Roskilde University

Stack Overflow Viewer Application (SOVA)

Project group: raw4
Özge Yaşayan
Shushma Devi Gurung
Ivan Spajić
Manish Shrestha

MSc. in Computer Science

Date: 14/10/2019

Table of contents

1. Introduction **2**

2. Normalization of tables of QA data model **3**

3. Design and creation of framework data model **6**

4. Relating QA & Framework Data Model **7**

4.1. Searching functionality 7

4.2. Assigning weights to words 8

4.3. Stored Procedures 9

4.3.1. Stored Procedure vs Stored Function 9

4.3.2. Implementation of stored procedure 10

5. Improving performance (Indexing) **11**

5.1. Implementation of indexing 11

6. Appendices **13**

1. QA data model - ER diagram 13

2. QA + Framework data model - ER diagram 14

3. Output of the test script 14

1. Introduction

The project Stack Overflow Viewer application (SOVA) is aimed towards providing the tools in Stack Overflow to help computer programmers. The goal is to support two complementary functions. The first is a keyword-based search which can be used to discover answers to questions about programming. The second is a search history along with a marking and note-taking option so that programmers can make a track of the most interesting answers already found. We use the data provided in the database-dump file "stackoverflow_universal.backup1" to create our own database and load the data into the database.

This database contains four tables:

- *posts_universal*
- *comments_ universal*
- *words*
- *stopwords*

The two tables, *posts_universal* and *comments_universal* contained the raw data fetched from Stack Overflow.

The *words* table is helpful in simplifying the indexing for information retrieval on columns in the Stack Overflow data. The important part of this table is the inverted index that provides on the post table. The *stopwords* table is a single column table listing a set of words usually considered as stop words.

2. Normalization of tables of QA data model

Some of the original database tables containing raw data from Stack Overflow were constructed in such ways that they violated one or more relational normalization forms. The issues were most obvious with the *posts_universal* and *comments_universal* tables, both of which contained data that pertained to more than just the entities they respectively represented. For example, both tables included data for a sort of user that served as an *author* in the case of *comments_universal*, or an *owner* in the case of *posts_universal*, even though they also already contained data regarding a post or a comment. Such violations of normalization forms produced a normalization requirement that had to be satisfied before continuing with the project.

Since all Normalization Forms (NFs) require that a table satisfies the preceding normalization form (or is unnormalized in the case of 1NF), it only made sense to start from 1NF. The *posts_universal* table contained a column called *tags* that represented Stack Overflow question tags. The problem with this column was that the data contained was a series of concatenated strings separated by a double colon separator (::). Since a table column cannot contain multiple values per column, these tag values were split up to normalize for 1NF. This

solution also proved more elegant as the *posts_universal* table contained both questions and answers while Stack Overflow only permits for tags on questions, resulting in many answers containing *null* values in the column. Since the relation between posts (questions) and tags is a Many-to-Many (M-M) relationship, the logical solution was to create a specific table for tags and connect it to the posts via *ID* references.

As mentioned, both *posts_universal* and *comments_universal* contained multiple columns not attributed to the entity their tables represented. To satisfy 2NF, these columns had to be separated since post *authors*, or comment *owners*, did not exclusively depend on their (whole) primary keys as many users could easily author multiple posts as well as comments. Thus, both authors and owners were decoupled from their tables, however a simple observation revealed that both entities contained identical attributes, and in some cases the same data. Queries revealed that certain *authors* existed as *owners* and vice-versa. This made sense since Stack Overflow only has one kind of user of both posts as well as comments, implying that these were, in fact, the same entity. Hence, after decoupling, a decision was made to combine the two into a single table, *so_members*.

Since *posts_universal* contained both questions and answers in one table, while also including specific columns for both data concepts, there existed a considerable number of *null* value cases. For example, *posts_universal* contained *acceptedanswerid* and *closeddate* columns, which only related to questions and not answers. Stack Overflow questions are the only type of posts that can have an accepted answer, as well as a date of closing. Likewise, the table contained a *parentid* column, which only belonged to *answers* as answer posts are the only type of post with a parent question.

posts_universal also consisted of a *posttype* column containing either a *1* or a *2* as values for distinguishing between questions and answers. Since answers are the only kind of post with a parent (question), these were therefore the only tuples containing a non-null value in the *parentid* column. This meant that all tuples with a non-null value in *parentid* also had *2*s in their *posttype* attribute, while all tuples with *null*s in *parentid* contained *1*s in their *posttype* column, respectively. Since these two attributes were strictly related and could be inferred, one from another, the table violated 3NF.

