**Project Outline and Planning**

**TEAM: Jon Mukaj, Fabio Marku, Kevin Tenolli, Enest Methasani, Kristi Rrapaj, Eugen Selenica**

Our project involves designing a contester database, which is an application oriented to programming competitions and coding exercises. The database should hold information related to the registered users, exercises, statistics etc. The users will sign in into the contester with their username and password. They will have access to exercises, together with their category and status.

During competitions, the entrants of the competition will have the privilege of accessing a special section with exclusive exercises according to the respective contest. Each exercise, will be accompanied with a brief description, simulation and a section for code submission. The compiler provided on the contester will include support for JAVA, C, C++, and Python. The validation of the code will be made by first checking the syntax (task done by compiler), and then checking the output of the script, compared to the output provided by the test case. In addition to this, the submission will include even constraints related to memory and time.

On the Research & Idea Phase, our team will discuss about different technologies that can help on the aspect of database speed and performance (query optimization, normalization). After this step, two members of the team will be chosen (Enest and Kevin) in order to thoroughly describe how these technologies function and how will they be implemented on our project, based on their research and on what the team discussed beforehand.

On the Project Execution Phase, the team will schedule a brainstorming session, where each member will propose at least 5 entities/relationships that he thinks should be specifically included on the database conceptual design. After this step, Jon and Fabio will take the product of all the proposals, make corrections and changes where needed and compile a list of requirements definitions for our database. The other members of the group based on the requirement definition will start creating the ER diagram of the database. Since, the conceptual design is one of the most crucial steps in the development of databases, three members of the group (Jon, Kevin, Eugen) will be assigned the task of checking the ERD to make sure no design pitfall is made before going to the next step.

The RS schema is a straightforward step, since most of the work and decision – making is supposed to be done on the ERD step. However, after Kristi and Eugen finish converting the ERD to RS, Fabio is tasked with the duty of checking the schema for any error. Jon will extract the functional dependencies and check if the database conforms to the Third Normal Form. For database dump phase, Fabio and Kevin have the duty of creating and inserting data (enough data only for test purposes), while the queries part will have contributions from everyone, because each member will be tasked to provide 5 managerial queries, that will deal with retrieving data from the database based on a certain purpose. During the project our team will use applications and services such as ERDplus, sqlliteonline, PopSQL and MySQL Workbench.

**Research and Idea Consolidation**

**Authors: Kevin Tenolli, Enest Methasani**

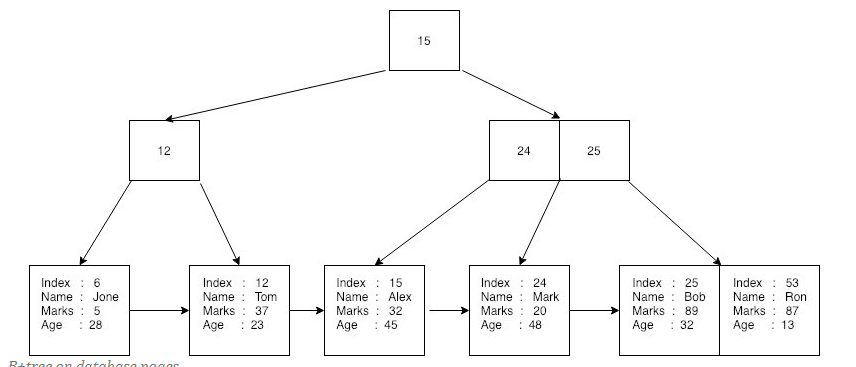
The aim of this section is to introduce and discuss about technologies, which can be implemented in our database in order to improve its speed and data redundancy.

The first subject is that of normalization. Normalization is the process of structuring a relational database from one normal form up to another, with the purpose of reducing data redundancy and integrity. Data redundancy deals with the pattern of unnecessarily repeated values in record or fields of the database. The existence of redundancy is a complicated issue, because on one hand it can be taken into advantage in order to improve query retrieval speed, based on the fact that more information is found condensed on the same table, but on the other hand, it can greatly affect storage size and more importantly increase the risk of corrupting data, as a direct consequence of update anomalies. To keep a fair balance between storage and speed, we have made the decision to normalize up to the Third Normal Form, to eliminate every transitive dependency.

The advantage of a normalized database is that, the operations for insertions, deletion and modification will cause no anomaly behavior. In addition, relations will be relatively smaller, and as such substantially improve the performance of the database (taking into consideration that the data will first go through a buffer during query execution). But the most crucial edge of normalization is making our database ACID compliant. ACID which is the acronym for atomicity, consistency, isolation and durability is a set of properties related to database transactions intended to guarantee validation of data no matter the scenarios. However, normalization will come with its own disadvantages. It will put a bigger pressure for our team on the correct creation of queries, since for the retrieval of data from different tables, JOINS is required. This complication of queries can even affect and hinder indexing read times, since the pointer to the physical memory will not work efficiently with JOINS.

The other technique for improving the performance of the database, specifically the part of searching for information is indexing. Indices are used to find rows of a column quickly, without reading through the entire table. There exist many methods for indexing, with the most naïve being creating a new relation for the index of a column, which will ensure binary search time complexity, and others more advanced such as Clustering, Hashing and B+ Trees. Since, MySQL supports B+ Trees, that is what our team will take advantage of for the implementation of our database.

