RAWDATA Portfolio Subproject 1

Picture: [QAPicture], **Source**: pexels.com, **License**: Creative Commons Public Domain

Authors: Anton Kjær Hansen, Frederik Zeilberger Thulstrup, Lukas Kucerik, Daniel Șerbănescu
Class: RAWDATA Autumn 2018
Institution: Roskilde University
Supervisors: Troels Andreasen, Henrik Bulskov Styltsvig
Project span: 20.09.2018 - 03.10.2018

Table of Contents

Introduction

This report revolves around the first of the four-part portfolio project presented during the Responsive Applications, Web services and Databases (RAWDATA) course at Roskilde University (RUC). It aims at describing the process and decisions that we have gone through as well as the requirements we have decided to implement in this course. This report also includes a short preliminary list of requirements which we identified for implementation in the coming course sections. The final product is accessible on the “rawdata.ruc.dk” database server in the database “raw10”.

Application design and adapted requirement list

Vision

Since the initials of the SOVA application are describing the concept of the final product in an elegant but rather vague manner, a slightly deeper look into the application’s vision will be provided in the following paragraphs.

SOVA stands for Stack Overflow Viewer Application and its main focus revolves around an easy-to-use web application for searching in Stack Overflow’s vast database of questions and answers. It is not meant to be developed as a part of the already well-established Stack Overflow website, nor as a plugin providing additional functionalities. Instead, the SOVA application is a standalone system whose goal is to provide a user-friendly way of searching among the questions and answers related to programming and software development in general. However, since this feature is a common aspect of Stack Overflow’s website itself, the application should contain other features that make programmer’s life easier. These would include marking a question as user’s favourite which would provide the user with a quick access to the issues they are currently struggling with. Taking notes (annotating) while marking a specific question should also offer certain clearance for the reasons behind marking the question or might simply serve as a note-taking tool within the application. Accessing the history of search results is also one the main features of the system, since its presence eases the process of finding the solutions that helped to solve (or understand) a particular issue from the past.

The product leaves space for improvements of certain aspects which will be described further in the document. Similarly, more features can be developed if time constraints would allow for such scenario. Since the development process follows agile principles to certain extent, some of the application’s features will be improved in future iterations, allowing for minor changes in the requirements or additional feature requests.

Application requirements

Functional requirements

Since the requirements given in the portfolio assignment description are explaining the application's main features well but leave certain amount of freedom in deciding how they should be implemented, a list of functional requirements specific to the SOVA application was built in a form of rational (easy to understand, logical and simplistic) user stories - RUS. This list is presented below.

ID	Requirements:
RUS-01	As a user I want the ability to sign up to SOVA with a username and a password.
RUS-02	As a user I want the ability to log in to the stand-alone SOVA application.
RUS-03	As a user I want the ability to see a Stack Overflow (further referred to as 'SO') question.
RUS-04	As a user I want the ability to see a list of SO questions.
RUS-05	As a user I want the ability to see the related answers for a SO question.
RUS-06	As a user I want the ability to see the related comments for a SO question.
RUS-07	As a user I want the ability to see the related comments for a SO answer.
RUS-08	As a user I want the ability to mark a specific SO question.
RUS-09	As a user I want the ability to annotate a specific SO question that is already marked (or mark and annotate the SO question at the same time)
RUS-10	As a user I want the ability to see a list of SO questions that I have previously marked.
RUS-11	As a user I want the ability to search among SO questions that I have previously marked.
RUS-12	As a user I want the ability to see annotations I wrote down while marking the SO questions.
RUS-13	As a user I want the ability to search for SO questions that contain a specific phrase.
RUS-14	As a user I want the ability to add a filter to my search, so I only search for SO questions that are tagged with a specific tag.
RUS-15	As a user I want the ability to search for SO answers that contain a specific phrase.
RUS-16	As a user I want the ability to search for SO comments that contain a specific phrase.

RUS-17	As a user I want the ability to see a list of my entire search history.
RUS-18	As a user I want the ability to see the author of a question, answer and a comment.
RUS-19	As a user I want the ability to see basic details about an author.
RUS-20	As a user I want the ability to see the specific answers, questions and comments which an author has written.
RUS-21	As a user I want the ability to see a visualization of popular searches, made by the entire user base.

[Figure 1 - Table of user stories]

Please note that the user stories above marked with yellow describe functional requirements which were identified in the initial phase but were decided not to be focused on in the portfolio subproject 1.

Non-functional requirements

The development of the SOVA application is focused mainly on its functional side. However, certain non-functional aspects should be considered in the final product. These are briefly described in the following paragraphs.

Security

Passwords of the application’s users should be stored safely, so they would not be easy for the machines (or humans) to guess or brute force. The application should also be generally resistant to at least the most common attacks such as cross-site scripting vulnerabilities in form validation or search inputs. Additionally, it shouldn't be possible to manipulate the database information directly from the interface (sql-injection and website code manipulation).

Performance

Since the core of the application is searching through big amounts of data, it should be optimized in a way that it can deliver the results in a reasonable time, without causing too much inconveniences to the end-user.

Maintainability

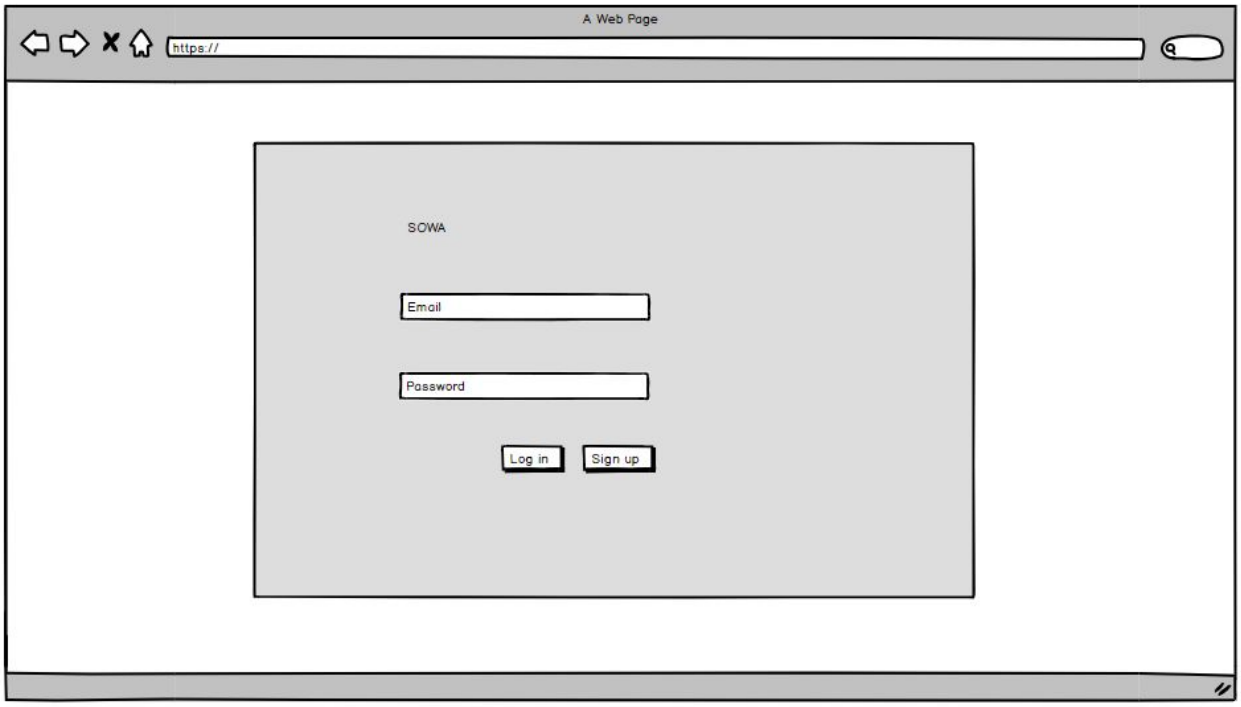
The application layers should be well defined and modularized to an extent where the system is relatively easy to be navigated and the potential issues can be discovered quickly. The maintenance in such system is also easier and clearer to the developers.

