RAWDATA Portfolio Subproject 1 Requirements

(Please refer to the note: *RAWDATA Project Portfolio* for a general introduction)

Below the *Portfolio Subproject 1* is described and the requirements are given. To set the context we describe first the overall goals of the Portfolio project as a whole. The specific goals and requirements for this Portfolio subproject 1 follows and finally instructions on a what to do and how to hand in are given. Two appendixes are included describing the provided data and commenting on functionality in the database, respectively.

Overall goals and direction for the RAWDATA Portfolio Project

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 – a keyword-based search and a search history that keeps track of what's already retrieved and what parts were the most interesting.

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 possibly other resources.

As a minimum the application should support the following.

RAWDATA Portfolio project - General requirements

1) Search for posts and comments in Stack Overflow
2) Present search results by lists of items that link to (or can be unfold for) details
3) Keep track of search history
4) Provide a marking option for posts of special interest among posts presented in the search result and allow optional annotation to marked posts
5) Support multiple users such that each user only has access to own history, markings and annotation
6) Deal with relevance of objects to queries and use this to provide better answers
7) Present search results with focus on words frequencies rather than on the texts they occur in, such as ranked lists of most frequent words
8) Provide visualizations of search results and subsets of the database by means of word clouds and word graphs showing significant words and their relations

As it appears these items are open for your own interpretation and all can be dealt with in several ways and calls for further elaboration. You are also welcome to add additional items of your own preference to the list. Some examples for inspiration are:

Examples of additional features, for inspiration

- Provide statistics and visualize frequent Stack Overflow topics
- Similar words search
- Phrase search
- Browse topics of interest
- Build and visualize networks of associated words and/or topics
- Provide visualizations of marked/annotated posts and history, to provide surveys to users
- …

However, the goal here is not to design and implement as many features as possible. And **you are NOT required to add anything** to the general requirements. In the end we'll evaluate quality, not quantity ☺.

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 and a Framework model, for storage of Stackoverflow QA (Question and Answer) data and for supporting the framework (users, markings, annotations, history) respectively. 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. The application design will of course also be subject for development in the following portfolio subprojects, but it is important to take functionality and features of the application into account from the beginning, when considering what data is needed. 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. Adapt a preliminary list of requirements that include the General requirements listed above (maybe slightly elaborated) as well as what you have decided to add. Since this is a preliminary design, it may obviously be subject to later changes – including later removal of some of the items from the list 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. In this subproject we will only use a small collection of sample data extracted from Stack Overflow including close to 14000 posts (out of more than around 15.5 million). Later we will provide a larger sample of data. The Stack Overflow data sample is described in appendix 1.

The data is provided as a script, that can be loaded into a database in your local Postgres server as a relational database with two tables. To do this follow the instructions in the first paragraph of appendix 1. When your version of the database is created, study the content and compare with the description in appendix 1. You can claim that the two-table data model and relational database 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.

B.1. Develop a data model to represent the provided QA-data from Stack Overflow. Use the two-table data model as a starting point, 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. Describe also central concepts and key aspects of the theory used.

B.2. Implement the model in B.1 as a relational database in Postgres. Create firstly your new tables in the same database as you imported the two source tables into. Then load data into your new tables from the two source tables and finally delete the two source tables. Collect everything in a script that can be executed repeatedly.

B.3. If further modifications/extensions to the QA-model are needed to meet your requirements listed under A, describe these, discuss their implications, present the result of your changes to the model and implement these in your database. Modify the script from B.2 so that it generates your final database.

B.4. Finally provide a description of the extended QA-model and support this description by a visualization using appropriate diagrams.

C. Framework model

In essence the framework model should support the following: registration of users of your application, storage of search history individually for users, marking of interesting comments/posts and optional 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. Proceed as in B (except for the data loading).
C.2. Combine the model developed in C.1 with the result of B.1 in such way that you make as few changes to the two models as possible. Describe the combined result.
C.3. Implement the combined model. Modify the result of B.2 (or B.3, if this extends B.2) basically by adding a new part that corresponds to the result of C.1. Extend the script from B such that it also generates tables resulting from your design in C.1.
C.4. Finally provide a description of the combined model and support this description by a visualization using appropriate diagrams.

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.

D.1. Again starting from your description and requirements in A, proceed with a design of functionality. Develop a set of functions, procedures and, if needed, triggers that meet the requirements and cover the needs for access to data in the database. Discuss alternatives, give arguments for your choices and, to the degree this appears to be relevant, refer and describe relevant methodology and theory. Make sure that you, as a minimum, cover functionality for search, search history, marking, annotation and management of multiple user.
D.2. Describe your API, that is, the result of D.1 in detail, specifying purpose, input, output, effect, side effects, etc.
D.3. Implement what's described in D.2 by writing functions and procedures in Postgres. Collect everything in a single SQL script that generates you full API.

E. 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.

How 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 3/10-2018**:

- A project report in size around 6-10 normal-pages[1] excluding appendices.
- An SQL script file for complete generation of your database, that is, the script file edited in C.3. You may assume that generation starts from a database with the two source tables.

[1] A normal-page corresponds to 2400 characters (including spaces). Images and figures are not counted.

