M2: Bike Insurance Provider

Konstantin Haas (01451754) & Michael Raffelsberger (11772903)

June 2021

Contents

1 Implementation Overview

We implemented IS for a small bike insurance customer. Bike insurance customers can request policies with different options, accept and decline offers as well as report claims for existing policies. Bike insurance Agents can access a fee report and a claim report in order to monitor the business.

1.1 Front-End

We did not implement a javascript frontend, but decided to use *jinja* bootstrap templates, as they are sufficient for the user interface we want to create and come at easy compatibility with our python back-end. We define a base template and every route extend this base template with custom content.

1.2 Back-End

We implemented the back-end as a REST API using the *Flask* framework, which builds on python and was a natural choice with both team members having background in python programming. Users are authenticated with the *Flask Login* Manager.

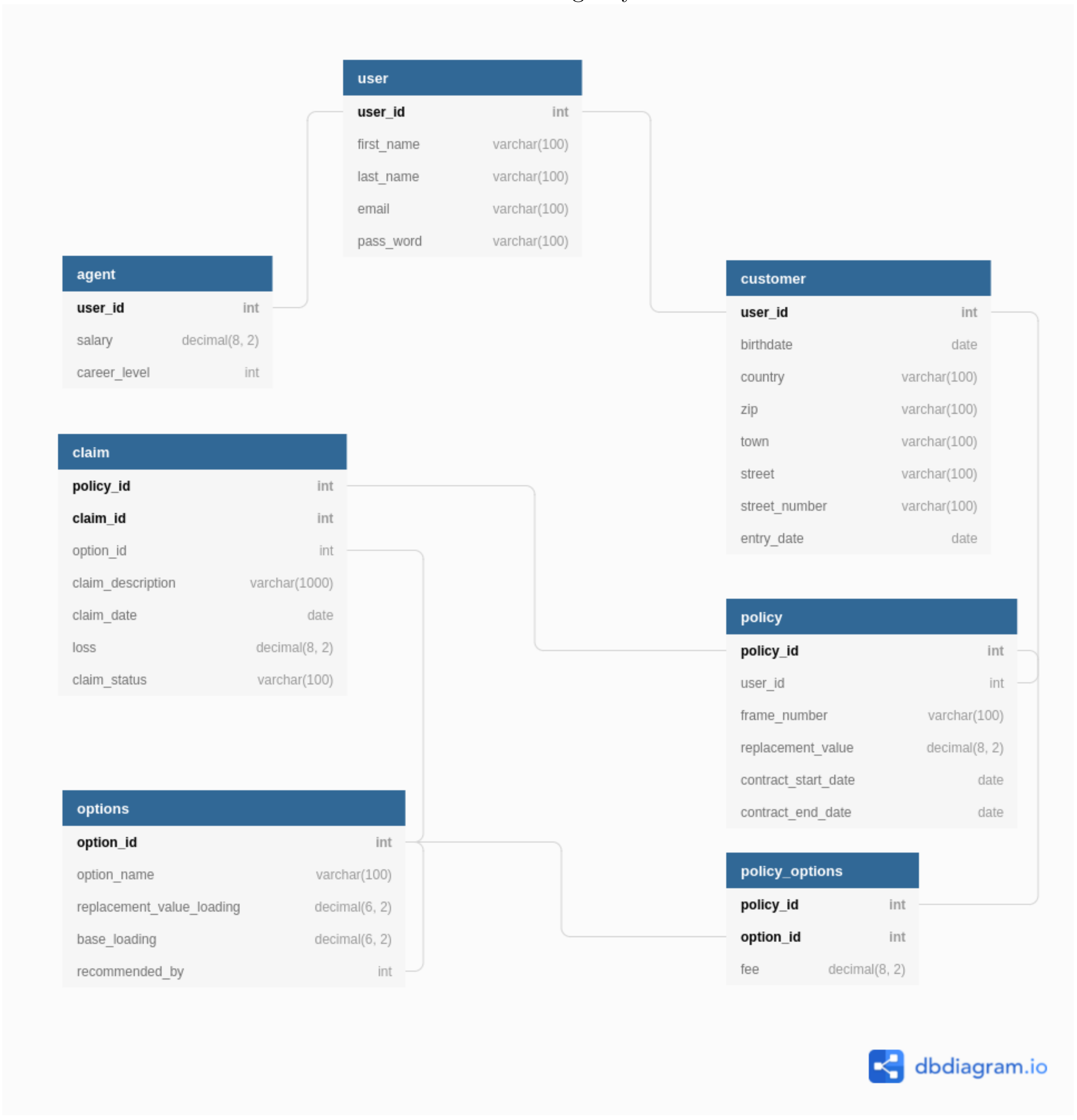
1.3 Database

We use the python packages *mysql.conncector* and *pymongo* for the MySQL and Mongo DB, respectively. Before the first login the MySQL database must be initialized via a buttom.

