Project outline and planning

For building an ACM database, we need to consider the requirements. The requirements for this database are:

1. **Users:** The users in this case would be the students, in which they do the assignments, solve problems and discuss with other people and many more; the teacher who evaluates the students' solutions and give grades to them and the administrator who manages the data and maintains the system. These users will have permission depending on the role they are given.

2. **Comments:** The students will ask questions if some exercises might be a bit hard or if there is something that he doesn't understand or need a hint.

3. **Statistics:** this section will have for each student the number of solutions he has done, the number of exercises he didn't correctly or how many solutions did he submit or which will be derived from the database.

4. **Competition:** this section will be for special events in which a group of questions are presented and the student who solves the most questions right, he becomes a winner. There will also be some sections of some contests from previous years in case the student wants more challenges.

5. **Volumes/Category:** in this section, the exercises are divided into categories so that the student could start from something easy to something challenging.

6. **Forums/ Issues:** if there is something that is needed to be discussed on an assignment or problem, the student can go to the forums section and ask for anything

7. **FAQ/Issues:** when the user is registered for the first time, he/ she is not familiar with the system yet and this can be found in the frequently asked questions or report if there is a problem with the system.

8. **Assignments/tasks:** the teacher can submit a group of questions as homework or assignment for the students and they can solve before the deadline. If the student misses the homework or task deadline, he is immediately evaluated with 0 points. Otherwise, depending on the solution, the judging system will evaluate it properly.

9. **Questions:** This section will have questions with their description and some sample test cases. They will be divided into categories depending on the volume and the difficulty level.

10. Announcements: This section will have the recent news divided in categories.

11. Ranking system: This section will show a table with all the names of the students and the total number of solutions they have done posted in descending order.

12. TODO: In this section, the users will be able to create a todo list which will be used as a reminder and it may contain a description and a list of questions of his choice to solve it later.

Depending on these requirements for the project we will build the appropriate tables and eliminate any possible data redundancy and make sure the queries are appropriate for the inquiry of the data presented. We will construct ERD and RS then we will implement the database in a RDBS which will be chosen in the next phases. We will do some research and find a proper RDBMS that suits our projects. We may use other DBMS paradigms besides RDBMS to improve the performance. And in the end, we will implement the database which we will include in final submission.

Project Execution

Before we begin with the erd and rs schema, we first need to show the entities and their attributes that we thought about:

Entities

(entity -> attributes)

Department -> department_id, name, acronym

User_level -> user_level_id, name, description

User_setting -> user_setting_id, language_id, programming_language_id, theme_id

Rank -> rank_id, name, minimum_question_count

Theme -> theme_id, name

Language -> language_id, name, prefix

User -> user_id, first_name, last_name, birthday, username, password, email, question_submitted, question_solved, questions_rejected, created_at, created_by, user_level_id, depeartment_id, user_setting_id, rank_id

Submission -> submission_id, code, status_name,created_at, question_id, user_id, assignment_id, programming_language_id, error_message

Programming_language -> programming_language_id, name, version, logo, compiler_path

Course -> course_id, name, acronym, code

Question Level-> question_level_id, name

Question Category -> question_category_id, name

Volume -> volume_id, name

Question -> question_id, title, description, created_at, hint, volume_id, user_id, question_level_id, question_category_id

Test Case -> test_case_id, input, output, question_id

FAQ -> faq_id, title, description, created_at

Contest -> contest_id, name, created_at, description

Student Group -> student_group_id, code, study_year, department_id

Group -> group_id, name, student_group_id, parent_group_id

ToDo -> todo_id, name, description, user_id

Assignment -> assignment_id, name, description, created_at, deadline, is_graded, user_id, programming_language_id, contest_id, course_id

Comment -> comment_id, title, description, created_at, user_id, question_id, replies_to_id

Announcement -> announcment_id, created_ at, title, description, announcment_category_id

Announcement Category -> announcment_category_id, name

Attachment -> attachtment_id, title, created_at, assigment_id

Relationships

Entity -> Entity One(Optional)-to- many(Mandatory) (Explanation)

User -> User Level (Many to one) (Many users has exactly one user level and one user level has many users)

User -> Rank (Many to one) (Many users can have the same rank and one rank can be the same among many users.)

User -> Department (Many to one) (Many users belong to the same department and one department has many users.)

User -> User setting id (One to one) (Each user has a unique setting and each user setting is unique to one user.)

User setting -> Language (Many to one) (One language can be used across many user settings and many user settings can use the same language.)

User setting -> Theme (Many to one) (Many user settings can have the same theme and one theme might be used on many user settings.)
Assignment -> User (Many to Many) (An user can have many assignments and an assignment can be assigned to many users.)
Assignment -> Programming language (Many to one) (Many assignments can have the same programming language with which they need to be solved and one programming language might be requested on many assignments.)
Assignment -> Contest (Many to one) (Many assignments can be part of a contest and one contest might have multiple assignments as part of it.)
Assignment -> Course (Many to one) (A course contains many assignments and many assignments are part of a course.)
Question -> Volume (Many to one) (A volume contains many questions and many questions are part of a volume.)
Question -> User (Many to one) (A user can create many questions, but might also create none. Many questions can be created by one user.)
Question -> Question Level (Many to one) (Many questions belong to the same level of difficulty, and one difficulty level can contain many questions.)
Question -> Question Category (Many to one) (Many questions belong to the same category, and one question category contains many questions.)
Comment -> User (Many to one) (A user can leave many questions, but isn’t required to leave any. Many comments can be left by the same user.)