B+ Tree is a form B Trees, which on itself is a generalized form of a binary tree, which differs from the fact that instead of a single entry, its node will point to an array of entries and have a reference to the child node of each entry. B+ Tree takes this even further by storing the data only on its leaf nodes, meaning that all values in non-leaf nodes is duplicated onto the leaves. This means that the leave will include all the records of the tree in a sorted order. The way this data structure is designed gives the advantage of having a logarithmic time complexity for searching, insertions and deletions. This property of the B+ tree means that they can be implemented directly for indexing on a database albeit with a little modification, because now the entry or payload will consist of a key and a value associated with that key, the value itself being a pointer or reference to the record of interest on our database. Below, there is an example of how a database will work using indexes with B Trees on MySQL.



**Requirements**

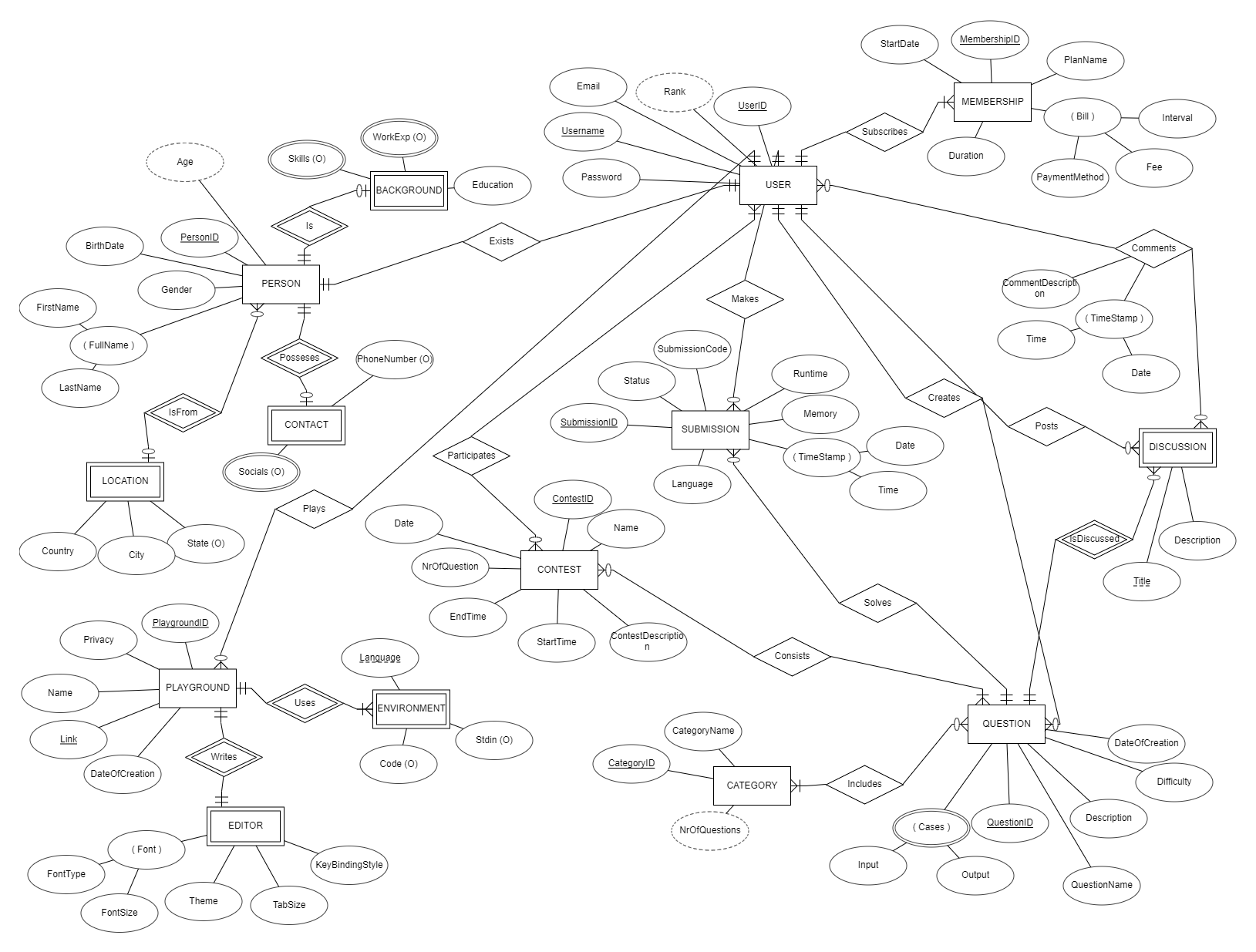
**Authors: Jon Mukaj, Fabio Marku**

CodeHero is a contester webpage where students, coders and other professionals have the opportunity to practice their coding skills, compete for contests and prepare for interviews. The ER diagram of the database for CodeHero is based on the following requirements:

