Querying Relational and NoSQL Databases Systems (MongoDB)
Introduction

- How do we interact with a database?

  => Data Quering

In database terms, a query is used to retrieve data from the database.
SQL - Overview

A Relational Database Management System

• ...uses predefined schemas
• ...organizes data in Tables
• ...stores data in rows
• ...is consistent and available (CAP)
• ...is a mature technology
• ...supports the powerful and established query language SQL

• Examples:
  - Open Source: MySQL, PostgreSQL, ...
  - Commercial: Oracle, IBM DB2, MS SQL Server, ...
SQL Sublanguages

- **DML-Commands** (Data Manipulation Language)
  - Select-/Insert-/Update- and Delete operations

- **DDL-Commands** (Data Definition Language)
  - Describe, update and delete datastructures

- **DCL-Commands** (Data Control Language)
  - Manage permissions
## Create, Read, Update, Delete Queries

### CRUD Queries

<table>
<thead>
<tr>
<th></th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>INSERT INTO <code>tabellenname</code> (spalte1 , spalte2, spalte3, ...) VALUES (wert1, wert2, wert3, ...);</td>
<td>INSERT INTO <code>User</code> (UserID , UserName) VALUES (1, “Max Mustermann”);</td>
</tr>
<tr>
<td>READ</td>
<td>SELECT col1, col2, ... FROM <code>table1</code>, <code>table2</code>, ... WHERE &lt;condition&gt;</td>
<td>SELECT * FROM <code>User</code> WHERE UserID=1</td>
</tr>
<tr>
<td>UPDATE</td>
<td>UPDATE <code>tabellenname</code> SET <code>spalte1</code> = <code>wert1</code>, <code>spalte2</code> = <code>wert2</code>, ... WHERE <code>bedingung</code>;</td>
<td>UPDATE <code>User</code> SET <code>UserName</code> = “John Doe”, WHERE UserID = 1;</td>
</tr>
<tr>
<td>DELETE</td>
<td>DELETE FROM <code>tabellenname</code> WHERE <code>bedingung</code>,</td>
<td>DELETE FROM <code>User</code> WHERE UserID = 1;</td>
</tr>
</tbody>
</table>
Read Queries

• Assembles a set of tuples → strong mathematical foundation

• Declarative: defines the result, but not the „how“

```
SELECT Text, Date 
FROM Message 
WHERE GroupID = 1
```

<table>
<thead>
<tr>
<th>Text</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi John</td>
<td>2018-01-23</td>
</tr>
<tr>
<td>How are you?</td>
<td>2018-01-23</td>
</tr>
</tbody>
</table>
Select Queries with Aggregations

- Produce one value based on a set
- COUNT, AVG, MAX, MIN

```sql
SELECT COUNT(MessageID) AS MessageCount, Date
FROM Message
GROUP BY Date
```

<table>
<thead>
<tr>
<th>MessageCount</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2018-01-23</td>
</tr>
<tr>
<td>7</td>
<td>2018-01-24</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Select Queries over Multiple Tables with Joins (1)

- Multiple tables in FROM clause represent the cartesian product
- Join condition: defines which tuples match; shrinks the result set

Type 1

```sql
SELECT g.Name, m.Text, m.Date
FROM Group g
INNER JOIN Message m ON g.GroupID = m.GroupID
```

Type 2

```sql
SELECT g.Name, m.Text, m.Date
FROM Group g, Message m
WHERE g.GroupID = m.GroupID
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Text</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avengers</td>
<td>Hi John</td>
<td>2018-01-23</td>
</tr>
<tr>
<td>Avengers</td>
<td>How are you?</td>
<td>2018-01-23</td>
</tr>
<tr>
<td>BBQ</td>
<td>Salat Suggestions?</td>
<td>2018-03-25</td>
</tr>
</tbody>
</table>
Other join types in SQL

**LEFT JOIN**

```
SELECT g.Name, m.Text, m.Date
FROM Group g
LEFT JOIN Message m ON g.GroupID = m.GroupID
```

**RIGHT JOIN**

```
SELECT g.Name, m.Text, m.Date
FROM Group g
RIGHT JOIN Message m ON g.GroupID = m.GroupID
```

**FULL OUTER JOIN**