Comment -> Question (Many to one) (A question can have many comments related to it, but can also have no comments. Many comments can be left to one question.)
Comment -> Comment (Self to many) (A comment might get replies from many comments and many comments might reply to one comment.)
User -> Question (Many to Many) (FAVORITE QUESTIONS) (A user can have many favorite questions and a question might be a favorite of many users.)
Test Case -> Question (Many to One) (Many test cases belong to one question and one question contains many test cases.)
Submission -> User (Many to one) (One user can submit many submissions and many submissions might be submitted by the same user.)
Submission -> Question (Many to one) (Many submissions might be submitted for one question and one question can have many submissions.)
Submission -> Assignment (Many to one) (Many submissions can be submitted for one assignment and one assignment can have many submissions.)
Submission -> Programming Language (Many to one) (Many submissions can be written in one programming language and one programming language can be used on many assignments.)
Group -> Student Group (One to one) (A group can be a student group and a student group is a kind of a group).
Group -> Group (Self to many) (A group can contain many subgroups and many subgroups can be part of a parent group.)
Student Group -> Department (Many to one) (Many student groups belong to one department and one department contains many student groups.)
ToDo -> User (Many to one) (Many ToDo notes can be noted by a user and a user can have many ToDo notes.)
Announcement -> Announcement Category (Many to one) (There can be many announcements of the same category and one announcement category can contain many announcements.)
Attachment -> Assignment (Many to one) (An assignment can have many attachments to it and many attachments can be linked to one assignment.)
Assignment -> Group (Many to Many) (A group can have many assignments, an assignment can be given to many groups.)
Assignment -> Question (Many to Many) (An assignment can have many questions and a question can be part of many assignments.)
To Do -> Question (Many to Many) (A question can be part of many ToDo notes and a ToDo note can contain many questions.)
Group -> User (Many to Many) (A group can have many users and a user can be part of many groups.)

ERD (Entity Relationship Diagram)

After detecting the entities and the relationships formed between them, we started working on the Entity Relationship Diagram. We first inserted all the entities on the schema, followed by their attributes. Following this, we introduced the relationships formed, along with the cardinality constraints for each of them. We then converted multivalued attributes into entities and formed new relationships. This allowed us to have an easier transition to the Relational Schema (RS) and normalization of tables.

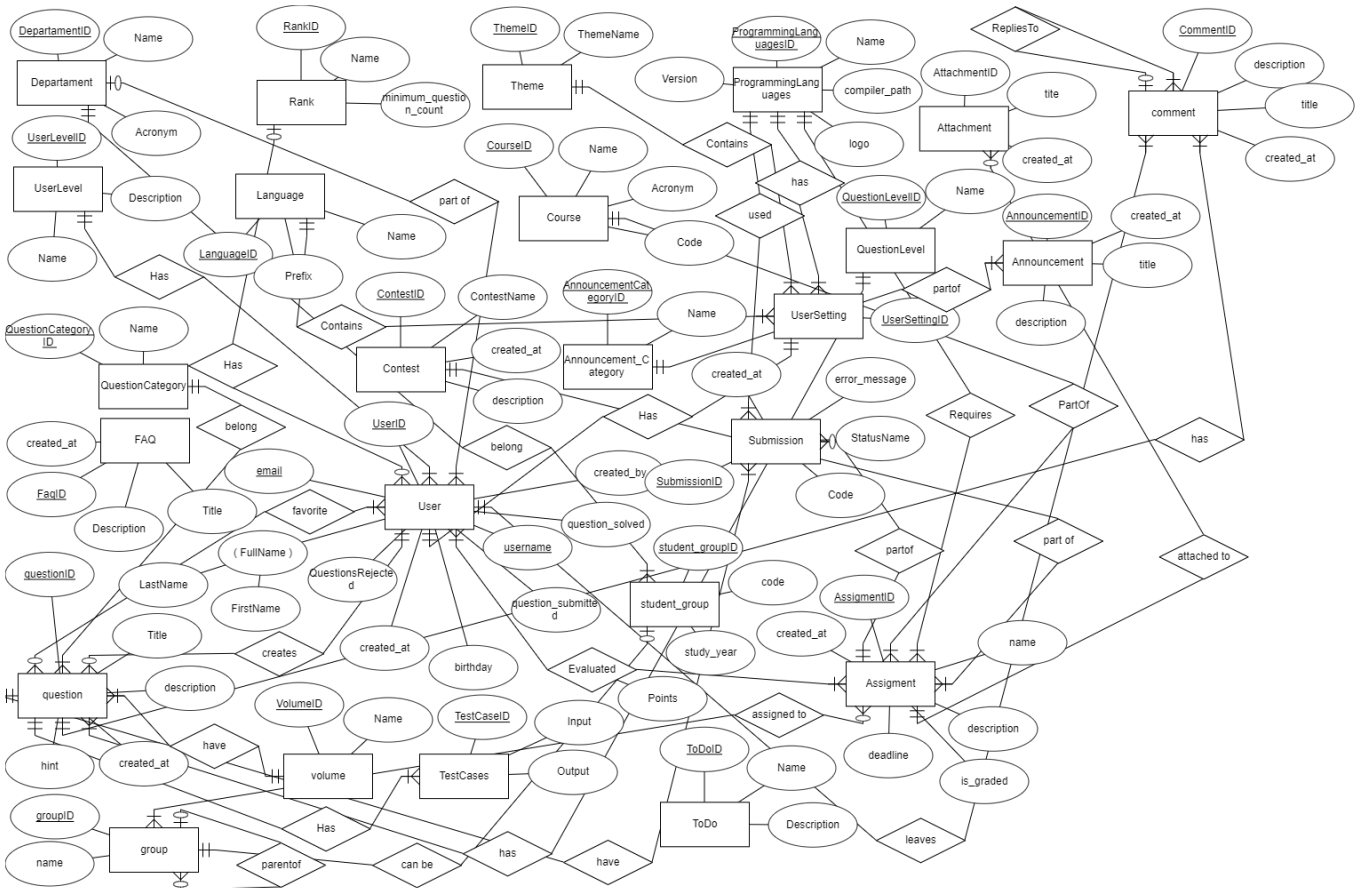

Figure 1. Entity Relation Diagram (ERD)