Wireframes

Wireframes are used to communicate an idea or a concept about the application. They are meant to be easily disposable sketches as they do not represent the final product in detail. Some of the details can be added at a later revision of the wireframe.

After we had the requirements in place, we brainstormed about how the end user application should look like and came up with some wireframes.
We envisioned the app with three different frames, the *Login and signup* frame, *Search and History* frame and *Question details* frame.

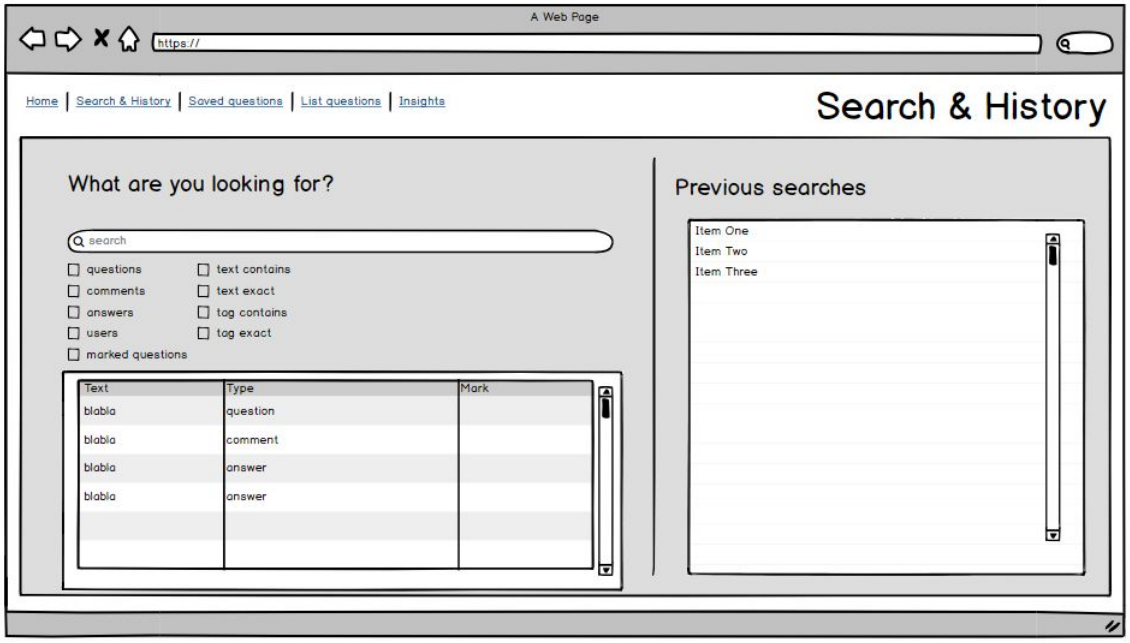
Login and signup



[Figure 2 - Wireframe of signup and login view]

The Login and Signup frame is a simple standard login screen that has two fields one for the email and the other for the password. We provide the user with two possibilities, one for login and the other for sign up.

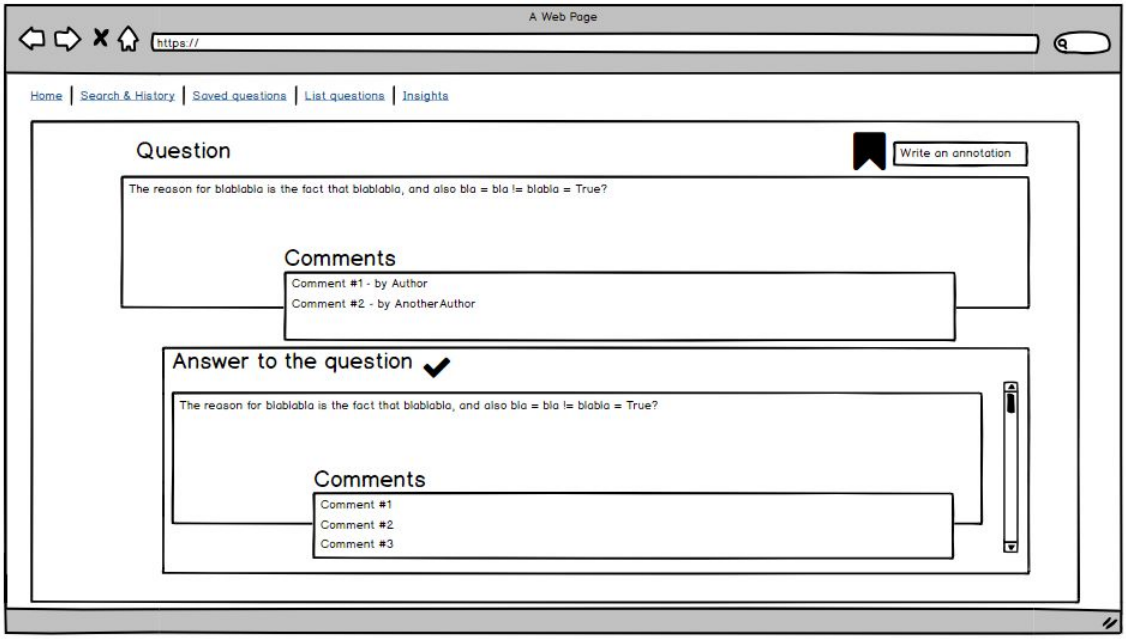
Search and history



[Figure 3 - Wireframe of the Search & History view]

The *Search and History* frame is meant to be the main frame of the application where the user is presented with a search area on the left side and a history of searches on the right side.
The search functionality comes with a search bar and some filters followed by the search result area while the search history is simply a list of previously searched terms.
A user can navigate forward from the *Search and History* frame by clicking on a search result item. If the user chooses to click on a previous search term instead, new search is performed and she still has to click on one of the search result items in order to go to the next frame.

Question and details



[Figure 4 - Wireframe of the Question and details view]

Question and details is the last frame of our application. At this point, the user is presented with a Stack Overflow question that is relevant to her previous search terms. The question is presented in a similar manner with the Stack Overflow layout where the question is on top of the screen and the answers come below, ordered by their relevance. Comments are foldable on both questions and answers.

Modelling the database

After the group has agreed on the application's functionality and visualized the potential final design, it became possible to create the structure for the data that will be stored in the database - model the database. Generally, it was agreed that it will consist of two parts.

1. The QA model, whose basis was provided in the assignment itself and which basically represents the Stack Overflow questions-and-answers data. It contains the information about posts together with their authors and other additional data.
2. The Framework model, which is the base for additional functionality that the SOVA application will introduce. It holds the information about user accounts, marked questions and search history of each user. This model will most likely be amended in the next phases of the portfolio project.

The QA model

Before we could start modelling the database for the application-specific features (the framework model), we had to look at the data from Stack Overflow's system we got in the assignment description (the QA model). The first part, therefore, consisted mainly of figuring out how the given database is structured and how we can split the two big tables, namely posts_universal and comments_universal, into smaller ones - making the database not only more efficient in data storage terms, but also faster to search in.

From the beginning, we tried to look into various ways of modelling the QA part of the database. Therefore, we have used three different ways for creation of this model.

