Milestone 2

of

Project:
Pharmaceutical Company

for

052400-1 VU Information Management & Systems Engineering (SS 2022)

Student names:	FILLIES Claas, A12047732
	WAGER Clemens, A01635477
Degree program:	Master of Computational Science
	Master of Computational Science
Supervisors:	Prof. Dipl. Ing . Dr. Erich Schikuta
	Dipl.-Ing . Ralph Vigne , Bakk

Vienna, 30.06.2022

Content

Disclaimer

For this IMSE project an existing DBS project was used which was developed by Clemens Wager in SS2021 for the course 051031 VU Database Systems (2021S).

Also with this Milestone M2 we submit a corrected and slightly altered version of M1.

2.1. The RDBMS Part (Phase 1)

2.1.0 Introduction

The web application centers around a company and enables not only registration of products and clients but also gives insight on the company's business. For detailed information on the domain see the submitted PDF file "*M1_FILLIES_WAGER_IMSE_SS22_corrected.pdf*".

2.1.1. Configuration of Infrastructure

To run the system and make changes to the database, the user downloads the submitted folder and navigates into it with a CLI. If the local machine is running on Windows, a docker daemon is required (f.e. Docker Desktop). Build and start the docker application with the command

``docker-compose up --build`` .

The docker application will automatically establish a connection to the two databases. The web interface of the MongoDB can be accessed by navigating to http://127.0.0.1:5000 in a web browser. All actions required are well described in comments and are executable by the press of the corresponding button. At first the user lands on the starting page, where they can rebuild and refill the SQL database. This process takes about two minutes and must not be interrupted. Then the user can fill the NoSQL database by migrating the data from the SQL to the NoSQL database. After both actions have been completed, the use will see a small notification. Then they can click the last button to proceed to the next page, to query the database. Here all functions are well described.

The docker application will also connect to the SQL database server. The corresponding interface can be accessed via http://127.0.0.1:8000 in web browser. This webpage is to be used once the SQL database is filled. All use cases and reports can be executed on this page and results will lead to a result page.

2.1.2. Structure of containers

The *Flask* container contains `app.py`, which sets up the SQL database (CREATE and INSERT), enables data migration to NoSQL database and the functionality for the NoSQL interface. The NoSQL interface is defined in php language in the same container. To implement the use cases and reports on the NoSQL database the python `Flask` framework was used. Here the library `mysql-connector` enables communication between the interface and the database.

The *sql_interface* container defines the web interface for the SQL database and also defines the functions to query the database for the functionality. The use cases and reports for the SQL database have been implemented in php using the library `mysqli` which enables communication between the php interface and the MySQL database.

2.1.3. Data import

The MySQL database is filled with randomly generated data stored in csv files. There will be 50 to 300 rows inserted in ten tables. This process takes about two minutes and is realized in `app.py`. This Python script connects to the database, creates the required tables, and inserts all data.

2.1.4. Implementation of a Web system

The system incorporates two cloud based databases. The first one is written in MySQL and hosted on *Free MySQL Hosting*[1]. The second database is defined in NoSQL and is hosted on *MongoDB Atlas*[2]. We chose this approach, because cloud computing has various upsides that we also discussed in the lecture. To name some: Our system has higher availability, easy configuration, more access possibilities and does not require additional hardware on the user side.

