INFORMATION MANAGEMENT & SYSTEMS ENGINEERING

MILESTONE 2

QueensmanDB - Espionage Agency Management Platform

Implementation Report

Table of Content

Project Initialization

Our project can be installed by opening the project folder and calling the following command in a terminal, after launching the docker software in the background.

- *docker compose build --no-cache*

The –no-cache option made it possible to make changes in development being loaded into the docker web containers immediately, without needing to rebuild the whole environment again. After this we need to start all the services, by running:

- *docker-compose up*

here we developed the environment like so, that the creation of the database, the tables, and the insertion of the random generated data all takes place automatically. After all servers are running the python server will try to execute the insertion script onto the mysql container, when it is ready to build communications. This typically takes about 20 seconds on our working machines. The system should be ready when the logs of the python server print out the output data frames of the insertion program in the log files.
Now the rest of the project can be tested through the web interface by visiting

- *http://localhost:8080/*
  or,
- *https://localhost/*

respectively. Alternatively can the user connect with the mysql or mongo databases through the terminal:

- *docker exec -i mysqldb mysql -u root -p*

password =‘Schikuta<3’

- *docker exec -it mongodb mongosh*

Technology Stack

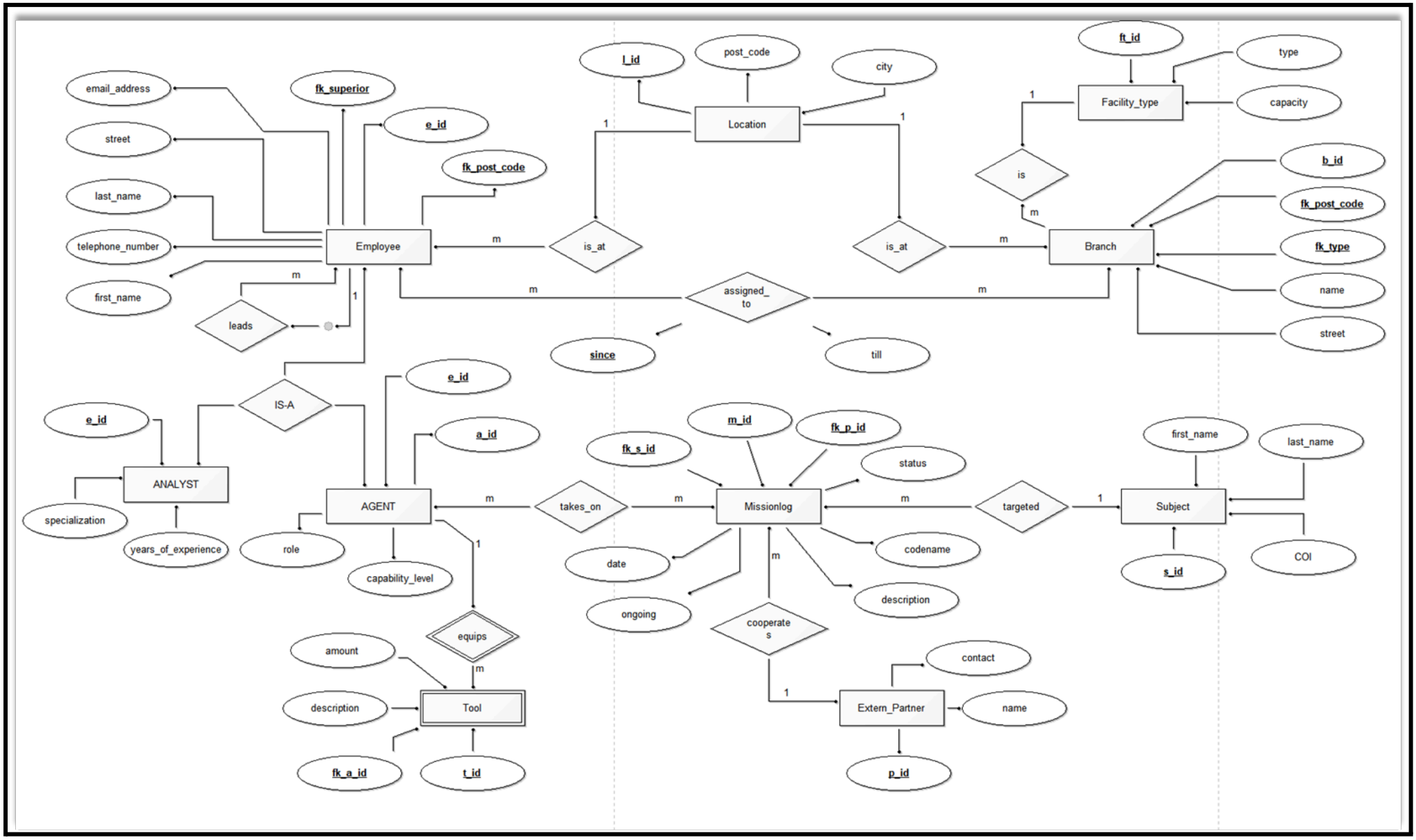
Our tech-stack consists of a Front-End, Back-End and two versions of databases, which each are ran in Containers through the Docker Software.