* The CodeHero Database will keep track of people, backgrounds, locations, users, memberships, contacts, questions, contests, categories, discussions, submissions, playgrounds, editors and environments.
* For each person, CodeHero will keep track of a unique person identifier, full name, date of birth and gender.
* For each location, CodeHero will keep track of country, city and optional state.
* For each user, CodeHero will keep track of a unique user identifier, unique username, password, email, and status.
* For each membership, CodeHero will keep track of a unique membership identifier, plan name, start date, duration, and billing info including fee, payment method and interval of payments.
* For each contact, CodeHero will keep track of optional phone number, and optional social addresses.
* For each background, CodeHero will keep track of multivalued optional skills, work experience and education.
* For each question, CodeHero will keep track of a unique question identifier, question name, description, testcases, date of creation and difficulty rating.
* For each contest, CodeHero will keep track of a unique contest identifier, name, description, number of question contest has, date, start and end time of contest.
* For each category, CodeHero will keep track of a unique category identifier, category name and number of problems in category.
* For each discussion, CodeHero will keep track of a partially unique discussion identifier, discussion title, and description.
* For each submission, CodeHero will keep track of a unique submission identifier, status, runtime, memory, code, language of submission and timestamp.
* For each playground, CodeHero will keep track of a unique PlaygroundID, unique link so playground can be shared, playground name and privacy status and date of creation.
* For each editor, CodeHero will keep track of font type and size, tab size, editor theme, and Keybinds style (VIM, EMAC, Standard).
* For each environment, CodeHero will keep track of partially unique language and optional codes and standard inputs given to the runtime environment.
* A user exists for exactly one person. A person exists for exactly one user.
* A person does not need to be of a background, but can be of exactly one background (identifying relationship). A background is for exactly one person.
* A person does not need to be from a location, but can be from exactly one location (identifying relationship). A location does not need to be for a person, but may be for many people.
* A person does not need to possess a contact, but can possess only one contact (identifying relationship). A contact can be possessed by exactly one person.
* A user does not need to make a submission, but may submit many times. A submission can be made by exactly one user.
* A user does not need to create a question, but may create many questions. A question can be created by exactly one user.
* A user does not need to post a discussion, but may post many discussions. A discussion can be posted by exactly one user.
* A user does not need to comment in a discussion, but may comment in many discussions many times. A discussion does not need to be commented by a user, but may be commented by many users.
* A discussion is discussed by exactly one question (identifying relationship). A question may be discussed by many discussions or by no discussion at all.
* A question does not need to be solved by a submission, but may be solved by many submissions. A submission can solve exactly one question.
* A category may include many or no question at all. A question must be included by at least one category.
* A contest consists of one or many questions. A question can be in many or no contests at all.
* An environment is used by exactly one playground (identifying relationship). A playground may use one or more environments.
* An editor writes exactly one playground (identifying relationship). A playground is written by exactly one editor.
* A user subscribes to one or many memberships. A membership can be subscribed by exactly one user.