RS (Relational Schema)

Having created the Entity Relation Diagram, we started working on providing a Relational Schema for our database. We mapped each entity into a table. Then we added the corresponding attributes to each entity. To represent relationships, we added foreign keys of related tables as appropriate, preferring non-null foreign keys over nullable ones. However, this construct has proven to be non sufficient to map many to many relationships to RS. To solve this problem, we introduced new tables for entities having many to many relationships and created one to many relationships between them.

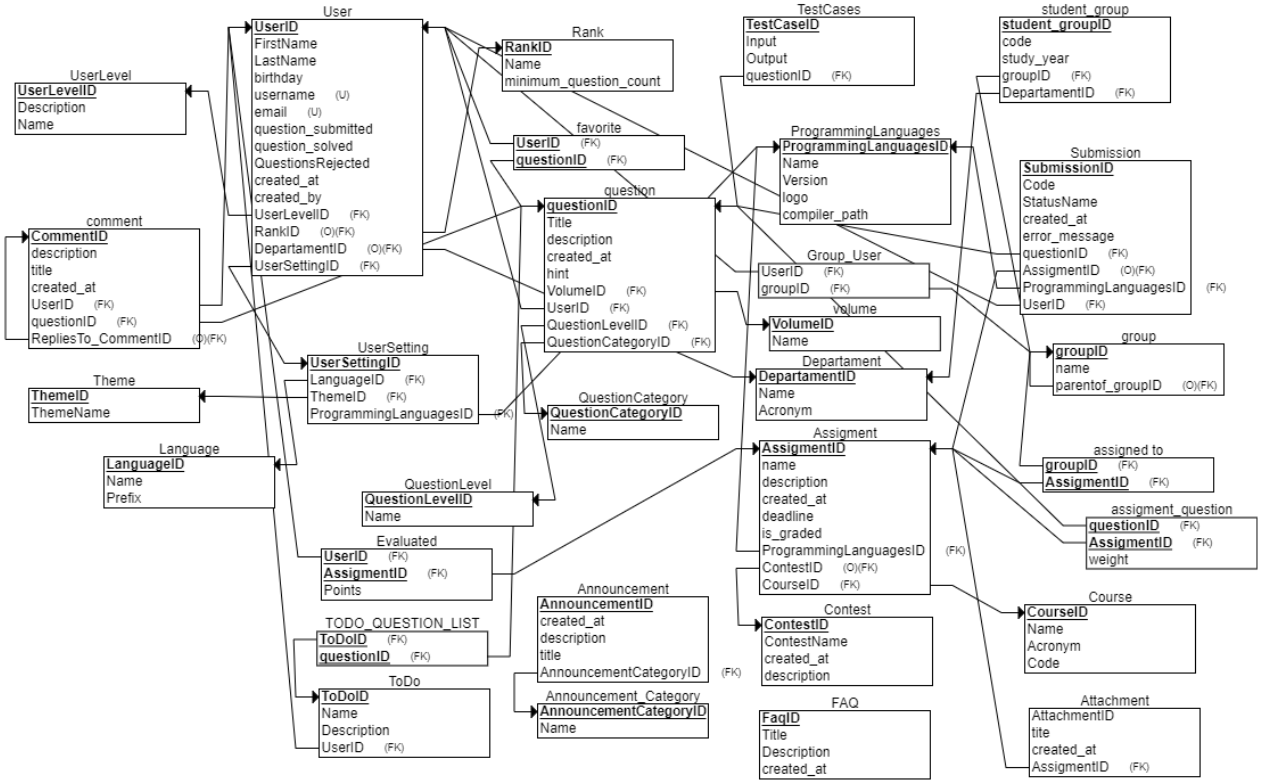

Figure 2. Relational Schema (RS)

EERD (Enhanced Entity Relationship Diagram)

