TU Kaiserslautern
FB Informatik, AG HIS
Prof. Dr.-Ing. Stefan Deßloch
M.Sc. Michael Hohenstein

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Worksheet 1 - DB Schema Design and Programming

"SQL Programming and Object-Relational Schemas"

Introduction

Goal. In this worksheet you will get familiar with the system environment, especially with the database management system DB2 Universal Database (short DB2). You will refresh your SQL knowledge and learn how to use object relational features, how to add custom functionality to the database, and how to connect to the database with Java.

Email Database

This worksheet is based on a database for emails, attachments, addresses, and users. Figure 1 shows the corresponding UML diagram of our database.

Every group works on its own database "dbgrpXY" (XY is referred to your group number, e.g. dbgrp01). Hints for accessing your database are given in the Wiki. Note that the evaluation in PRISE will not access your own group database, but rather an internal copy.

Under `/dbprak/share/ws1` on Lara or from our Wiki, you will also find the DDL statements to load the sample data in your database. Once you created the schema, make sure the sample data is loaded before starting the development. Also note that the sample data will be used to test your submitted queries.

General Rules

- If not explicitly requested or permitted in the exercise, you are not allowed to create new DB objects. This includes also the usage of temporary tables. Furthermore, any modification of the data in the email database is prohibited.
- You have to ensure that your queries return only the requested columns, in the order specified on each task, i.e., use the proper ORDER BY clause. Further, queries should be formulated in such a way that data pool changes or schema extensions (i.e. new subtypes or subtables) do not affect their correctness.
- Take care of writing clear and readable code (both SQL and Java). Use proper indentation, consistent formatting, and meaningful names. Comment your code. Refactor your code whenever necessary to keep it readable and avoid needless casts (in both SQL and Java); e.g., for IDs utilize the proper reference type.
- Remember that the cleanliness of your code and queries is part of the evaluation. Solutions without any comments or with a poor visual separation of logical blocks (e.g., through indentation) will have points deducted. The higher-level design of the algorithms (i.e., "elegance" of the solutions) is also considered in this process.