The Back-End main purpose is to establish the connection between database and front-end functionalities. The main programming language used is Python to create a connection with MySQL and running the create script for establishing our relational design. Furthermore, the Python Library Faker is used to generate sensical fake data for every table in our system. Additionally, the migration script is also done in Python, to successfully migrate the MySQL data to MongoDB with our defined collections.

For the Front-End we decided to keep it simple and chose PHP as our language to ensure the Front-End correctly calls and displays the data from the database. The display itself is done in HTML and CSS.

Database Design

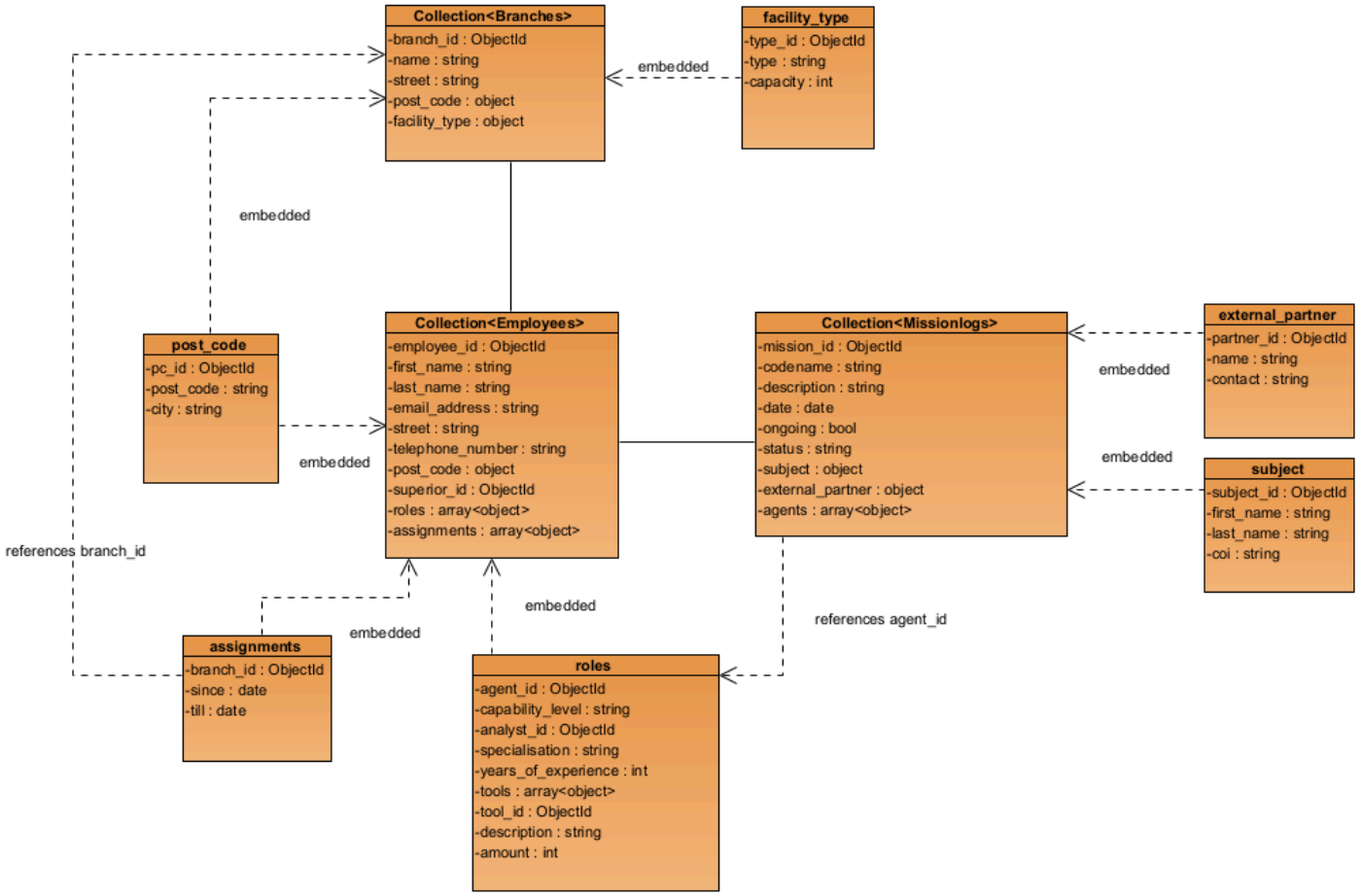
RDBMS - MySQL



Note: In comparison to our milestone 1 design, we have changed a few minor details about our logical design. Our entity *Facility_type* has the additional attribute *capacity.* The *ANALYST* entity now correctly incorporates the primary key *e_id* as child from its parent *Employee.* Most importantly, to make our data analytics use case for branch & transfer history possible, the *since* attribute in the *assigned_to* table is now a primary key.

The MySQL database consists of 12 tables including 2 tables *assigned_to* and *takes_on* that hold the relationship between Employee and its Branch (assigned_to) and Agent with its Missionlog (takes_on).

NoSQL - MongoDB



Our MongoDB design consists of three primary collections.

- Employees
- Branches
- Missionlogs

Our goal was to reduce the amount of entities (12 in RDBMS) as much as possible and denormalize all related data to the relevant collection by embedding.

The Employees collection has the general attributes while having roles, assignments and post_code embedded. This way it has all relevant info about employee types and their tools, what branches they are assigned to and their locations. Embedding all this information into one collection allows us to be efficient in our queries, as the complete profile with all its details about an employee can be acquired in a single query. For our use cases in our QueensmanDB, this helps with acquiring relevant information of employees roles and assignments without multiple joins, like in the RDBMS design.