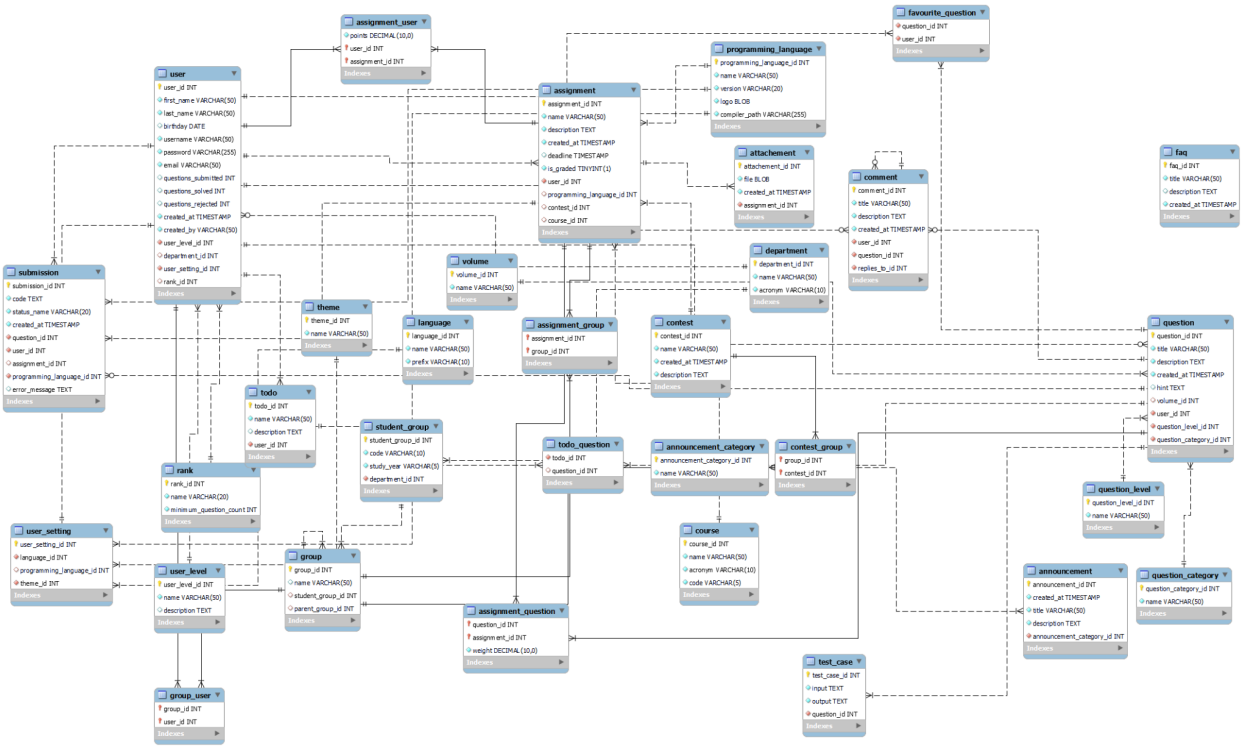

Figure 3. Enhanced Entity Relationship Diagram (EERD)

SQL

DDL, DML

You can find them in APPENDIX A Section with all other sources.

DQL (Managerial Queries) as Views

1. Fetches all announcements from category 2 (patches).

```
CREATE VIEW `Announcements for Patches` AS
SELECT announcement_id,
title as 'Announcement Title',
`description` AS 'Announcement Description',
`name` AS 'Category Name'
FROM announcement AS an
JOIN announcement_category AS an_cat
ON an.announcement_category_id = an_cat.announcement_category_id
WHERE an.announcement_category_id = 2;
```

2. Fetches all assignments per Group

```
CREATE VIEW `Assignments per Group` AS
SELECT assignment.assignment_id,
`group`.group_id,
assignment.`name` AS 'Assignment',
`group`.`name` AS 'Group'
FROM assignment_group, assignment, `group`
WHERE assignment_group.assignment_id = assignment.assignment_id
AND assignment_group.group_id = `group`.group_id;
```

3. Fetches from user table specific user based on credentials

```
CREATE VIEW `Authentication` AS
SELECT user_id, `first_name`, `last_name`
FROM `user`
WHERE `password` =
'$2a$12$D3sepOMfL5NAYBDA8Nfk1OS/4uxtYCTekD2JDJZ3gmEsNbbqSuRw.'
AND username = 'ibreti';
```

4. Displays all average points per group.

```
CREATE VIEW `Average points per group` AS SELECT group_id, `name`,
AVG(points)
FROM (
      SELECT u.user_id, g.group_id, `name`
      FROM group_user AS g_u
      JOIN `user` AS u
      ON g_u.user_id = u.user_id
      JOIN `group` AS g
      ON g.group_id = g_u.group_id
      WHERE g.student_group_id IS NOT NULL
) AS g_u
JOIN assignment_user
ON assignment_user.user_id = g_u.user_id
GROUP BY g_u.group_id;
```

5. Fetches favorite questions of a user.

```
-- Get favorite questions of a user
CREATE VIEW `Favorite question of user` AS
SELECT u.user_id,
CONCAT(u.first_name, ' ', u.last_name) AS `User Full Name`,
q.question_id,
q.title AS `Question`
FROM `user` AS u
JOIN favourite_question as fav
ON u.user_id = fav.user_id
JOIN question as q
ON fav.question_id = q.question_id
WHERE u.user_id = 2
AND CONCAT(u.first_name, ' ', u.last_name) = 'Drin Karkini';
```

6. Fetches most preferred programming language.

```
-- Most preferred programming language
CREATE VIEW `Most preferred programming language` AS
SELECT p.programming_language_id,
p.`name` AS 'Programming Language',
COUNT(p.programming_language_id) AS 'Number of uses'
FROM `user` AS u
JOIN user_setting AS u_s
ON u.user_setting_id = u_s.user_setting_id
JOIN programming_language AS p
ON u_s.programming_language_id = p.programming_language_id
GROUP BY p.programming_language_id
ORDER BY COUNT(p.programming_language_id) DESC LIMIT 1;
```

7. Fetches top 3 most used themes.

```
-- Top 3 used themes
CREATE VIEW `Top 3 preferred themes` AS
SELECT t.theme_id,
t.`name` AS 'Theme',
COUNT(t.theme_id) AS 'Number of uses'
FROM `user` AS u
JOIN user_setting AS u_s
ON u.user_setting_id = u_s.user_setting_id
JOIN theme AS t
ON u_s.theme_id = t.theme_id
GROUP BY t.theme_id
ORDER BY COUNT(t.theme_id) DESC
LIMIT 3;
```