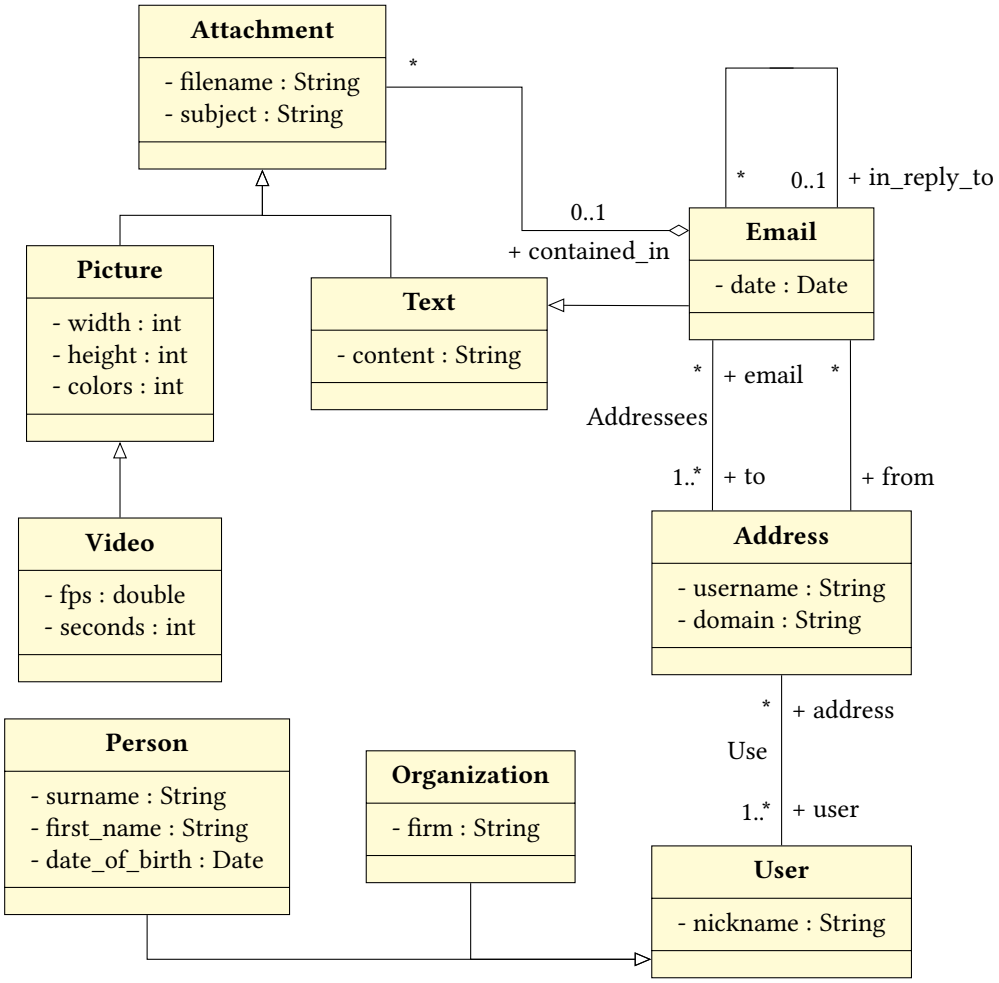

Figure 1: Email Database

The Schema Explained

- This schema is based on *object-relational SQL*, as you can see in the inheritance relationships, e.g., between User and Person. Please have a look for the references in the Wiki if you are not familiar with it. Most importantly, make sure you understand the differences between structured types and typed tables. Some tasks can only be solved with object-relational operations, and thus a basic understanding is crucial. We also **strongly recommend** you to study the file `schema.sql`, which contains the CREATE statements for the given schema.
- Relationships are expressed in terms of *reference types*, which are the object-relational equivalent of foreign keys.
- Users of the system can be either persons or organizations, but they all share the *nickname* attribute of the parent User type. Note that SQL allows a user to also exist simply as an instance of the User type, i.e., neither a person nor an organization.
- Users have a **n:m** relationship to email addresses, i.e., a user may have multiple addresses and an address can be shared by multiple users. This is reflected in the *Use* table.
- An email is sent *from* a specific address, (optionally) *in reply to* a specific email (1:1 relationships) and *to* multiple addresses (n:m relationship). The multiple destination addresses of an email are stored in the *Addressee* table.

- An email may contain multiple attachments. This 1:n relationship is reflected in the *contained_in* attribute of attachments.
- Attachments can be either a *Picture* or a *Text*, and from there further specialize into Video and Email, respectively. Note that making Email a subtype of Attachment allows us to include whole Email messages as attachments of another email. As you will see, this will also give rise to interesting recursive queries in the exercises.

Tips for Solving the Exercises

- Load the data on your own database and develop solutions locally before submitting it to PRISE. We provide several alternatives to work with the database in the Wiki.
- A good SQL programmer naturally thinks in terms of joins. Whenever you need to produce results that potentially come from multiple tables (or even cross-references within the same table), it is most likely that you will need a join. Keep that in mind when you get stuck.
- Some exercises require recursive SQL queries. Check the references in the Wiki if you are not familiar with this concept.
- Break complex queries into meaningful and self-contained sub-queries. For exercise **??**, for example, first think about how to generate a ranking table from a generic table of names and values, and then build on that to solve further requirements. In exercises that require recursion, first think about how to build the basic recursive set, and only then introduce further operations like filters or aggregations. The `WITH` construct of SQL is very useful for this purpose.

Exercises

Exercise 1: SQL

Use SQL queries to solve the following seven problems based on our email database. Only a single query per problem is allowed. Of course, you are allowed to use subqueries. You are free to resolve relationships either with the dereference operator (`->`) or in the classical way, using joins.