The Branches collection has the general attributes while having post_code and facility_type embedded, with assignments being referenced through branch_id. Having all information in one collection about one branch helps optimizing the query process, requiring only one query for a branch.
The Missionlogs collection has all relevant attributes about the mission while having external partners and subjects embedded, with a reference to roles for agent_id. Embedding those details helps with requiring only one query to get all mission related information, enhancing performance. The reference to agent_id helps for efficient lookups and causes no redundancy when updating a agents details. This helps to have a fast access to what agents belongs to what mission.
Our design's priority was to simplify the complexity of entities and data into related collections, optimizing the query process by needing just one query overall and having fast access to specific information through indices.

NoSQL - Indices

- Employees

```
db.employees.create_index("employee_id")
db.employees.create_index("roles.agent_id")
db.employees.create_index("assignments.branch_id")
```

We have 3 indices for our employees collection. Employee_id is the primary key for identifying employees, which helps for many queries to fetch information about specific employee profiles, so viewing and updating employee profiles becomes faster, which is very helpful in our use cases regarding employee information. Roles.agent_id is an index that is very helpful for acquiring data about agents, which are a main entity in our database. This way lookups for agents are very fast and helpful in finding specific agents that are assigned to specific missions. Assignments.branch_id is an index for increasing our query performance about retrieving all employees linked to a branch, helping with our use case for histories.

- Branches

```
db.branches.create_index("branch_id")
db.branches.create_index("name")
```

The branches collection used branch_id and name as indices, which help with lookups and joins since those are the main identifiers for branches. Branch_id helps with performance since we can use it to find branch details and thus assigning employees to branches easier.