The *posts_universal* table was therefore decomposed into two specialized entities, *questions* and *answers*, where *questions* contained only the question-specific attributes (e.g. *accepted_answer_id*, *closed_date*), while answers contained only answer-specific ones (e.g. *parent_id*). In the case of *answers* an additional boolean column, *accepted*, was created to signify whether or not the respective answer is accepted for its parent question. This solution avoids keeping many *null* values for questions that do not have accepted answers. These changes inherently made it so that the database became more efficient as the size on disk became smaller with the removal of specialized columns having to be represented by *null*s for all the cases where they were not applicable. This was most severe in *parentid* and *closeddate* columns, since questions cannot have a parent and answers cannot be closed, meaning for all tuples not adhering to the attributes, these values were always *null*. The disadvantage with

specializing these entities was that querying for data (e.g. textual content) regarding either questions or answers comes at the cost of either a join operation in a single query, or multiple queries involving where clauses.

posts_universal tuples also contained a *linkpostid* attribute (weakly) referencing the *ID* of the post they were linked to. Since many posts can be linked to a single post, this represented an M-1 relationship. Some posts had no linked posts, and thus contained *null*s, so for the sake of consistency, a *link_posts* relation was created, referencing appropriate posts (questions) to others.

Having decoupled *comments* entities from *so_members* entities, it became apparent that Stack Overflow comments are nothing more than posts, containing identical attributes (e.g. *text*, *score*, *creation date*, etc.). This presented the option of decomposing the *comment* relation into its respective identical attributes and the relation with the post they belong to. The posts and comments were thus merged into a single *submissions* table, keeping the relevant user submissions in one place, while separating their respective specializations (*questions*, *answers*, *comments*) in a separate relation.

An advantage to this approach is that any comment-specific queries will be more optimized, and thus faster. Regarding a count query, the results will be fetched faster due to the much smaller size of the table, and the way it is stored.[1] A query returning the number of comments for certain types of posts or users will have to join smaller tables and thus load far less into memory as opposed to fetching all the entity attributes. Since *comments* only contains two columns, the overhead for the table is much smaller. Another advantage lies in the fact that any textual indexes built around post contents and searching can now seamlessly incorporate comments into the functionality. A disadvantage, however, would be that queries regarding any functionality to do with the contents of the comments themselves are automatically slower, since any and all such operations would require joins with the main *submissions* relation. It should be noted that a workaround could be built using materialized views, where such queries were already pre-processed and would therefore require no additional execution time, but this solution thus requires more available space.

An issue subsequently presented with populating the comments into the *submissions* table was that some comments had colliding *ID* values with some posts, as they used to be separate entities. This was resolved by giving all comments a negative *ID* value – based on the one they originally had.

Due to time constraints, the *words* table was left untouched, but it should be noted that this table became obsolete after decoupling the previous relations. The *tablename* and *what*

---

[1] https://dba.stackexchange.com/questions/83756/columns-count-in-select-query-and-sql-server-buffer-cache

columns were now unnecessary as all the references to any forms of text came from the same table, *submissions*.

3. Design and creation of framework data model

The database for SOVA is to be built on two different data models. The previously discussed QA data model and a data model to be created, named Framework model. This model is meant to support the framework which includes users, markings, annotations and history. The goal is to combine and implement the two different data models onto a single database.

To achieve this, we created four different tables, aiming the new data model's goal.