1.1. Multiple addresses
Which users have more than 4 addresses? The result should contain the nickname in the first column, and the amount of addresses in the second (which should be greater than 4). It must be *ordered* by the amount column as first criteria and the nickname as second, both in ascending order. (Tip: to practice your object-relational skills, try to answer this query without using joins)

1.2. Total numbers
How many users of each class are registered in the system? The result should contain four rows and two columns. The first column contains the keywords 'users', 'persons', 'organizations', and 'total'. The second column contains the corresponding amount, and the result should be *ordered* by it in ascending order. For the 'users' row, you should count only the pure instances of the User type (i.e., that are neither person nor organization), and for the 'total' row the total count of User instances. Your solution must be general in the sense that adding a new type and table to the hierarchy does not falsify the query results. For example, if we were to add a new type *admin* as a subtype of *user*, the column 'users' must remain unchanged, while 'total' may include the new table. Note that the total should be equal to the sum of the first three rows, but you are **NOT** allowed to compute the amounts using arithmetic operations.

1.3. Most commonly used letter

Which letter occurs most frequently as the first character of attached files (filename)? How often? You do not have to distinguish between lower and upper case letters. If two or more letters have the same frequency and come first, provide all of them in your result. The use of the directive `FETCH FIRST ROWS ONLY` is **NOT** allowed! The result should contain one column for the letter and another column with the number of times it occurs, and it should be in ascending order on the letter.

1.4. Address sharing

Which users share an email address with the youngest (i.e., most recent date of birth) person? If there is a tie (two or more youngest persons with the same birthday), you must consider the addresses of all youngest persons. Results should contain the nickname of the persons in ascending order.

1.5. Birthday greetings

Who received emails on their birthday? Produce a result with the user's nickname, the date of birth, and the amount of emails they received. Exclude from your result the emails received from organizations, which are probably SPAM. You do not have to check the contents of the emails for words of congratulations. Furthermore, if a person received the same birthday email on multiple addresses, it must only be counted once. Note that a birthday is an annual event unlike the real date of birth recorded in the database. Results should be *ordered* by nickname in ascending order.

1.6. Top-10

Create a "Top-10" list of the users with the shortest average response time for their emails. To measure the response times, use the average amount of weeks between incoming email and outgoing response (use the `TIMESTAMPDIFF()` function). Consider all response emails sent by any address of the respective user (we do not care who is the receiver of these emails). To qualify for the Top-10, users must have responded to at least 3 messages (less than 3 values are not significant enough).

To handle ties, calculate the *dense rank* of each user. This means that two users with the same value will share the same rank, and the following rank is incremented by one. For example, four users with response times of 2, 4, 4, and 5 weeks would produce the ranks 1, 2, 2, and 3, respectively. The *Top-k list* is then defined as all rows with rank $\leq k$. Note, this means that the Top-10 may have more than 10 rows.

The result should contain the nickname and the ranks of the users, with rows *ordered* by the ranks in the ascending direction. You are **NOT** allowed to use OLAP functions like **`RANK()`**, **`DENSE_RANK()`**, or **`ROW_NUMBER()`**!

1.7. Top-10 variation

Produce the same Top-10 list as above, but this time with a *regular rank*. This means that ranks following repeated values are incremented by the amount of such repeated values. For the example given above with the four users, the ranks would be 1, 2, 2, and 4. Again, it is possible that more than 10 rows are produced, and OLAP functions are **NOT** allowed.

1.8. Thread decomposition

Build a *thread* representation of all emails in the system. A thread is defined as the largest set of emails that are connected by their *in_reply_to* relationships. Since different emails may respond to the same message, a thread actually produces a tree structure. To represent this in a table, you must assign the same *thread ID* value to all emails that belong to the same thread and attach a column containing the depth of each email in the tree. The thread ID is defined as the ID of the root email in the thread. Rows with the same thread ID must appear contiguously in the result set (i.e., sort the rows by thread ID in ascending order). Within the same thread, the emails should be displayed in ascending chronological order.

Instead of displaying all threads, you should filter them by searching for only those that contain an

email with a squared image (width = height) in the attachments. Note, even if multiple emails in a thread contain such an attachment, the thread should occur only once in the output.
The result should contain the following columns, in *order*: thread ID, email ID, sender username, in_reply_to, depth, and date.

Exercise 2: User-defined Functions

Now you have to extend the database with new functions written in SQL.

