COL362 - Project - Milestone 2

Databoys - Tanish Gupta, Rishi Jain, Daksh Khandelwal

17th April 2023

Contents

1 Introduction

We plan to develop a clone of StackExchange (specifically the CS domain - the original can be found here). In this milestone, we write the database part of our application i.e. the queries that will be used to run our application. We present all these queries, indexes that we use to boost their performance, the new features that we add over and above the existing website, and database size and performance parameters.

2 Some Added Features

On top of the existing website, we plan to add some new features to it.
Firstly, we create a new role on stackexchange, which will have admin powers. An admin is allowed to:

1. Delete an account of a user (if (s)he finds the user to be violating any terms and conditions, or the user has been reported by many other users)
2. Delete some specific post(s) if they do not follow the standards.
3. Edit any question/comment to meet the expectations of the platform.
4. Have powers similar to that of the owner of the post including marking an answer as accepted, marking a question as duplicate etc.

Another interesting feature that we add is that users can follow other users. This will help build the "feed" (home page) for the user. We show content of the users that the current user follows on the feed. (The feed may also show the posts you follow. We also show other type of content including interesting questions etc. All of these different types of feeds can be selected using a drop-down menu box).

We also recommend new people to follow. This is done using an SQL query that figures out people whom the user doesn't follow already, has some common interest in tags, and who has recently used the platform (We use the lastaccessdate parameter for this).

We can also recommend new people in other ways.

3 Challenges

1. Since many of the data including password and some email IDs are not publically available due to the privacy reasons, we are unable to provide authentication with the existing data. For this we have some fix. If we register a new user, (s)he can signup with a password, and then use it as to login too. All users also have another way to login, which is based on their account ID and display name.
2. StackExchange internally implements some ML algorithms to find relevant questions to the text being typed by the user (Similar to a google search). We do not include this feature (as we feel this is not needed for the purposes of this course). Instead, we allow the user to either search for a question using its ID, or enter a text which we match directly to the contents of the question.

4 Who will be the application users?

4.1 Users

Users first need to login to make the best use of the platform. Without logging in, they can search for a question, view answers and not do much other than that. But if a user is logged in, (s)he can ask questions, answer the questions of other people, follow people, upvote questions, follow posts (we added this features as found on Piazza, if you follow a post, you get regular updates). Users can add votes, can earn badges, setup their profiles (tell people about themselves - like an Instagram Bio, or follow tags they are interested in, fetch questions that are unanswered). The exact semantics of checking if a user is logged in or not are defered until the next milestone.

4.2 Admin

An admin has some super-powers that a normal user doesn't have. Firstly, there is no option to signup for an admin. An admin may chose upto 1 more user to make admin. Admin may decide to delete a post (a question or an answer) or a comment if it finds (s)he finds it to be violating any of the terms and conditions. Admin may perform normal user tasks too. (S)He may ask questions, may answer questions, add comments, earn badges etc. An admin is the one who assigns badges to users. Admin is like the root user of the application who has many more priviledges than a normal user.

5 Some Features

5.1 User's View of the System

- **Welcome Page**

  1. This will be the home page for everyone. It will contain three buttons - login and signup buttons for users, and an admin login button.
  2. The home page will also contain some most recently asked questions (even if they have been answered) - along with some popular tags (popular being defined by the number of posts with that tag).
  3. The home page will also have a search bar to search for questions using text.
  4. Clicking on any question redirects to the question page.
  5. Click on login, signup or admin login buttons redirect on their respective pages.

- **Login Page**

  1. To login a user, we have 2 options. Since the passwords for the data downloaded are not available, we allow those users to login using their account ID and display name. For all the new users, we ask for passwords, which they may alternately use to log in. We use the below query to alter the definiton of our relation accordingly.

```
1
2   ALTER TABLE users
    ADD COLUMN Password VARCHAR(255);
3
```

  2. Once we have the details, we check if the user with such username and password combination exists in the DB, and if yes, then return that entry.
  3. Note that if we do not find such an entry, we do not give more information to the user that if the username was not found or the password was incorrect. We do this so that any person trying to maliciously log in does not get any additional information about the username.
  4. Once a user is successfully able to log in, we redirect him to a (richer) home page.