**Figure 1. ERD**

**Authors: Kevin Tenolli, Enest Methasani, Kristi Rrapaj, Eugen Selenica**



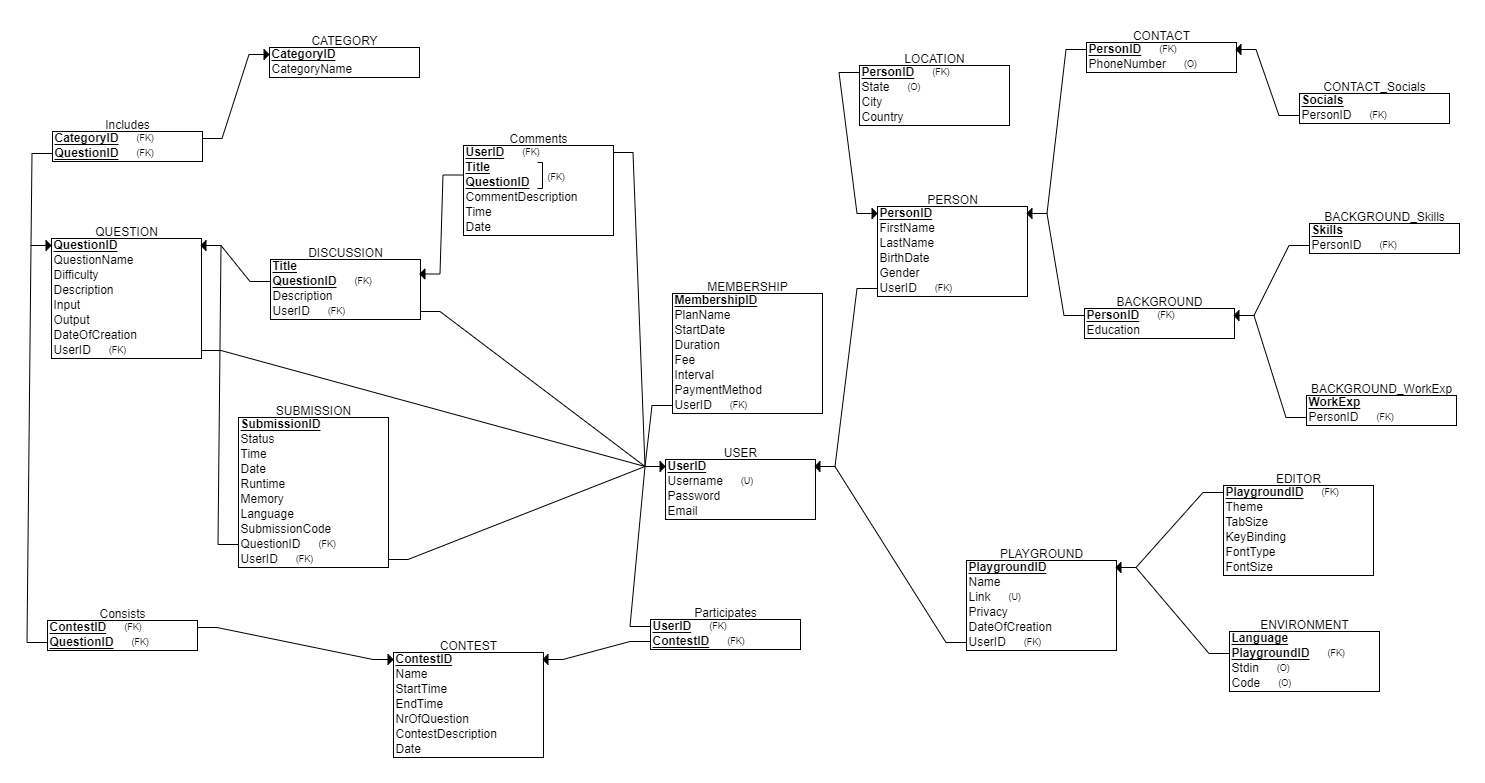
**Rationales**

**Author: Kevin Tenolli**

* We decided to make Discussion and Environment weak entities, since they cannot be uniquely identified by only their attributes, but they are identifiable in combination with the question for which the discussion is made at the first place and the playground for which the environment is being used. Therefore, via an identifying relationship isDiscussed and uses, Discussion is related to Question while Environment is related to Playground.
* The entity Editor includes no fully or partially unique key as its attribute. While typically a weak entity has a composite key consisting of the key of its parent entity plus one or more additional attributes, it doesn't have to be so since in our case, the Playground entity can have at most only one editor, meaning that PlaygroundID can serve on its own as an identifying key for the Editor. The same applies for Location, Contact and Background.
* The attributes Code and Stdin of entity Environment are left as optional, since there could be scenarios where the users leave the environment empty or where no input is supposed to be taken from the keyboard.
* The relationship of person with entities such as Location, Contact and Background was made as optional, since we took into consideration the fact that some individuals prefer to be discrete and not provide too much personal information.
* The relationship of category with question is optional, since new categories can be inserted, questions can be removed during the life cycle of the database.
* The relation Comments has attributes for description and timestamp of the comment, since users can comment many times on the same discussion thread.
* The entity Submission has attributes related to memory consumed, runtime and language of script, since memory and runtime depend on the different solutions that exists for the same question. CodeHero supports many programming languages, so the language attribute is included to make a distinction between different submissions of the same question.

**Figure 2. RS**

**Authors: Kristi Rrapaj, Eugen Selenica**



**Functional Dependencies**

**Author: Jon Mukaj**

Now, we define functional dependencies for each table, which are the relationships between the attributes of a table.

**Table: Question**

**PK:** QuestionID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** QuestionID => QuestionName, Difficulty, Description, Input, Output, DateOfCreation

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Question** is already in 1NF, since no multi-values are present.
2. The table **Question** is already in 2NF, since no partial key dependency exists.
3. The table **Question** is already in 3NF, since no transitive dependency exists.

**Table: Submission**

**PK:** SubmissionID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** SubmissionID => Status, Time, Date, Code, Runtime, Memory, Language.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Submission** is already in 1NF, since no multi-values are present.
2. The table **Submission** is already in 2NF, since no partial key dependency exists.
3. The table **Submission** is already in 3NF, since no transitive dependency exists.

**Table: Contest**

**PK:** ContestID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** ContestID => Name, StartTime, EndTime, NrOfQuestion, ContestDescription, Date.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Contest** is already in 1NF, since no multi-values are present.
2. The table **Contest** is already in 2NF, since no partial key dependency exists.
3. The table **Contest** is already in 3NF, since no transitive dependency exists.

**Table: Category**

**PK:** CategoryID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** CategoryID => CategoryName

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Category** is already in 1NF, since no multi-values are present.
2. The table **Category** is already in 2NF, since no partial key dependency exists.
3. The table **Category** is already in 3NF, since no transitive dependency exists.

**Table: Discussion**

**PK:** Composite (Title + QuestionID (FK))

**Partial Key:** None, even though the relation has a composite primary key.

**Full Key:** QuestionID, Title => Description

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Discussion** is already in 1NF, since no multi-values are present.
2. The table **Discussion** is already in 2NF, since no partial key dependency exists.
3. The table **Discussion** is already in 3NF, since no transitive dependency exists.

**Table: Comments**

**PK:** Composite (Title + QuestionID (FK) + UserID)

**Partial Key:** None, even though the relation has a composite primary key.

**Full Key:** QuestionID, Title, UserID => CommentDescription, Time, Date.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Comments** is already in 1NF, since no multi-values are present.
2. The table **Comments** is already in 2NF, since no partial key dependency exists.
3. The table **Comments** is already in 3NF, since no transitive dependency exists.

**Table: Membership**

**PK:** MembershipID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** MembershipID => PlanName, StartDate, Duration, Fee, Interval, PaymentMethod.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Membership** is already in 1NF, since no multi-values are present.
2. The table **Membership** is already in 2NF, since no partial key dependency exists.
3. The table **Membership** is already in 3NF, since no transitive dependency exists.

**Table: User**

**PK:** UserID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** UserID => Username, Password, Email.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **User** is already in 1NF, since no multi-values are present.
2. The table **User** is already in 2NF, since no partial key dependency exists.
3. The table **User** is already in 3NF, since no transitive dependency exists.