8. Fetches all points for a specific group for an assignment

```
-- Take all points for a specific group for an assignment
CREATE VIEW `Points for specific assignment and specific group` AS
SELECT `user`.user_id, CONCAT(`user`.first_name, " ", `user`.last_name) AS
'User Full Name',
assignment.assignment_id,
assignment.`name` AS 'Assignment Name',
`group`.group_id,
`group`.`name` AS 'Group name',
points
FROM assignment, `group`, assignment_group, assignment_user, `user`
WHERE `user`.user_id = assignment_user.user_id
AND `group`.group_id = (
      SELECT group_id
      FROM `group`
      WHERE `name` = 'CEN 2020 B'
)
AND assignment.assignment_id = (
      SELECT assignment_id
      FROM assignment
      WHERE `name` = 'Homework week 1'
);
```

9. Fetches Questions count per question level

```
-- Questions count per question level
CREATE VIEW `Questions count per question level` AS
SELECT qlev.question_level_id,
qlev.`name` AS 'Level',
COUNT(q.question_id) AS 'Number of questions'
FROM question AS q
JOIN question_level AS qlev
ON q.question_level_id = qlev.question_level_id
GROUP BY qlev.question_level_id;

CREATE VIEW `Questions For Assignments` AS
SELECT question.question_id, `name` AS 'Assignment Name',
title AS 'Question',
assignment.`description` AS 'Assignment Description',
question.`description` AS 'Question Description'
FROM assignment, question, assignment_question
WHERE question.question_id = assignment_question.question_id
AND assignment.assignment_id = assignment_question.assignment_id;
```

10. Fetches Replies to a comment

```sql
-- Replies to a comment
CREATE VIEW `Replies to comment` AS
SELECT u.user_id,
CONCAT(u.first_name, ' ', u.last_name) AS `User Full Name`,
c.comment_id,
c.title AS 'Comment Head',
c.`description` AS 'Comment Body',
q.question_id,
q.title AS 'Question'
FROM `comment` AS c
JOIN `user` AS u
ON c.user_id = u.user_id
JOIN question AS q
ON c.question_id = q.question_id
WHERE c.replies_to_id = 1;
```

11. Fetches Student count per Department.

```sql
CREATE VIEW `Student count per Department` AS
SELECT `user`.department_id, `name`, COUNT(*) as members
FROM `user`, department
WHERE `user`.department_id = department.department_id
GROUP BY `user`.department_id;
```

12. Fetches Submissions 1 day before assignment creation

```sql
CREATE VIEW `Submissions 1 day before assignment creation` AS
SELECT u.user_id,
CONCAT(first_name, ' ', last_name) AS 'User Full Name',
`name` AS 'Assignment Name',
TIMESTAMPDIFF(HOUR, a.created_at, s.created_at) AS 'Hour Difference'
FROM `user` AS u
JOIN submission AS s
ON U.user_id = s.user_id
JOIN assignment AS a
ON s.assignment_id = a.assignment_id
WHERE TIMESTAMPDIFF(HOUR, a.created_at, s.created_at) < 24;
```

13. Fetches submissions after the deadline.

```
CREATE VIEW `Submissions after deadline` AS
SELECT u.user_id,
CONCAT(first_name, ' ', last_name) AS 'User Full Name',
`name` AS 'Assignment Name',
TIMESTAMPDIFF(HOUR, s.created_at, a.deadline) AS `Hour Difference`
FROM `user` AS u
JOIN submission AS s
ON u.user_id = s.user_id
JOIN assignment AS a
ON s.assignment_id = a.assignment_id
WHERE TIMESTAMPDIFF(HOUR, s.created_at, a.deadline) >= 0;
```

14. Fetches Test cases of question

```
CREATE VIEW `Test cases of question` AS
SELECT t_c.test_case_id,
t_c.input,
t_c.output,
q.question_id,
q.title AS 'Question'
FROM test_case AS t_c
JOIN question AS q
ON t_c.question_id = q.question_id
WHERE q.question_id = 1;
```

15. Fetches all To Do's with at least 5 questions

```
CREATE VIEW 'To Do's with at least 5 questions` AS
SELECT t.todo_id,
t.`name` AS 'TODO Name',
t.`description` AS 'TODO Description',
u.user_id,
CONCAT(u.first_name, ' ', u.last_name) AS 'User Full Name',
COUNT(q.question_id) AS 'Number of questions'
FROM todo_question AS t_q
JOIN todo as t
ON t_q.todo_id = t.todo_id
JOIN question AS Q
ON t_q.question_id = q.question_id
JOIN `user` AS u
ON t.user_id = u.user_id
GROUP BY t.todo_id
HAVING COUNT(q.question_id) >= 5;
```

16. Fetches Top 5 solved volumes

```
-- Top 5 accepted volumes
CREATE VIEW `Top 5 solved volumes`AS
SELECT v.volume_id,
v.`name` AS 'Volume',
COUNT(DISTINCT q.question_id, s.status_name, s.user_id) AS 'Number of
accepted submissions'
FROM submission AS s
JOIN question AS q
ON s.question_id = q.question_id
JOIN volume AS v
ON q.volume_id = v.volume_id
WHERE s.status_name = 'Accepted !'
GROUP BY v.volume_id
ORDER BY COUNT(DISTINCT q.question_id, s.status_name, s.user_id) DESC
LIMIT 5;
```

17. Fetches User with most solved questions

```
-- User with most solved questions
CREATE VIEW `Top 10` AS
SELECT user_id,
CONCAT(first_name, " ", last_name) AS 'User Full Name',
questions_solved FROM `user`
WHERE questions_solved IS NOT NULL
ORDER BY questions_solved DESC, CONCAT(first_name, " ", last_name) ASC
LIMIT 10;
```

18. Fetches User and rank

```
CREATE VIEW `User and rank` AS
SELECT u.user_id,
CONCAT(u.first_name, ' ', u.last_name) AS 'User Full Name',
r.`name` AS 'Rank'
FROM `user` AS u
JOIN `rank` AS r
ON u.rank_id = r.rank_id
ORDER BY r.rank_id DESC, CONCAT(u.first_name, ' ', u.last_name) ASC;
```