2 RDBMS Design

2.1 Visualization

We can visualize the RDBMS schema in the following way[1]:



user	
user_id	int
first_name	varchar(100)
last_name	varchar(100)
email	varchar(100)
pass_word	varchar(100)

agent	
user_id	int
salary	decimal(8, 2)
career_level	int

customer	
user_id	int
birthdate	date
country	varchar(100)
zip	varchar(100)
town	varchar(100)
street	varchar(100)
street_number	varchar(100)
entry_date	date

claim	
policy_id	int
claim_id	int
option_id	int
claim_description	varchar(1000)
claim_date	date
loss	decimal(8, 2)
claim_status	varchar(100)

policy	
policy_id	int
user_id	int
frame_number	varchar(100)
replacement_value	decimal(8, 2)
contract_start_date	date
contract_end_date	date

options	
option_id	int
option_name	varchar(100)
replacement_value_loading	decimal(6, 2)
base_loading	decimal(6, 2)
recommended_by	int

policy_options	
policy_id	int
option_id	int
fee	decimal(8, 2)

[1] `https://dbdiagram.io/`

2.2 Create Table Statements

The MySQL database was created with the following statements:

```
CREATE TABLE IF NOT EXISTS user (
user_id int unsigned NOT NULL AUTO_INCREMENT,
first_name varchar(100) NOT NULL,
last_name varchar(100) NOT NULL,
email varchar(100) NOT NULL UNIQUE,
pass_word varchar(100) NOT NULL,
PRIMARY KEY (user_id)
);

CREATE TABLE IF NOT EXISTS agent (
user_id int unsigned NOT NULL,
salary decimal(8, 2) unsigned NOT NULL,
career_level int unsigned NOT NULL,
PRIMARY KEY (user_id),
FOREIGN KEY (user_id) REFERENCES user(user_id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS customer (
user_id int unsigned NOT NULL,
birthdate date NOT NULL,
country varchar(100) NOT NULL,
zip varchar(100) NOT NULL,
town varchar(100) NOT NULL,
street varchar(100) NOT NULL,
street_number varchar(100) NOT NULL,
entry_date date NOT NULL,
PRIMARY KEY (user_id),
FOREIGN KEY (user_id) REFERENCES user(user_id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS policy (
policy_id int unsigned NOT NULL AUTO_INCREMENT,
user_id int unsigned,
frame_number varchar(100) NOT NULL,
replacement_value decimal(8, 2) unsigned NOT NULL,
contract_start_date date NOT NULL,
contract_end_date date DEFAULT NULL,
CHECK (contract_end_date IS NULL OR contract_end_data ¿ contract_start_date),
PRIMARY KEY (policy_id),
FOREIGN KEY (user_id) REFERENCES customer(user_id) ON DELETE SET NULL
);

CREATE TABLE IF NOT EXISTS options (
option_id int unsigned NOT NULL AUTO_INCREMENT,
option_name varchar(100) NOT NULL,
replacement_value_loading decimal(6, 2) unsigned NOT NULL,
base_loading decimal(6, 2) unsigned NOT NULL,
recommended_by int unsigned,
PRIMARY KEY (option_id),
FOREIGN KEY (recommended_by) REFERENCES options(option_id)
);

CREATE TABLE IF NOT EXISTS policy_options (
policy_id int unsigned NOT NULL,
option_id int unsigned NOT NULL,
fee decimal(8, 2) unsigned NOT NULL,
PRIMARY KEY (policy_id, option_id),
FOREIGN KEY (policy_id) REFERENCES policy(policy_id),
FOREIGN KEY (option_id) REFERENCES options(option_id)
);

CREATE TABLE IF NOT EXISTS claim (
policy_id int unsigned NOT NULL,
claim_id int unsigned NOT NULL,
option_id int unsigned NOT NULL DEFAULT '6',
claim_description varchar(1000),
claim_date date NOT NULL,
loss decimal(8, 2) NOT NULL,
claim_status varchar(100) NOT NULL DEFAULT 'reported',
PRIMARY KEY (policy_id, claim_id),
FOREIGN KEY (policy_id) REFERENCES policy(policy_id) ON DELETE CASCADE,
FOREIGN KEY (option_id) REFERENCES options(option_id)
)
```

2.3 Remarks

- The id of user, policy and option are auto-incremented.
- The email of the user would also be a key candidate. It has to be unique and must not be NULL.
- An agent/customer is deleted in a cascading way whenever the corresponding entry in the user table is deleted. This is basically also the case in NoSQL where the information of the three tables is stored in just one collection.
- The contract end date of a policy has the default value NULL. With this choice we do not have to care about setting a contract end date in our create policy use-case.
- There is also a input control to prevent an end date from being lower than the start date.
- When a customer is deleted, the customer id in the corresponding policies are set to NULL. This way, we do not lose the business information, but policies are just kind of anonymized.