- Missionlogs

```
db.missionlogs.create_index("mission_id")
db.missionlogs.create_index("agents.agent_id")
db.missionlogs.create_index("date")
db.missionlogs.create_index("status")
db.missionlogs.create_index("subject.subject_id")
```

Missionlogs has 5 indices in total that generally increase query performance and are helpful in our data analytics use case. Mission_id gets a specific mission faster to get all related information about it, so it can be updated and retrieved. Agents.agent_id is an index that helps with querying much faster regarding what agents are assigned to a mission and their involvements. Date is a helpful index that can provide results faster when querying about date

ranges, like in our report where we specify in what date range we want to measure the success of our agents and generating reports about time in general. Status is another index that is helpful with our report to quickly filter missions through their status to get a real time overview of missions and for our report to find success rates amongst agents. Subject_id is the general index to improve query performance to find related subject data faster.
Our indices aim to improve query performance by indexing the main entities and improve our report performance for details like date ranges, status rates and branch histories.

Web System

PHP - Apache

Our website supports a login page where a user can log in through a username and password. Currently, we have on user that can access the webpage.
The login-credentials are:

- Username: Merlin
- Password: secret

The webpage supports two buttons that can fill the empty database through Python and a migration button that migrates currently existing data that is supported through MySql to a MongoDB. Once the fill button acts as a reset button that reinitializes the whole database by drop, create and insert scripts. The migration button can be considered as a similar feature after already being migrated once. If the user migrates from the relational setting to the non-relational MongoDB environment, the fill button gets hidden.
The websystem runs on PHP-Apache and the main functionalities are supported with a DatabaseHelper.php file.
A nginx docker container supports the web server further with the ability to also be reachable by the https protocol.

Use Case & Report Implementation

Main Use Case 1 - Assign Mission To Agent

Our first main use case is to make users of the platform able to assign a mission to an agent. The user presses a button that shows all mission files as a list. A radio button can be pressed to select a mission and the user can press a button below the list to show all available agents for this mission. The user can select one or more agents and with a last button click, the mission is assigned. To verify the assignment, the webpage is also capable of the reverse, as one can select an agent and display all his missions he is assigned to.

MySql-Statements

The following SQL statements describe the process of implementing this feature successfully. We have a query that first select all missions for our Mission-List display. We then want to make sure all already assigned agents to a selected missions will be pre-selected, so no inconsistencies appear. Lastly, we want to assign the selected agent to the mission, and insert the new entry.

- Select All Missions

```
SELECT
    m.M_ID,
    m.CODENAME,
    m.DESCRIPTION,
    m.M_DATE, m.ONGOING, m.STATUS
    FROM MISSIONLOG m
```

- Select Agents Assigned To Mission

```
SELECT a.A_ID, e.FIRST_NAME, e.LAST_NAME FROM AGENT a
JOIN TAKES_ON t ON a.A_ID = t.FK_A_ID
JOIN EMPLOYEE e ON a.E_ID = e.E_ID WHERE t.FK_M_ID = ?
```

- Assign Agents To Selected Mission

```
DELETE FROM TAKES_ON WHERE FK_M_ID = ?
INSERT INTO TAKES_ON (FK_A_ID, FK_M_ID) VALUES (?, ?)
```

The statements are simple and straightforward, however since multiple joins are used, larger datasets for our webpage would require more optimization in our mysql approach.

MongoDb-Statements

- Select All Missions

```
db.missionlogs.aggregate([
    {
        $project: {
            mission_id: "$mission_id",
            codename: "$codename",
            description: "$description",
            date: "$date",
            ongoing: "$ongoing",
            status: "$status"
        }
    }
]).toArray();
```

Select Agents Assigned To Mission

```
db.missionlogs.aggregate([
    {
        $match: { mission_id: 1 }
    },
    {
        $unwind: "$agents"
    },
    {
        $lookup: {
            from: "employees",
            localField: "agents.agent_id",
            foreignField: "employee_id",
            as: "agent_info"
        }
    },
    {
        $unwind: "$agent_info"
    },
    {
        $project: {
            agent_id: "$agents.agent_id",
            first_name: "$agent_info.first_name",
            last_name: "$agent_info.last_name"
        }
    }
]).toArray();
```