```
SELECT g.Name, m.Text, m.Date
FROM Group g
FULL OUTER JOIN Message m ON g.GroupID = m.GroupID
```
MongoDB is a document-oriented database management system which

• ...organizes data in Collections (similar to Tables in SQL)
• ...stores data in JSON Documents
• ...is scalable out of the box
• ...is consistent and partition tolerant (CAP)
• ...increases availability through replica sets
• ...divides workload through sharding
• ...has a less powerful query language than SQL
CRUD Interface

- „Simple“ command set in a JavaScript shell
  - Insert(o)
  - Delete(o)
  - Find(o, p)

- How to do joins?
  - Not at all: Denormalized storage of aggregate objects
  - At the application level: transfer the data
  - Do joins with MongoDB’s aggregate and lookup operations
  - Implement the join in a map-reduce statement
Example: A Chat Application

```
Message
  ID
  Text

User
  ID
  Name
  has
  0..*
  1..*
  0..*

Group
  ID
  Name
  has
  0..*

Avatar
  ID
  Name

GroupMessage
  MessageNumber
  0..*

UserMessage
  TimeReceived
  0..*

UserMessage
  from
  1

User
  to
  1

User
  admin
  1

User
  partOf
  0..*
```
Persistence Options

- Normalized Data Model / Relational Model (Multiple Collections)

- Denormalized Data Model / Aggregate Model (One collection)
Pros & Cons of Aggregate Data Model

Pros & Cons: Normalized Data Model
- Smaller footprint (especially true for SQL Databases, not so much for MongoDB)
- No duplicates -> updates are fast, inserts are fast
- Reads are fast on single tables (less data)
- Less need for „group by“ or „distinct“
- Read with joins are slow

Pros & Cons: Denormalized/Aggregate Data Model
- Excellent performance on reads (no joins necessary)
- Efficient Index Usage
- Updates/Inserts complex and costly
Example for Pros & Cons of Denormalized Data

When a denormalized data model rules:

```json
{   "GroupId": 1,
    "CreationDate": "2018-01-19",
    "Name": "Avengers",
    "Messages": [   {   "MessageID": "1",
                    "Text": "Hi John",...},
   {   "MessageID": "2",
                    "Text": "How are you?,...},
   ...  ]
}
```

Whole group including all messages can be retrieved with one read operation from the database

When a denormalized data model is problematic:

```json
{   "GroupId": 1,
    "Name": "A group",
    "Users": [   {   "UserID": "1",
                    "UserName": "John",...},...
    ]
}
```

```json
{   "GroupId": 2,
    "Name": "A 2nd group",
    "Users": [   {   "UserID": "1",
                    "UserName": "John",...},...
    ]
}
```

What happens if the user wants to update his name?

### Create, Read, Update, Delete Commands

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CREATE</strong></td>
<td></td>
</tr>
</tbody>
</table>
| `db.collection.insertOne(<document>)`      | `db.user.insertOne(
  { userID: 1,
    userName: "Max Mustermann" }
)` |
| **READ**                                   |         |
| `db.collection.find(…)`                   | `db.collection.find(  
  {userName : "John Doe"},
  {userId: 1})` |
| **UPDATE**                                 |         |
| `db.collection.updateOne(<filter>, <update>, <options>)` | `db.user.updateOne(
  { userID: 1 },
  { $set: {
    "userName": "John Doe"
  }}
)` |
| **DELETE**                                 |         |
| `db.collection.deleteOne(<filter>, {writeConcern: <document>, collation: <document> })` | `db.user.deleteOne(  
  { "userID" : 1 }
)` |
Simple Query on Normalized Data

- Selects documents in a collection or view and returns a cursor (a pointer to the result set) to the selected documents.
- Operators for filter: $gt, $eq, $gte, $lt. E.g.: {"GroupID": { $gt: 5 } }
- Logical operations: $and, $or. E.g.: {$and: [{operation1},{operation2}]}
Simple Query on Denormalized Data

- To retrieve the necessary data in a denormalized dataset we need to do the projection on a nested document.
- However, result is not exactly the same (Messages Collection returned)

```
SELECT Text, Date
FROM Message
WHERE GroupID = 1
```

```
Text         Date           
Hi John      2018-01-23   
How are you? 2018-01-23   
```

```
db.Message.find(
  {
    "groupID": 1,
    "messages.text": 1,
    "messages.date": 1
  }
)
```