**Table: Person**

**PK:** PersonID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** PersonID => FirstName, LastName, BirthDate, Gender.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Person** is already in 1NF, since no multi-values are present.
2. The table **Person** is already in 2NF, since no partial key dependency exists.
3. The table **Person** is already in 3NF, since no transitive dependency exists.

**Table: Location**

**PK:** PersonID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** PersonID => State, City, Country.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Location** is already in 1NF, since no multi-values are present.
2. The table **Location** is already in 2NF, since no partial key dependency exists.
3. The table **Location** is already in 3NF, since no transitive dependency exists.

**Table: Contact**

**PK:** PersonID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** PersonID => PhoneNumber

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Contact** is already in 1NF, since no multi-values are present.
2. The table **Contact** is already in 2NF, since no partial key dependency exists.
3. The table **Contact** is already in 3NF, since no transitive dependency exists.

**Table: Contact\_Socials**

**PK:** Socials

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** None.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Contact\_Socials** is already in 1NF, since no multi-values are present.
2. The table **Contact\_Socials** is already in 2NF, since no partial key dependency exists.
3. The table **Contact\_Socials** is already in 3NF, since no transitive dependency exists.

**Table: Background**

**PK:** PersonID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** PersonID => Education

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Background** is already in 1NF, since no multi-values are present.
2. The table **Background** is already in 2NF, since no partial key dependency exists.
3. The table **Background** is already in 3NF, since no transitive dependency exists.

**Table: Playground**

**PK:** PlaygroundID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** PlaygroundID => Name, Link, Privacy, DateOfCreation.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Playground** is already in 1NF, since no multi-values are present.
2. The table **Playground** is already in 2NF, since no partial key dependency exists.
3. The table **Playground** is already in 3NF, since no transitive dependency exists.

**Table: Editor**

**PK:** PlaygroundID

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** PlaygroundID => Theme, TabSize, KeyBinding, FontType, FontSize.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Editor** is already in 1NF, since no multi-values are present.
2. The table **Editor** is already in 2NF, since no partial key dependency exists.
3. The table **Editor** is already in 3NF, since no transitive dependency exists.

**Table: Environment**

**PK:** PlaygroundID, Language.

**Partial Key:** None, since the relation has a single column primary key.

**Full Key:** PlaygroundID, Language => Stdin, Code.

**Transitive:** None, since no non-key column defines another column.

**Normalization:**

1. The table **Environment** is already in 1NF, since no multi-values are present.
2. The table **Environment** is already in 2NF, since no partial key dependency exists.
3. The table **Environment** is already in 3NF, since no transitive dependency exists.

**Database dump**

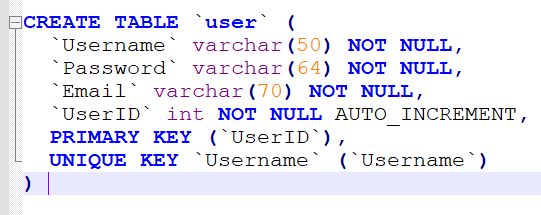
**Creating Tables**

**Author: Kevin Tenolli**

Based on the Relational Schema, I wrote the CREATE statements in SQL for our database. The order of the CREATE statements in conformity with the Referential Integrity Constraint, meaning that relations with foreign keys are created after relations being referenced by foreign keys.

The tool I used mostly to create the tables of our database was MySQL Workbench. It is an IDE which integrates database design, implementation, administration and SQL execution. In addition to this, it implements by default Indexing for candidate keys using B trees, which gives the opportunity for a greater speed on query retrievals.

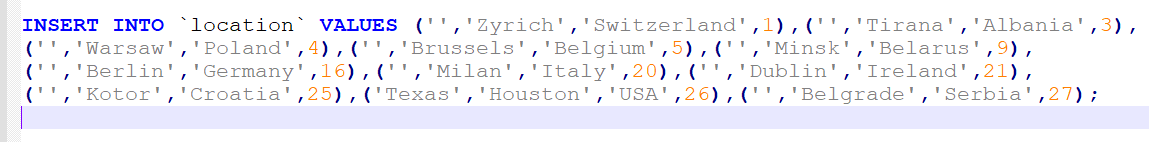
For each primary key of our relations, I included the keyword AUTO\_INCREMENT on the CREATE TABLE statement. This means that the ID will be auto-generated and auto-incremented for each record and there is no need to give the value of id when populating the relations with data.



**Populating Tables**

**Author: Fabio Marku**

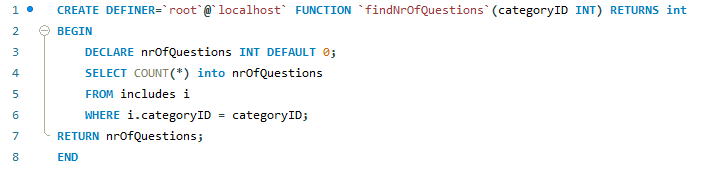
For our database to have a meaning and for the managerial queries to be tested properly and basically give a result, the tables need to be populated with data. This operation was done by executing queries using INSERT INTO statement. I decided that 10 to 15 records for each table was enough to serve as a sample of the database.