[1] Free MySQL Hosting www.*freemysqlhosting.net*

[2] MongoDb Atlas www.mongodb.com/atlas/database

2.2 NoSQL Design (Phase 2)

2.2.1. Logical/Physical database design of your RDBMS

```
-- ------------------------------------------------------------------------------
-- CREATE TABLE
-- ------------------------------------------------------------------------------
 CREATE TABLE IF NOT EXISTS Client(
    ID_client          INTEGER AUTO_INCREMENT,
    Client_Name        VARCHAR(50) NOT NULL,
    Country_Name       VARCHAR(25) NOT NULL, -- FK
    CONSTRAINT PK_client PRIMARY KEY (ID_client)
);

CREATE TABLE IF NOT EXISTS Product(
    ID_product          INTEGER AUTO_INCREMENT,
    Product_Name        VARCHAR(50) UNIQUE NOT NULL,
    Price               DECIMAL(8,2) NOT NULL,
    Indication          VARCHAR(40) NOT NULL,
    CONSTRAINT PK_product PRIMARY KEY (ID_product)
);

CREATE TABLE IF NOT EXISTS Orders(
    ID_Orders          INTEGER AUTO_INCREMENT,
    ID_Product         INTEGER NOT NULL, -- FK
    ID_Client          INTEGER NOT NULL, -- FK
    Order_Date         DATE    NOT NULL,
    Quantity           INTEGER NOT NULL,
    CONSTRAINT PK_orders PRIMARY KEY (ID_orders),
    CONSTRAINT quantity_range_check CHECK(Quantity >= 10 AND Quantity <= 999999)
);

CREATE TABLE IF NOT EXISTS Campaign(
    ID_Product          INTEGER NOT NULL, -- FK
    Campaign_Name       VARCHAR(40) NOT NULL, -- not UNIQUE because weak entity
    Start_date          DATE NOT NULL,
    End_date            DATE NOT NULL,
    CONSTRAINT PK_campaign PRIMARY KEY (ID_product, Campaign_Name),
```

```
    CONSTRAINT camp_date CHECK(Start_date < End_date) -- CHECK CONSTRAINT
);



CREATE TABLE IF NOT EXISTS Employee(
    ID_employee         INTEGER AUTO_INCREMENT,
    Firstname           VARCHAR(25) NOT NULL,
    Lastname            VARCHAR(25) NOT NULL,
    Gender              VARCHAR(1)  NOT NULL,
    Salary              DECIMAL(8,2) DEFAULT 1500,
    Team_leader         INTEGER, -- FK
    Hire_date           DATE NOT NULL,
    CONSTRAINT PK_emp PRIMARY KEY (ID_employee),
    CONSTRAINT emp_gender CHECK(Gender IN ('F','M','D')) -- CHECK CONSTRAINT
);

CREATE TABLE IF NOT EXISTS Marketing_emp(
    ID_employee     INTEGER, -- FK
    Occupation      VARCHAR(50),
    CONSTRAINT PK_mark PRIMARY KEY (ID_employee)
);

CREATE TABLE IF NOT EXISTS Advertises(
    ID_employee         INTEGER, -- FK
    ID_product          INTEGER, -- FK
    CONSTRAINT PK_adv PRIMARY KEY (ID_employee, ID_product)
);

CREATE TABLE IF NOT EXISTS General_Manager(
    ID_employee         INTEGER, -- FK
    ID_region           INTEGER UNIQUE NOT NULL, -- FK
    CONSTRAINT PK_GM PRIMARY KEY (ID_employee)
);



CREATE TABLE IF NOT EXISTS Region(
    ID_region           INTEGER AUTO_INCREMENT,
```

```
    Region_Name         VARCHAR(25) UNIQUE NOT NULL,
    CONSTRAINT PK_region PRIMARY KEY (ID_region)
);

CREATE TABLE IF NOT EXISTS Country(
    Country_Name        VARCHAR(25) NOT NULL, -- PK
    ID_region           INTEGER NOT NULL,  -- FK
    CONSTRAINT PK_country PRIMARY KEY (Country_Name)
);

-- ------------------------------------------------------------------------------
-- ADD FOREIGN KEY CONSTRAINTS
-- ------------------------------------------------------------------------------
ALTER TABLE Client
    ADD CONSTRAINT FK_client_country
    FOREIGN KEY (Country_Name) -- referencing table
    REFERENCES Country (Country_Name) -- referenced table
    ON DELETE CASCADE -- integrity
;

ALTER TABLE Orders
    ADD CONSTRAINT FK_orders_product   FOREIGN KEY (ID_product)
    REFERENCES Product (ID_product) ON DELETE CASCADE
;

ALTER TABLE Orders
    ADD CONSTRAINT FK_orders_client FOREIGN KEY (ID_client)
    REFERENCES Client (ID_client) ON DELETE CASCADE
;

ALTER TABLE Campaign
    ADD CONSTRAINT FK_camp_product FOREIGN KEY (ID_product)
    REFERENCES Product (ID_product) ON DELETE CASCADE
;
-- Team_leader ID
ALTER TABLE Employee
    ADD CONSTRAINT FK_emp_leader FOREIGN KEY (Team_leader)
```

```
    REFERENCES Employee (ID_employee) ON DELETE CASCADE
;

ALTER TABLE Marketing_emp
    ADD CONSTRAINT FK_mark_emp FOREIGN KEY (ID_employee)
    REFERENCES Employee (ID_employee) ON DELETE CASCADE
;

ALTER TABLE Advertises
    ADD CONSTRAINT FK_adv_prod FOREIGN KEY (ID_product)
    REFERENCES Product (ID_product) ON DELETE CASCADE
;

ALTER TABLE Advertises
    ADD CONSTRAINT FK_adv_mark FOREIGN KEY (ID_employee)
    REFERENCES Marketing_emp (ID_employee) ON DELETE CASCADE
;

ALTER TABLE General_Manager
    ADD CONSTRAINT FK_gm_emp FOREIGN KEY (ID_employee)
    REFERENCES Employee (ID_employee) ON DELETE CASCADE
;

ALTER TABLE General_Manager
    ADD CONSTRAINT FK_gm_reg FOREIGN KEY (ID_region)
    REFERENCES Region (ID_region) ON DELETE CASCADE
;

ALTER TABLE Country
    ADD CONSTRAINT FK_country_reg FOREIGN KEY (ID_region)
    REFERENCES Region (ID_region) ON DELETE CASCADE
;
```