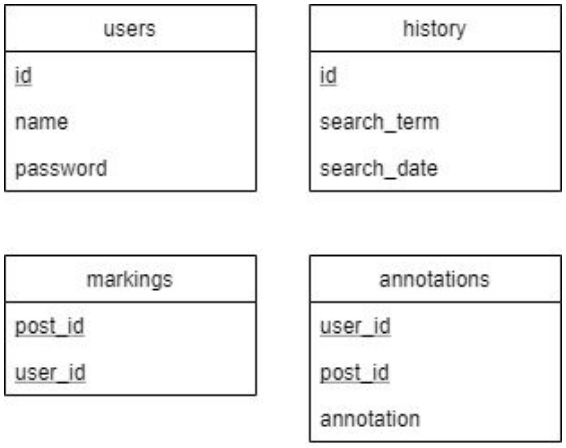


Figure 1: Framework data model

The intended goal of the Framework data model is to:

- allow users to search for submissions
- keep history of user's search terms
- mark submissions for users to find them in the future
- make annotations on the submissions.

As this is a preliminary design and the aim is to keep the tables unsophisticated, the *users* table consists of three attributes. *id* as the unique, auto-incrementing primary key, *name* as the user's "username" or display name which is meant to be unique and their *password*, stored after it gets salted and hashed. The user's name is designed to be unique as it is the only differentiating themselves from another user.

Next table is *history*, which is supposed to keep logs of user's search terms. It also has three attributes, *id* as the unique, auto-incrementing primary key, *search_term* as the string which the user queried for and *search_date* which is a timestamp of when the string was queried for.

There is a requirement of another table to show the relation between user and their history of searched terms. Hence, the relation *user_history* is introduced to store a user's specific search history. It consists of *user_id* and *history_id*.

Figure 2: A relation between QA and Framework data model

Next relation defined is called *markings* which is meant to be the relationship between *submissions* and user; after the two frameworks are combined. As it is a relationship between two tables, it consists of the primary keys between the said table; i.e *user_id* and *submission_id*.

The final table is called *annotations*. The purpose of this table is to allow user to write annotations on submissions for personal use. As the *user_history* table, annotations is also a relationship between *submissions* and *users*, with the addition of the annotation itself. Hence, this table consists of user_id, *submission_id* and *annotation* attributes.

4. Relating QA & Framework Data Model

4.1. Searching functionality

Besides the simple search algorithm (D1) that finds posts which contain a search term them, we needed to implement functions that would make it possible to retrieve posts that are related to search terms.

For questions D2, D3 and D4, we followed the instructions from the subproject requirements document. How we assigned weights to words for question D5 will be explained in the following subsection.

For question D6, after assigning weights to words and having a weighted index table, we decided to first retrieve the posts that contained at least one of the search terms, and added the weights of the contained search terms which would later become their "ranks", then ordered the posts by their ranks in descending order.

For question D7, we went with the suggested implementation and first found the posts that contained at least one of the search terms, and then found the word frequencies in those posts and ordered the words by their frequencies in descending order, ignoring the stop words. For this, we used the inverted index table that did not include the weights, because we realized we didn't need to use the weights, but only the word frequencies, so there was no need to use the "weight" column which was the only attribute that weighted word index table had and the

inverted index table didn't. Therefore, we decided that using Boolean indexing would be sufficient.

Here are some illustrations of the results we gathered from the solution of question D7:

