CITP – Portfolio Subproject 1 Requirements

(Check also the notes: *CITP Introduction to Project Portfolio* for an overall introduction and *CITP Project Portfolio Source 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 *CITP Introduction to Project Portfolio*, we aim to build a tool to search for, browse, rate and compare movies and actors. The tool should be developed as a multi-user web-application and, apart from search/browse functionality, it should also keep track of search history and support various database updates including rating and bookmarking of interesting movies.

The tool should develop into one or more web applications that draws on functionality made available through web services, which in turn draws on data from one or more databases.

Goals and requirements for Portfolio subproject 1

The goal for this portfolio subproject 1 is to provide a database for the Movie application and to prepare key functionality of the application. The database must be built based on two partly independent data models, a Movie Data Model for storage of movie data collected from external sources and a Framework Model for supporting the framework (users, bookmarkings, local ratings, 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 separate servers.

A. Application design

Sketch a preliminary design of the application you intend to develop and the features that you aim to provide. Study the domain by considering the provided data and by browsing movie apps on the web. Based on this, develop your own ideas and describe these in brief. Develop and describe also a first sketch of how to show the history, ratings and bookmarkings in your application. Give arguments for your design decisions and discuss and describe the implications on your data model. Since this is a preliminary design, it may obviously be subject to later changes – including later simplifications due to time limitations.

B. The Movie Data Model

The data model for the movie data part must be designed so that the provided data can be represented. The provided data is described in the note *CITP – Project Portfolio Source data* that also explains how to get access to the data and load the data into a database in your own database server.

You can claim that the provided tables already comprise a movie data model as requested, but if you want to achieve a good design that doesn't violate common conventions concerning relational database design, a redesign is needed.

B.1. Develop a data model to represent the provided data. Try to develop the model independently of the structure in the source data, but make sure that all provided data can be represented. Apply database design related methodology and theory learned from the database part of the CITT 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 **using the notation** described **in the DB-book**. If you prefer another notation, you can present your data model in that notation as well.

B.2. Implement the model developed in B.1 as a relational database in PostgreSQL. Create firstly a database and import all the source data into that database as described in the note *CITP – Project Portfolio Source data*. Secondly, create the new tables corresponding to your model from B.1., and distribute the data from the source data tables into your new tables. Finally, delete the source data tables. Collect all your commands for creating tables, distributing data and deleting source data in a single **SQL script** called for instance **B2_build_movie_db.sql**. Make this script such that it can be modified and executed repeatedly with psql until you are

satisfied with the result and have tested that all data is in the right place. To make a script so that it can be executed repeatedly, simply include "wipe-out" of all the old stuff – either by **drop**ping things to start with or using **create or replace** … rather than just **create**. The major part of your SQL script will be create table and insert statements (maybe like this: insert into xxx select … from …). However, you may also need to include do-blocks in your SQL script, for instance if you need to iterate through a source table and split data. When you are done, generate a **reverse engineered ER-diagram**[1] and include this as well as your **SQL script** in your documentation.

If your considerations from A calls for changes to the Movie-data 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, storage of rating history, bookmarking of titles and names (personalities). Feel free to extend the Framework with your own additions (such as support addition of personal notes to titles).

C.1. Develop a data model that is appropriate for the purpose of the framework. Provide (and include in your report) an ER diagram of your framework model. Indicate how your Framework Model connects to the Movie Data Model by including the entities from your Movie Data Model in B.1 that are directly connected with relationships from the Framework Model. It must be clear how the two models combine. Optionally, provide a complete ER-diagram with all entities and relationships from both models. Use the **ER notation** described **in the DB-book**. If you prefer another notation, you can present your data model in that notation as well.

C.2. Implement the Framework Model. Write a SQL script that adds the Framework Model to an already created Movie Data Model database (result from B.2). Name this script for instance **C2_build_framework_db.sql**. Since there is no data for the Framework part yet, the script will be mostly Create Table statements. Generate a **reverse engineered ER-diagram**[1] of your full (Movie Data + Framework) database and include this as well as your **SQL script** in your documentation.

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 movie data. To start, develop a simple approach as follows.

D.1. **Basic framework functionality**: Consider what is needed to support the framework and develop functions for that. You will need functions for managing users and for bookmarking names and titles. You could also consider developing functions for adding notes to titles and names and for retrieving bookmarks as well as search history and rating history for users.

D.2. **Simple search**: Develop a simple search function for instance called **string_search()**. This function should, given a search sting S as parameter, find all movies where S is a substring of the title or a substring of the plot description. For the movies found return id and title (tconst and primarytitle, if you kept the attribute names from the provided dataset). Make

[1] Open your database in Navicat and right click on the schema (probably called "public"). Then select "Reverse Schema to Model ..." and click the Diagram tab (or double click the diagram line). You can edit the diagram by moving boxes and connectors, if you would like to change the layout. When you are done, you can save it or just copy the graphics by taking a screenshot and paste this into your report.

sure to bring the framework into play, such that the **search history** is updated as a side effect of the call of the search function.