```
{
  "_id": ObjectId("5afeb9753eccdda7b317e583"),
  "messages": [
    {
      "text": "Hi John", "Date": "2018-01-23"
    },
    {
      "text": "How are you?", "Date": "2018-01-23"
    }
  ]
}
```
More complex Queries on Denormalized Data

- For more complex queries, we need the `aggregate` function, which implements data processing pipelines

  \[
  \text{db.collection.aggregate( [ } \{ \text{ <stage> } \}, \ldots ] \text{ )}
  \]

- **Example stages include**
  - `$unwind`: deassembles an array field to output a document for each element
  - `$sort`: Reorders document stream by specified sort key
  - `$project`: to add/remove fields
  - `$match`: for filtering (like find())
Example for Query with „aggregate“

To retrieve the exact same result as with SQL, we need to use *aggregate* operation. Similar to pipe & filter architecture

```javascript
db.Group.aggregate([
  {$match: {"GroupID" : 1 }},
  {$unwind: "$Messages"},
  {$project:
    {
      "_id" : 0,
      "Text" : "$Messages.Text",
      "Date" : "$Messages.Date"
    }
  }
])
```

{{"Text" : "Hi John", "Date" : "2018-01-19"}}

{{"Text" : "How are you?", "Date" : "2018-01-19"}}
Aggregate with Selection on Nested Data

```javascript
db.Group.find(
  {
    "Messages.Date": "2018-01-19"
  }
)

```

```javascript
{…
  "Messages": [
    {…, "Text": "Hi John"},
    {…, "Text": "How are you?"},
    {…, "Text": "Wrong Date?"}
  ]
}
```

```javascript
db.Group.aggregate([
  { $unwind: '$Messages' },
  { $match: {
      "Messages.Date": "2018-01-19"
    }},
  { $project: {
      "Text": "$Messages.Text"
    }
  }
]

```

```javascript
{…
  "Text": "Hi John"
}
```

```javascript
{…
  "Text": "How are you?"
}
```

```javascript
{…
  "Text": "Wrong Date?"
}
```

From 20th of January
Join Query on Normalized Data

- Only one join possible as part of aggregate: Left Outer Join

```
{  "GroupID" : 1,
   "Name" : "Avengers",
   "GroupMessages" : [
      {...,"Text" : "How are you?"},
      {...,"Text" : "Hi John"}
   ]
}
```

```
db.Group.aggregate({
   $lookup:
   {
      from: "Message",
      localField: "GroupID",
      foreignField: "GroupID",
      as: "GroupMessages"
   }
})
```
Join Query on Denormalized/Aggregated Data

- Same simple query as before can be used

```javascript
db.Message.find(
{
  "groupID": 1,
  "messages.text": 1,
  "messages.date": 1
}
)
```

```javascript
{
  "_id": ObjectId("5afeb9753eccdda7b317e583"),
  "messages": [
    {
      "text": "Hi John", "Date": "2018-01-23",
    },
    {
      "text": "How are you?", "Date": "2018-01-23"
    }
  ]
}
```
To Join or Not To Join? (Rules of thumb)

- Avoid joins and use denormalized model when:
  - There is a contains relationship
  - There is a one-to-many relationship on which the „childs“ always appear with the „parent“
  - Maybe the use case allows duplicated/redundant data

- Use joins and a normalized model when:
  - If embedding results in duplicated/redundant data which might be problematic (update anomalies) and read performance does not dramatically increase
  - For many-to-many relationships
  - For data sets with deep nesting (>5 levels)
MongoDB – Summary

- MongoDB has a simple data model and a quite powerful query language (compared to other NoSQL systems)
  - The syntax as JavaScript objects is too complex
- Aggregate operation allows implementation of complex queries as data processing pipelines
- MongoDB can be fast, if data model and queries are applied efficiently
  - ... but do not trust performance comparisons stating something like „MongoDB is 5x faster than MySQL“
  → Database tuning is complex
Takeaways

- Relational Database Systems
  - Basic Concepts of SQL
  - Simple Queries (C(R)UD)
  - Select Queries & Aggregations
  - J o i n s  (Left/Right/Inner)

- MongoDB
  - NoSQL Database System
  - Basic Concepts
  - Simple Queries (C(R)UD)
  - Persistence Options
  - Aggregation Pipeline
  - Left Join