This also corresponds to how it is handeled in MongoDB where we decided to have a separate policy collection (partially for this reason!).

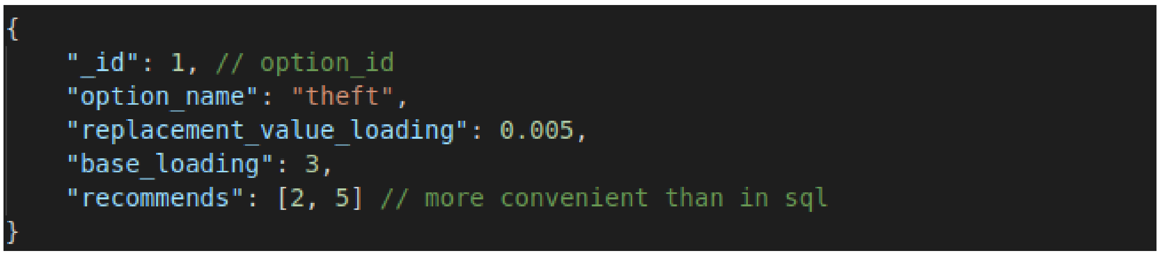
- In the claim table, the default option id is 6. This is the helper option ”undefined”. With this trick we do not have to ask the customer to choose the correct option of his or her policy that is affected by the reported claim. In practice, the insurance agent dealing with the case would then decide what is the correct option if the claim is justified at all.
- A claim as a weak entity of the policy is also deleted in a cascading way.

3 NoSQL Design

First of all, we used MongoDB for our project as recommended by the lecturers. We decided to use three collections: option, user and policy.

3.1 Example Documents

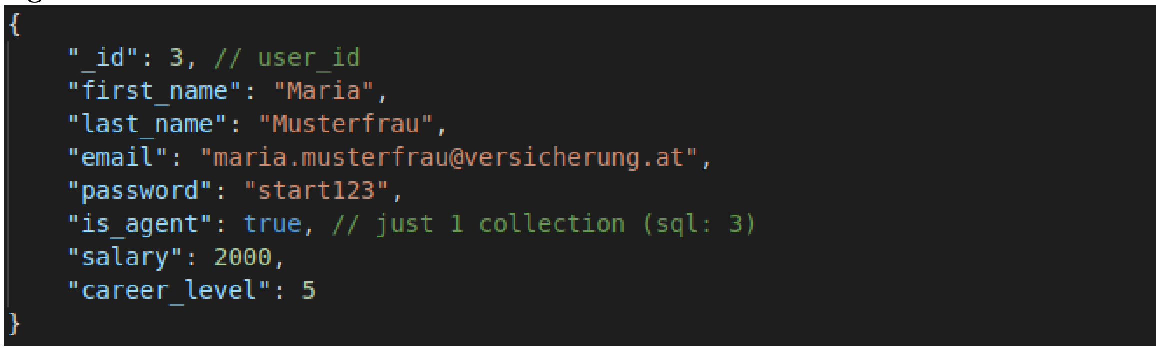
3.1.1 Option



```
{
    "_id": 1, // option_id
    "option_name": "theft",
    "replacement_value_loading": 0.005,
    "base_loading": 3,
    "recommends": [2, 5] // more convenient than in sql
}
```