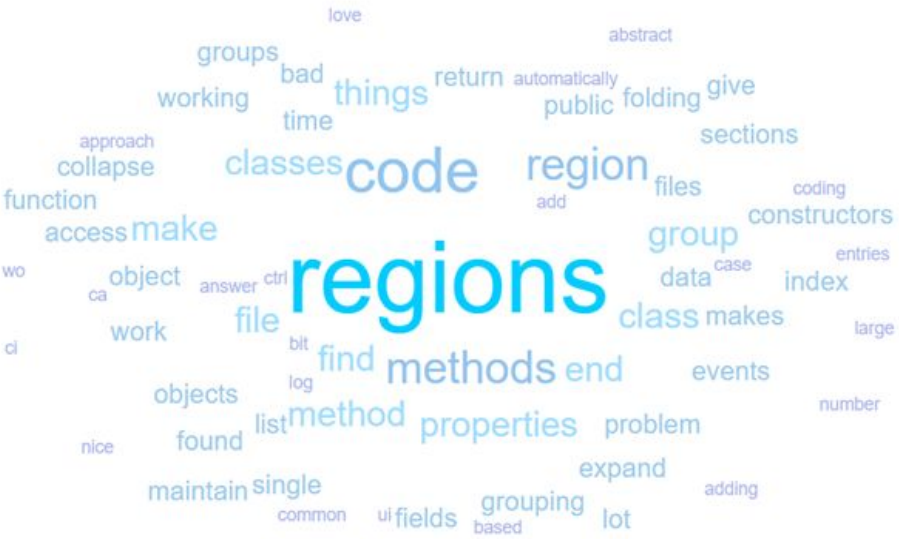


Figure 3: Using "regions" as the search term

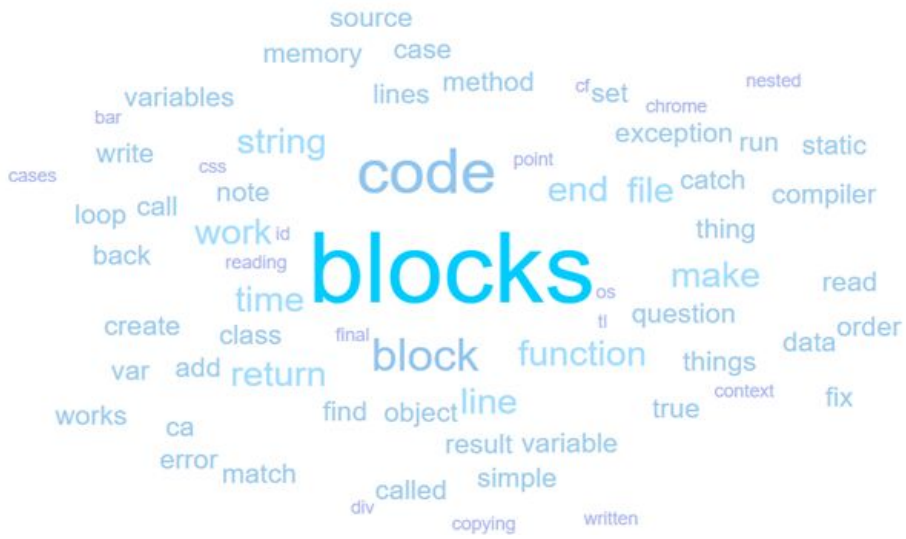


Figure 4: Using "blocks" as the search term

4.2. Assigning weights to words

Before assigning weights, the first thing we did was to remove the stop words from the initial weighted index table, because there were more than 800.000 rows originally and we realized that more than 300.000 of those were actually stop words. Therefore, removing stop words

saved us the time and effort to assign weights to words that didn't add much to the meaning of a post.

After removing the stop words from the table, we decided to let the weight values be from 0 to 1. Then we decided that words that were a part of the titles would be more important than words that were part of the bodies of the posts. Our reasoning for this was because first, titles describe the question in a few words that are the most important and relevant, so if a title contains a certain search term, it is very likely that we will want to display the post with that title as a result of the search query. Second, we would want to bring the user more questions as results rather than answers, so that the users can relate their problems with the ones that exist on the database as questions and they can learn valuable information through viewing the answers. For this reason, we gave words that are included in the post title 0.3 weight and the words that were included in the post body 0.1 weight (the numbers are arbitrary).

After that, we talked about the "part of speech" attribute in the words table. We discussed which words are more important when we are trying to convey our thoughts. We concluded that nouns are the most important, followed by verbs. Therefore, we multiplied the weights of the words that were nouns by 1.5 and verbs by 1.2 (the numbers are arbitrary).

Finally, we applied a variant of TF-IDF by calculating the ratio of the occurrence of a word divided by the word count in a post and finding out how many posts contain that word. We used the same formula in the slides. We decided to use TF-IDF because it seemed to help us get more relevant results.

An alternative to TF-IDF that we found was TF-PDF. TF-PDF is typically used for news or emerging topics. We decided not to use it because after discussing it among ourselves, we concluded that the relevance for our posts are not determined by how recent they are. There are also other alternatives such as vector space model and cosine similarity however for the ease of implementation reasons, we decided to go with the TF-IDF approach.