1. Decomposing the two provided tables into smaller ones, using the normalization principles based on functional dependencies. A functional dependency is a constraint between two sets of attributes in a relation from a database. They play a key role in finding issues with a database design and helped us with making the difference between a good and a bad database design.
2. Looking at the real Stack Overflow website and trying to work out how it could work on the database level.
3. Thinking about how we would model the database from start if there was no such system in place yet - using the real website as a reference.

Chosen approach

We chose to apply the ER model making a database design from scratch. Having our requirements at hand and some of the constraints from the product owner, we proceeded to discover new constraints and design an ER diagram.
The given relations were:

comments_universal(commentid, postid, commentscore, commenttext, commentcreatedate, authorid, authordisplayname, authorcreationdate, authorlocation, authorage);

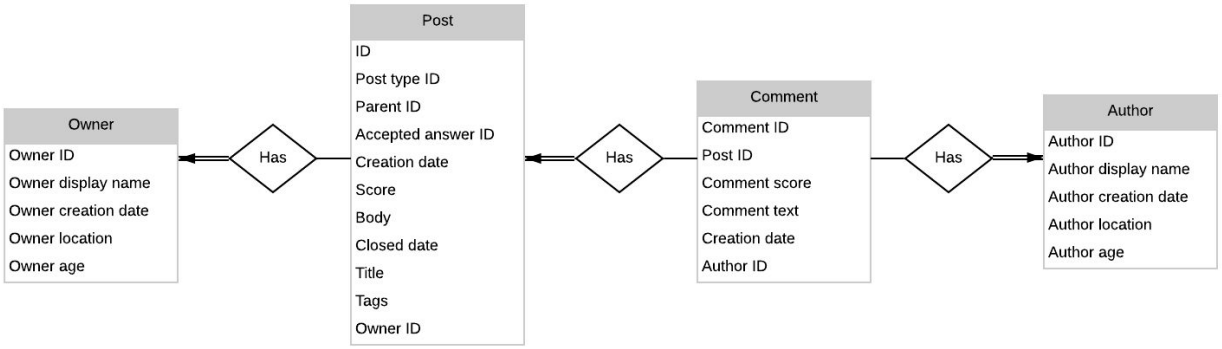
posts_universal(id, posttypeid, parentid, acceptedanswerid, creationdate, score, body, closeddate, title, tags, ownerid, ownerdisplayname, ownercreationdate, ownerlocation, ownerage, linkpostid);

The aforementioned relations have the following constraints:

comments_universal-relation
commentid → postid, commentscore, commenttext, commentcreatedate, authorid
authorid → authordisplayname, authorcreationdate, authorlocation, authorage

posts_universal-relation
id → posttypeid, parentid, acceptedanswerid, creationdate, score, body, closeddate, title, tags, ownerid
ownerid → ownerdisplayname, ownercreationdate, ownerlocation, ownerage

This can be visualized by the following E-R diagram



[Figure 5 - Preliminary E-R diagram of the QA model]

However this diagram is not the normalized version of the database model.

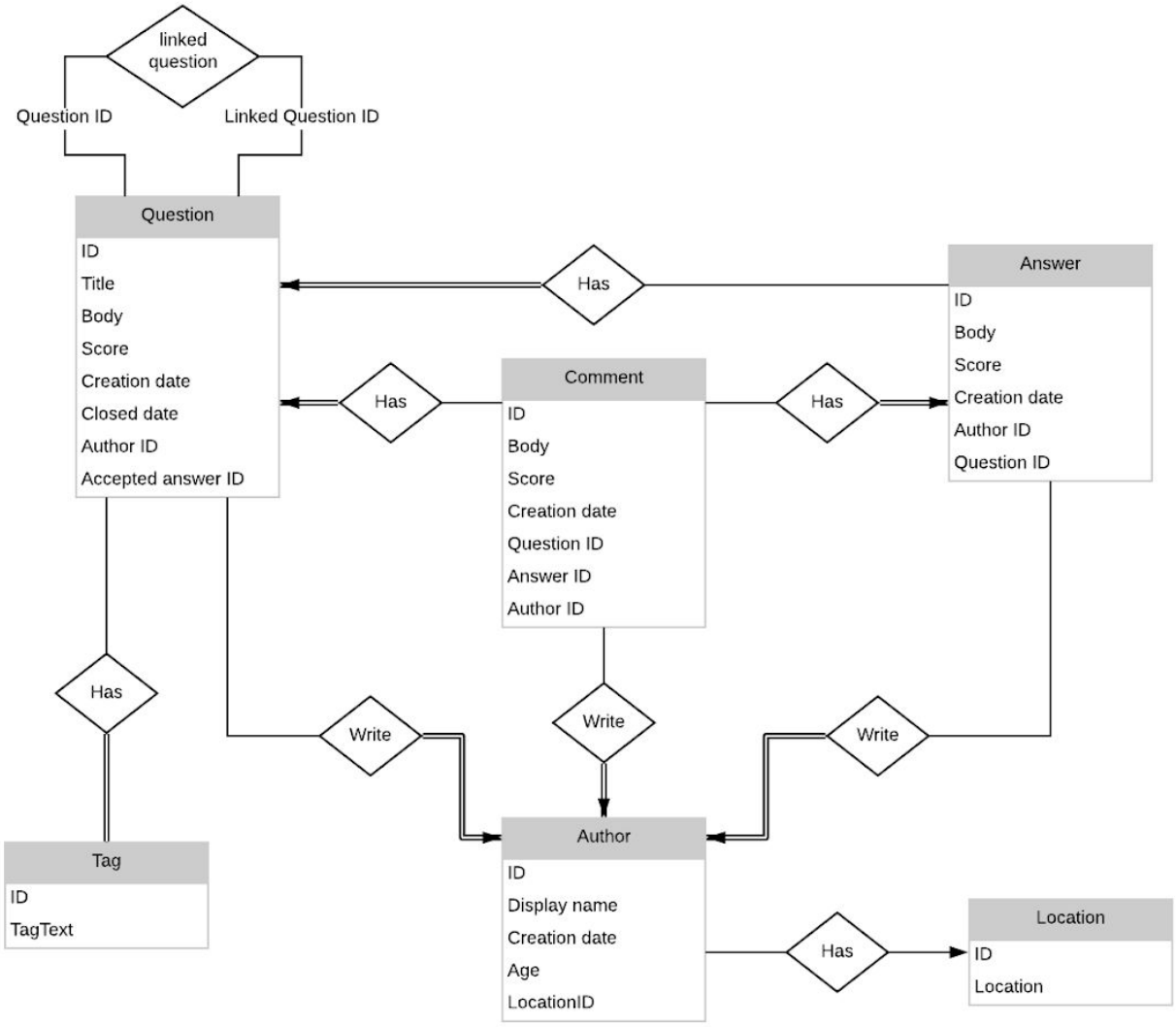
Based on the data given we noticed a few other constraints:
id → linkpostid
ownerid → ownerlocation
id, posttypeid → tags

Final relations

In order to fulfill these constraints, we splitted the existing relations into the following relations:
questions (**id**, title, body, score, creation_date, closed_date, author_id, accepted_answer_id);
answers (**id**, body, score, creation_date, author_id, question_id);

comment (**id**, body, score, creation_date, question_id, answer_id, author_id);
author (**id**, display_name, creation_date, age, location_id);
tag (**id**, tag);
location (**id**, location);

By analyzing the data, we found out that the concept of author from the comments_universal table is the same as the owner from the posts_universal table. Furthermore, Location could be divided further into country, state, city but the data provided is very unreliable and badly formatted.
Below is the final E-R diagram of the QA database.