2.2.2. NoSQL: Reasons for design (Collections, documents, ...)

The tables of the SQL schema are merged and embedded into different NoSQL Collections. The tables "Client" and "Country" are merged into the collection "**Client**" where the *client_name* was chosen as index. NoSQL databases structures do not require foreign keys or specific IDs. As *client_names* are unique in this domain, we can safely drop the unique numerical identifier *ID_client*. The collection "Client" was designed to hold all the Data needed for the second usecase as well the second report, that is the reason why it holds the *client_name* and the country in which the client resides.

The "**Employee**" table is merged into the "Employee" collection together with the information of the recursive relation *team_leader* . The *firstname* and *lastname* are combined into *name*. The unique id was replaced by the Employee name. The Employee collection is not required for any for the usecases or repots but since all the data from the SQL DB needs to be migrated to the NoSQL DB and there is no intuitive place to embed every employee into a different collection, we decided to save them into their own collection. The Employee collection also comes in handy for the dropdown bar of usecase 2 in the NoSQL interface.

The collection "**Order**" comprises data from the table "Orders", "Client" and "Product" and is designed to hold all the data needed for usecase four. The latter two tables are used to replace the foreign keys with the corresponding names. Use case four requires an order ID and the time of the order.

The collection "**Product**" is more complex than the first few collections. The complete table "Campaign" is embedded as array for each Product. As "Campaign" is a weak entity in the SQL schema that depends on "Product", this was a logical choice. Likewise the tables "Advertises" and "Marketing_emp" are embedded into this collection. Their id was also replaced by their complete employee name, because rows from "Marketing_emp" are part of the table "Employees". The "product" collection is logically split into two parts. The first part is the information needed for usecase one and report one, which are the name of the product its indications and the price. The second parts is the data which needs to be stored in the NoSQL DB like "Campaign" or "Marketing_emp", however it this data must not be accessed. Since they relate only in one-to-n relationships there is no need for a separate collection.

The collection "**Region**" contains the data of "Country", which becomes an embedded array. The collection is designed to hold all the data for the third usecase. Regions contain at least 2 countries but always less than 100. Since a country can also exist in one region queries will not have any problems searching through this collection. The "General_Manager" table, which has a one-to-one relation in the SQL schema is naturally also part of the "Regions" collections. The latter is part of the corresponding document with all employee information of that entity.

Our fundamental idea was to design one collection for every usecase. That means that we have data redundancy, like the name of all the countries which are saved in the “Region” and in the “Client” collection. And that we have an in “inefficient” migrating progress when we setup the NoSQL DB, because we are saving the same data in two different locations. However, with theses specially designed collections we can efficiently query the database for our usecases, because we do not need any join-like commands (e.g. $lookup or related) in our NoSQL queries.