- Assign Agents To Selected Mission

```
db.missionlogs.updateOne(
    { mission_id: 1 },
    { $set: { agents: [] } }
);

const agent_ids = [ 1,2,3];

agent_ids.forEach(agent_id => {
    db.missionlogs.updateOne(
        { mission_id: 1 },
        { $push: { agents: { agent_id: agent_id, role: "Assigned" } } }
    );
});
```

The approach with MongoDb does not work in our webpage and a solution could not be found - Selection of Missions is working, so is a display of available Agents - The selection process and assign process does not work. However, here is the theoretical explanation what we thought it should work: In our selection we read the data and put it out as array. In the selection of already assigned agents, we look for the specific mission ID and unwind it in individual documents. We lookup with employees and get the agents and their information, which is again unwinded and we acquire the id, first name and last name successfully and perform. We then clear and update each agent. The queries are efficient as we use mission, agent and employee indices, which make each query faster. The amount of assigned agents is usually low, so the update operations remain efficient, as we are not expecting large datasets here.

Main Use Case 2 - Register Employee To Branch

The second main use case consisted of enabling the user to assign employees of the agency to a specific branch through the web interface. After switching to the employee page the user can select an individual employee through a “Assign to Branch”-Button. After the click, a modal window gets launched which acts as a input form for the user, after the input gets submitted the the system checks the user submission for validation and inserts the new entry and updates the previously last entry accordingly into the database. Finally the user gets notified for the (un-)successful compilation of the operation.

MySql-Statements

In this section I highlight all the SQL queries which were essential to complete the given use case. At first we needed to get all available branches to place them into a selection box. Then after the user submitted the transfer data for the employee, we first need to retrieve the most recent assigned posts entry date of said employee(as to check if the new assignment lies not behind the new one). The next Query is for updating the most recent post termination to the

new entry date if the TILL field is empty/null. And at last we insert the new assignment, depending if the user also already submitted a termination date, the query will look different.

```
SELECT B_ID, NAME FROM BRANCH;
SELECT MAX(SINCE) AS last_since FROM ASSIGNED_TO WHERE ASS_E_ID = ?;
UPDATE ASSIGNED_TO SET TILL = ? WHERE ASS_E_ID = ? AND TILL IS NULL;
INSERT INTO ASSIGNED_TO (ASS_E_ID, ASS_B_ID, SINCE, TILL) VALUES (?, ?, ?, ?);
INSERT INTO ASSIGNED_TO (ASS_E_ID, ASS_B_ID, SINCE) VALUES (?, ?, ?);
```

MongoDb-Statements

In these queries we first retrieve the branches and refactor the result so it gets compatible with the web application. Then we start a query to retrieve the last entry of an employee assignment by first searching for objects with the matching id. Then we unwind the assignments and sort them in desc order and cut everything out except the first entry with a limit - before building a resulting document. In the end we update the last entries TILL field and push a new entry onto the assignments list.

```
# projects branch info
db.branches.find(
    {},
    {
        branch_id: 1,
        name: 1
    }
).map(function(branch) {
    return {
        B_ID: branch.branch_id,
        NAME: branch.name
    };
});

# retrieve last entry
db.employees.aggregate([
    { $match: { employee_id: parseInt(e_id) } },
    { $unwind: "$assignments" },
    { $sort: { "assignments.since": -1 } },
    { $limit: 1 },
    { $project: { last_since: "$assignments.since" } }
]).toArray();

# previous entries till
db.employees.updateOne(
        {
            employee_id: parseInt(e_id),
            "assignments.since": last_since
        },
        {
            $set: { "assignments.$.till": since_date }
        }
    );

# insert new entry
db.employees.updateOne(
    { employee_id: parseInt(e_id) },
    { $push: { assignments: new_assignment } }
);
```

Secondary Use Case 4 - Promoting an Employee

That use case is about being able to promote a specified employee in our web application. After choosing a promising employee out of the employee list, the user is able to show the current superiors and its team(people that have the same superior in the agency). A dedicated button lets the user promote the chosen employee and gives the user a list of all available other employees and checkboxes. These checkboxes serve the purpose to select the subordinates of the about to be promoted employee, its previous team members are preselected. Once the input is submitted the employee discards its reference to the current superior and all selected subordinates get their superior reference updated. The now promoted employee has no superior and no team.

