RAWDATA Portfolio Subproject 1 Requirements

(Please refer to the note: *RAWDATA F2018 Project Portfolio* for a general introduction)

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 history and marking model – such that the result combines the two models. However, it is important that the database later can be separated, if need should arise to host these models on different servers. The functionality to be exposed to the service layer should include support for search for posts and personal notes as well as updates in connection with note-taking and activity history.

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 is more of the key issue in portfolio subproject 2 and 3, 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 the history and marking part. 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 "required requirements" from the note *RAWDATA Portfolio project* as well as what you have decided to add.

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 14.5 million). Later we will provide a larger sample of data. The Stack Overflow data sample is described below in appendix 1.
The data is provided as a SQL script, that can be loaded into your MySQL server as a relational database with two tables. Study the content of the data 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 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 developed above as a relational database in MySQL. To add content to your new database, develop and execute code to load data from the provided two-table relational database.

B.3. If further modifications/extensions to the QA-model are needed to meet your requirements listed under A[1], describe these, discuss their implications, present the result of your changes to the model and implement these in your database.

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

[1] It is not required that such need for extension should arise from A.

C. History and marking model

In essence the history and marking functionality of your application implies that the system needs to keep track of previous searches (search history) and supports marking of, as well as optional addition of personal notes to, posts retrieved in search results.

C.1. Develop a data model to support the history and marking functionality. 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.
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 MySQL.

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.
D.2. Describe the result of D.1 in detail, specifying purpose, input and output, side effects, etc.
D.3. Implement what's described in D.2 by writing functions and procedures in MySQL.

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 the modifications. Include, as an appendix to your report, your script as well as the result of running it (use "Execute All (or Selection) to Text" from the Query menu in MySQL Workbench or run it from a command line interface). You need only to proof by examples that you code is runnable here. A more elaborate approach to testing is an issue in Portfolio Project 2 (and section 2 of the RAWDATA course).

The project report

You are supposed to work in groups and each group must submit their final Portfolio project 1 report by upload to the course website on Moodle, while the product, including the database and the implemented functionality, should be uploaded to your database[2] on the course server on wt-220.ruc.dk.

The report should in size be around 8 normal-pages[3] excluding appendices. The submission deadline for the report as well as the product is 5/3-2018.

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 changes can be included in Portfolio project report 2 or 3.

[2] The group databases on wt-220.ruc.dk are raw1 for group 1, raw2 for group 2 etc.
[3] A normal-page corresponds to 2400 characters (including spaces). Images and figures are not counted.

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, return to do just that later on. 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.

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 11 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 can be loaded into a MySQL-server (for instance your own local server) by running the SQL script "stackoverflow_sample_universal.sql" provided on Moodle.

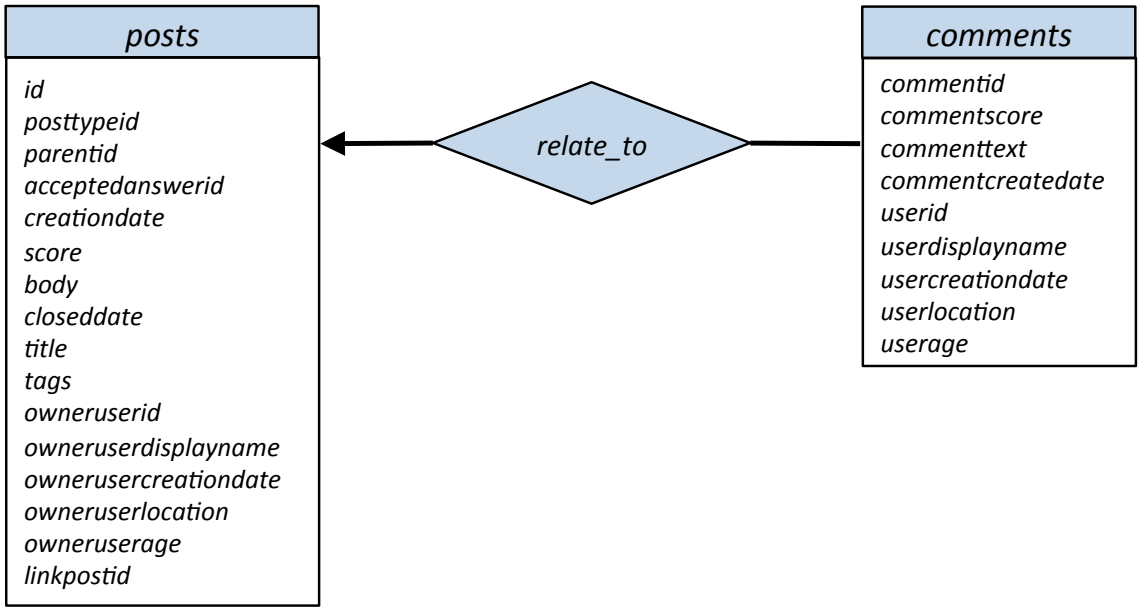

Figure 1: The preliminary data model

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

posttypeid
1: Question
2: Answer

A question post will 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(commentid, postid, commentscore, commenttext, commentcreatedate, userid, userdisplayname, usercreationdate, userlocation, userage)

posts(id, posttypeid, parentid, acceptedanswerid, creationdate, score, body, closeddate, title, tags, owneruserid, owneruserdisplayname, ownerusercreationdate, owneruserlocation, owneruserage, 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*-relation**

- *commentid -> postid, commentscore, commenttext, commentcreatedate, userid*
- *userid -> userdisplayname, usercreationdate, userlocation, userage*

***posts*-relation**

- *id -> posttypeid, parentid, acceptedanswerid, creationdate, score, body, closeddate, title, tags, owneruserid*
- *owneruserid -> owneruserdisplayname, ownerusercreationdate, owneruserlocation, owneruserage*

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.