even though it is not necessary nor recommended in most cases, we may change the _id property of any document to whatever we like without breaking anything.

4.2 Working with MongoDB

Adding a Document to the Database

Adding a document to a MongoDB database is as easy as calling a method on the database instance. If the instance we are referencing has not been created at this point, MongoDB will automatically create it for us.

```
1 db.collection("Tasks").insert({content: "Eat Lunch", done: false}).then((
      result) => {
2   console.log(result);
3 })
```

Getting a Document from the Database

In order to query data from a MongoDB collection we can use the built in find method. When we call the find method without providing any arguments all documents in the collection will be returned in an unstructured way.

```
1 db.collection("Tasks").find().then((result) => {
2   console.log(result); //Will print all documents
3 })
```

When we would only like to receive a single item we simply have to pass in some criteria to the find method. In the example below we would like to receive a document from the collection that has a content property with a value of "Eat Lunch".

```
1 db.collection("Tasks").find({content: "Eat Lunch"}).then((result) => {
2   console.log(result);
3 })
```

Deleting Documents from Database

The deleteMany method can be used to delete all documents that fit the criteria we pass in. In this example we are deleting all documents that have a content property with the value "Eat Lunch".

```
db.collection("Tasks").deleteMany({content: "Eat Lunch"}).then((result) => {
  console.log(result);
})
```

On the contrary we have the deleteOne method if we would only like to remove the first item from the database that matches our criteria. When an item has been found the method immediately stops.

```
db.collection("Tasks").deleteOne({content: "Eat Lunch"}).then((result) => {
  console.log(result);
})
```

The findOneAndDelete method works very similar to the previous one. Though the difference is that it does not only delete the first item that matches but also returns it. In this case we are deleting the first document that has a property done with the value false.

```
db.collection("Tasks").findOneAndDelete({done: false}).then((result) => {
  console.log(result);
})
```

When we would like to delete a document by its unique identifier we have to create a new ObjectID by providing the id string value of the document to the ObjectID constructor and pass it into our findOneAndDelete method. This can be done in the following way.

```
...
var ObjectID = mongoose.Schema.Types.ObjectId;
...
db.collection("Users").findOneAndDelete({_id: new ObjectID("5
    af6e4721342c78d496ddce6")}).then((result) => {
  console.log(result);
})
```

Updating Documents in the Database

When updating documents inside MongoDB we can use MongoDB's built in update operators to simplify the process. A complete list of all update operators can be found at `https://docs.mongodb.com/manual/reference/operator/update/`. The example below demonstrates a basic update tasks. We call the findOneAndUpdate method to update a value in a document using the built in update operator $set. In this case we want to set the done property of the matching element to true.

```
db.collection("Tasks").findOneAndUpdate({
    _id: new ObjectID("5afb4d7d924f0d0154a3c27e")
```

```
3   }, {
4       $set: {
5         done: true
6       }
7     }, {
8       returnOriginal: false //prevents returning the original document and
      returns the updated one instead
9   }).then((result) => {
10    console.log(result);
11  });
```

5 Mongoose

"Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box." [6] It is an npm library for MongoDB that simplifies development and reduces the amount of code we need to write. We can structure our data and use custom validators to ensure a clean and consistent database. Furthermore, the built in object relation mapping system allows quick and simple data exchange between our JavaScript applications and our databases. While MongoDB reinitialises its connection to the database for every action Mongoose keeps the connection open at all times.

5.1 Mongoose Models and Schemas

A Mongoose "Model is an object that gives you easy access to a named collection, allowing you to query the collection and use the Schema to validate any documents you save to that collection. It is created by combining a Schema, a Connection, and a collection name." [7]. When a model is being created (line 3 below), we pass in a model name as well as a Mongoose Schema to use for that model. Mongoose schemas are used to define the basic structure of the entities inside a MongoDB collection. In a way they are very similar to the table definition in relational databases. The code snippet below defines a simplified version of the Task collection that we have created for our WeekMe Application. We have a total of four properties, each with a different datatype. In line 6 we set the content property of a task document to required, meaning that it won't be possible to save a document to the collection without setting a value here. One line further below we use a constrain to ensure that the minimum length of the content property is one character. There is a variety of constrains and validators available which can be found on the official mongoose website at `https://mongoosejs.com/docs/validation.html`. In line 12 and line 16 we use the default keyword to define a fallback value whenever no value is passed in. The done property of the task document obviously should default to false as most Tasks won't be completed yet when a user is submitting them to our application. In line 19 we use a mongoose ObjectId as the datatype. This is usually done when we want to reference another document in a different collection using a foreign key. In this case we reference the owner of the task by providing the primary key of a document in the users collection.

```
1  var mongoose = require("mongoose");
2
```

```
3  var Task = mongoose.model("Task", {
4    content: {
5      type: String,
6      required: true,
7      minlength: 1,
8      trim: true //Removes any whitespaces in before or after the string
9    },
10   done: {
11     type: Boolean,
12     default: false
13   },
14   doneAt: {
15     type: Number,
16     default: null
17   },
18   _user: {
19     type: mongoose.Schema.Types.ObjectId,
20     required: true
21   }
22 });
23
24 module.exports = {Todo};
```