MySql-Statements

This section provides an overview of how the previously stated use case can be realized by using the following SQL queries. First we will need the personal data of the superior and the team member of a selected employee in the employee list. To do the former we need to self join the employee with its superior foreign key, to do the latter we get all employees that have the same superior as the selected person but not the employee itself. Once we choose to promote the employee we need to fetch all employees that are not the person which is about to be promoted, those will be potential subordinates. For preselection of the checkboxes we retrieve the former superior of the promotee and use it to get the other team members, similar to the queries before. Once the user submits the input we update the superior foreign keys of the employees, the subordinates reference their new boss and the promoted employee relinquishes the reference to its former superior.

```
SELECT s.E_ID as SUPERIOR_ID, s.FIRST_NAME as SUPERIOR_FIRST_NAME, s.LAST_NAME as SUPERIOR_LAST_NAME
FROM EMPLOYEE e
JOIN EMPLOYEE s ON e.SUPERIOR_FS = s.E_ID
WHERE e.E_ID = ?;
SELECT E_ID, FIRST_NAME, LAST_NAME
FROM EMPLOYEE
WHERE SUPERIOR_FS = (SELECT SUPERIOR_FS FROM EMPLOYEE WHERE E_ID = ?) AND E_ID NOT LIKE ?;

SELECT * FROM EMPLOYEE WHERE E_ID != ?;

SELECT SUPERIOR_FS FROM EMPLOYEE WHERE E_ID = ?:
SELECT E_ID, FIRST_NAME, LAST_NAME
FROM EMPLOYEE
WHERE SUPERIOR_FS = ?;

UPDATE EMPLOYEE SET SUPERIOR_FS = NULL WHERE SUPERIOR_FS = ?;
UPDATE EMPLOYEE SET SUPERIOR_FS = NULL WHERE E_ID = ?;
UPDATE EMPLOYEE SET SUPERIOR_FS = ? WHERE E_ID = ?;
```

MongoDb-Statements

```
# retrieve superior of employee
db.employees.findOne(
    { employee_id: parseInt(e_id) },
    { projection: { superior_id: 1 } }
);
db.employees.findOne(
        { employee_id: superior_id },
        { projection: { employee_id: 1, first_name: 1, last_name: 1 } }
    );


# retrieve team members
db.employees.find(
        {
            superior_id: employee.superior_id,
            employee_id: { $ne: parseInt(e_id) }
        },
        { projection: { employee_id: 1, first_name: 1, last_name: 1 } }
    ).toArray();


# formatting result
{
    superior: superior_info ? {
        SUPERIOR_ID: superior_info.employee_id,
        SUPERIOR_FIRST_NAME: superior_info.first_name,
        SUPERIOR_LAST_NAME: superior_info.last_name
    } : null,
    team: team_info.map(function (team_member) {
        return {
            E_ID: team_member.employee_id,
            FIRST_NAME: team_member.first_name,
            LAST_NAME: team_member.last_name
        };
    })
};
```

```
# Query to select all employees except the one with the given employee ID
db.employees.find(
    { employee_id: { $ne: parseInt(e_id) } }
).map(function(employee) {
    return {
        E_ID: employee.employee_id,
        FIRST_NAME: employee.first_name,
        LAST_NAME: employee.last_name,
        EMAIL_ADDRESS: employee.email_address,
        STREET: employee.street,
        TELEPHONE_NUMBER: employee.telephone_number,
        POST_CODE: employee.post_code,
        SUPERIOR_FS: employee.superior_id,
        roles: employee.roles,
        assignments: employee.assignments
    };

# retrieve superior_id
db.employees.findOne(
    { employee_id: parseInt(e_id) },
    { projection: { superior_id: 1 } }
).superior_id;

# team members with the same boss
db.employees.find(
    { superior_id: superior_fs },
    { projection: { employee_id: 1, first_name: 1, last_name: 1 } }
).map(function(employee) {
    return {
        E_ID: employee.employee_id,
        FIRST_NAME: employee.first_name,
        LAST_NAME: employee.last_name
    };
});

# reset all team members of the current employee
db.employees.updateMany(
    { superior_id: e_id },
    { $set: { superior_id: null } }
);

db.employees.updateOne(
    { employee_id: e_id },
    { $set: { superior_id: null } }
);
```

```
# update the selected team members
db.employees.updateMany(
        { employee_id: { $in: team_member_ids } },
        { $set: { superior_id: e_id } }
    );
```

To make the 7 relational queries work in our new non-relational environment of mongodb work we needed to write the queries differently to before. with the above pseudo-esque NoSQL code I want to illustrate how we came to our solution.