19. Fetches Users, groups and contests

```
-- Users by groups by contest
CREATE VIEW `Users groups and contests` AS
SELECT u.user_id,
CONCAT(u.first_name, ' ', u.last_name) AS `User Full Name`,
g.group_id,
g.`name` AS 'Group',
c.contest_id,
c.`name` AS 'Contest'
FROM `user` AS u
JOIN group_user AS g_u
ON u.user_id = g_u.user_id
JOIN `group` as g
ON g_u.group_id = g.group_id
JOIN contest_group AS c_g
ON g.group_id = c_g.group_id
JOIN contest AS c
ON c_g.contest_id = c.contest_id
ORDER BY CONCAT(u.first_name, ' ', u.last_name) ASC;
```

Procedures

```
CREATE PROCEDURE UpdateAssignmentDeadline(
      IN assignment_id INT,
   IN new_deadline TIMESTAMP
)
BEGIN
      UPDATE assignment AS a
   SET a.deadline = new_deadline
   WHERE a.assignment_id = assignment_id;
END
```

```
CREATE PROCEDURE UpdateQuestionDescription(
      IN question_id INT,
   IN new_question_description TEXT
)
BEGIN
      UPDATE question AS q
   SET q.`description` = new_question_description
   WHERE q.question_id = question_id;
END
```

```
CREATE PROCEDURE UpdateUserGroup(
      IN user_id INT,
   IN old_group_id INT,
   IN new_group_id INT
)
BEGIN
      UPDATE group_user AS g_u
   SET g_u.group_id = new_group_id
   WHERE g_u.user_id = user_id
   AND g_u.group_id = old_group_id;
END
```

```
CREATE PROCEDURE UpdateUserPassword(
      IN user_id INT,
   IN `password` VARCHAR(255)
)
BEGIN
      UPDATE `user` AS u
   SET u.`password` = `password`
   WHERE u.user_id = user_id;
END
```

```
CREATE PROCEDURE UpdateUserRank(
      IN user_id INT,
   IN new_rank_id TEXT
)
BEGIN
      UPDATE `user` AS u
   SET u.rank_id = new_rank_id
   WHERE u.user_id = user_id;
END
```

Research and Idea Consolidation

Below we explained all the improvements

Improvements applied

We considered different optimizations and improvements for our database.
First, we looked at normalization, as mentioned earlier. By normalizing our database up to 3NF, we noticed the following benefits:

- Elimination of redundant data: Repeated (redundant) entries were removed from the tables of the database, allowing for smaller amounts of data to be used.
- Data consistency: Having removed redundant data, we replaced them with references (FK) to the needed tables. This allowed us to apply changes on a single table and reflect it on all tables depending on the changed information, thus making our database more consistent.
- Better and quicker execution: We have reduced the number of columns per table and increased the number of tables on our database. This allows us to query only the data and tables we need, thus making our queries faster.
- Greater database organization: Having split our data into specific tables, we have provided a logical separation of attributes, thus making it easier for us to create a mental model for how the system works. This allows for an increased understanding of the system and ease of operation with it.

We were careful to pick integer primary keys for our tables, allowing for more efficient indexing. Looking at attributes, we tried to design our tables such that we reduce the number of optional attributes to a minimum. In our effort to avoid storing optional data, we preferred non-null foreign keys when creating relational tables. This increased the efficiency of the database, as it removed null checks for each optional column we query. Furthermore, this reduced the number of null fields on the tables, thus effectively filling the table entries.
Moving forward, we decided to invest in a higher number of tables stored. As previously mentioned, this allowed us to make some specific queries faster, given the tables have a reduced number of columns. Another benefit of this approach was that we can trivially collect analytical data from our tables, without extra overhead.
Since we have many tables with 1:M relationships, in order to make the queries slightly more efficient we used joins to eliminate the **N+1** select problem. We have detected some cases where we selected a number of rows using O/R [Outright (foreign exchange transactions)] naive implementation, in which it required **N+1** selection statements (steps)

in order to fetch the same data. This causes more queries to be executed and has an impact on the performance.
But note that this may not be a good idea since reducing the join statements in queries is beneficial since some statements with a poorly designed pattern that involves a lot of joins may not work well.

Database tools and SQL plugin allows the developer to query and manage databases when the database connection is created. They provide support for all the features that are available in DataGrip, the standalone database management environment for developers. With the plugin, you can query, create and manage databases. Databases can work locally, on a server, or in the cloud. The plugin supports MySQL which we have used.
MySQL tops the list of robust transactional database engines available on the market. With features like complete atomic, consistent, isolated, durable transaction support, multi-version transaction support, and unrestricted row-level locking, it is the go-to solution for full data integrity. Many of the world's largest and fastest-growing organizations including Facebook, Twitter, Booking.com, and Verizon rely on MySQL to save time and money powering their high-volume Web sites, business-critical systems and packaged software.

Advantages of Using MySQL

- MySQL server has a much more sophisticated ALTER TABLE.
- MySQL server has support for tables without transactions for applications that need all the speed they can get. The tables may be memory-based, HEAP tables or disk based MyISAM.
- MySQL server has support for two different table handlers that support transactions, InnoDB, and BerkeleyDB. Because every transaction engine performs differently under different conditions, this gives the application writer more options to find an optimal solution for his or her setup, if need be per individual table.
- MERGE tables gives you a unique way to instantly make a view over a set of identical tables and use these as one. This is perfect for systems where you have log files that you order, for example, by month.
- The option to compress read-only tables, but still have direct access to the rows in the table, gives you better performance by minimizing disk reads. This is very useful when you are archiving things.
- MySQL server has internal support for full-text search.