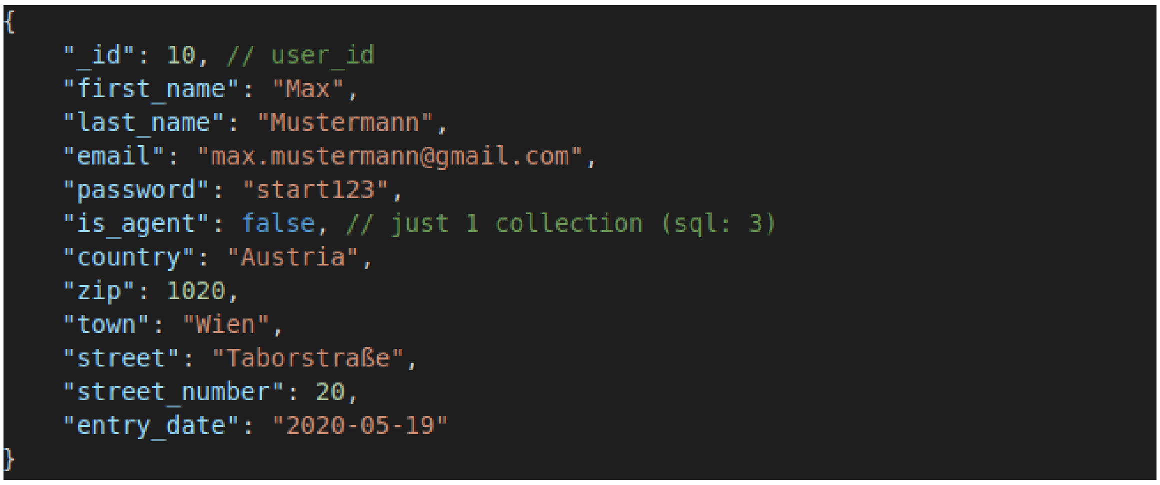
3.1.2 User

Agent



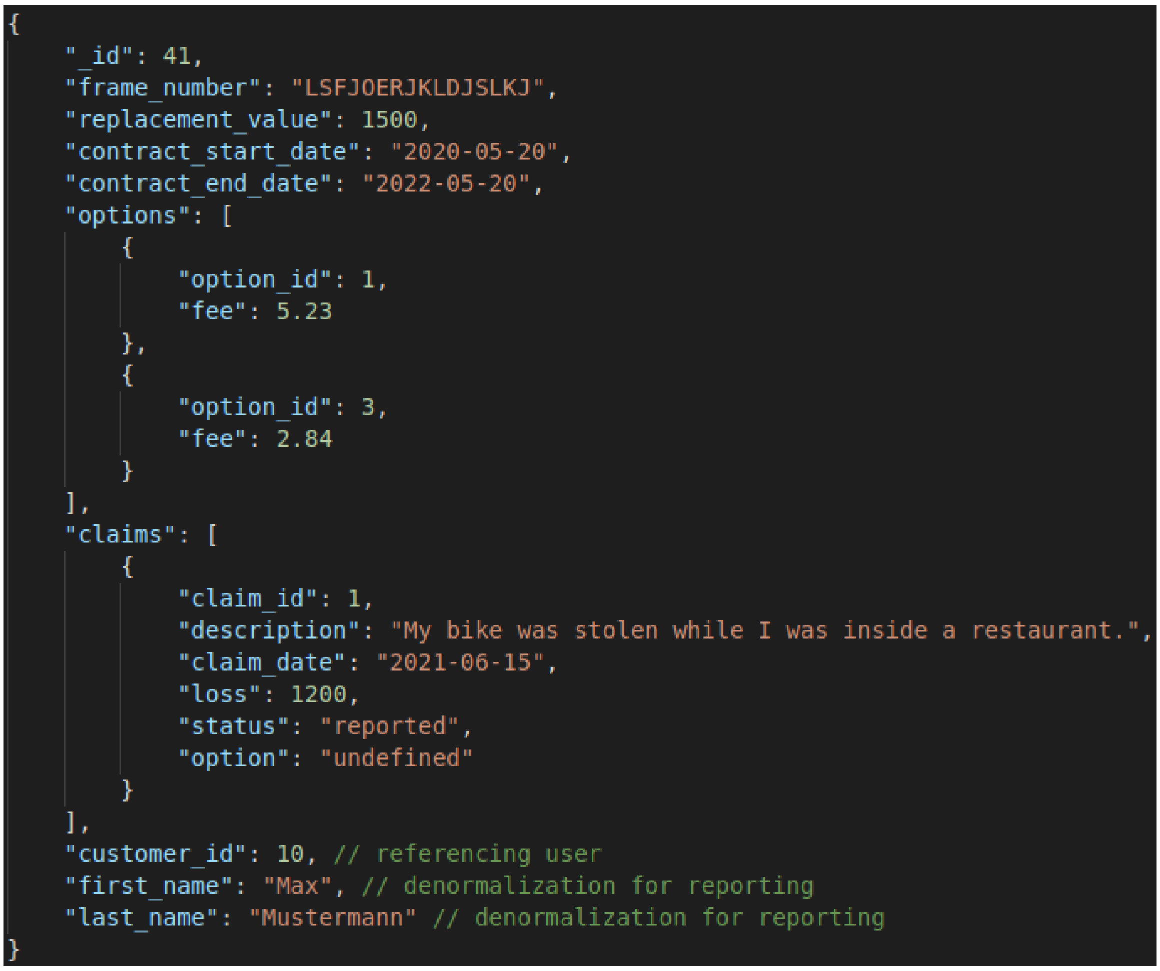
```
{
    "_id": 3, // user_id
    "first_name": "Maria",
    "last_name": "Musterfrau",
    "email": "maria.musterfrau@versicherung.at",
    "password": "start123",
    "is_agent": true, // just 1 collection (sql: 3)
    "salary": 2000,
    "career_level": 5
}
```