In the first query we retrieve the superior from the given employee and later extract its details. In the later part we retrieve all team members excluding the employee. In the end, to make it compatible with the front end we construct an object with the newly found data with the matching attribute names expected by the web interface.

The second query and bigger one retrieves all employees except the chosen employee, like with the first query we map the result to match it with the latter processing.

The 3rd query now retrieves the superior reference from an employee again, get the team member of a specific employee and maps the result again.

Finally with the 4th query we now update the employee and all its former team members accordingly.

Reports

Report 1 - Find Most Successful Agents Based on Unique Subjects

Our QueensmanDB keeps track of Missions and Agents. Every agent has missions assigned to him and the job of an agent is to fulfill the mission's objective. A subject is always the main focus in a mission and depending on the outcome, a mission will be deemed SUCCESSFUL or FAILED under its status value. Ongoing missions remain PENDING. With our report, we want to help making a educated choice which agent is most suitable for the next mission and we base this on his total amount of successful missions and specifically on unique subjects, as a agent that has various successful missions is even more probable to be best suited for the case. In the following, we discuss our MySql and MongoDb statements and how they compare to achieve this report.

MySql-Statement

```sql
SELECT
    a.A_ID,
    e.LAST_NAME,
    COUNT(m.M_ID) AS successful_missions,
    COUNT(DISTINCT m.FK_S_ID) AS successful_missions_unique_subjects
FROM
    AGENT a
JOIN
    EMPLOYEE e ON a.E_ID = e.E_ID
LEFT JOIN
    TAKES_ON t ON a.A_ID = t.FK_A_ID
LEFT JOIN
    MISSIONLOG m ON t.FK_M_ID = m.M_ID AND m.STATUS = 'SUCCESSFUL' AND m.M_DATE >= DATE_SUB(CURDATE(), INTERVAL 1 YEAR)
GROUP BY
    a.A_ID, e.LAST_NAME
ORDER BY
    successful_missions_unique_subjects DESC, successful_missions DESC
```

This statement requires several joins to ensure consistency and accuracy of results. We select the ID and last name of an agent, as well as the two counts required for the report. We join agents with employees to get the last name, join then to acquire the missions they are assigned to and join with Missionlog to filter based on success and date range. Since we want to find the most successful agents in the past year, the range is 1 YEAR.

MongoDb-Statement

```
db.missionlogs.aggregate([
    {
        $match: {
            status: 'SUCCESSFUL',
            date: {
                $gte: new ISODate(new Date(Date.now() - 365 * 24 * 60 * 60 * 1000).toISOString())
            }
        }
    },
    {
        $unwind: '$agents'
    },
    {
        $group: {
            _id: {
                agent_id: '$agents.agent_id'
            },
            total_successful_missions: { $sum: 1 },
            unique_subjects: { $addToSet: '$subject.subject_id' }
        }
    },
    {
        $lookup: {
            from: 'employees',
            localField: '_id.agent_id',
            foreignField: 'roles.agent_id',
            as: 'agent_info'
        }
    },
    {
        $unwind: '$agent_info'
    },
    {
        $group: {
            _id: {
                agent_id: '$_id.agent_id',
                last_name: '$agent_info.last_name'
            },
            successful_missions: { $first: '$total_successful_missions' },
            successful_missions_unique_subjects: { $first: { $size: '$unique_subjects' } }
        }
    },
    {
        $project: {
            agent_id: '$_id.agent_id',
            last_name: '$_id.last_name',
            successful_missions: 1,
            successful_missions_unique_subjects: 1
        }
    },
    {
        $sort: {
            successful_missions_unique_subjects: -1,
            successful_missions: -1
        }
    }
]).toArray()
```

We walk through the missionlog collection and filter documents that only include successful missions over the past year. We unwind the agents to get the individual documents. Continued by grouping the mission by agent ID and count the total amount of successful missions and the amount of unique subjects of the agent. A lookup is done to join with the

employees collection to get the specified agent information. We unwind again for individual documents and group by agent ID. This way, we acquire the last name of the agent and take the first entry of successful missions and unique subjects count. We finalize and sort the results by successful missions amount and the same, but based on unique subjects.