4.3. Stored Procedures

A stored procedure can be defined as a set of SQL statement that can be written, given a name, and stored directly in the database, which can be reused and shared by multiple functions and programs. It functions like methods in other programming languages and help to protect the database from sql injection attack. It also preserves data integrity and improves productivity as statements in a stored procedure only must be written once. SQL can manage stored procedures like CREATE, ALTER, or DROP.

4.3.1. Stored Procedure vs Stored Function

In stored procedure, we can use both SELECT and Data Manipulation Language (DML) statements but in the stored function only SELECT is allowed. The stored function is mostly

used to make quick tasks inside SQL statements where procedures are used for doing big tasks that is independent of any context .

4.3.2. Implementation of stored procedure

In our project, the implementation of stored procedure is used to log search results with the timestamp. This means, every time a search function is called, we perform a stored procedure to log the search results.

The stored procedure is defined as:

```
create or replace function log_search(string_to_log text)
    returns void as
$$
begin
    insert into history(search_term, search_date)
    select string_to_log, now();
end
$$ language plpgsql;
```

As it can be shared and used multiple times, all we need to do is *perform* this stored procedure when needed, and it inserts into the history table with its unique and auto-generated id, the search term and the timestamp of when the search was performed.

For example, we log the search history in section D.1 of the subproject, where define and call a *simple_search* function.

```
-- create a function to do the simple_search
create or replace function simple_search(search_string text)
    returns table (
                post_id int4,
                body    text
                ) as
$$
begin
    return query
        select s.id, s.body
        from submissions s,
             (select id from wi where word like '%' || search_string
|| '%') t
        where s.id = t.id;

-- call the previously created procedure
```

```
    perform log_search(search_string);
end ;
$$ language plpgsql;

-- use the simple_search fn to search for a term
select *
from simple_search('nuget');
```

In the code snippet above, the previously defined stored procedure is performed right after making the query. The result of that stored procedure is to insert the search term, ***nuget***, on to the table *history*, as shown in figure 5.

	id	search_term	search_date
1	1	nuget	2019-10-13 12:28:55.562345

Figure 5: Insert operation performed by stored procedure

A known task to do is to update the table *history* with a new attribute *user_id*, and populate it with the id of the user who is currently authenticated and performed the search.

5. Improving performance (Indexing)

Indexing in database is introduced to optimize searching and accessing the data. As the database grows, the performance is affected when there is no indexing. Therefore, index in data structure is helpful to reach or get the specific row in a large table in the least period of time.

In PostgreSQL, the primary key always has an associated index and B-tree indexes are created by default when using the **`CREATE INDEX`** command.

5.1. Implementation of indexing

To test the performance improvement introduced by indexing, we chose section D3 (exact match querying). That query compares on the *wi* table on the *word* column more than once. So we created an index on that table.

For testing purposes, we dropped the index and ran the query for comparison, as shown on figure 6. The total execution time is 901 ms.

```
SELECT * from exact_match('regions', 'blocks', 'constructors')
1 row retrieved starting from 1 in 901 ms (execution: 868 ms, fetching: 33 ms)
```

Figure 6: Query from D3 execution without indexing

And we repeated the query again, but after creating the index first as shown on figure xxx. The final execution time is 93 ms. That's 10x performance improvement with indexing.

```
create index wi_word_inx on wi (word)
completed in 5 s 39 ms
SELECT * from exact_match('regions', 'blocks', 'constructors')
1 row retrieved starting from 1 in 93 ms (execution: 76 ms, fetching: 17 ms)
```