[Figure 6 - Final E-R diagram of the QA model]

The Framework model

The main part of the application's database is undoubtedly the question-and-answers model which was described in the previous sections. However, since the application is a stand-alone system using the data from Stack Overflow's website, it needs its own database model to support different users and to keep track of their favourite questions and search history. This model is substantially smaller and easier to understand than the previously described one, mainly because of the simple application's features it is agreed to fulfill.

What the framework model should describe is the functionality of the SOVA application which is to keep track of a user's search history as well as the opportunity to mark and annotate posts. Keep in mind that the mentioned users are users of the SOVA application and not of Stack Overflow website itself.

Identified entities

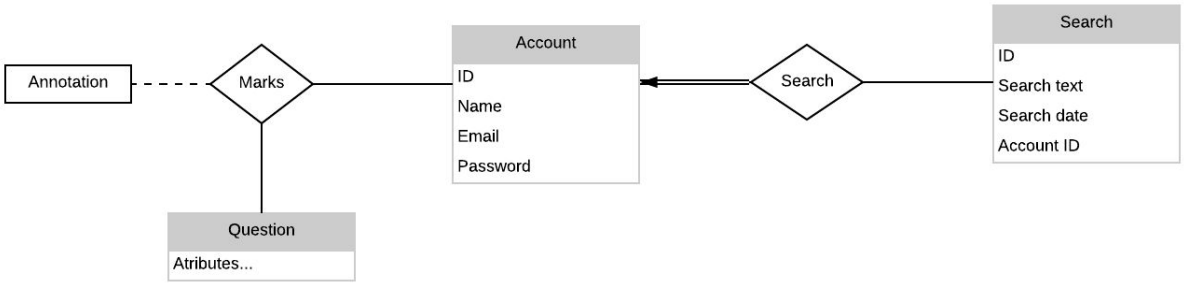
For the application framework, we have decided to use account table as the storage for information about users. When a user creates an account in the application, their email, name and hashed password are stored in this table. Each user is identified with a unique identifier to prevent data collision.

To keep the history of search results, it was decided to create a search table where the information about search phrases is stored. It is connected with the account that the search was performed from, so each users see only their own search history. A date of the search is kept for future references.

An account can also mark and annotate a question and this is the link to the QA part of the database.

E-R diagram for the framework

The diagram below consists of 3 entities with 2 relationships and 1 attribute. The account entity contains ID, name, email and password and an account is able to mark questions from the QA part with the annotation attribute. The search entity has an ID, search text, search date and account ID which links to the account entity.



[Figure 7 - E-R diagram of the Framework model]

The account-search relation is a one to many relation, but the search must have a relation to account if it exists. This is marked with the double arrow pointing from search to account.

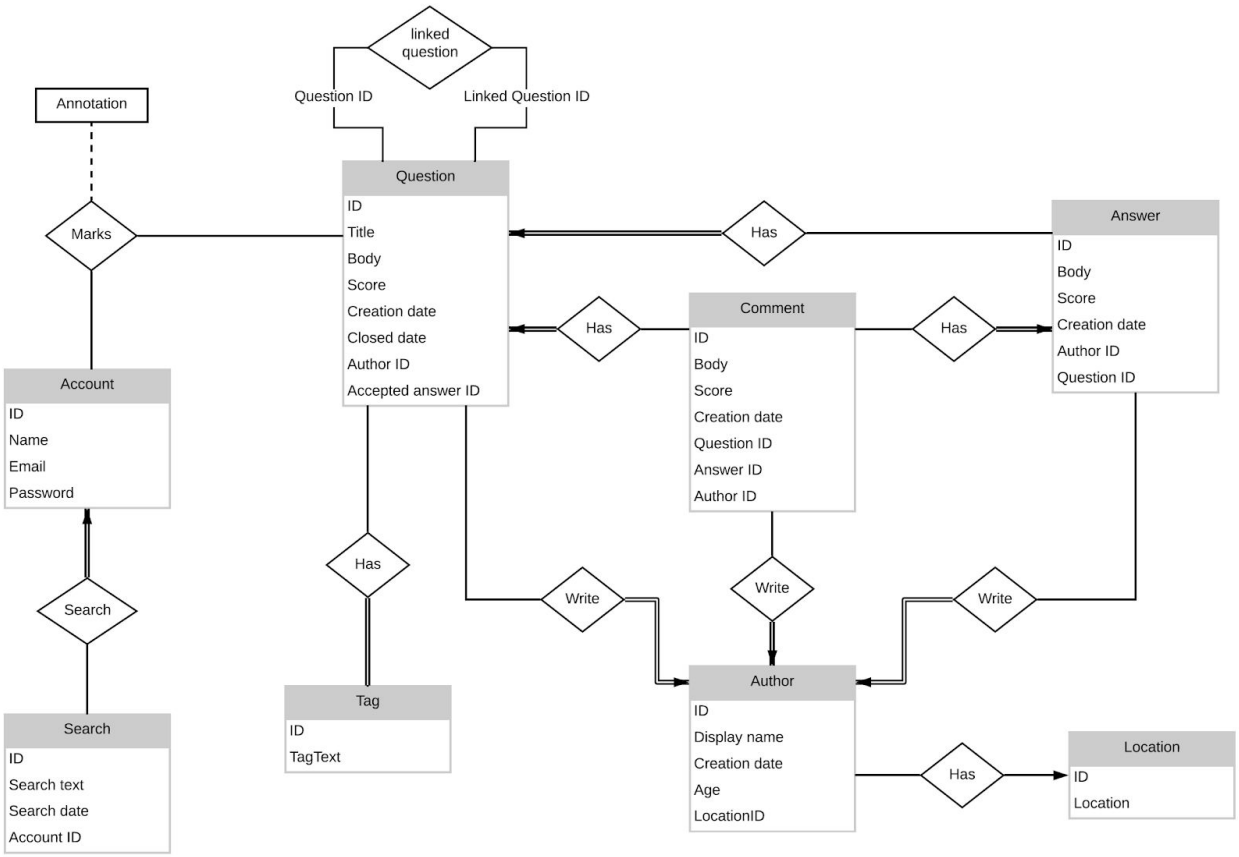
As we agreed, the only link between the two models needs to be the question ID. However, there are two important facts to keep in mind:

1. each user (account, in our case) can mark or annotate multiple questions
2. each question can be marked or annotated by multiple users.

This means that the relationship between a question and an account can be described as many-to-many, in database-modelling terms. This relationship is described by a diamond titled “marks” in the diagram below, since an account (user) marks a question. The relationship has a little addition which is presented as a box connected with the relationship diamond by a dashed line. It represents a relationship attribute of “annotation” whose value can either be filled while the user is marking the question, leaving a note for themselves; or left empty, leaving the question marked without any additional notes.

The combined model

Since the beginning of the application’s development, we have agreed to keep the QA and Framework model as separate as possible. The separation might prove useful if there comes a time in the future, where major changes would have to be done to the Framework database part, especially because it would affect the QA part only to a small degree. Similarly, if the data structure from Stack Overflow’s database would change for whatever reason, the SOVA application would stay almost unaffected. In case the id’s of questions get switched with each other or a question gets deleted, the Framework part is unaffected. Below in figure 8, the combined model is shown with the link between the question entity and the account entity.



[Figure 8 - The combined E-R model]

Database implementations

The database implementations are based on the combined E-R diagram for the QA and the framework as seen above. We chose to split the implementation of the new database into two parts. First script, called the “database creation script”, creates all the necessary tables without any data. The second script, called the “database migration script”, migrates all the data from the old tables (posts_universal and comments_universal) we were given into the newly created tables.

Database creation script

The tables we created are all based on the E-R diagram presented earlier in the report. When we created the new tables, we knew that the order of creation is playing an important role. In SQL there are foreign key constraints that need to refer to other tables. Therefore, it is important to have the tables we refer to before we create the referring tables.