Statements-Comparison

MySQL requires a more complex join behavior that is expensive on the performance when the dataset grows large enough. We improve it by proper indexing through IDs and foreign key structure, but the process and the aggregation remains complex enough to cause performance impacts on larger datasets.

On the contrary, the MongoDb statements pass the documents as required through the pipeline by walking through the collections. Due to the functionalities of MongoDb, the operations such as unwind can be handled more effectively and show a performance advantage over our Sql-Statement. Our indices agent_id and date are used and optimize the entire process even further.

Report 2 - List of History Of Specific Employees Past Assigned Branches And Possible Transfers

For the secret service agency it is paramount that it knows where its employees are stationed at any given moment, whereas be an agent in the fields or an analyst in the computer labs. The life in the agency is dynamic and fast paced, therefore employees are encouraged to transfer to different branches within the agency. Branches can be of different type, some are official headquarters others are secret hideouts. It is also sometimes the case that employees get transferred back into past stationed branches. With this Report we want to achieve that the agency gets detailed information about their employees. The resulting output of an employee's assignment history consists of the branch, its type, where it is located and from when, and if specified till when the employee is stationed there.

MySql-Statement

To achieve the above described information we will need to Join across 4 entities. We need data from the assigned_to table which links the employees to their stations and then fetch the more illustrative information out of the other tables.

```
SELECT ASS_ID, NAME, ft.`TYPE`, CITY, SINCE, TILL
FROM ASSIGNED_TO
LEFT JOIN BRANCH ON ASS_B_ID = B_ID
LEFT JOIN FACILITY_TYPE ft ON FK_TYPE=ft.FT_ID
LEFT JOIN POST_CODE pc ON FK_POST_CODE  = PC_ID
WHERE ASS_E_ID = ?
```

MongoDb-Statement

In this query we first try to match every document that has the right index attribute in it, after that we unwind the embedded assignments list to inspect it further. Once that is done we now join the objects with the lookup-operation by binding objects through the assignments.branch_id and branch_id/branch_info. After unwinding the branch_info we now build the new document and format the date values “SINCE” and ”TILL” into an acceptable format for the website.

```
db.employees.aggregate([
    {
        $match: {
            employee_id: parseInt(e_id)
        }
    },
    {
        $unwind: "$assignments"
    },
    {
        $lookup: {
            from: "branches",
            localField: "assignments.branch_id",
            foreignField: "branch_id",
            as: "branch_info"
        }
    },
    {
        $unwind: "$branch_info"
    },
    {
        $project: {
            ASS_ID: "$assignments.branch_id",
            NAME: "$branch_info.name",
            TYPE: "$branch_info.facility_type.type",
            CITY: "$branch_info.post_code.city",
            SINCE: {
                $dateToString: {
                    format: "%Y-%m-%d",
                    date: "$assignments.since"
                }
            },
            TILL: {
                $cond: {
                    if: { $eq: ["$assignments.till", null] },
                    then: null,
                    else: {
                        $dateToString: {
                            format: "%Y-%m-%d",
                            date: "$assignments.till"
                        }
                    }
                }
            }
        }
    }
]).pretty();
```

Statements-Comparison

While both queries realize the same result they operate quite differently in their approaches to achieve these. While the relational SQL script focuses on joining associated tables together, the non-relational NoSQL query works with objects and relies on aggregation pipelines to transform and mold the query into a satisfying output.
While join operations can be considered as optimized in relational database systems, they are still considered computationally expensive if the data domain is of certain size. The same can be said about the aggregation pipeline operations for NoSQL queries, but by providing the non-relational database with indexes to improve performance we think we made the best out of it, because the MongoDB search doesn’t have to search through the whole document, reducing I/O operations.