Except from the “employee” collection this goal was achieved. We are sacrificing runtime in the one-time migration process to have efficient queries for the reoccurring use cases.

2.2.3. NoSQL indexing

Propose how data should be indexed in MongoDB to be beneficial for your application.

Instead of using numerical identifiers the NoSQL collections use the entitys’ names. The ID of the table “Client” becomes the *client_name* of the client. The ID in “Employee” becomes the combined *firstname* plus *lastname* of each employee. The *product_name* is used as ID for each product. The ID_orders of the “Order” table is maintained in the NoSQL database. As described above there is no reason to connect any collections for our queries so there is also not a reason for any keys which would enable an easy access of the DB while quiring thought a different one. Without the need of keys and with unique names we where able to replace every IDs besides the Order ID. The Order ID was caped because our orders do not have any other unique attributes like a name.

2.2.4. Comparison of main use cases and reports (SQL vs. NoSQL)

Main use case 1 – Register new client (by Clemens Wager)

SQL Code:

```
INSERT INTO Client (client_name, country_name)
VALUES ('{$client_client_name}', '{$client_country_name}');
```

NoSQL Code in flask:

```
@app.route("/Af2", methods=['GET'])
def Af2():
	#------------ABFRAGE2 ----------------------
	nameOfCountry = str(request.values.get("country"))
	nameOfClient = str(request.values.get("name"))
	newDoc = { "$set":{ "name": nameOfClient, "country_name":nameOfCountry }}
	collection=dbDrug['Client']
	collectionRegion=dbDrug['Region']
	res= f"The Client with the Name {nameOfClient} was inserted"

	filter = { 'name': nameOfClient }
	count =  (collectionRegion.count_documents({ "Countries" : nameOfCountry }))
	if count == 1:
		filter = { 'name': nameOfClient }
		collection.update_one(filter, newDoc,  True)
	else:
		res= "The compnay does not supply the country"

	session["res2"] = res

	return render_template('credits.html',t=title,h=heading,
	 results= resultFromSession())
```

Comparison:

In the MySQL design the insert statement is rather simple. The table “Client” has an automatically incremented ID, so only the name and the country are given in the statement.

The NoSQL statement uses the update_one() function, because if a row already exists, there will be no error thrown, but the existing data will be updated. As Input data every string entry is accepted for a name and a non existing country will not be updated. Also this function avoids data redundancy, as only one client with a certain name is stored in the NoSQL database.

Main use case 2 – Register new product (by Claas Fillies)

SQL Code:

```
INSERT INTO Product (Product_Name, Price, Indication)
VALUES ('{$product_name}', '{$price}', '{$indication}');
```

NoSQL Code in flask:

```
@app.route("/Af1", methods=['GET'])
def Af1():
	#------------ ABFRAGE1 ----------------------
	nameOfProduct = str(request.values.get("name"))
	indication = str(request.values.get("indication"))
	try:
		price = int(request.values.get("price"))
	except Exception:
		session["Af1Info"] = f"Price need to be an int3"
		return render_template('credits.html',t=title,h=heading,
		results = resultFromSession())
	try:
		campaignslocal = session["campaigns"]
	except Exception:
		campaignslocal = []

	try:
		marketingEmplocal = session["marketingEmp"]
	except Exception:
		marketingEmplocal = []
```

```
        newDoc = { "$set":{ "name": nameOfProduct, "indications": indication, "price":
        price , "Campaigns": campaignslocal, "Marketing_Emp": marketingEmplocal}}

        collection=dbDrug['Product']
        filter = { 'name': nameOfProduct }
        collection.update_one(filter, newDoc,  True)
        res = "The following product was inserted: " + nameOfProduct
        print(f"[INFO] {res}")

        session["res1"] = res

        return render_template('credits.html',t=title,h=heading, results=
        resultFromSession())
```

Comparison:

In the MySQL design the insert statement is rather simple. The table “Product” has an automatically incremented ID, so only the name, price and the indication are given in the statement.