The creation script is thoroughly described in the Appendix A. We chose to present only the most complex step of the database creation script below.

```
CREATE TABLE question_tag (
      question_id INTEGER NOT NULL,
      tag_id INTEGER NOT NULL,
      PRIMARY KEY(question_id, tag_id),
      CONSTRAINT question_tag_question_question_id_fk FOREIGN
KEY(question_id)
            REFERENCES question(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
      CONSTRAINT question_tag_tag_tag_id_fk FOREIGN KEY(tag_id)
            REFERENCES tag(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

[Figure 9 - Step 7 from the database creation script (Appendix A)]

Database migration script

The full migration script is presented in the Appendix B. We chose to present only the step 7 of the migration script where we populate the newly created question tag table with data from the old posts_universal. We split the value of the tags attribute and make a new row for each tag.

Here, the question_tag table gets populated. This table contains the relations between which questions got which tags.

```
-- Step 7. Populate question_tag WITH DATA FROM posts_universal table.
DO $$
DECLARE tag_number integer := (SELECT count(tag) FROM tag); -- the tag
number
            tag_name varchar(100) := '';
            question_ids integer[];
BEGIN
      FOR i IN 1..tag_number
      LOOP
            tag_name := (SELECT tag FROM tag WHERE id = i);
            question_ids := ARRAY(
                              SELECT DISTINCT id
                              FROM posts_universal
                              WHERE tags LIKE '%::' || tag_name || '::%'
                              OR tags LIKE tag_name || '::%'
                              OR tags LIKE '%::' || tag_name
                              OR tags = tag_name
                              );
            FOR j IN ARRAY_LOWER(question_ids,
1)..ARRAY_UPPER(question_ids, 1)
            LOOP
                  INSERT INTO question_tag(question_id, tag_id)
                  VALUES (question_ids[j], i);
            END LOOP;
      END LOOP;
END;
$$;
```

[Figure 10 - Step 7 of the database migration script (Appendix B)]

Functionality

The product owners had prior to specification of functionality required that the following areas should to be covered:

- search
- search history
- marking
- annotation
- management of multiple users

We decided to add these functional specifications to the table seen in the section below, with the functional specifications related ID and supplied arguments.

Functionalities

Short description of the different functionalities of our API

ID	Functionality	Arguments
F-01	Search for questions with text phrases in questions, answers and comments	text, userid
F-02	Save search history	text, userid
F-03	Mark question	user_id, account_id
F-04	Annotate question	user_id, account_id, text
F-05	Sign up	name, email, password
F-06	Sign in	email, password

[Figure 11 - Table of API-functions with description and input arguments]

We implemented the functions listed above in the following way:
F-01 and F-02 are combined in a function called ‘search’ which logs data about each function call into the search table.
F-03 and F-04 are combined in a function called ‘mark_question’ which take the arguments: account_id, question_id and text with the default value of NULL.
F-05 functionality is built within the function called ‘sign_up’ which creates an account with provided data.
F-06 is not suited as a database specific task… Therefore we will not focus on it in this portfolio project, but in a later portfolio project

In regard to F-01 and F-02 we considered an approach of using a trigger to store searches made by users in a “search_history”-table. Conceptually, this approach would work in a way that a trigger function (with the search input as argument) would be called every time an user made a search. But since a trigger is only triggered by an INSERT, UPDATE or DELETE statement, and since our search function would only require a SELECT statement, we decided to disregard the trigger-based approach.

The search function

The purpose is to locate the most relevant questions given the search phrase. We look in questions, answers and comments but we only return questions matching relevant answers and comments. As an input, our function needs the search_texand the account_id that executed the search.

```
CREATE OR REPLACE FUNCTION search(search_text TEXT, account_id INTEGER)
RETURNS TABLE (
      title TEXT,
      body TEXT,
      display_name TEXT
)
AS $$
BEGIN
INSERT INTO search(search_text, search_date, account_id)
VALUES(search_text, NOW() AT TIME ZONE 'UTC', account_id);
RETURN QUERY SELECT question.title, question.body, author.display_name
FROM question
JOIN author ON question.author_id=author.id
JOIN answer ON answer.question_id=question.id
JOIN comment ON comment.question_id=question.id OR
comment.answer_id=answer.id
WHERE LOWER(question.title) LIKE LOWER('%' || search_text || '%')
      OR LOWER(question.body) LIKE LOWER('%' || search_text || '%')
    OR LOWER(answer.body) LIKE LOWER('%' || search_text || '%')
    OR LOWER(comment.text) LIKE LOWER('%' || search_text || '%');
END;
$$
LANGUAGE 'plpgsql';
```

[Figure 12 - The search function (F-01 and F-02)]

First we define what the function should receive as input and what output variables to give. Then the search query is saved in the search table. At last the actual function begins by first joining the question, answer and comment tables and then returns the title, body and display_name of the questions related to the search_text.

The mark_question function

The mark_question function does both the marking and annotation of a question at the same time. If a user marks a question it will be annotated with NULL, and if the user annotates the question, the annotation will of course be the the annotation text. If the user unmarks the question, the whole entry should just be deleted from the database. This will, however, happen in another function.

```
CREATE OR REPLACE FUNCTION mark_question(account_id INTEGER, question_id
INTEGER, annotation TEXT DEFAULT NULL)
RETURNS VOID
AS $$
INSERT INTO marked_question(account_id, question_id, annotation)
VALUES(mark_question.account_id, mark_question.question_id,
mark_question.annotation)
ON CONFLICT(account_id, question_id) DO UPDATE
      SET annotation=mark_question.annotation;
$$
LANGUAGE SQL;
```

[Figure 13 - The mark_question function (F-03)]

The function takes an account_id, a question_id and the annotation with a default of NULL. Then it inserts the values into the marked_question table. If there is a conflict meaning that there already is a row containing the specified account_id and question_id, it will update the annotation instead of adding another row.

The sign_up function

The sign_up function is fairly simple. All it has to do is to save the account information in the database.

```
CREATE OR REPLACE FUNCTION sign_up(name TEXT, email TEXT, password
CHARACTER VARYING(60))
RETURNS VOID
AS $$
INSERT INTO account(name, email, password) VALUES(sign_up.name,
sign_up.email, sign_up.password);
$$
LANGUAGE SQL;
```

[Figure 14 - The sign_up function (F-05)]

The function takes a name, an email and a password and inserts them into the account table.

Possible improvements

We have a major improvement we could make to our creation script. The relation between questions and answers is circular because they both refer to each other. This causes problems for us when we define foreign keys. We solve it by altering the table by adding constraints after creating both tables.

The same problem occurs in the migration script where we migrate data into the question and answer table. Since we need to specify the parent for an answer we need to link to that question, but since the question table is not yet migrated and we have a constraint on the answer table we need to add the parent of the answer after we create the two tables.

Both cases could be solved by creating an additional table “question_answer” which keeps track of all relations between questions and answers.

Additionally we could improve the migration script in step 0 where we use a for loop instead of a while loop. The for loop loops through 5 times and it might only need 4 or less loops. We couldn’t find a smart way to specify the conditions within the while loop needed to solve this problem.

Looking at the functions implemented, specifically F-01 revolving around search, there is room for improvement. The function uses three JOIN statements that negatively affects performance. A possible solution to mitigate this performance issue would be to implement indexes, that allows us to collect, parse and store data in a way that facilitates fast information retrieval. Another possible approach would be to analyze the query for an instance by estimating join cardinalities.

The F-05 functionality can be improved by returning an error message if the user already exists.

Testing

We test by executing a set of queries that checks the old QA model and the new one for data retrieval and data insertion to assure that data integrity is not affected and functional dependencies are still represented.

We have written a test script which tests for data consistency of the new tables and testing the functionalities. The test for data consistency is done by comparing the data from the new tables with the original tables and looking for differences. It does look like it’s consistent. The way we test the functionalities is by running each function once with some input data and testing whether or not the correct actions happen.

The database testing script can be found in Appendix C.

Bibliography

[WikiFD]: https://en.wikipedia.org/wiki/Functional_dependency
[WikiERM]: https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model
[DSC] Database System Concepts 6th Edition Abraham Silberschatz, Henry Korth, S. Sudarshan ISBN: 978-0073523323
[QAPicture]: https://www.pexels.com/photo/questions-answers-signage-208494/

Appendix

Appendix A.

The purpose of this appendix is to serve as documentation on how we made the creation script.

The first part of the script is dropping tables if they exist as we need to create the tables from scratch and not worry whether already created. This is also the complete list of tables we will add.

