RAWDATA Portfolio Subproject 1 Requirements

(Please refer to the note: *RAWDATA Project Portfolio* for a general introduction and to the note *RAWDATA Project Portfolio Data* for a description of the provided data)

Below the *Portfolio Subproject 1* is described and the requirements are given.

As mentioned in the note *RAWDATA Project Portfolio*, we aim to provide a tool to help computer programmers develop skills while they are working. The tool should support two complementary functions. The first is a keyword-based search that can be used to find answers to questions related to programming. The second is a search history as well as a marking and note-taking option to keep track of the most interesting answers already found.

The tool should develop into a responsive web application that draws on functionality made available through web services, which in turn draws on data from one or more databases and functionality embedded therein.

Goals and requirements for Portfolio subproject 1

The goal of this portfolio subproject 1 is to provide a database for the SOVA application (Stack Overflow Viewer Application) and to prepare the key functionality of the application. The database must be built based on two independent data models, a QA-data model for storage of Stackoverflow QA (Question and Answer) data and a Framework model for supporting the framework (users, markings, annotations, history). The two models should be combined into a single database when implemented. However, it is important that the database later can be separated, if need should arise to host these models on different servers.

A. Application design and adapted requirement list

Sketch a preliminary design of the application you intend to develop and the features that you aim to provide. Study the domain – content from Stack Overflow – and the possibilities for search and browsing provided by the Stack Overflow website. Based on this, develop your own ideas and describe these in brief. Develop and describe also a first sketch of how to provide access to the history, annotations and markings in your application. Give arguments for your design decisions and discuss and explain the implications on your data model. Since this is a preliminary design, it may obviously be subject to later changes – including later removal of items due to time limitations.

B. The QA-model

The data model for the QA-part must be designed so that all provided data can be represented. The provided data is described in the note *Portfolio Project Data* that also includes instructions on how to load the data into your own preliminary database.

You can claim that the two provided tables with post and comments already comprise a solution for the QA-part. But, if you want to achieve a good design that doesn't violate the most common conventions for good database design, a thorough redesign is needed. Some of the problems may be revealed by considering the following functional dependencies, that you can assume will hold in the two tables.

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*

An important step towards a better design is to take these dependencies into consideration while developing a new model. However, while developing your own model, you should consider and discuss whether there are other problems than those relating to the dependencies above.

B.1. Develop a data model to represent the provided QA-data from Stack Overflow. Start with the two tables with posts and comments, and proceed from there. Try to apply database design related methodology and theory learned from the database part of the RAWDATA course in the process. It is important that you document intermediate and final models, present alternatives and argue for your choices. Provide an ER diagram of your final design.

B.2. Implement the model developed in B.1 as a relational database in Postgres. Create firstly your new tables in the database you created for the source data. Then load data into your new tables from the two source tables and finally delete the two source tables. Collect everything in a script (using psql) that can be executed repeatedly on a database that includes only the two initial tables.

If your considerations from A calls for changes to the QA-part model, you can just include these changes in B.1 and B.2. Just remember to add a note about what the changes are.

C. Framework model

In essence the framework model should support the following: registration of users of your application and (for users individually) storage of search history, marking of interesting comments / posts retrieved in search results as well as addition of personal notes to posts retrieved in search results.

C.1. Develop a data model that is appropriate for the purpose of the framework.

C.2. Combine the model developed in C.1 with the result of B.1 in such a way that you make as few changes to the two models as possible. Describe the result (also with diagram(s)).

C.3. Implement the combined model. Modify the result of B.2 by adding a new part that corresponds to C.1. Extend the script from B such that it also generates the tables designed in C.1.

D. Functionality

An important part of this subproject is, in addition to the modeling and implementation of the database, to develop key functionality that can be exposed by the data layer and applied by the service layer. The goal here is to provide this functionality as, what can be considered, an Application Programming Interface (API) comprising a set of functions and procedures developed in PostgreSQL.

First of all, what is needed is functionality to support search in the stackoverflow data. To start, develop a simple approach as follows.

D.1. **Simple search & framework**: Develop a simple search function that, given a search sting S, will return all posts where S is a substring of the title or the body. Make sure to bring the framework into play, that is, storage of search history, and the possibility adding markings and annotations to posts, individually for users.

The following elaborates on search going beyond simple string matching. Modify the database and develop a set of functions and procedures that meet the requirements. Discuss alternatives, give arguments for your choices and, to the extend this appears to be relevant, refer and describe relevant methodology and theory. Make sure, again, to **bring the framework into play where relevant**.

D.2. **Boolean indexing:** Build and a Boolean (= unweighted) inverted index to be applied in you search functions. You can take advantage of the provided table **words** and simplify to build your own.

D.3. **Exact-match querying:** Introduce an exact-match querying function that takes one or more keywords as arguments and returns posts that match all of these. Use the inverted index from D.2 for this purpose.