**Derived Attributes**

**Author: Jon Mukaj**

Since derived attributes are not shown in the Relational Schema, they are not present in the CREATE statements. However, they are part of our database and we should present them by using queries and functions to calculate the attributes.



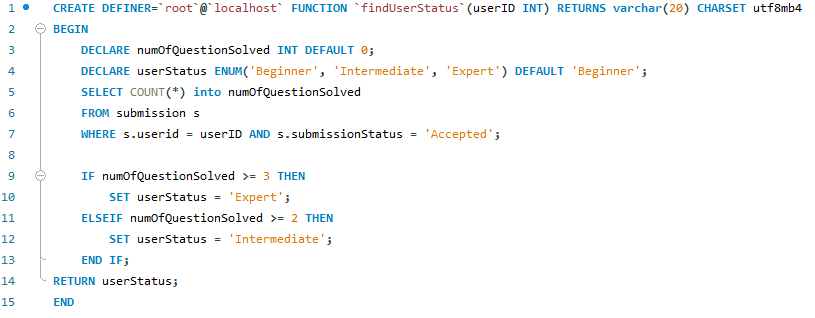
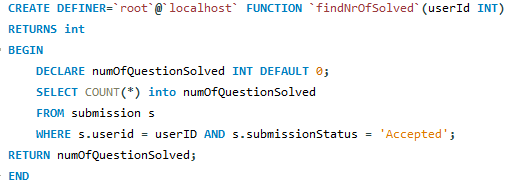


Figure 3. Functions used to calculate derived attributes.

****

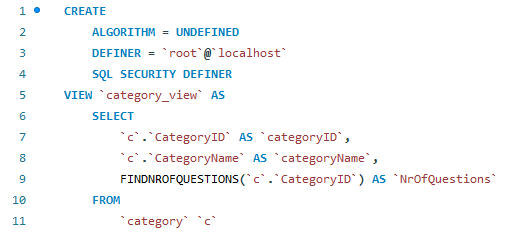
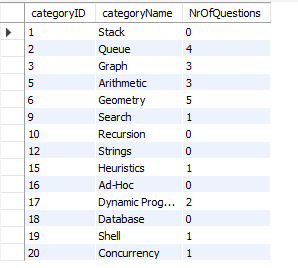
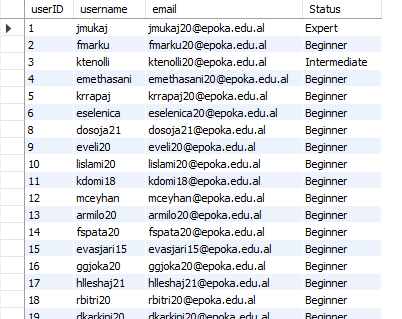
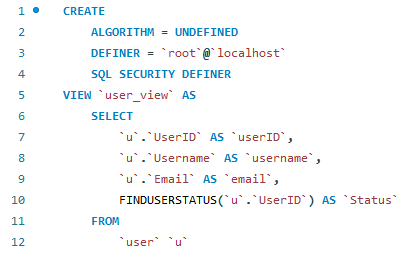
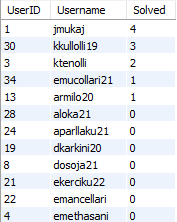
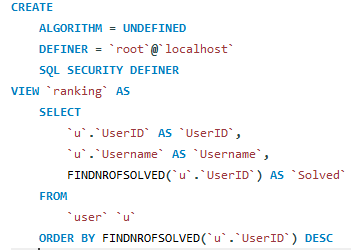


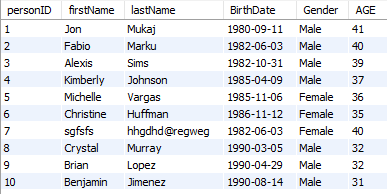
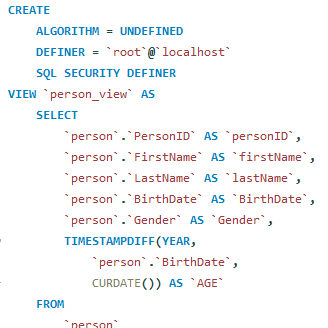
Figure 4.1. View to show category with number of questions

Figure 4.2. View to show user with its status





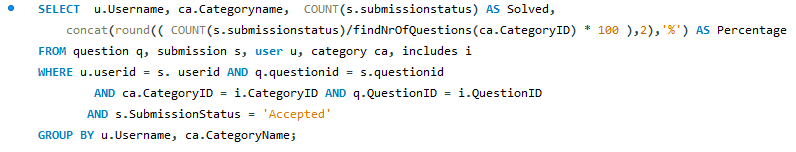


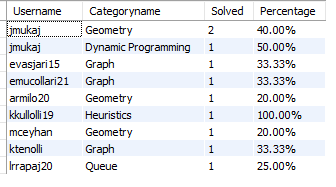


**Managerial Queries**

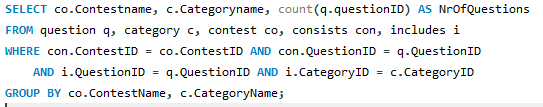
* **Kevin Tenolli**

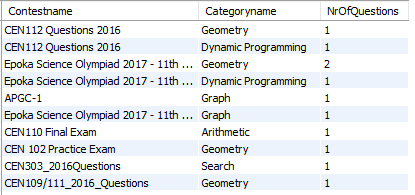
**Description: Shows nr of solved questions in a category from each user**