The NoSQL insert-statement can insert more detailed information. Thus, the user can input a campaign and a responsible marketing employee before submitting the product information. That way when the product is stored to the database, the new document will also hold the data on the corresponding campaigns and marketing employees. The function update_one() is used here again, to avoid creation of redundant documents, where e.g. one has a campaign and the other one does not. If a product with a certain name already exists in the database, its information will be overwritten.

Product report: What are the most expensive products under the given price limit?
(by Claas Fillies)

A report on the five most expensive products, which cost less than the given price limit. Users can filter by setting the price limit.

SQL Code:

```
SELECT p.ID_product, p.Product_Name, p.Indication, p.Price
     FROM Product p
     WHERE Price IN
       ( SELECT Price
         FROM Product
         WHERE Price < '{$price_limit}')
     ORDER BY Price DESC
     LIMIT 5;
```

NoSQL Code in flask:

```
@app.route("/Rep1", methods=['GET'])
def Rep1():
       #------------Rep1 ----------------------
       collection=dbDrug['Product']
       resRp1 = []
       try:
              numberMonth = int(request.values.get("limit"))
       except Exception:
              session["Rep1Info"] = f"Limit need to be an int32"
              return render_template('credits.html',t=title,h=heading,
              results= resultFromSession())

       coursor = collection.find( { "price": { "$lt": numberMonth } } ).sort("price",
       -1).limit(5)
       for document in coursor:
              resRp1.append([document["name"], str(document["price"])])
       if len(resRp1)> 0:
              session["resRp1Done"] = True
              session["resRp1"] = resRp1
       else:
              session["Rep1Info"] = f"No Below this limit {numberMonth}"
       return render_template('credits.html',t=title,h=heading,
       results= resultFromSession())
```

Comparison:

The SQL query first filters all products that are below the given price limit. Then those products are sorted in descending order by their price. So before sorting the data, all rows are omitted which have a price above the limit. This way the query executes faster.

The NoSQL statement also filters all documents that are within the limit and them in descending order. The two database queries are very similar and follow the same idea.

Client report: Report on the clients in alphabetical order filtered by country (by Clemens Wager)

Report of clients that reside in a given country. Users can filter clients by the country they are located in.

SQL Code:

```
SELECT * FROM Client
      WHERE Country_Name = '{$country_param}'
      ORDER BY Client_Name;
```

NoSQL Code in flask:

```
@app.route("/Rep2", methods=['GET'])
def Rep2():
      #------------Rep2-----------------------
      collection=dbDrug['Client']
      resRp2 = []
      countryName = request.values.get("country")

      coursor = collection.find( { "country_name": countryName }
       ).sort("country_name", 1)
      for document in coursor:
            resRp2.append([document["name"]])

      if len(resRp2)> 0:
            session["resRp2Done"] = True
            session["resRp2"] = resRp2
      else:
            session["Rep2Info"] = f"No Clients in the country: {countryName}"
      return render_template('credits.html',t=title,h=heading,
       results= resultFromSession())
```

Comparison:

This report is realized by a fairly elegant SQL query. As countries are an attribute of the table “Client” the statement is simply defined.

The NoSQL collection of Clients was designed after the same idea and also allows for a simple query.

2.3 NoSQL Implementation (Phase 3)

2.3.1. NoSQL Database design

Client:

_id [object-id]
country_name [string]
Name [string]

Employee:

_id [object-id]
Gender [string]
Hir_Date [string]
Name [string]
Salary [int32]
Team_Leader [int32]

Order:

_id [object-id]
Client [string]
Order_Date [date]
orderID [int32]
Price [int32]
Product [string]
Quantity [int32]

Product:

_id [object-id]
Campaigns [array: (string), (date), (date)]
Indications [string]
Marketing_Emp [array. (string), (string)]
name [string]
Price [int32]

Region:

_id [object-id]
Countries [string]
General_Manager [array: (string), (string), (int32), (int32), (date)]
name [string]

2.3.2. Data migration

The data migration functionality is defined in `app.py`. A connection to both databases is established and the NoSQL Collections are deleted in the beginning. Then the structure of the first collection and its documents is defined in python. Depending on the structure different SQL tables are queried. Their results are inserted into documents to fill up the collection.

This process is repeated for each collection.