Customer



```
{
    "_id": 10, // user_id
    "first_name": "Max",
    "last_name": "Mustermann",
    "email": "max.mustermann@gmail.com",
    "password": "start123",
    "is_agent": false, // just 1 collection (sql: 3)
    "country": "Austria",
    "zip": 1020,
    "town": "Wien",
    "street": "Taborstraße",
    "street_number": 20,
    "entry_date": "2020-05-19"
}
```

3.1.3 Policy



```
{
    "_id": 41,
    "frame_number": "LSFJOERJKLDJSLKJ",
    "replacement_value": 1500,
    "contract_start_date": "2020-05-20",
    "contract_end_date": "2022-05-20",
    "options": [
        {
            "option_id": 1,
            "fee": 5.23
        },
        {
            "option_id": 3,
            "fee": 2.84
        }
    ],
    "claims": [
        {
            "claim_id": 1,
            "description": "My bike was stolen while I was inside a restaurant.",
            "claim_date": "2021-06-15",
            "loss": 1200,
            "status": "reported",
            "option": "undefined"
        }
    ],
    "customer_id": 10, // referencing user
    "first_name": "Max", // denormalization for reporting
    "last_name": "Mustermann" // denormalization for reporting
}
```

3.2 Design Motivation

In the following, you will find out about the reasons for this design choice:

- We decided to make a separate option collection since there was no reasonable alternative. Storing the option information in every policy, would lead to an absurd redundancy that would really hurt whenever we want to update our option portfolio (adjust prices, introduce new products, etc.) and might also undermine consistency if we make a mistake. This way we could also lose information if one option is not used by any customer at all. As the option collection is likely a very small one anyway, a join on policy also will not hurt too much.
- We decided to make a single user collection out of user, customer and agent in SQL. This is possible due to the schemaless character of MongoDB. It makes sense and is convenient since the id system is the same for both groups anyway and allows for different fields in different documents in the same collection. We can now find out about whether a user is an agent with the explicit is_agent attribute.
- It is important not to nest the policy collection into the users since we want to retain the (anonymized) policy information (on delete set null in sql) even when a user decides to have his or her personal information deleted.
- We use referencing on the policy side for the user-policy relationship and denormalize the first name as well as the last name of the user. Fortunately, a name does not change too often (few updates) and this way we save joins for both reports where we need the customer names and also get a cheap updates for the use-cases (new policy and report claim).
- We do not have "infinitely" growing arrays anywhere. A scenario where this could have happened in our application is if we would have stored the agent-claim relationship in a way that an agent has an array with all claims that he or she has been responsible for. Over time, this array would expand and hold mostly unimportant information (claims already processed).

3.3 Indexing

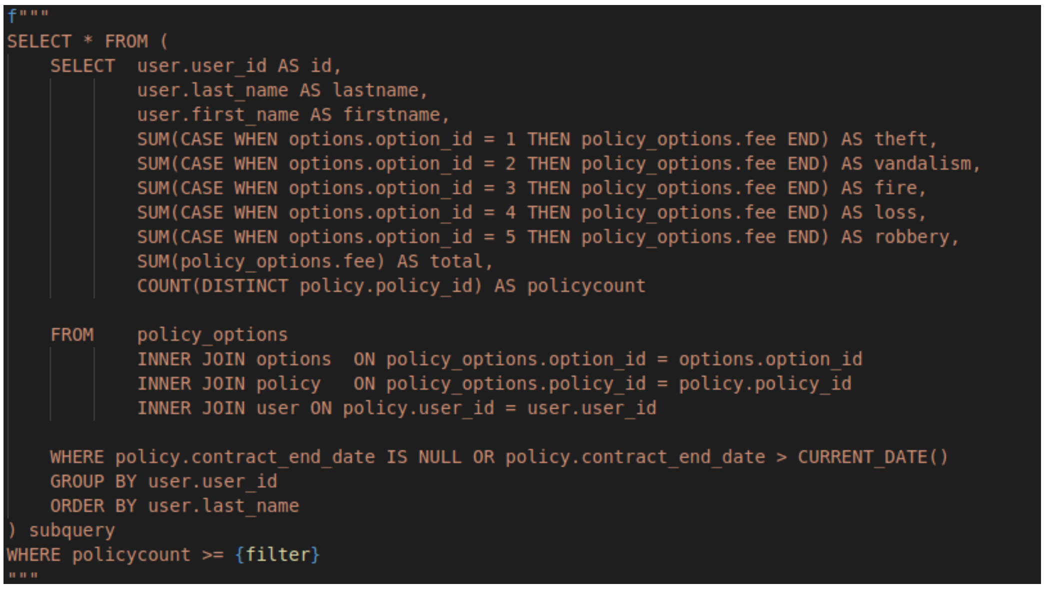
To speed up our implementation, we can make use of the following index concept:

- For all three collections, we use the default unique MongoDB attribute "_id" (which would otherwise be created automatically anyway) instead of an additional option_id, user_id or policy_id. Since we have to query (e.g. by user_id for the login logic of the app) and join (e.g. by option_id for the feereport) by them sometimes, that (hopefully) makes sense.
- **Policy:** We (would) add an ascending index by the last_name because we have to sort the reports by it. We could also use a compound index here including the first_name if we want to sort by both last name and first name. On top of that, we can use an index on the customer_id. The reason is that for the claim-use-case we have to find the policies of a specific user.
- **User:** We can further use an index on the email attribute here, since we have to find users by their email address quickly for the login logic in Flask.