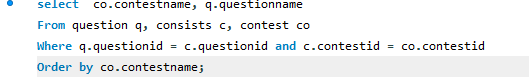
****

****

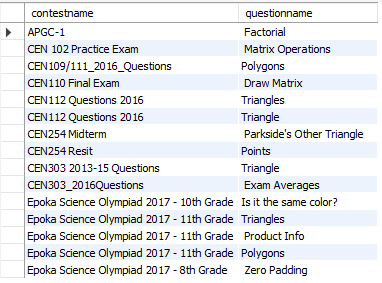
**Description: Shows the number of questions from each category included in each contest**

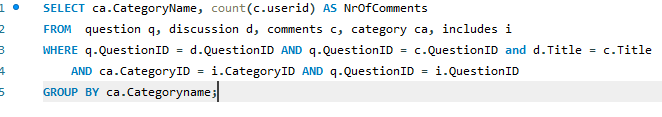
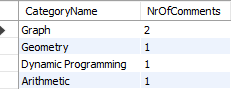
****

****

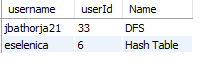
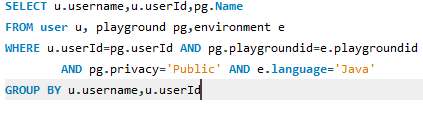


**Description: Shows each contest and its question**



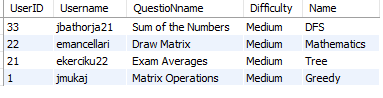
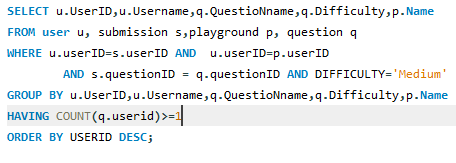
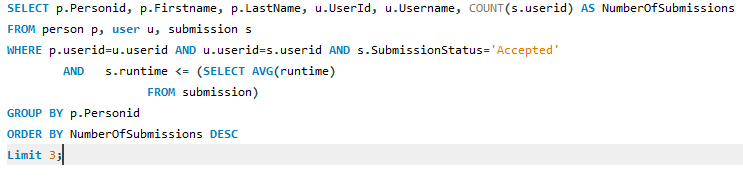
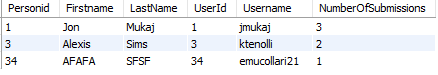


**Description: Shows number of comments for each category**

* **Eugen Selenica**

**Description: Select the username, userid, playground name for all users that**

**use java in public playgrounds**

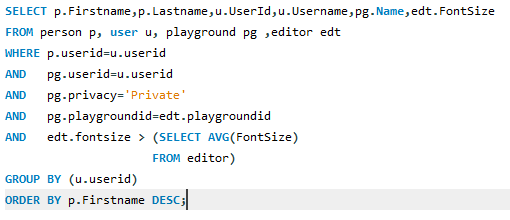
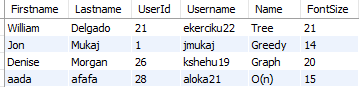


**Description: Select the id , username, playground-name and the name of the question of all the users that have attempted to solve at least 1 medium question.**

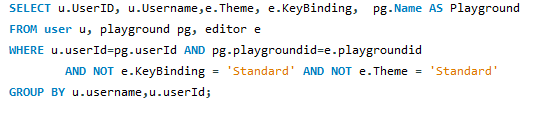
**Description: Select the real name username id, for the top 3 users with the**

**most questions submitted and whose runtime is faster than the average**

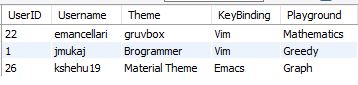
**Description: Select the firstname, lastname ,userid,username,playgroundname for all users that use private playgounds where the fontsize is greater than the average**



‘

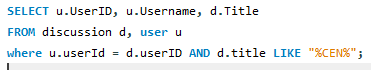
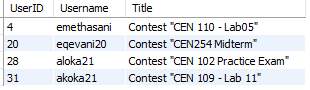
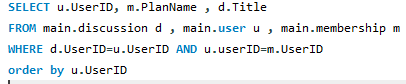
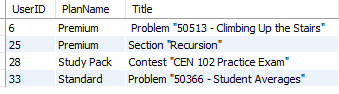
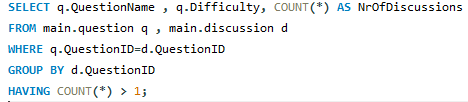


**Description: Select the username, userid, password, theme, keybinding and playground name for all the users whose editor keybinding and theme is different from standard**



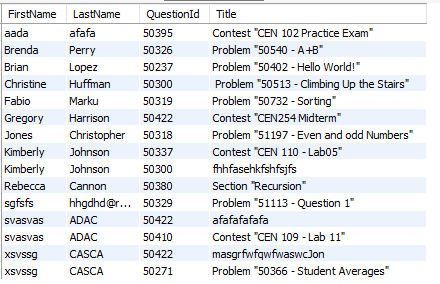
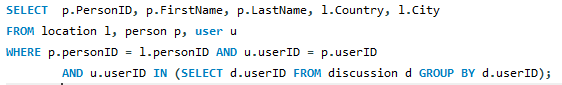
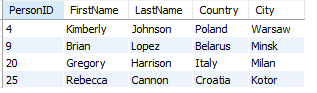
* **Fabio Marku**