```
DROP TABLE IF EXISTS search;
DROP TABLE IF EXISTS marked_question;
DROP TABLE IF EXISTS account;
DROP TABLE IF EXISTS comment;
DROP TABLE IF EXISTS question_tag;
DROP TABLE IF EXISTS tag;
DROP TABLE IF EXISTS linked_question;
DROP TABLE IF EXISTS question;
DROP TABLE IF EXISTS answer;
DROP TABLE IF EXISTS author;
DROP TABLE IF EXISTS location;
```

Then we create the location table which is pretty simple. It needs an id which is primary key and the location which cannot be null. The smart thing is that we give the id the datatype Serial which automatically gives it an id as you add data to the database.

```
CREATE TABLE location (
      id SERIAL PRIMARY KEY,
      location TEXT NOT NULL
);
```

In the author table we defined the variables id (which is primary key), display name, creation date, age and location id which links to the location table. We define a foreign key constraint

which makes sure to update affected linked tables in case of deletion or update of the location id.

```
CREATE TABLE author (
      id INTEGER PRIMARY KEY,
      display_name TEXT NOT NULL,
      creation_date TIMESTAMP WITHOUT TIME ZONE NOT NULL,
      age INTEGER,
      location_id INTEGER,
      CONSTRAINT author_location_location_id_fk FOREIGN KEY(location_id)
            REFERENCES location(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

Just like the location table, a tag table is created to manage tags. There is an id and the tag text in this table.

```
CREATE TABLE tag (
      id SERIAL PRIMARY KEY,
      tag TEXT NOT NULL
);
```

The question table gets created with the different attributes. There is a foreign key defined to link the author id to the id of the author table.

```
CREATE TABLE question (
      id INTEGER PRIMARY KEY,
      title TEXT NOT NULL,
      body TEXT NOT NULL,
      score INTEGER NOT NULL,
      creation_date TIMESTAMP WITHOUT TIME ZONE NOT NULL,
      closed_date TIMESTAMP WITHOUT TIME ZONE,
      author_id INTEGER NOT NULL,
      accepted_answer_id INTEGER,
      CONSTRAINT question_author_author_id_fk FOREIGN KEY(author_id)
            REFERENCES author(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

Same as question the answer table gets created with the different attributes. There are two foreign keys defined to author id and question id.

```
CREATE TABLE answer (
      id INTEGER PRIMARY KEY,
      body TEXT NOT NULL,
```

```
      score INTEGER NOT NULL,
      creation_date TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    question_id INTEGER,
      author_id INTEGER NOT NULL,
      CONSTRAINT answer_author_author_id_fk FOREIGN KEY(author_id)
            REFERENCES author(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
    CONSTRAINT answer_question_question_id_fk FOREIGN KEY(question_id)
            REFERENCES question(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

Since defining the accepted answer id foreign key in the question table requires the answer table. We had to add this constraint after both tables had been created. The constraint is a foreign key linking accepted answer id from question with the id from answer.

```
ALTER TABLE question ADD CONSTRAINT question_answer_accepted_answer_id_fk
FOREIGN KEY(accepted_answer_id)
      REFERENCES answer(id)
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

Since the relationship between questions and tags is many-to-many we have to create a table to manage the connections. The table consists of question id and tag id and there are defined the foreign keys for both.

```
CREATE TABLE question_tag (
      question_id INTEGER NOT NULL,
      tag_id INTEGER NOT NULL,
      PRIMARY KEY(question_id, tag_id),
      CONSTRAINT question_tag_question_question_id_fk FOREIGN
KEY(question_id)
            REFERENCES question(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
      CONSTRAINT question_tag_tag_tag_id_fk FOREIGN KEY(tag_id)
            REFERENCES tag(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

This is also a table to manage a many-to-many relationship between question id and linked question id.

```
CREATE TABLE linked_question (
      question_id INTEGER NOT NULL,
      linked_question_id INTEGER NOT NULL,
      PRIMARY KEY(question_id, linked_question_id),
      CONSTRAINT question_question_id_fk FOREIGN KEY(question_id)
            REFERENCES question(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
      CONSTRAINT question_linked_question_id_fk FOREIGN
KEY(linked_question_id)
            REFERENCES question(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

As with the question and answer table the comment table is created with some foreign keys.

```
CREATE TABLE comment (
      id INTEGER PRIMARY KEY,
      text TEXT NOT NULL,
      score INTEGER NOT NULL,
      creation_date TIMESTAMP WITHOUT TIME ZONE NOT NULL,
      author_id INTEGER NOT NULL,
      question_id INTEGER,
      answer_id INTEGER,
      CONSTRAINT comment_author_author_id_fk FOREIGN KEY(author_id)
            REFERENCES author(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
      CONSTRAINT comment_question_question_id_fk FOREIGN KEY(question_id)
            REFERENCES question(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
      CONSTRAINT comment_answer_answer_id_fk FOREIGN KEY(answer_id)
            REFERENCES answer(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

Now the tables for the framework are being created. At first the account table is created which stores information about the users that use the SOVA software. It has an id, a name and an email.

```
CREATE TABLE account (
      id SERIAL PRIMARY KEY,
```

```
      name TEXT NOT NULL,
      email TEXT NOT NULL,
      password CHARACTER VARYING(60) NOT NULL
);
```

The search table is created with id as a bigint, with the reasoning that the number of searches could potentially become huge. The other attributes are defined and the foreign key is set.

```
CREATE TABLE search (
      id BIGSERIAL PRIMARY KEY,
      search_text TEXT NOT NULL,
      search_date TIMESTAMP WITHOUT TIME ZONE NOT NULL,
      account_id INTEGER NOT NULL,
      CONSTRAINT search_account_account_id_fk FOREIGN KEY(account_id)
            REFERENCES account(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

The marked question table contains information about which account id and question id that an annotation belongs to. The primary key is both account id and question id together.

```
CREATE TABLE marked_question (
      account_id INTEGER NOT NULL,
      question_id INTEGER NOT NULL,
      annotation TEXT,
    PRIMARY KEY (account_id, question_id),
      CONSTRAINT marked_question_account_account_id_fk FOREIGN
KEY(account_id)
            REFERENCES account(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
    CONSTRAINT marked_question_question_question_id_fk FOREIGN
KEY(question_id)
            REFERENCES question(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

Appendix B.