When the built in validators are not enough or we need to do some very specific validation we can define a custom validator in our schema. Within the user schema of the WeekMe application we use a custom validator (lines 9-12) to validate that the string value that is stored inside the email property matches basic email formatting. For this purpose we are using an npm module with the name validator, a library of string validators and sanitizers (more infos at: `https://www.npmjs.com/package/validator`).

```
1  var UserSchema = new mongoose.Schema({
2    email: {
3      type: String,
4      required: true,
5      lowercase: true,
6      minlength: 1,
7      trim: true, //Removes any whitespaces in before or after the string
8      unique: true, //Can only exist once in Collection
9      validate: {
10       validator: validator.isEmail,
11       message: '{VALUE} is not a valid email'
12     }
13   },
14   ...
```

5.2 Model Methods & Instance Methods

Mongoose provides us with the ability to define methods for our models as well as for the documents stored inside our collections. Model methods are defined using the statics keyword. They act on the entire collection while instance methods are defined using the methods keyword and only act on an

instance within a collection. Below I added one example of each type from the WeekMe application. The findByToken method is a model method that finds and returns a user from the collection based on the token we pass in. The generateAuthToken method is an instance method that generates a unique token for an instance of a user inside the user collection.

```
1  UserSchema.statics.findByToken = function (token){
2    var User = this;
3    var decoded;
4
5    try {
6      decoded = jwt.verify(token, process.env.JWT_SECRET);
7    } catch (e) {
8      return Promise.reject();
9    }
10
11   return User.findOne({
12     "_id": decoded._id,
13     "tokens.token": token,
14     "tokens.access": "auth"
15   });
16 };
17
18 UserSchema.methods.generateAuthToken = function () {
19   var user = this;
20   var access = "auth";
21   var token = jwt.sign({_id: user._id.toHexString(), access}, process.env.
       JWT_SECRET);
22
23   user.tokens = user.tokens.concat([{access, token}]);
24   return user.save().then(() => {
25     return token;
26   });
27 };
```

5.3 Mongoose Middleware

"Middleware (also called pre and post hooks) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins. Mongoose 4.x has 4 types of middleware: document middleware, model middleware, aggregate middleware, and query middleware. Document middleware is supported for the following document functions. In document middleware functions, this refers to the document." [8] Basically Mongoose middleware lets us run certain code before or after certain events occur (e.g. before we update a model). In the WeekMe application we define a Mongoose middleware on the schema to hash a password before saving it to the collection. This is achieved by using the .pre method and passing in a string value of "save" to tell the middleware at what event it should be executed. Similar to the way Express middleware is working we need to signal that our middleware has finished executing by calling the next() method in line 11 so that the application can continue in its execution.

```
1  UserSchema.pre("save", function (next) {
```

```
  var user = this;
  if(user.isModified("password")){
      bcrypt.genSalt(10, (err, salt) => {
        bcrypt.hash(user.password, salt, (err, hash) => {
        user.password = hash;
        next();
      });
    });
  } else {
    next(); //tell middleware to complete (mandatory)
  }
});
```

6 Advantages and Disadvantages of NoSQL

Working with NoSQL databases comes with a set of advantages and disadvantages when compared to working with relational databases. While relational databases all share a certain standard and mostly the same query language, each NoSQL database can be implemented very differently. Relational databases are also considered more stable and offer a larger set of tools. Due to their maturity and the fact that they have been around for almost 40 years there are much more skilled and experienced developers available. NoSQL databases on the other hand have different strengths. As the different types of NoSQL databases are very different by nature, different NoSQL databases are suited for different projects. In general their initial setup process is much quicker and due to their easy management database administrators can become obsolete. Due to their flexible schemas NoSQL databases, when used without a document modeling layer (ODM) like Mongoose, are well suited for projects where the structure of the data is expected to change frequently. Any large scaled system that handles and generates a lot of data should be considered to be implemented using a NoSQL database. Since these implementations can run in clusters of cheap machines they offer an improved scalability and are cost efficient while avoiding bottlenecks. Whenever data consistency is crucial, eg. for an accounting or payment management system, or when there are many complex relationships between the data, a relational database system should be preferred.

List of Figures

List of Tables

References

[1] [Online]. Available: http://www.nosql-database.org

[2] T. Trelle, *MongoDB*. Heidelberg: dpunkt.verlag, 2014, page 3.

[3] E. E. Ian Robinson, Jim Webber, *Graph Databases*. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly, 2015, page 5.

[4] [Online]. Available: https://dzone.com/articles/nosql-database-types-1

[5] [Online]. Available: https://www.hadoop360.datasciencecentral.com/blog/advantages-and-disadvantages-of-nosql-databases-what-you-should-k

[6] [Online]. Available: https://mongoosejs.com

[7] [Online]. Available: https://stackoverflow.com/questions/9127174/why-does-mongoose-have-both-schemas-and-models

[8] [Online]. Available: http://mongoosejs.com/docs/middleware.html