D.4. **Best-match querying:** Develop a refined function similar to D.3, but now with a "best-match" ranking and ordering of objects in the answer. A best-match ranking simply means: the more keywords that match, the higher the rank. Posts in the answer should be ordered by decreasing rank.

D.5. **Weighted indexing:** Build a new inverted index for weighted indexing similar to your index from D.2, but now with added weights. A weight for an entry in the index should indicate the relevance of the post to the word. Discuss and decide on a weighting strategy. A good choice would probably be a variant of TFIDF, but apart from the details of this you should also consider other related issues for instance what exactly to index and whether or not to take stopwords into consideration. Be aware that some of the entries in **words** stem from an imperfect tokenization (the splitting of text into words that was done to build the **words** table), so filtering to omit some of the "noise" could also be considered.

D.6. **Ranked weighted querying:** Develop a refined function similar to D.4, but now with a ranking based on a relevance weighting (TFIDF or similar) provided by the indexing derived in D.3.

D.7. **Word-to-words querying:** An alternative, to providing search results as ranked lists of posts, is to provide answers in the form of ranked lists of words. These would then be weighted keyword lists with weights indicating relevance to the query. Develop functionality to provide such lists as answer. One option to do this is the following: 1) Evaluate the keyword query and derive the set of all matching posts, 2) count word frequencies over all matching posts, 3) provide the most frequent words (in decreasing order) as an answer (the frequency is thus the weight here). The Boolean as well as the weighted indexing can be applied for this purpose. Compare and try to clarify what's better.

D.8. **Word cloud visualization:** Provide one or a few examples of visualizations of weighted keyword list answers (results of D.7). At this point (this subproject) no complex coding is needed. The purpose is to provide an illustration to be included in your report and you can do that by copying a weighted keyword list answer into a word cloud visualization tool[1]. Notice that this is intended to be a subject for further elaboration in Portfolio subproject 3.

Consider, *if time allows*, one or two of the following issues.

D.9. [OPTIONAL] Introduce a similarity search, such that evaluation of keyword-match is independent of the word form (ending) used in the text in the posts.

D.10. [OPTIONAL] Consider and suggest an approach to relevance feedback that allows the user to select most interesting documents in the first result set and the "system" to generate a second result set taking the users preference into consideration.

D.11. [OPTIONAL] Consider and suggest an approach to query expansion, where the query is modified before the answer is derived. The general principle is to add words to the query that are similar to words already there – e.g. by being synonyms, by being frequently co-occurring or by being inflections of the same word.

D.12. [OPTIONAL] If you have an idea of your own, you can plug it in here …

E. Improving performance by indexing

One of the advantages of using a relational database in the data layer is that query performance can be improved and tailored to the actual needs (most frequent and/or most important queries/query types) at any state of the development process (even after the development has finished). Without need to reconsider other parts of the system, performance can be improved by adding or modifying the **indexing** of tables in the database.

E.1. Consider the extensions developed under **D** and discuss/explain what may potentially provide significant performance improvements. Describe how your database is indexed (observe: this concerns database indexing, not textual inverted indexing – even though the latter may build on the former).

[1] One option is to use the word cloud visualization tool made available on Moodle.

F. Testing

Demonstrate by examples that the results of **D** work as intended. Write a **single** script that activates all the written functions/procedures and, for those that modify data, add selections to show before and after for the modifications. In this subproject you need only to proof by examples that you code is runnable. A more elaborate approach to testing is an issue in Portfolio Project 2.

Include, as an appendix to your report, the output from running your testing script. See descriptions in assignment 1 and 2 on how to do this.

What to hand in

You are supposed to work in groups and each group (one member of each group) should hand in the following **on Moodle with deadline 14/10-2019**:

- A project report in size around 6-12 normal-pages[2] excluding appendices.
- A (data) script file for complete generation of your database, that is, the script file edited in C.3 with the additions you made in D.2 and D.5. You may assume that generation starts from an initial database as described in the note *RAWDATA Portfolio Data*.
- A (code) script file for complete generation of the API to your database, that is, the functionality you developed in D.
- A (test) script file testing your API, that is, the testing script file edited in E.

In addition, each group should make their product available **on rawdata.ruc.dk**. Thus (also with deadline 14/10-2019), make sure to:

- Re-implement your complete database as your group database[3] on the course database server on rawdata.ruc.dk

Notice that the report you hand in for Portfolio project 1 is not supposed to be revised later. However, your design and implementation can be subject to revision later. Documentation for later changes can be included in later Portfolio project reports.

[2] A normal-page corresponds to 2400 characters (including spaces). Images and figures are not counted.
[3] The group databases on rawdata.ruc.dk are raw1 for group 1, raw2 for group 2 etc.