Step 0

In step 0 we deal with inconsistent data in the original dataset, meaning data that refers to non-existent values. Since the database we received is a small part of the entire Stack Overflow database there are examples of comments linking to questions that is not in this part of the database. And since we demand referential integrity, we decided to remove the inconsistent data. The way this is done is by looping through the database and checking if the different data is linking to other data. If not it will be deleted. The procedure loops through 5 times since we didn't quite know how to do the condition in the while statement. 5 times should be enough to remove all inconsistencies. A while loop is a improvement to be made.

```
DO $$
BEGIN
      -- LOOP 5 times TO ensure all dependecies of non-existend data are
removed.
      -- 303 Rows from posts_universal are removed.
      -- 616 Rows from comments_universal are removed.
      FOR i IN 1..5
      LOOP
      -- Removes all the questions that refer to a non-existent linked
question.
      DELETE FROM posts_universal WHERE linkpostid NOT IN (SELECT id FROM
posts_universal);
      -- Removes all the questions that refer to a non-existent accepted
answer.
      DELETE FROM posts_universal WHERE acceptedanswerid NOT IN (SELECT id
FROM posts_universal);
      -- Removes all the answers that refer to a parentid of a non-existing
question.
      DELETE FROM posts_universal WHERE parentid NOT IN (SELECT id FROM
posts_universal);
      -- Remove all the questions that link to a non-question id.
      DELETE FROM posts_universal WHERE linkpostid NOT IN (SELECT id FROM
posts_universal WHERE posttypeid = 1);
      -- Remove all the comments that do not link to a post id.
      DELETE FROM comments_universal WHERE postid NOT IN (SELECT DISTINCT
id FROM posts_universal);
      END LOOP;
END;
$$;
```

Step 1

In this step the location table gets populated by adding the distinct locations to the location table. As locations gets added the id will be assigned automatically since we gave it the serial data type..

```
-- Step 1. Find all the distinct locations and populate location table
INSERT INTO location(location)
SELECT DISTINCT ownerlocation
FROM posts_universal
WHERE ownerlocation IS NOT NULL;
```

Step 2a

This step populates the author table with data from the posts_universal table where the had specified a location. In the posts_universal table the author of posts and comments is known as “owner”.

```
-- Step 2a. Populate the author table using the already populated locations
INSERT INTO author(id, display_name, creation_date, age, location_id)
SELECT DISTINCT ownerid, ownerdisplayname, ownercreationdate, ownerage,
"location".id
FROM posts_universal
JOIN "location" ON posts_universal.ownerlocation = "location"."location";
```

Step 2b

This step populates the author table with owners from the posts_universal table where the user does not have a specified a location.

```
-- Step 2b. Populate the author table with authors that have NULL location
values
INSERT INTO author(id, display_name, creation_date, age)
SELECT DISTINCT ownerid, ownerdisplayname, ownercreationdate, ownerage
FROM posts_universal
WHERE ownerlocation IS NULL;
```

Step 3

This step populates the answer table with data from the posts_universal table. A posttypeid with the value “2” indicates an answer.

```
-- Step 3. Populate the answer table using the already populated author
table with data from posts_universal.
-- Check if it still works. we changed the join condition.
```

```
INSERT INTO answer(id, body, score, creation_date, author_id)
SELECT DISTINCT posts_universal.id, body, score, creationdate, ownerid
FROM posts_universal
WHERE posttypeid = 2;
```

Step 4a

This step populates the question table with data from posts_universal with posttypeid of “1” which indicates questions.

```
-- Step 4. Populate the question table using the already populated answer
and author table with data from posts_universal.
INSERT INTO question(id, title, body, score, creation_date, closed_date,
author_id, accepted_answer_id)
SELECT DISTINCT id, title, body, score, creationdate, closeddate, ownerid,
acceptedanswerid
FROM posts_universal
WHERE posttypeid = 1;
```

Step 4b

This step specifies the question that the answers relate to.

```
-- Step 4b. After the question and answer tables have been created, we can
proceed by updating the question_id (FK) with the referenced value
(parentid / answer.id) in the answer table
UPDATE answer
SET question_id = posts_universal.parentid
FROM posts_universal
WHERE posts_universal.posttypeid = 2
and answer.id = posts_universal.id;
```

Step 5

This step populates the linked_question table by finding the questions linked to each question and storing that relation.

```
-- Step 5. Populate linked_question table with data from posts_universal
table.
INSERT INTO linked_question (question_id, linked_question_id)
SELECT id, linkpostid
FROM posts_universal
WHERE linkpostid IS NOT NULL
AND posttypeid = 1;
```

Step 6

The tag table will be populated with data from the posts_universal table. The challenge is due to the way tags are stored in the database. It is a long text string separated by two colons "::". First the text string is split into an array of strings containing the tags. Then the function loops through that array and inserts tags into the tag table if they don't already exist.

```
-- Step 6. Populate tag table with data from posts_universal table.
DO $$
DECLARE tag_row varchar(100)[];
            tag_string varchar(100) := '';
BEGIN
      FOR tag_row IN (SELECT DISTINCT regexp_split_to_array(tags, TEXT
'::') FROM posts_universal WHERE tags IS NOT NULL)
      LOOP
            FOR i IN ARRAY_LOWER(tag_row, 1)..ARRAY_UPPER(tag_row, 1)
            LOOP
                  tag_string := tag_row[i];
                  IF NOT EXISTS(SELECT tag FROM tag WHERE tag.tag =
tag_string) THEN
                        INSERT INTO tag(tag) VALUES (tag_string);
                  END IF;
            END LOOP;
      END LOOP;
END;
$$;
```

Step 7

Here the question_tag table gets populated. This table contains the relations between which questions got which tags.

```
-- Step 7. Populate question_tag WITH DATA FROM posts_universal table.
DO $$
DECLARE tag_number integer := (SELECT count(tag) FROM tag); -- the tag
number
            tag_name varchar(100) := '';
            question_ids integer[];
BEGIN
      FOR i IN 1..tag_number
      LOOP
            tag_name := (SELECT tag FROM tag WHERE id = i);
            question_ids := ARRAY(
```

```
                              SELECT DISTINCT id
                              FROM posts_universal
                              WHERE tags LIKE '%::' || tag_name || '::%'
                              OR tags LIKE tag_name || '::%'
                              OR tags LIKE '%::' || tag_name
                              OR tags = tag_name
                              );
            FOR j IN ARRAY_LOWER(question_ids,
1)..ARRAY_UPPER(question_ids, 1)
            LOOP
                  INSERT INTO question_tag(question_id, tag_id)
                  VALUES (question_ids[j], i);
            END LOOP;
      END LOOP;
END;
$$;
```

Seen within the outer loop there are four distinct cases for matching tags in regard to their inherent order...

Step 8a

This step populates the location table with data from the comments_universal table.

```
-- Step 8a. Populate the location table with data from comments_universal
INSERT INTO location(location)
SELECT DISTINCT authorlocation
FROM comments_universal
WHERE authorlocation IS NOT NULL
ON CONFLICT DO NOTHING;
```

Step 8b

Here the author table gets populated with data from comments_universal where the author has a location specified. If the author already exists nothing will happen due to the ON CONFLICT DO NOTHING statement.

```
-- Step 8b. Populate the author table with data from comments_universal
INSERT INTO author(id, display_name, creation_date, age, location_id)
SELECT DISTINCT authorid, authordisplayname, authorcreationdate, authorage,
"location".id
FROM comments_universal
JOIN "location" ON comments_universal.authorlocation =
"location"."location"
ON CONFLICT DO NOTHING; -- avoid conflicting duplicates.
```

Step 8c

Same as with 8b the author gets populated with data from comments_universal, but this time where authors didn't specify a location.

```
-- Step 8c. Populate the author table with data from comments_universal
that have a null location.
INSERT INTO author(id, display_name, creation_date, age)
SELECT DISTINCT authorid, authordisplayname, authorcreationdate, authorage
FROM comments_universal
WHERE authorlocation IS NULL
ON CONFLICT DO NOTHING; -- avoid conflicting duplicates.
```

Step 9a

Step 9a populates the comment table which has not been populated yet. The data is coming from comments_universal. Only the comments related to questions will be populated here.

```
-- Step 8d. Populate the COMMENT TABLE WITH DATA FROM comments_universal
table.
-- Populate all the question comments:
INSERT INTO "comment" (id, "text", score, creation_date, author_id,
question_id)
SELECT DISTINCT commentid, commenttext, commentscore, commentcreatedate,
authorid, question.id
FROM comments_universal
JOIN question ON comments_universal.postid = question.id
ORDER BY commentid;
```

Step 9b

In the last step the comments related to answers will be inserted into the comment table.

```
-- Populate ALL the answer COMMENTS:
INSERT INTO "comment" (id, "text", score, creation_date, author_id,
answer_id)
SELECT DISTINCT commentid, commenttext, commentscore, commentcreatedate,
authorid, answer.id
FROM comments_universal
JOIN answer ON comments_universal.postid = answer.id;
```

Appendix C.