- **Sign Up Page**

  1. The sign up page will ask for some of the user details that will be used to create an account in the database.
  2. If the user already exists (there are heuristics to decide this) - then we do not let new user register.
  3. On successful registration, the user will be either redirected to home page or to the login screen (This decision we shall take in the next milestone).

- **Profile Page**

  1. The profile page will house information about the logged in user.
  2. This information will be the one that is stored in the DB.
  3. There will also be an option to edit this info, if required.

- **Question Page**

  1. There will be a page for every question - which will have the question, the answers and comments (if any). This will be similar to the one on the original website.

2. We will also show similar questions on this page. (Similar will be defined by similar tags).
3. There will be options to upvote a question, upvote an answer or a comment, add an answer or a comment, or to follow a post. We will try to include more features in the next milestone to make it more similar to the original design.

- **Question Revisions Page**

1. Just like we have for Piazza, we plan to also show the revisions to a particular post.
2. For the frontend, the user will be able to see the history of the post, the revisions made etc.
3. For the backend, we will query the posthistory table, and show all the revisions of the post.

- **Admin Login Page**

1. This will be a simple page demanding the necessary information for an admin login. Nothing fancy here.
2. On successful login, admin will be redirected to the home page.

- **Admin Home Page**

1. This page is different from the user home page because an admin has some super-powers which are not available to a normal user.

- **Tags Page**

1. This page will contain some information about some of the popular tags (popular again being defined by the number of posts it is part of).
2. Clicking any of the tags will redirect user to a questions page with a filter on tag.
3. The tags page will also have a search bar similar to a questions search bar to search for a tag with a name similar to the one being searched for.
4. There will be three options to search for tags. We can ask for most popular tags, or tags in the sorted order (by their name), or the newest tags (with a simple assumption that newer tags have larger ID number).

- **Add a new Question Page**

1. This page will be to add a new question.
2. It will ask for relevant fields including title, question content, tags, etc. and add a new question in the posts table in the database.

- **Users page**

1. This page will show other users of the platform.
2. We plan to show users based on many parameters. We can show users on the basis of reputation, the users that are admins etc.
3. We may also show recommended users to follow. This will be done based on similar tags interests etc.

6 Database-specific Functionality

6.1 Constraints

Following constraints are imposed on our database design apart from primary key constraints:

- All the foreign key constraints - for example the relations which have a `userid` attribute must have corresponding entry in the users table i.e. there must exist some user with that value of `userid`.
- The `count` attribute of a tag must be equal to the actual number of occurences of that tag in the `Posts` table.

6.2 Access Privileges

We can categorize the people who access our application into 3 types:

- Person with no account : We will call them non-users
- Person with an account : We will call them users
- Admins

Following are the access privileges for each category of people:

- Non-users: They are allowed to see and read posts (questions , answers , comments) but they are not allowed to post anything. Neither are they allowed to comment or vote on any post.
- users: They can post, comment,vote, answer and ask questions.
- Admins: They can delete accounts of users who do not meet or violate the Terms and Conditions of the platform. Admin is the one who assigns badges to users based on some pre-defined criteria.

7 List of Queries

In this milestone submission, we submit the broad overview of queries we have planned that will drive our application. There might be other small subtleties and minor queries that might be written afterwards to present more information to a user.

Here, we present queries that will help render each page of the website, and also the queries that can be used for special functionality.

