Unit 9: Introduction to NoSQL Databases

- NoSQL
- Classification of non-relational databases
- MongoDB
- Data types
- Data insertion
- Queries
- Updates
- Deletions
- Aggregations
- Indices

Introduction

- Relational databases (based on tables) are the most important and widespread model within this field
- However, there are other databases that use different ways of representing and managing information
- These databases do not make use the SQL language

NoSQL

- This term refers to those non-relational databases not using SQL
- It can also be understood as “Not Only SQL”
- Data do not require fixed structures (such as tables) to be stored
- As a consequence of the above, they do not allow operations such as the natural product (*join*)

NoSQL

- NoSQL database systems step back in time to decades ago, when object-oriented databases (like programming) started to be discussed
- It has been in the 21st century (in the middle of the Internet-driven digital age) when they have become more popular and important

NoSQL

- They are neither better nor worse than SQL based systems → they might just be better suited to certain situations or cases
- In general, they sacrifice flexibility in exchange for performance and scalability → optimization of recovery and aggregate operations

Classification of non-relational databases

Main types:

- Object-based (ObjectDB)
- Document-based (MongoDB)
- Graph-oriented (Neo4j)
- Key/value pairs (Redis, Cassandra)
- Tabular distributed (Hadoop, HBase)

MongoDB

- www.mongodb.com
- Distributed database
- Document-based
- Free and open source
- More flexible and easier to handle than a classic relational database
- Designed to scale well by adding servers

MongoDB

- Once installed, MongoDB consists of two parts: the server and the clients
- The server must provide service in the background while attending requests
- Every client connects to the running server and displays a shell interface to perform operations
- In its command line, it is possible to execute Javascript code, as well as to carry out operations on the databases

Type of data

- Documents stored in MongoDB databases use the JSON format. Example:
- {

  "first_name": "John",

  "last_name": "Doe",

  "age": 21,

  "skills": ["Programming", "Databases", "API"]

  }

Data types

- The data can embrace the following main types:
  - null → no value
  - Boolean → true, false
  - Numeric → basically integer or float values
  - String → text strings delimited by “ or '
  - Date → dates (expressed as Unix time)
  - Array → ordered collections of data (delimited by [])
  - Object ID → _id field of each document

Data types

- A document is kinda equivalent to a record or a row in the relational model
- Each document is stored in a collection → the equivalent of a table, so to speak
- Collection names have some syntax rules (they cannot contain like $, \0, start with system, …)
- Collections are referenced by their namespace → DB name + . + collection name

Let’s get to it!

- show dbs → list the available databases
- use <db name> → connect to the given database to work with it (it will create the database if it does not exist)
- db.createCollection(name, options) → create a new collection with the given parameters
- Nevertheless, collections may be created automatically simply by inserting new documents (at some named referenced database yet to be created)

Data insertion

- db.collection.insert(document) → [deprecated]
- db.collection.insertOne(document) → add a single document to the collection
- Example:

```
db.movies.insertOne({
    "title": "Avengers: Endgame"
})
```

Data insertion

- If the movies collection does not exist, it is created on the fly containing the previous document
- The document will always have an _id field, regardless of whether or not it is pointed out by the user

Data insertion

- db.collection.insertMany([document-1, …, document-n])
  → add multiple documents
- Example:

```
db.movies.insertMany({
    "title": "The Man of Steel"
}, {
    "title": "The Dark Knight"
})
```

Queries

- db.collection.findOne(query, projection)
- db.collection.find(query, projection)
- Examples:
  - db.movies.findOne()
  - db.movies.findOne({_id: 1})
  - db.movies.find()
  - db.movies.find({_id: 1})
  - db.movies.find({name: “Avengers: Endgame”})

Updates

- db.collection.updateOne(filter, update, options)
- Example:

```
db.movies.updateOne({
    _id: 1
}, {
    $set: {
        year: 2019
    }
})
```

Updates

- db.collection.updateMany(filter, update, options)
- Example:

```
db.movies.updateMany(
    { year: 2019 },
    { $set: { price: 20 } }
)
```

Deletions

- db.collection.deleteOne(filter, option)
- db.collection.deleteMany(filter, option)
- Examples:
  - db.movies.deleteOne()
  - db.movies.deleteOne({_id: 1})
  - db.movies.deleteMany({year: 2019})

Aggregations

- Operators: $avg, $count, $sum, $max, $min
- There are even more operators! (check docs)
- The aggregations are carried out via a pipeline
- Multiple chained documents are processed in stages
- Each stage performs an operation on input documents → then output documents are returned

Aggregations

- Operators: $avg, $count, $sum, $max, $min
- There are even more operators! (check docs)
- The aggregations are carried out via a pipeline
- Multiple chained documents are processed in stages
- Each stage performs an operation on input documents → then output documents are returned

Aggregations

- Example:

```
db.movies.aggregate([
    { $match: { year: 2019 } },
    { $count: "Number of movies" },
])
```

Aggregations

- Example:

```
db.movies.aggregate([
   {
      $group: {
         _id: "$year",
         numMoviesByYear: { $count: {} },
      },
   },
])
```

Aggregations

- Example:

```
db.movies.aggregate([
   {
      $group: {
         _id: "$year",
         priceByYear: { $avg: "$price" },
      },
   },
])
```

Aggregations

- Example:

```
db.movies.aggregate([
   {
      $group: {
         _id: "$year",
         priceByYear: { $avg: "$price" },
      },
   },
])
```

Aggregations

- Example:

```
db.movies.aggregate([
    { $match: { price: 20 } },
    { $group: {
        _id: "$year",
        totalIncomes: { $sum: "$income" }
     } },
     { $sort: { totalIncomes : -1} }
]);
```

Indices

- Indices speed up searches and make them more efficient
- db.collection.getIndexes() → see indices within a given collection
- db.collection.explain() → information and stats about the query plan in a given collection
- Example:

```
db.movies.find({
    title: 'Pirates of the Caribbean'
}).explain('executionStats')
```

Indices

- To create an index on a field and optimise data access and retrieval:
- db.movies.createIndex({title: 1}) → the index is created on the “title” field
- We can use a second parameter: {unique: true} → this enforces the field title to contain non-repeated values (error raised on insert otherwise)

Indices

- In order to delete an existing index, use: db.collection.dropIndex(index)
- Example: db.movies.dropIndex({‘title_1’})
- Alternative: db.movies.dropIndex({‘key’: { title: 1}})
- We can use a .dropIndexes() operation to remove all the indices within a collection but the _id one
- The default index on the _id field cannot be removed