We checked the integrity of the data by writing a test script. The script covers some of the migration steps we executed earlier in order to check if we did not lose any data.

```
-- Tests of data integrity
-- Step 1 Compare the location data in location table with the one in
post_universal table
      (SELECT location FROM "location")
EXCEPT
      (SELECT DISTINCT ownerlocation FROM posts_universal WHERE
ownerlocation IS NOT NULL)
UNION
      (SELECT DISTINCT ownerlocation FROM posts_universal WHERE
ownerlocation IS NOT NULL)
EXCEPT
      (SELECT location FROM "location");
 location
----------
(0 rows)

-- Step 2 Compare the author data in author table with the one in
post_universal table
      (SELECT author.id, display_name, creation_date, age,
"location"."location"
      FROM author
      JOIN "location" ON author.location_id = "location".id)
EXCEPT
      (SELECT DISTINCT ownerid, ownerdisplayname, ownercreationdate,
ownerage, ownerlocation
      FROM posts_universal
      WHERE ownerlocation IS NOT NULL)
UNION
      (SELECT DISTINCT ownerid, ownerdisplayname, ownercreationdate,
ownerage, ownerlocation
      FROM posts_universal
      WHERE ownerlocation IS NOT NULL)
EXCEPT
      (SELECT author.id, display_name, creation_date, age,
"location"."location"
      FROM author
```

```
      JOIN "location" ON author.location_id = "location".id);
 id | display_name | creation_date | age | location
----+--------------+---------------+-----+----------
(0 rows)

-- Step 3 Compare the answer table data with the one from posts_universal
table.
-- This revealed those answers where the author had a missing location
hence added Step 2b)
      (SELECT answer.id, answer.body, answer.score, answer.creation_date,
author.display_name, author.creation_date, answer.question_id
      FROM answer
      JOIN author ON author.id = answer.author_id)
EXCEPT
      (SELECT DISTINCT posts_universal.id, body, score, creationdate,
ownerdisplayname, ownercreationdate, parentid
      FROM posts_universal
      WHERE posts_universal.posttypeid = 2)
UNION
      (SELECT DISTINCT posts_universal.id, body, score, creationdate,
ownerdisplayname, ownercreationdate, parentid
      FROM posts_universal
      WHERE posts_universal.posttypeid = 2)
EXCEPT
      (SELECT answer.id, answer.body, answer.score, answer.creation_date,
author.display_name, author.creation_date, answer.question_id
      FROM answer
      JOIN author ON author.id = answer.author_id);
 id | body | score | creation_date | display_name | creation_date |
question_id
----+------+-------+---------------+--------------+---------------+--------
-----
(0 rows)

-- Step 4 Compare the question table data with the one from post_universal
table.
-- Prerequisite: PERFORM step 0. IN ORDER TO have a clean DATABASE
      (SELECT id, title, body, score, creationdate, closeddate,
ownerdisplayname, acceptedanswerid
      FROM posts_universal
      WHERE posttypeid = 1)
EXCEPT
```

```
      (SELECT question.id, title, body, score, question.creation_date,
closed_date, author.display_name, question.accepted_answer_id
      FROM question
      JOIN author ON question.author_id = author.id)
UNION
      (SELECT question.id, title, body, score, question.creation_date,
closed_date, author.display_name, question.accepted_answer_id
      FROM question
      JOIN author ON question.author_id = author.id)
EXCEPT
      (SELECT id, title, body, score, creationdate, closeddate,
ownerdisplayname, acceptedanswerid
      FROM posts_universal
      WHERE posttypeid = 1);
 id | title | body | score | creationdate | closeddate | ownerdisplayname |
acceptedanswerid
----+-------+------+-------+--------------+------------+------------------+
------------------
(0 rows)


-- Step 5 Compare the linked_question data with the one from
posts_universal table.
      (SELECT question_id, linked_question_id
      FROM linked_question)
EXCEPT
      (SELECT id, linkpostid FROM posts_universal WHERE linkpostid IS NOT
NULL)
UNION
    (SELECT id, linkpostid FROM posts_universal WHERE linkpostid IS NOT
NULL)
EXCEPT
      (SELECT question_id, linked_question_id
      FROM linked_question);
 question_id | linked_question_id
-------------+--------------------
(0 rows)


-- Step 6+7 Compare the tag and question_tag DATA WITH the NONE FROM
posts_universal TABLE.
-- ISSUE c and r tags are added to innocent questions due to the LIKE
```

```
condition on migration step 7
SELECT tags FROM posts_universal WHERE id = 14352483;
                    tags
-----------------------------------------------
 python::class::methods::couchdb::decorator
(1 row)

SELECT tag
FROM tag
JOIN question_tag ON tag.id = question_tag.tag_id
JOIN question  ON question.id = question_tag.question_id
WHERE question.id = 14352483;
    tag
-----------
 python
 class
 methods
 couchdb
 decorator
(5 rows)

-- CHECK IF we have the same number OF posts AS there were INITIALLY:
SELECT count(DISTINCT id) FROM posts_universal; -- 13380
 count
-------
 13380
(1 row)

SELECT count(*) FROM answer; -- 11182
 count
-------
 11182
(1 row)

SELECT count(*) FROM question; -- 2198
 count
-------
  2198
(1 row)

-- Check if we have the same number of comments as there were initially,
SELECT count(DISTINCT commentid) FROM comments_universal; -- 31426
```

```
 count
-------
 31426
(1 row)

SELECT count(id) FROM "comment"; -- 31426
 count
-------
 31426
(1 row)

-- testing functions and procedures
-- sign up a user
SELECT sign_up('Troels', 'troels@ruc.bug', 'troels123');
 sign_up
---------

(1 row)

-- search -- very slow.
SELECT * FROM search('element', 1);
title                                                |
-----------------------------------------------------------------------------
------------------+
 2.9999999999999999 >> .5?
|
| Zen
 404 when calling a c# web api from another c# web forms application
|
| Ronald
 Abuse of C# lambda expressions or Syntax brilliance?
|
| Remus Rusanu
 Acceptable CSS hacks/fixes
|
| Mark
 AccessControlException when using Spring Security with OpenID
|
| Ralph Kretzler
 "Access Denied" when trying to access file in web app
|
-----------------------------------------------------------------------------
```

```
------------------+
(10 rows)

-- mark question and then update the question with annotation
SELECT mark_question(1, 13724063);
 mark_question
---------------

(1 row)

SELECT mark_question(1, 13724063, 'This is an amazing thing to know');
 mark_question
---------------

(1 row)

SELECT * FROM marked_question;
 account_id | question_id |            annotation
------------+-------------+----------------------------------
          1 |    13724063 | This is an amazing thing to know
(1 row)
```