7.1 Login and Signup pages

7.1.1 Query 1

Depending on whether the user already exists or not, either we successfully login the user or we take him/her to the signup page. In case the login is successful, the `LastAccessDate` attribute of `users` table is updated to reflect the current date. This can be represented by the SQL query shown below.

```
DECLARE username VARCHAR(255) = 'johndoe'
    accid INT = 99
    pass VARCHAR(255) = 'samplepass'
IF EXIST(SELECT * FROM users WHERE DisplayName = @username and (AccountId = @accid or Password = @pass) )
BEGIN
    Print 'User login successful'
    UPDATE users
    SET LastAccessDate = CURRENT_DATE
    WHERE DisplayName = @username and (AccountId = @accid or Password = @pass)
END
ELSE
BEGIN
    INSERT INTO users (AccountId, Reputation , Views, DownVotes,UpVotes, DisplayName,Location, WebsiteUrl, AboutMe, CreationDate ,LastAccessDate, Password) VALUES(@accid, 0, 0,0,0,@username , NULL, NULL,NULL,CURRENT_DATE,CURRENT_DATE , @pass )
END
```

7.1.2 Query 2

The corresponding query for admin login will be slightly different because we do not permit admin sign-up.

```
DECLARE adminname VARCHAR(255) = 'johndoe'
    pass VARCHAR(255) = 'samplepass'
IF EXIST(SELECT * FROM admins WHERE DisplayName = @adminname and Password = @pass)
BEGIN
    Print 'Admin login successful'
    UPDATE admins
    SET LastAccessDate = CURRENT_DATE
```

```
        WHERE  DisplayName = @adminname and Password = @pass
    END
    ELSE
    BEGIN
        Print 'Admin sign up not permitted'
    END
```

7.2 Welcome Page

7.2.1 Query 1

- This page will initially have the interesting answered questions.
- The query for fetching the interesting answered questions will be:

```
WITH RelevantPosts
AS ( SELECT Id, Score, OwnerUserId
FROM posts
WHERE PostTypeId = 1 AND CreationDate > :date AND AcceptedAnswerId IS NOT NULL
),

upvoteCount AS
(
SELECT RP.Id, RP.Score, RP.OwnerUserId, Count(*) AS upvoteCount
FROM RelevantPosts AS RP
LEFT OUTER JOIN (
    SELECT PostId, VoteTypeId
    FROM votes
    WHERE VoteTypeId = 2)AS V
ON RP.Id = V.PostId
GROUP BY RP.Id, RP.score, RP.owneruserid
),

downvoteCount AS
(
SELECT RP.Id, RP.Score, RP.OwnerUserId, Count(*) AS downvoteCount
FROM RelevantPosts AS RP
LEFT OUTER JOIN (
    SELECT PostId, VoteTypeId
    FROM votes
    WHERE VoteTypeId = 3) AS V
ON RP.Id = V.PostId
GROUP BY RP.Id, RP.score, RP.owneruserid
),

AllVotes AS(
SELECT upvoteCount.Id, upvoteCount.Score, upvoteCount.OwnerUserId, upvoteCount.upvoteCount,
downvoteCount.downvoteCount
FROM upvoteCount, downvoteCount
WHERE upvoteCount.Id = downvoteCount.Id
)

SELECT AllVotes.Id, users.Reputation/10 + AllVotes.upvoteCount*10 + AllVotes.downvoteCount*10
+ AllVotes.Score*50 AS totalScore
FROM AllVotes, users
WHERE AllVotes.OwnerUserId = users.Id
ORDER BY totalScore DESC
LIMIT 100;
```

The above query first fetches the relevant posts (posts that are questions, have been created after a certain date and have an accepted answer). Then it fetches the upvote count and downvote count for each of these posts. Then it calculates the total score for each of these posts. Finally, it sorts the posts based on the total score and returns the top 100 posts. The criterion for toal Score is: Reputation/10 + upvoteCount*10 + downvoteCount*10 + Score*50.

We have used the reputation of the user as a factor in the total score because we want to give more weightage to the users who have a good reputation.

We can index CreationDate and PostTypeId in the posts table to speed up the query.We can also index the PostId and VoteTypeId in the votes table to speed up the query.