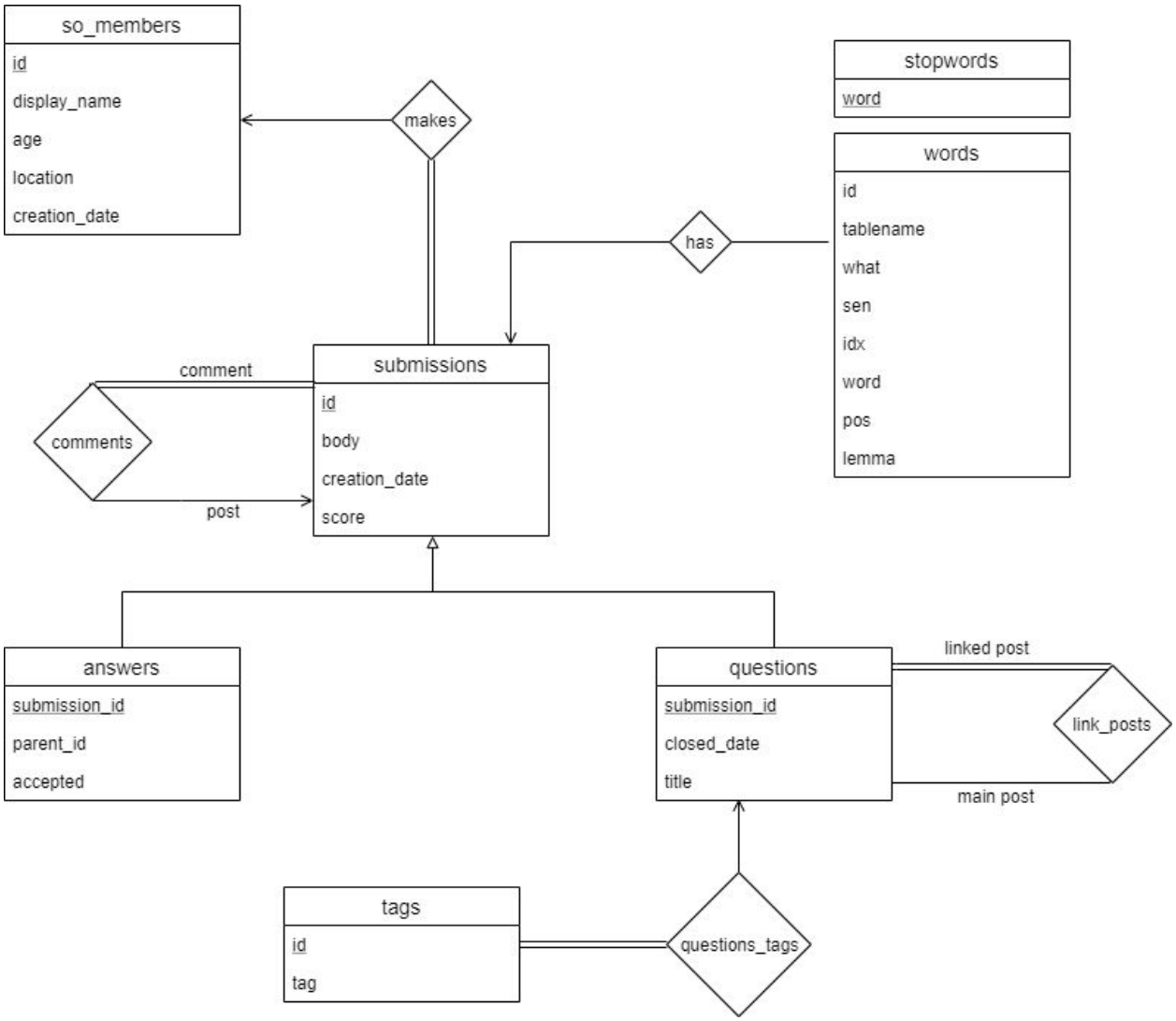
Figure 7: Query from D3 execution without indexing

There is an important line from the above output in figure 7 to notice. On line 2, we can see that creating the index took 5 seconds 39 ms. If the index was created for this specific use case, then it would have been counter-productive. But in our use case, the indexing benefits other queries as well and since it needs to be created once, it is a beneficial investment in the long run.

Note: These output were from the test run on our local machines. It may differ when run on ruc server.