4 Query Comparison

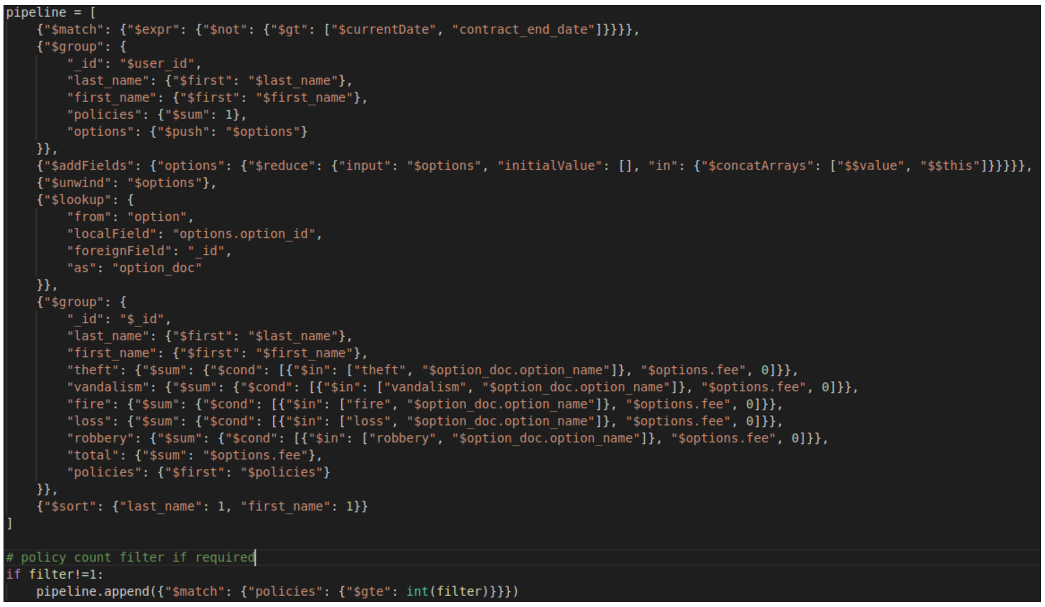
4.1 Feereport

4.1.1 SQL



```
f"""
SELECT * FROM (
    SELECT  user.user_id AS id,
            user.last_name AS lastname,
            user.first_name AS firstname,
            SUM(CASE WHEN options.option_id = 1 THEN policy_options.fee END) AS theft,
            SUM(CASE WHEN options.option_id = 2 THEN policy_options.fee END) AS vandalism,
            SUM(CASE WHEN options.option_id = 3 THEN policy_options.fee END) AS fire,
            SUM(CASE WHEN options.option_id = 4 THEN policy_options.fee END) AS loss,
            SUM(CASE WHEN options.option_id = 5 THEN policy_options.fee END) AS robbery,
            SUM(policy_options.fee) AS total,
            COUNT(DISTINCT policy.policy_id) AS policycount

    FROM    policy_options
            INNER JOIN options  ON policy_options.option_id = options.option_id
            INNER JOIN policy   ON policy_options.policy_id = policy.policy_id
            INNER JOIN user ON policy.user_id = user.user_id

    WHERE policy.contract_end_date IS NULL OR policy.contract_end_date > CURRENT_DATE()
    GROUP BY user.user_id
    ORDER BY user.last_name
) subquery
WHERE policycount >= {filter}
"""
```