7.2.2 Query 2

- If a user is logged in, they can also get the unanswered questions of their fellow users who have answered their questions in the past.
- The query is as follows:

```
With helpers as (
SELECT post2.OwnerUserId, COUNT(post2.OwnerUserId) AS answerCount
FROM (SELECT * FROM posts WHERE OwnerUserId = :userId AND PostTypeId = 1) AS post1
JOIN (SELECT * FROM posts WHERE PostTypeId = 2) AS post2
ON post1.Id = post2.ParentId
GROUP BY post2.OwnerUserId
ORDER BY answerCount DESC
LIMIT 100
)

SELECT Id FROM posts WHERE OwnerUserId in (SELECT OwnerUserId FROM helpers) AND PostTypeId = 1
ORDER BY CreationDate DESC;
```

The above query first fetches the users who have answered the most questions of the user with id :userId. Then it fetches the most recent questions asked by these users.
We can index the OwnerUserId and PostTypeId in the posts table to speed up the query.

7.2.3 Query 3

- Each post will have votes associated with it. These votes will also be displayed on the home screen.
- The query for fetching the votes for a post will be:

```
WITH upvoteCount AS (
SELECT Count(*) AS upvoteCount
FROM votes
WHERE PostId = :postId AND VoteTypeId = 2
),

downvoteCount AS (
SELECT Count(*) AS downvoteCount
FROM votes
WHERE PostId = :postId AND VoteTypeId = 3
)

SELECT upvoteCount, downvoteCount
FROM upvoteCount, downvoteCount;
```

The above query fetches the upvote count and downvote count for a post with id postId.

We can index the PostId and VoteTypeId in the votes table to speed up the query.

7.2.4 Query 4

- On the side we will also show the leaderboard of the users. The query for fetching the leaderboard will be:

```
WITH acceptedAnswerCount AS (
SELECT post2.OwnerUserId, COUNT(post2.OwnerUserId) AS answerCount
FROM (
SELECT ParentId, Id, OwnerUserId FROM posts WHERE PostTypeId = 2 AND CreationDate > '
2019-01-01') AS post2
INNER JOIN (SELECT Id, AcceptedAnswerId, OwnerUserId FROM posts WHERE PostTypeId = 1) AS post1
ON post2.ParentId = post1.Id WHERE post2.Id = post1.AcceptedAnswerId
GROUP BY post2.OwnerUserId
),

```

```
    answerCount AS (
    SELECT post2.OwnerUserId, COUNT(post2.OwnerUserId) AS answerCount
    FROM Posts AS post2 WHERE post2.PostTypeId = 2 AND post2.CreationDate > '2019-01-01'
    GROUP BY post2.OwnerUserId
    ),

    questionCount AS (
    SELECT OwnerUserId, COUNT(OwnerUserId) AS questionCount
    FROM posts
    WHERE PostTypeId = 1 AND CreationDate > '2019-01-01'
    GROUP BY OwnerUserId
    ),

    commentCount AS (
    SELECT UserId, COUNT(UserId) AS commentCount
    FROM comments
    WHERE CreationDate > '2019-01-01'
    GROUP BY UserId
    ),

    totalScore AS (
    SELECT acceptedAnswerCount.OwnerUserId, acceptedAnswerCount.answerCount*40 + answerCount.
    answerCount*10 + questionCount.questionCount*5 + commentCount.commentCount AS totalScore
    FROM acceptedAnswerCount, answerCount, questionCount, commentCount
    WHERE acceptedAnswerCount.OwnerUserId = answerCount.OwnerUserId AND acceptedAnswerCount.
    OwnerUserId = questionCount.OwnerUserId AND acceptedAnswerCount.OwnerUserId = commentCount.
    UserId
    ORDER BY totalScore DESC
    )

    SELECT DisplayName, Id, totalScore
    FROM users
    INNER JOIN totalScore
    ON users.Id = totalScore.OwnerUserId
    ORDER BY totalScore DESC;
```

The above query first fetches the users who have the most accepted answers. Then it fetches the users who have the most answers. Next it queries the database for the users who have the most questions. Then it fetches the users who have the most comments. Finally, it calculates the total score for each of these users. and sorts the users based on the total score (Returns only the top 100 users).