A short example for the "Employee" collection looks like this:

```
# ---------------------- Employee ------------------------
def insertEmployee(name, gender, salery, teamLeader, hirDate):
      collection = dbDrug['Employee']
      doc = { "$set":{"Name": name, "Gender": gender,
            "Salary": int(salery), "Team_Leader": teamLeader,
            "Hir_Date": creatDate(hirDate)}}
      filter = { 'Name': name }
      collection.update_one(filter, doc,  True)


cursorSQL.execute(
"""
SELECT * FROM Employee
""" )
allEmployees = cursorSQL.fetchall()
employeeCount = 0
for eachEmployee in allEmployees:
      name = eachEmployee[1] + " " +eachEmployee[2]
      gender = eachEmployee[3]
      salery = int(eachEmployee[4])
      teamLeader = eachEmployee[5]
      hirDate = eachEmployee[6]
      insertEmployee(name, gender, salery, teamLeader, hirDate)
      employeeCount += 1


print(f"[INFO] {employeeCount} new Employees inserted to DB")
```

Work protocol

Date	Duration (hours)	Task	Notes	Responsible
24.04.2022	4,00	experimenting with docker	getting a feeling for docker; first online tutorials	Clemens Wager
15.05.2022	4,00	orientation + planning	gather information on docker, mongoDB and ideas to implement DBS project	Claas Fillies
15.05.2022	4,00	orientation + planning	gather information on docker, mongoDB and ideas to implement DBS project	Clemens Wager
28.05.2022	6,00	sql db docker creation	Tried to setup oracle sql docker; no useful images online	Clemens Wager
29.05.2022	5,00	setup sql db docker	Tried to setup oracle sql docker; no useful images online; Oracle SQL part of DBS project has to be rewritten in MySQL!	Clemens Wager
29.05.2022	5,00	setup sql db docker	Tried to setup oracle sql docker; no useful images online; Oracle SQL part of DBS project has to be rewritten in MySQL!	Claas Fillies
30.05.2022	3,00	MySQL is good	rewrite to MySQL; install MySQL Workbench	Clemens Wager
30.05.2022	3,00	MySQL server db	mysql server for cloud db;	Claas Fillies
31.05.2022	4,00	use cases in MySQL		Clemens Wager
01.06.2022	3,00	reports in MySQL		Clemens Wager
01.06.2022	4,00	use cases in MySQL		Claas Fillies
02.06.2022	6,00	reports in MySQL; db filling	Tried to setup java executable to fill up SQL DB; (Q&A was too early)	Clemens Wager
03.06.2022	5,00	db filling	Tried to setup java executable to fill up SQL DB; switched to python SQLalchemy an rewrite most of the backend into php for interface	Clemens Wager

04.06.2022	5,00	db filling	omit SQLalchemy and implement filling with mysql-connector; rewrite db filling in python; python is much easier to use on such a project	Clemens Wager
08.06.2022	3,00	design of NoSQL DB; implementation of use cases	MongoDb Compass installed	Clemens Wager
08.06.2022	3,00	design of NoSQL DB; implementation of use cases	restructure project files and folders; MongoDB Atlas for cloud db; MongoDb Compass installed	Claas Fillies
09.06.2022	3,00	design of NoSQL DB; implementation of use cases and reports		Clemens Wager
09.06.2022	4,00	experimenting with php interface		Claas Fillies
26.06.2022	6,00	rework some MongoDB design; connect to frontend; flask	flask prove very useful for nice frontend-backend communication	Claas Fillies
26.06.2022	4,00	rework some MongoDB design; connect to frontend; flask		Clemens Wager
27.06.2022	5,00	docker; tutorials; flask		Claas Fillies
28.06.2022	8,00	setup docker for NoSQL part; merge of python DB filling with NoSQL docker	app.py hosts much of the functionality; creation of tables, filling, migration, mongodb queries	Claas Fillies
29.06.2022	8,00	implement data migration;		Clemens Wager
29.06.2022	8,00	NoSQL web interface; implement data migration; debug	interface format and linking in php	Claas Fillies
30.06.2022	8,00	NoSQL web interface; presentation slides; testing	buttons, colors, format in php	Clemens Wager
30.06.2022	3,00	presentation slides; testing		Claas Fillies