**Description: Select all the users from THE CEN (Computer Enginnering) question that have made a disscusion about a question. Display them by id, username and title.**

****

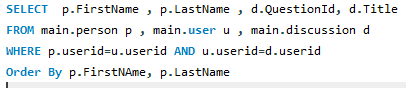
**Description: Diplay the QuestionName, its difficulty and number of discussions for questions which were discussed more than once.**

**Description: Display UserID , membership plan the user has paid , and the Title of the discussion this user has made . Order them by userid.**



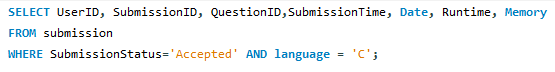
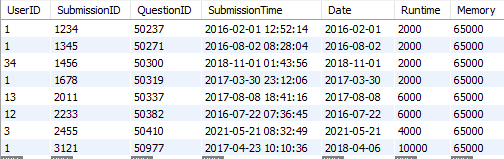
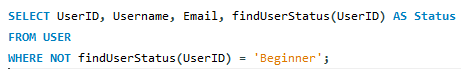
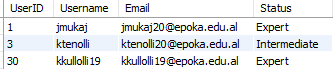
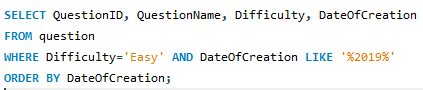
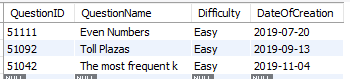
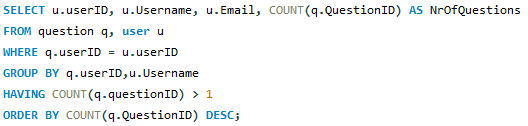
**Description: Display the firstname, lastname, id, country and city of all the users that have made discussion in the contester**

**Description: For all the persons who have made discussion about an exercise, display ther FirstName , LastName, the QuestionID and title. Sort them by first name .**



* **Enest Methasani**

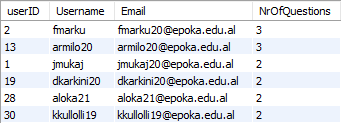
**Description: Show all the submissions on the C language whose status is accepted.**

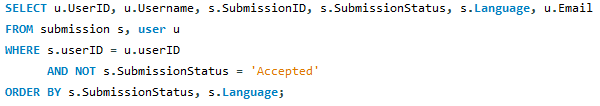
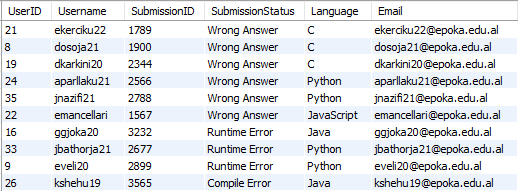


**Description: Show the userID, email and the number of created questions for every user that has created more than one question.**

**Description: Retrieve QuestionID, QuestionName, Difficulty and DateOfCreation for each question in the Easy' questions category that have been created in '2019' sorted by date of creation.**

**Description: Retrieve the Username, Email, UserID and status for each user whose status is different than beginner.**

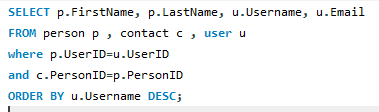
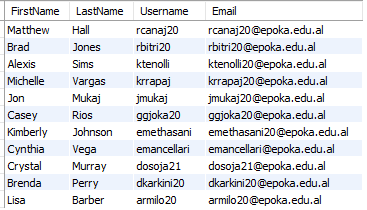
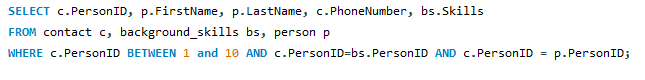
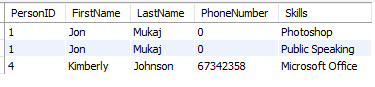




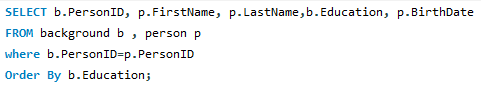
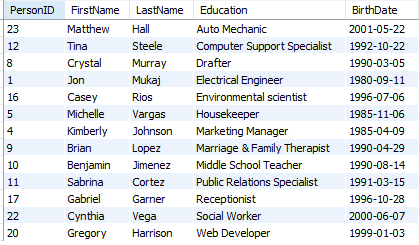
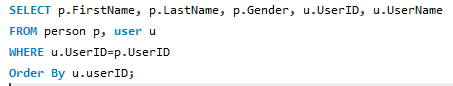
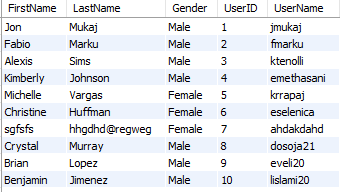
**Description: Show the userId, username, SubmissionID, SubmissionStatus, Language and email of the student for each submission whose status is not 'Accepted'. Order by status and language.**

* **Kristi Rrapaj**

**Description: Select the first name, last name, username and email and order them by username in descending order.**



**Description: Select person id , person name, phone number and skills where person id is between 1 and 5.**



**Description: Select the first name, last name, gender, user id and user name and order them by their userid.**

**Description: Select name, education, person id, and birthdate and order them by education.**