- An SQL script file for complete generation of the API to your database, that is, the script file edited in D.3.
- An SQL 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, **with deadline 3/10-2018**, make sure to:

- Re-implement your complete database as your group database[2] 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, if you find good reasons for this, your design and implementation can be subject to revision later. Documentation for later changes can be included in later Portfolio project reports.

Appendix 1 – Stack Overflow data to be used as source

A small collection of sample data extracted from Stack Overflow including close to 14000 posts (out of more than 15 million) and 33000 comments is provided in a two-table relational database for this subproject. The data model for this database is sketched in figure 1. The database is provided with the file **stackoverflow_universal.backup** (can be downloaded from Moodle). Create a database, e.g. "stackoverflow", and run the script to import the two relations. Using psql this would be:

```
psql -U postgres -c "create database stackoverflow"
psql -U postgres -d stackoverflow -f stackoverflow_universal.backup
```

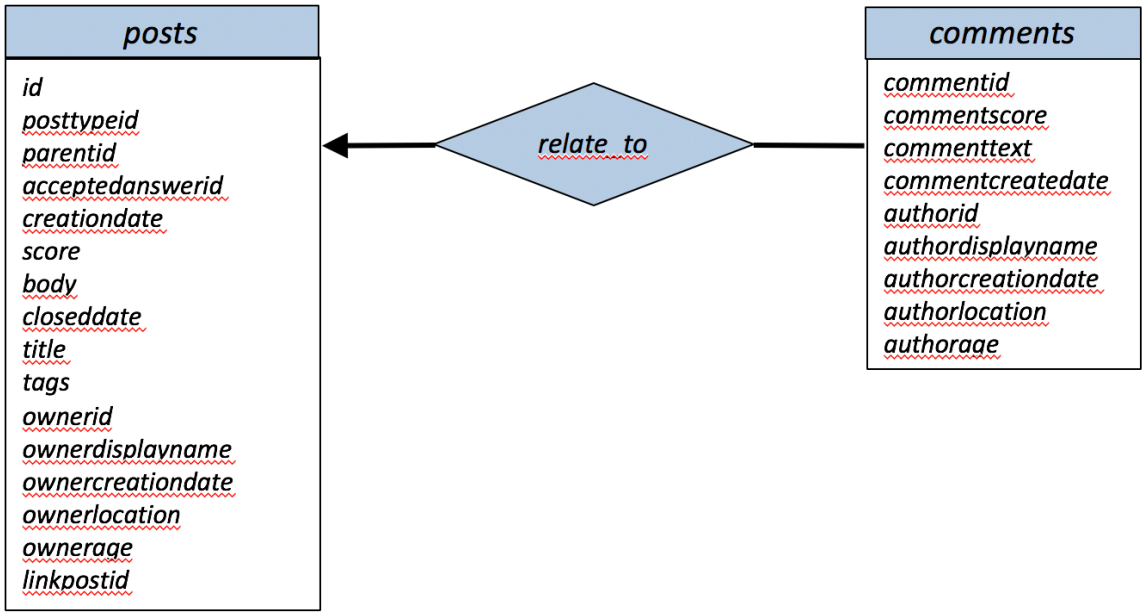

Figure 1: The preliminary data model

The attribute *posttypeid* can take two values to be read as follows:

posttypeid

1: Question
2: Answer

[2] The group databases on rawdata.ruc.dk are raw1 for group 1, raw2 for group 2 etc.

A question post will not have a parent (thus *parentid* is null), while an answer post always refer to a parent post via *parentid*, which thus is a reference to the question post that the answer post refers to.

Designing your own database

As indicated above, the two-table model cannot be considered good design. In predicate notation the schema is the following:

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);

Some of the problems may be revealed by considering the following functional dependencies, that you can assume will hold in the two source 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.

Obviously, since our main purpose is to search for answers, the text attributes are the most important. But info about users, links between posts, scores, etc. may potentially also be applied. So, even though you may only use a smaller part of the model, try to represent the full content in your design.

Appendix 2: Why functions and procedures in the data layer?

Whenever you design multi-layered systems you need to decide where to put the logic. Should it be in this layer or that layer? Or should it be divided between the layers? Obviously, there is no general answer to such questions, and it will often be a question of preferences in the development team combined with concrete requirements to the system under development. Among the important considerations to include in the decision about the placement of business logic, is not only the functionality of the system, but also usage, deployment, and future maintainability.

It is of course an option to access the data layer by means of queries expressed directly in SQL, and you may, for some needs, do just that. However, to break down the functional requirements into pieces corresponding to needs for access to the database may in itself be an important step in the design process and, in addition, encapsulating multiple actions in one function or procedure may contribute to data consistency as well as relieve the programmer from tedious details when accessing the database. One example is the need to track history. When a keyword query is given, the main task in the data layer is to collect the answer – a set of posts with certain attributes – but to keep track of the history, we also need to store the query itself with time stamps, user-id, etc. in the database. If queries are processed through a function in the database, this function can, apart from returning the answer, simply take care of storing the history data in the database. Obviously, this approach will also imply that the application code becomes less dependent on the database

structure. When the schema is modified or even the database completely replaced, only the API needs to be modified. The applications accessing the database will work as soon as this has happened.