The criterion for total Score is: acceptedAnswerCount*40 + answerCount*10 + questionCount*5 + commentCount.

We can index the CreationDate, OwnerUserId in the posts table to speed up the query. We can also index the CreationDate, UserId in the comments table to speed up the query. Index on the Id in the users table can also be used to speed up the query.

7.2.5 Query 5

- This page will also have options to fetch the most interesting unanswered questions.
- The query for it will be similar to one of the above queries, but here we will not consider the posts that have an "accepted" answer.

```
    WITH RelevantPosts
    AS ( SELECT Id, Score, OwnerUserId
    FROM posts
    WHERE PostTypeId = 1 AND CreationDate > '2019-01-01' AND AcceptedAnswerId IS NULL
    ),

    upvoteCount AS
    (
    SELECT RP.Id, RP.Score, RP.OwnerUserId, Count(*) AS upvoteCount
    FROM RelevantPosts AS RP
    LEFT OUTER JOIN (
        SELECT PostId, VoteTypeId
```

```
        FROM votes
        WHERE VoteTypeId = 2)AS V
    ON RP.Id = V.PostId
    GROUP BY RP.Id, RP.score, RP.owneruserid
    ),

    downvoteCount AS
    (
    SELECT RP.Id, RP.Score, RP.OwnerUserId, Count(*) AS downvoteCount
    FROM RelevantPosts AS RP
    LEFT OUTER JOIN (
        SELECT PostId, VoteTypeId
        FROM votes
        WHERE VoteTypeId = 3) AS V
    ON RP.Id = V.PostId
    GROUP BY RP.Id, RP.score, RP.owneruserid
    ),

    AllVotes AS(
    SELECT upvoteCount.Id, upvoteCount.Score, upvoteCount.OwnerUserId, upvoteCount.upvoteCount,
    downvoteCount.downvoteCount
    FROM upvoteCount, downvoteCount
    WHERE upvoteCount.Id = downvoteCount.Id
    )

    SELECT AllVotes.Id, users.Reputation/10 + AllVotes.upvoteCount*10 + AllVotes.downvoteCount*10
    + AllVotes.Score*50 AS totalScore
    FROM AllVotes, users
    WHERE AllVotes.OwnerUserId = users.Id
    ORDER BY totalScore DESC
    LIMIT 100;
```

7.3 Question Page

7.3.1 Query 1

This query will fetch the similar questions based on the tags of the question.

```
    WITH tagsForGivenQuestion AS (
    SELECT *
    FROM (
        SELECT regexp_split_to_table(tags, '[><]') AS tag
        FROM posts
        WHERE Id = :questionId
    ) AS tags WHERE tag !='';
    )
    SELECT posts.Id, count(*) AS count
    FROM posts, tagsForGivenQuestion
    WHERE posts.Tags LIKE '%' || tagsForGivenQuestion.tag || '%'
    AND posts.Id != :questionId
    GROUP BY posts.Id
    ORDER BY count DESC
    LIMIT 10;
```

First we fetch the tags of the given question. Then we fetch the posts that have the same tags as the given question. Then we count the number of times each post is present in the posts. Finally, we sort the posts based on the count and return the top 10 posts.
We can index the Id in the posts table to speed up the query.

7.3.2 Query 2

We can also upvote or downvote a question which will lead to:

```
--upvote
INSERT INTO votes (Id, PostId, 2, CreationDate, UserId, BountyAmount)
VALUES (:id, :postId, CURRENT_DATE, :userId, :bountyAmount);
--downvote
INSERT INTO votes (Id, PostId, 3, CreationDate, UserId, BountyAmount)
VALUES (:id, :postId, CURRENT_DATE, :userId, :bountyAmount);
```

In similar we can insert a new answer or a new comment.