2.1. Scalar Functions

Define two scalar functions, which take an ID as input parameter and return a formatted string according to the following rules:

a) **FORMAT_ADDRESS** function
Given an address ID, return its string representation in the format "username@domain". Example: "john@foobar.com"

b) **FORMAT_USER** function
Given a user ID, return the name of the user depending on which class they belong to. Persons must be formatted as "*first_name surname*", and organizations simply as the *firm* name. Both must then be followed by the *nickname* in parentheses. For users that are neither persons nor organizations, only the nickname without parentheses should be displayed. Examples: "John Doe (johnny81)", "ACME INC. (acme)", "peggy79"

2.2. ADDRESS_BOOK function

Define a table function which computes an address book for a specific user. The function should take a user ID as input parameter and return the computed address book as a table with usernames and email addresses as columns. Use the scalar functions from the previous exercise to format the usernames and email addresses in a human-readable format.
The address book must consist of all addresses which can be found in all emails and their attachments written or received by the given user. You have to consider both the sender and the recipients of the emails. The addresses of the user given as parameter to the function will obviously be within these results as well. We do not require that you remove the user's own addresses.
If an email has further emails attached, you have to search for further addresses in these emails too and so forth. Note that you do **not** have to search in the **content** of emails for email addresses.

All functions **MUST** take object references as the argument type. Solutions using integer arguments will be rejected.

Exercise 3: SQL Programming in Java (JDBC)

Develop a small JDBC program that searches for specific attachments in the database and displays the results to the user in an interactive manner. The given file `dbprak.search.jar` contains the basic application logic that invokes the search and displays the results. All you have to do is implement the interface method that performs the search in the DB. We strongly recommend you to look at the source code embedded in the JAR file (you can open it as a ZIP file) before proceeding.

The program works as following: Using the console input, the user provides a nickname of an email user in the database together with a time span (`from`, `to`; in the format YYYY-MM-DD) and a search string. The program searches for this string in the attachments of all emails written or received by the selected user in this time span. The result of the search is then presented to the user as a list where each entry consists of the filename and the subject of a found attachment.

However, the basic program does not return any results, because the database logic has not been implemented. Your job is to provide an implementation of the `AttachmentSearch` interface, where the

database interaction takes place. The name of the implementation class MUST be `Search`. **The solution consists of a single `Search.java` file**, where this class is implemented. You are allowed to use inner classes if necessary, as long as all your code is contained in this single file.

To test whether the program works with your provided implementation, run the `Main` class in the JAR file with the required arguments, namely server, port, database, user, and password.

The program makes use of simple "bean" classes to represent database objects like user, email, attachment, etc. These are contained in the `dbprak.ws1.ex3` package of the JAR file, together with all basic classes. Your implementation must use these classes to represent any data coming from the database. Therefore, refrain from using strings or generic JDBC values for manipulating DB objects in your implementation.

The following methods have to be implemented:

- `setConnection(Connection connection)`: This method is invoked by the program at start-up. It provides your implementation with a JDBC Connection object, which you can use to access the database.
- `listAttachments(User user, Date from, Date to, String searchString)`: This method is invoked when the user performs a search. The input parameters directly correspond to the ones explained above. The method must return a list (`java.util.List`) of `Attachment` objects, which are the results of the search query.

The implementation of the `listAttachments` method must satisfy the following conditions:

- The search in email attachments has to be **transitive**, i.e. you have to search in all attachments of an email as well as in the attachments of attached emails and so forth.
- The given search string **must** match any substring in either the filename or the subject or—if available—the content of an attachment.
- SQL is powerful enough to evaluate this search. Therefore, you are **NOT ALLOWED** to simply fetch all attachments from the database and implement the search logic in Java.

Submission

Once again, check that you considered all constraints and hints, the general ones and the task specific ones. Submit your solution using the PRISE system and please pay attention to the hints given in the Wiki. Follow the links on `http://lara.informatik.uni-kl.de/`.

Submission deadline is Sunday, May 19, 2019 at 23:59:59.
Lots of fun and success!