4.1.2 MongoDB



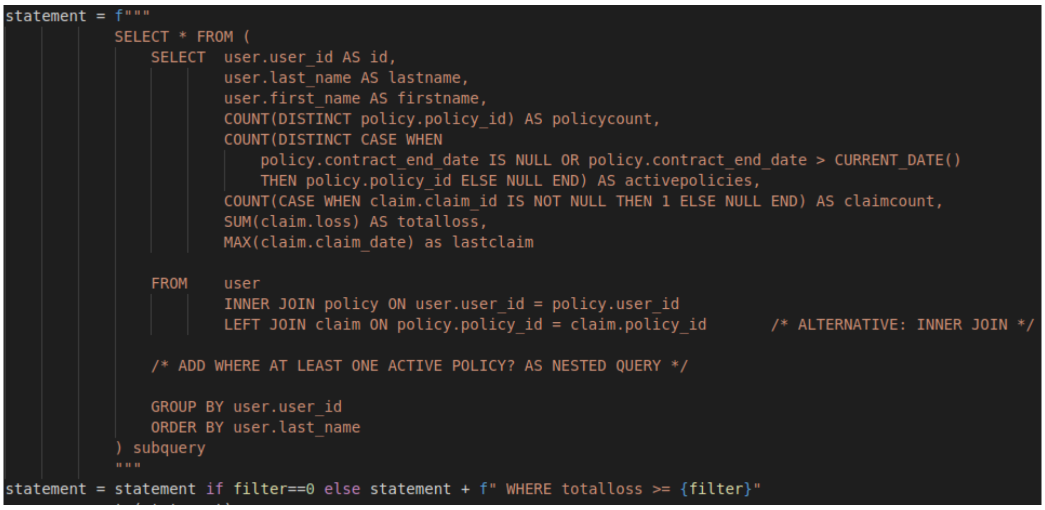
```
pipeline = [
    {"$match": {"$expr": {"$not": {"$gt": ["$currentDate", "contract_end_date"]}}}},
    {"$group": {
        "_id": "$user_id",
        "last_name": {"$first": "$last_name"},
        "first_name": {"$first": "$first_name"},
        "policies": {"$sum": 1},
        "options": {"$push": "$options"}
    }},
    {"$addFields": {"options": {"$reduce": {"input": "$options", "initialValue": [], "in": {"$concatArrays": ["$$value", "$$this"]}}}}},
    {"$unwind": "$options"},
    {"$lookup": {
        "from": "option",
        "localField": "options.option_id",
        "foreignField": "_id",
        "as": "option_doc"
    }},
    {"$group": {
        "_id": "$_id",
        "last_name": {"$first": "$last_name"},
        "first_name": {"$first": "$first_name"},
        "theft": {"$sum": {"$cond": [{"$in": ["theft", "$option_doc.option_name"]}, "$options.fee", 0]}},
        "vandalism": {"$sum": {"$cond": [{"$in": ["vandalism", "$option_doc.option_name"]}, "$options.fee", 0]}},
        "fire": {"$sum": {"$cond": [{"$in": ["fire", "$option_doc.option_name"]}, "$options.fee", 0]}},
        "loss": {"$sum": {"$cond": [{"$in": ["loss", "$option_doc.option_name"]}, "$options.fee", 0]}},
        "robbery": {"$sum": {"$cond": [{"$in": ["robbery", "$option_doc.option_name"]}, "$options.fee", 0]}},
        "total": {"$sum": "$options.fee"},
        "policies": {"$first": "$policies"}
    }},
    {"$sort": {"last_name": 1, "first_name": 1}}
]

# policy count filter if required
if filter!=1:
    pipeline.append({"$match": {"policies": {"$gte": int(filter)}}})
```

4.1.3 Comparison

For the feereport, we have to join four tables in SQL: options (for the option names), policy_options (for the fees), policy (for the contract end date) and user (for the first and last name). In MongoDB, we just have to lookup the the option names from the option collection. Apart from that, all necessary is already present in the policy collection thanks to nesting and denormalization. Unfortunately, the query was quite hard and is likely a little unorthodoxly implemented. In particular, we hard-coded the option names. Ideally, one would implement it in a more generic way.