7.4 Tags Page

This page will just have 2 queries.

7.4.1 Query 1

- To fetch the tags with the most questions. The query for it will be:

```
SELECT TagName, Id, Count FROM tags
ORDER BY Count Desc LIMIT 20;
```

We can index the Count in the tags table to speed up the query.

7.4.2 Query 2

To fetch the tags with the most questions after a given date. The query for it will be: -

```
With RelevantPosts AS (
    SELECT Id, Tags
    FROM posts
    WHERE PostTypeId = 1 AND CreationDate > :date
)
SELECT TagName, tags.Id, Count(*) AS Count
FROM RelevantPosts, tags
WHERE RelevantPosts.Tags LIKE '%' || tags.TagName || '%'
GROUP BY TagName, tags.Id
ORDER BY Count Desc LIMIT 20;
```

First we fetch the posts after a given date. Then we fetch the tags that are present in the posts. Then we count the number of times each tag is present in the posts. Finally, we sort the tags based on the count and return the top 20 tags.
We can index the CreationDate in the posts table to speed up the query.
We can also index the Id in the tags table to speed up the query.

7.5 Admin Homepage

Here we list the additional features available to the admin.

7.5.1 Query 1

- Admin can check the user signup rates for different months and years. This will help montior the growth and usage of the platform over time.
- This is the query -

```
select Year(CreationDate) year,
   Month(CreationDate) month,
   count(*) users
from Users
group by Year(CreationDate), Month(CreationDate)
order by Year(CreationDate), Month(CreationDate);
```

7.5.2 Query 2

- Admin can delete posts with score < -5.
- Score is the difference of the number of upvotes and downvotes. Note that since we define `ON DELETE CASCADE` in our relation definition, so deletion of a post implies deletion of tuples in the relation that references the deleted post.
- The following is the query -

```
DELETE *
FROM posts
WHERE score <= -5;

```

7.6 Profile Page

This is a simple page that will have 2 very specific features - one to view profile and the other to edit profile. (We may also display other information, but that will be same as homepage, so we don't include those queries again here).

7.6.1 Query 1

- To fetch the profile of the user, we use a very simple command:

```
SELECT * FROM users
WHERE users.id = 5;

```

The user id = 5 is just a placeholder in the above query. The exact semantics will be based on the front-end and back-end integration, when python will be used to replace that placeholder value (coming from the user session).

7.6.2 Query 2

- The other query to edit the profile of the user is also just simple:

```

UPDATE users
SET
    DisplayName = 'myName',
    Location = 'newLocation',
    websiteUrl = 'myUrl',
    aboutMe = 'This is me :)'
WHERE users.id = 5;


```

We can only allow a user to change few of the fields, as mentioned above. If the user doesn't want to change some of the fields, we simply pass in the old value.

7.7 Users Page

This page will display other users of the platform. There are many parameters on the basis of which we may show users.

7.7.1 Query 1

- We can show other users who have the maximum reputation.

```

SELECT * FROM users
ORDER BY reputation
DESC LIMIT 50;


```

7.7.2 Query 2

- Similar to above, we may also show users that are admins on our platform.

```

SELECT * FROM admins LIMIT 50;


```

7.7.3 Query 3

- We also plan to recommend new users that can be followed. This can be done on various heurisitics.
- For now, we stick to this:

```
WITH v1 AS
(
    SELECT person_id, tag_id FROM Person_like_tag WHERE person_id = 5
),

v2 AS
(
    SELECT person_id, tag_id FROM Person_like_tag WHERE person_id != 5 AND tag_id IN (SELECT tag_id FROM v1)
),

v3 AS
(
    SELECT DISTINCT person_id FROM v2 WHERE person_id NOT IN (SELECT person_id_followed FROM Person_follows_person WHERE person_id = 5)
),

v4 AS
(
    SELECT id, lastaccessdate FROM users WHERE id IN (SELECT person_id FROM v3) ORDER BY lastaccessdate DESC LIMIT 10
),

v5 AS
(
    SELECT id FROM v4
)

SELECT * FROM users
WHERE id IN (SELECT id FROM v5);


```

We suggest new people to follow that the user doesn't already follow based on interests in tags (i.e. they should have atleast one common interest in a tag) and lastaccessdate for that user is latest (top 10 for now).