D.3. **Title rating**: Introduce functionality for rating by a function, called for instance rate(), that takes a title and a rate as an integer value between 1 and 10, where 10 is best (see more detailed interpretation at What are IMDb ratings?). The function should update the (average-)rating appropriately taking the new vote into consideration. Make sure to bring the framework into play, such that the **rating history** is updated as a side effect of the call of the rate function. Also make sure to treat multiple calls of rate() by the same user consistently. This could be for instance by ignoring or blocking an attempt to rate, in case a rating of the same movie by the same user is already registered. Alternatively, an update with the new rate can be preceded by a "redrawing" of the previous rating, recalculating the average rating appropriately.

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

D.4. **Structured string search:** Develop a search function, for instance called **structured_string_search()**, that take 4 string parameters and return titles that match these on the title, the plot, the characters and the person names involved respectively. Make the function flexible in the sense that it don't care about case of letters and argument values are treated as substrings (to match they should just be included in the column value in question). For the movies found, return id and title (in the source data called tconst and primarytitle). Make sure to bring the framework into play, such that the search history is stored as a side effect of the call of the search function.

D.5. **Finding names:** The above search functions are focused on finding titles. Try to add to these by developing one or two functions aimed at finding names (of for instance actors).

D.6. **Finding co-players:** Make a function that, given the name of an actor, will return a list of actors that are the most frequent co-players to the given actor. For the actors found, return their nconst, primaryname and the frequency (number of titles in which they have co-played).
Hint: You may for this as well as for other purposes find a view helpful to make query expressions easier (to express and to read). An example of such a view could be one that collects the most important columns from title, principals and name in a single virtual table.

D.7. **Name rating**: Derive a rating of names (just actors or all names, as you prefer) based on ratings of the titles they are related to. Modify the database to store also these name ratings. Make sure to give higher influence to titles with more votes in the calculation. You can do this by calculating a weighted average of the averagerating for the titles, where the numvotes is used as weight.

D.8. **Popular actors:** Suggest and implement a function that makes use of the name ratings. One example could be a function that takes a movie and lists the actors of the movie in order of decreasing popularity. Another could be a similar function that takes an actor and lists the co-players in order of decreasing popularity.

D.9. **Similar movies:** Discuss and suggest a notion of similarity among movies. Design and implement a function that, given a movie as input, will provide a list of other movies that are similar.

D.10. **Frequent person words:** The **wi** table provides an inverted index for titles using the 4 columns: primarytitle, plot and, from persons involved in the title, characters and primaryname. So, given a title, we can from **wi** get a lot of words, that are somehow characteristic for the title. To retrieve a list of words that are characteristic for a person we can do the following: find the titles the person has been involved in, and find all words associated with these titles (using **wi**). To get a list of unique words, you can just group by

word in an aggregation by count(). Thereby you'll get a list of words together with their frequencies in all titles the person has been involved in. Use this principle in a function **person_words()** that takes a person name as parameter and returns a list of words in decreasing frequency order, limited to fixed length (e.g. 10). Optionally, add a parameter to the function to set a maximum for the length of the list. You can consider the frequency to be a weight, where higher weight means more importance in the characteristics of the person.

D.11. **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 **wi** for this purpose. You can find inspiration on how to do that in the slides on Textual Data and IR.

D.12. **Best-match querying:** Develop a refined function similar to D.11, 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. Titles in the answer should be ordered by decreasing rank. See also the Textual Data and IR slides for hints.

D.13. **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 titles, 2) count word frequencies over all matching titles (for all matching titles collect the words they are indexed by in the inverted index **wi**), 3) provide the most frequent words (in decreasing order) as an answer (the frequency is thus the weight here).

Consider, *if time allows*, the following issues.

D.14. **Weighted indexing** [OPTIONAL]**:** Build a new inverted index for weighted indexing similar to the **wi** index, but now with added weights. A weight for an entry in the index should indicate the relevance of the title to the word. As weighting strategy, a good choice would probably be a variant of TFIDF.
Ranked weighted querying: Develop a refined function similar to D.12, but now with a ranking based on a relevance weighting (TFIDF or similar) provided by the weighted indexing.

Finally, feel free to elaborate.

D.15. **Own ideas** [OPTIONAL] If you have some ideas of your own, you can plug them 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 **database 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).

F. Testing using the imdb database

Demonstrate by examples that the results of **D** work as intended. Write a **single** SQL 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. Generate an output file from running your test script file. (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 7/10-2024**:

- A project report in size around 6-12 normal-pages[2] excluding appendices.
- The two (data) script files from B.2 and C.2 for complete generation of your database. You may assume that generation starts from an initial database loaded with the content of **imdb.backup**, **omdb_data.backup** and **wi.backup**.
- One or more (code) script files that generate your result from D (functions and procedures that comprise your API to your database as well as generates additional data, if relevant).
- Your testing script and test output files for the testing requested in F.

In addition, each group should make their product available on **cit.ruc.dk**. Thus (also with deadline 7/10-2024), make sure to:

- Re-implement your complete database as your group database[3] on the course database server on cit.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 cit.ruc.dk are cit01 for group 1, cit02 for group 2 etc.