- With MySQL you can access many databases from the same connection (depending, of course, on your privileges).
- MySQL server is coded from the start to be multi-threaded, while PostgreSQL uses processes. Context switching and access to common storage areas is much faster between threads than between separate processes. This gives MySQL server a big speed advantage in multi-user applications and also makes it easier for MySQL server to take full advantage of symmetric multiprocessor (SMP) systems.
- MySQL server has a much more sophisticated privilege system than PostgreSQL. While PostgreSQL only supports INSERT, SELECT, and UPDATE/DELETE grants per user on a database or a table, MySQL server allows you to define a full set of different privileges on the database, table, and column level. MySQL server also allows you to specify the privilege on host and user combinations.
- **On-Demand Scalability** - MySQL offers unmatched scalability to facilitate the management of deeply embedded apps using a smaller footprint, even in massive warehouses that stack terabytes of data. On-demand flexibility is the star feature of MySQL. This open-source solution allows complete customization to eCommerce businesses with unique database server requirements
- **Round-the-Clock Uptime** - MySQL comes with the assurance of 24×7 uptime and offers a wide range of high-availability solutions, including specialized cluster servers and master/slave replication configurations.
- **Complete Workflow Control** -With an average download and installation time of less than 30 minutes, MySQL means usability from day one. Whether your platform is Linux, Microsoft, Macintosh or UNIX, MySQL is a comprehensive solution with self-management features that automate everything from space expansion and configuration to data design and database administration.
- **Scalability and Flexibility** - The MySQL database server provides the ultimate in scalability, sporting the capacity to handle deeply embedded applications with a footprint of only 1MB to running massive data warehouses holding terabytes of information. Platform flexibility is a stalwart feature of MySQL with all flavors of Linux, UNIX, and Windows being supported. And, of course, the open source nature of MySQL allows complete customization for those wanting to add unique requirements to the database server.
- **High Performance** - A unique storage-engine architecture allows database professionals to configure the MySQL database server specifically for particular applications, with the end result being amazing performance results. Whether the intended application is a high-speed transactional processing system or a high-volume web site that services a billion queries a day, MySQL can meet the most demanding performance expectations of any system. With high-speed load utilities, distinctive memory caches, full text indexes, and other

performance-enhancing mechanisms, MySQL offers all the right ammunition for today's critical business systems.

- **High Availability** - Rock-solid reliability and constant availability are hallmarks of MySQL, with customers relying on MySQL to guarantee around-the-clock uptime. MySQL offers a variety of high-availability options from high-speed master/slave replication configurations, to specialized Cluster servers offering instant failover, to third party vendors offering unique high-availability solutions for the MySQL database server.
- **Robust Transactional Support** - MySQL offers one of the most powerful transactional database engines on the market. Features include complete ACID (atomic, consistent, isolated, durable) transaction support, unlimited row-level locking, distributed transaction capability, and multi-version transaction support where readers never block writers and vice-versa. Full data integrity is also assured through server-enforced referential integrity, specialized transaction isolation levels, and instant deadlock detection.
- **Web and Data Warehouse Strengths** - MySQL is the de-facto standard for high-traffic web sites because of its high-performance query engine, tremendously fast data insert capability, and strong support for specialized web functions like fast full text searches. These same strengths also apply to data warehousing environments where MySQL scales up into the terabyte range for either single servers or scale-out architectures. Other features like main memory tables, B-tree and hash indexes, and compressed archive tables that reduce storage requirements by up to eighty-percent make MySQL a strong standout for both web and business intelligence applications.
- **Strong Data Protection** - Because guarding the data assets of corporations is the number one job of database professionals, MySQL offers exceptional security features that ensure absolute data protection. In terms of database authentication, MySQL provides powerful mechanisms for ensuring only authorized users have entry to the database server, with the ability to block users down to the client machine level being possible. SSH and SSL support are also provided to ensure safe and secure connections. A granular object privilege framework is present so that users only see the data they should, and powerful data encryption and decryption functions ensure that sensitive data is protected from unauthorized viewing. Finally, backup and recovery utilities provided through MySQL and third party software vendors allow for complete logical and physical backup as well as full and point-in-time recovery.
- **Comprehensive Application Development** - One of the reasons MySQL is the world's most popular open source database is that it provides comprehensive support for every application development need. Within the database, support can be found for stored procedures, triggers, functions, views, cursors, ANSI-standard

SQL, and more. For embedded applications, plug-in libraries are available to embed MySQL database support into nearly any application. MySQL also provides connectors and drivers (ODBC, JDBC, etc.) that allow all forms of applications to make use of MySQL as a preferred data management server. It doesn't matter if it's PHP, Perl, Java, Visual Basic, or .NET, MySQL offers application developers everything they need to be successful in building database-driven information systems.