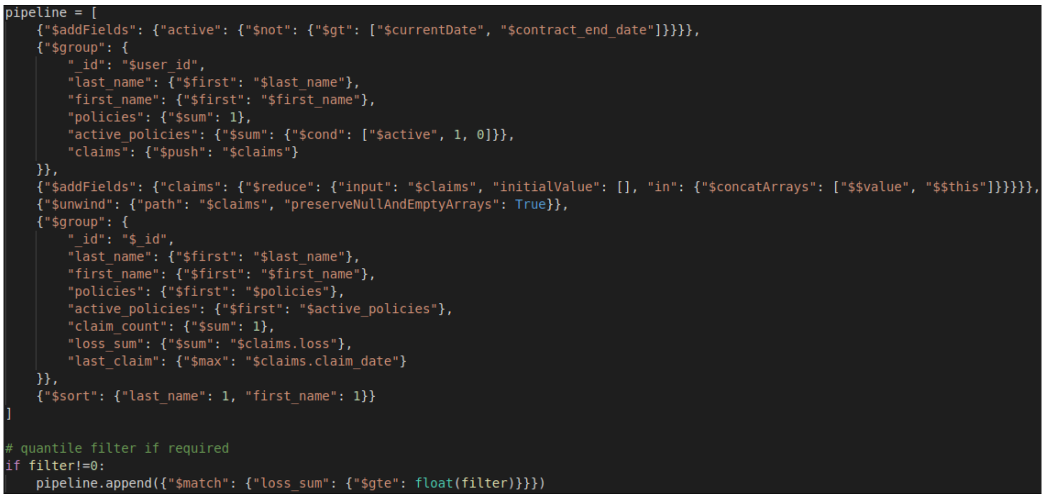
4.2 Claimreport

4.2.1 SQL



```
statement = f"""
            SELECT * FROM (
                SELECT  user.user_id AS id,
                        user.last_name AS lastname,
                        user.first_name AS firstname,
                        COUNT(DISTINCT policy.policy_id) AS policycount,
                        COUNT(DISTINCT CASE WHEN
                            policy.contract_end_date IS NULL OR policy.contract_end_date > CURRENT_DATE()
                            THEN policy.policy_id ELSE NULL END) AS activepolicies,
                        COUNT(CASE WHEN claim.claim_id IS NOT NULL THEN 1 ELSE NULL END) AS claimcount,
                        SUM(claim.loss) AS totalloss,
                        MAX(claim.claim_date) as lastclaim

                FROM    user
                        INNER JOIN policy ON user.user_id = policy.user_id
                        LEFT JOIN claim ON policy.policy_id = claim.policy_id       /* ALTERNATIVE: INNER JOIN */

                /* ADD WHERE AT LEAST ONE ACTIVE POLICY? AS NESTED QUERY */

                GROUP BY user.user_id
                ORDER BY user.last_name
            ) subquery
            """
statement = statement if filter==0 else statement + f" WHERE totalloss >= {filter}"
```

4.2.2 MongoDB



```
pipeline = [
    {"$addFields": {"active": {"$not": {"$gt": ["$currentDate", "$contract_end_date"]}}}},
    {"$group": {
        "_id": "$user_id",
        "last_name": {"$first": "$last_name"},
        "first_name": {"$first": "$first_name"},
        "policies": {"$sum": 1},
        "active_policies": {"$sum": {"$cond": ["$active", 1, 0]}},
        "claims": {"$push": "$claims"}
    }},
    {"$addFields": {"claims": {"$reduce": {"input": "$claims", "initialValue": [], "in": {"$concatArrays": ["$$value", "$$this"]}}}}},
    {"$unwind": {"path": "$claims", "preserveNullAndEmptyArrays": True}},
    {"$group": {
        "_id": "$_id",
        "last_name": {"$first": "$last_name"},
        "first_name": {"$first": "$first_name"},
        "policies": {"$first": "$policies"},
        "active_policies": {"$first": "$active_policies"},
        "claim_count": {"$sum": 1},
        "loss_sum": {"$sum": "$claims.loss"},
        "last_claim": {"$max": "$claims.claim_date"}
    }},
    {"$sort": {"last_name": 1, "first_name": 1}}
]

# quantile filter if required
if filter!=0:
    pipeline.append({"$match": {"loss_sum": {"$gte": float(filter)}}})
```

4.2.3 Comparison

For the claimreport, we have to join three tables in SQL: user (for the names), policy (connects user and claim and for contract end date) and claim (obvious). A hard task here was to have both the number of total policies and the number of active policies for the user. This was also not so easy in MongoDB. Let us highlight, that MongoDB does not require a single join for this report because most information is naturally included thanks to nesting the claims and the customer names are contained due to denormalization.

5 References

- `https://www.youtube.com/`
- `https://hub.docker.com/`
- `https://www.w3schools.com/`
- `https://stackoverflow.com/`
- `https://github.com/CoreyMSchafer`
- `https://getbootstrap.com/`
- `https://docs.mongodb.com/manual/`
- `https://pymongo.readthedocs.io/en/stable/`
- and many more ...
- for details see: `Protocoll` file in `documentation` folder

6 Work Protocoll

30.04	MR	get familiar with docker	8h
03.05	MR	get familiar with docker/flask	5h
04.05	MR	sql scheme	5h
05.05	MR	datagenerator and check simple queries + boilerplate	5h
10.05	MR	get familiar with flask	5h
11.05	MR	create table and first inserts	5h
12.05	MR	mysql fill button	5h
13.05	MR	login/session logic and report query	5h
21.05	KH/MR	team merge	0h
01.06	KH	html templates and header bar and https	8h
03.06	MR/KH	update routes and sql report queries and report2 table and buttons	8h
04.06	KH	user account functions and route	8h
05.06	KH/MR	use-case 1 and 2 and user-account site	8h
06.06	MR	finish reporting and add undefined option	6h
09.06	KH	finish use-cases (write)	8h
12.06	MR	set up mongodb and data migration	8h
14.06	KH/MR	mongo use-cases	8h
15.06	MR/KH	mongo reporting	8h
17.06	KH	finish code	8h