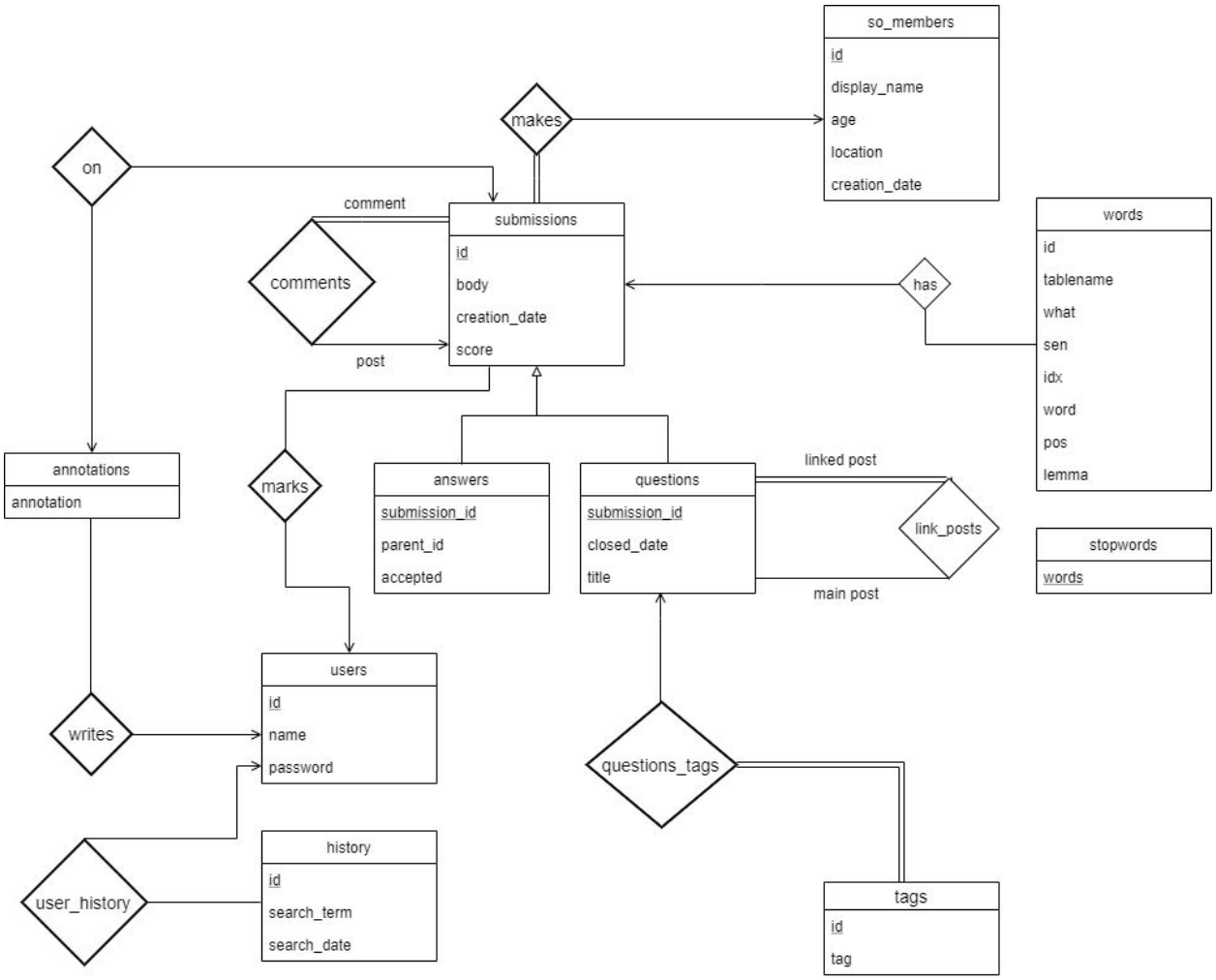
6. Appendices

1. QA data model - ER diagram



Also hosted in GitHub for an enlarged view:
https://github.com/ivanspajic/SOVA/blob/master/subproject1%20hand-ins%20and%20appendix/QA%20data%20model%20-%20ER%20diagram.png

2. QA + Framework data model - ER diagram



Also hosted on GitHub for an enlarged view:
https://github.com/ivanspajic/SOVA/blob/master/subproject1%20hand-ins%20and%20appendix/QA%20%2B%20Frameworks%20data%20model%20-%20ER%20diagram.png

3. Output of the test script

The output is provided in the zip file named as “test_output.txt”.
As an alternative, it is also hosted on GitHub. Link to GitHub:
https://github.com/ivanspajic/SOVA/blob/master/subproject1%20hand-ins%20and%20appendix/test_output.txt