- **Management Ease** - MySQL offers exceptional quick-start capability with the average time from software download to installation completion being less than fifteen minutes. This rule holds true whether the platform is Microsoft Windows, Linux, Macintosh, or UNIX. Once installed, self-management features like automatic space expansion, auto-restart, and dynamic configuration changes take much of the burden off already overworked database administrators. MySQL also provides a complete suite of graphical management and migration tools that allow a DBA to manage, troubleshoot, and control the operation of many MySQL servers from a single workstation. Many third party software vendor tools are also available for MySQL that handle tasks ranging from data design and ETL, to complete database administration, job management, and performance monitoring.
- **Open Source Freedom and 24 x 7 Support** - Many corporations are hesitant to fully commit to open source software because they believe they can't get the type of support or professional service safety nets they currently rely on with proprietary software to ensure the overall success of their key applications. The questions of indemnification come up often as well. These worries can be put to rest with MySQL as complete around-the-clock support as well as indemnification is available through MySQL Enterprise. MySQL is not a typical open source project as all the software is owned and supported by Oracle, and because of this, a unique cost and support model are available that provides a unique combination of open source freedom and trusted software with support.
- **Lowest Total Cost of Ownership** - By migrating current database-drive applications to MySQL, or using MySQL for new development projects, corporations are realizing cost savings that many times stretch into seven figures. Accomplished through the use of the MySQL database server and scale-out architectures that utilize low-cost commodity hardware, corporations are finding that they can achieve amazing levels of scalability and performance, all at a cost that is far less than those offered by proprietary and scale-up software vendors. In addition, the reliability and easy maintainability of MySQL means that database administrators don't waste time troubleshooting performance or downtime issues, but instead can concentrate on making a positive impact on higher level tasks that involve the business side of data.

Mysql + Redis

Mysql and **Redis** are both well known DBMS. We are familiar with MySQL and relational databases, but what is Redis?
Redis (Nosql) is an in-memory data structure store, used as a distributed, in-memory key–value database, cache and message broker, with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indices.

But why should we use these 2 different DBMS paradigms together?

As mentioned above Redis is an in-memory database since it makes it good for caching. Sometimes frequently used data we need to cache somewhere for faster retrieval. Since MySQL stores data in the harddrive it takes a longer time to retrieve them so that's the case when Redis comes into play.

One of the things that's so great about Redis is how well it can complement and extend other databases in your ecosystem. While replacing legacy backend databases with newer ones is often seen as expensive and risky, they don't easily scale in the linear fashion required for users' 'instant experience' expectations.

Some of the things that make it difficult to accommodate modern application needs using a traditional MySQL architecture are:

1. Read/write speeds of traditional databases are not good enough for use cases such as session stores.
2. Introducing new tables or modifying an existing schema can be extremely complex, which makes adding new features and applications extremely difficult.
3. Legacy databases are limited by the number of operations you can perform per second and by the number of concurrent connections you can have. As a result, you end up having to invest in more database instances – increasing your infrastructure and maintenance costs.

On the other hand, in-memory databases like Redis can help you future-proof and increase the value of your investment. For instance, if your application stores data in MySQL or any other relational database, you can easily insert Redis as a front-end database in between the application and MySQL. By designing look-aside and write-through caching solutions, session stores and rate-limiters with Redis, you'll boost performance, speed innovation, scale with fewer resources and ultimately deliver the best experience for your users.

Redis as a ‘System of Engagement’

Redis’ in-memory key-value data store can deliver low latency responses for your users because it’s extremely fast and flexible, and includes built-in data structures (e.g. Lists, Hashes, Sets, Sorted Sets, Bitmaps, Hyperloglog, and Geospatial Indices) that perform some data operations more efficiently and effectively than relational databases. We recommend using Redis behind the data access layer as a ‘system of engagement’ to store the hot data users engage with, while designating your MySQL as the ‘system of record.’

To avoid creating a bottleneck in their application, database or network layers, many developers use Redis for the following use cases:

Cache: This provides a tiered model for memory access, in which applications store common, repeatedly read objects in Redis. Caching helps applications retrieve data quickly and limit the load on the database server.

Session Store: In all interactive apps, the server maintains a unique session for each active user. Rather than relying on MySQL-like relational databases to persist session data, a single cluster of Redis on decently sized servers with sufficient RAM can manage thousands, if not millions of sessions.

Real-time Analytics: Gamification through leaderboards, dashboards, polls, messages, counters and other real-time aggregators require constant processing and communication with end-users. Redis’ powerful and highly efficient data structures enable you to collect, process and dissipate millions of simultaneous activities or objects to thousands of active users in real time.

Metering: Redis can also help developers cost-effectively manage the load on legacy servers during peak usage times by rate limiting the number of calls applications make every few seconds.

In addition to the examples above, Redis can be used as a message broker, data structure store and temporary data store for a variety of use cases. Essentially, Redis gets your data closer and faster to your end user, while collecting their data more quickly. Taking things to the next level, Redis Enterprise offers high availability, in-memory replication, auto-scaling and re-sharding, along with leading-edge CRDT-based active-active support for distributed databases and built-in Redis modules such as RediSearch, ReJSON, Rebloom and Redis Graph.

Thanks to this powerful database, today's 'instant experience' demands can be met without ripping out your legacy solutions. Using a bit of creativity, you can instead leverage Redis to balance the need for business continuity and time-to-market with the benefits of superior performance, flexibility, extensibility and scalability.

Query Speed Measuring

Below are provided the measurements for some of the queries.

It took **0.535555 seconds** to create the database and the tables.

Managerial Queries Speed

1. 0.016 seconds
2. > 0.001 seconds
3. > 0.001 seconds
4. 0.016 seconds
5. 0.016 seconds
6. 0.016 seconds
7. > 0.001 seconds
8. 0.016 seconds
9. 0.016 seconds
10. > 0.001 seconds
11. 0.015 seconds
12. 0.015 seconds
13. > 0.001 seconds
14. > 0.001 seconds
15. 0.015 seconds
16. 0.014 seconds
17. 0.015 seconds
18. 0.016 seconds
19. > 0.001 seconds
20. 0.016 seconds

The speed of the queries are measured using MySQL Workbench. As we can see the queries executed very fast. These results are not affected only by the design of sql statements or database design. Depending on the hardware and the amount of data you may get different results.