7.8 New Question page

This page will be used to add new questions. Only logged in users will be able to access this page. Note that we increment the count of the corresponding tags (listed in the question) in the `tags` table.

7.8.1 Query 1

```
DECLARE owneruserid INTEGER := 99
        title TEXT := "sampletitle"
        contentlicense TEXT := "CC BY-SA 4.0"
        body TEXT := "samplebody"
        tagsss := ["tag1", "tag2", "tag3", "tag4"]

FOR r in tagsss:
    UPDATE tags
    SET Count = Count + 1
    WHERE tags.TagName = r

INSERT INTO posts(OwnerUserId, LastEditorUserId, PostTypeId, AcceptedAnswerId, Score, ParentId, ViewCount, AnswerCount, CommentCount, OwnerDisplayName, LastEditorDisplayName, Title, Tags, ContentLicense, Body, FavoriteCount, CreationDate, CommunityOwnedDate, ClosedDate, LastEditDate, LastActivityDate)
VALUES
    (@owneruserid,@owneruserid,1,NULL,0,NULL,0,0,0, (SELECT DisplayName from users where Id = @owneruserid),(SELECT DisplayName from users where Id = @owneruserid), @title, @tagss, @contentlicense, @body, NULL,CURRENT_DATE,NULL,NULL, CURRENT_DATE, CURRENT_DATE);
```

7.9 Questions Revisions Page

Similar to Piazza (where we can see the history of posts), we can also see history of posts on our platform.

7.9.1 Query 1

- Given a post, we would like to show all of its history. This history can be found in the postHistory Table.
- The exact way of processing the history will be taken care by the backend. We just query the database for the history of that particular post.

```
1
2    SELECT * FROM post_history
3    WHERE postid = 5;
4
5
```

This contain multiple rows according to the revisions. For a single edit (for example, changing the title) - the table records 2 rows, one the old title, and other the new title, both with the same RevisionGUID. We defer the semantics of handling these output tuples for the next milestone.

No more functionality is needed for this page.

7.10 Edit Post Page

- To edit a post, we need to make 2 changes.

7.10.1 Query 1

- Firstly, we need to update the post in the posts table. (If some of the parameters need not be changed, then just fill in the old values in the update query)

```
1    UPDATE posts
2    SET lasteditoruserid = 5, tags = '<newTags>', body = 'This is new body', title = 'This is
      new title', lasteditdate = CURRENT_DATE
3    WHERE id = 5;
4
```

7.10.2 Query 2

- Secondly, after doing this, we also need to add (a min of) 2 new tuples to the postHistory table (where we record the old and the new edited post).
- The basic structure of this query looks like -

```
1    INSERT INTO
2    post_history (id, posthistorytypeid, postid, revisionguid, creationdate, userid,
     userdisplayname, comment)
3    VALUES (0, 1, 5, '1', CURRENT_DATE, 5, '', 'This is a comment');
4
```

7.11 Other Basic Queries and Uses of Tables

This section includes some other queries that will be used to drive the application in general. We mention these queries relation wise for better understanding purposes:

7.11.1 Badges

The badges table stores the badges that users earn. For every badgeid, there is a user to whom it belongs, and what kind of a badge it is. Only admins can generate new badges and assign them to the user.

1. To create a new badge -

```
    INSERT INTO
        badges (id, userid, class, name, tagbased, date)
    VALUES
        (1, 5, 1, 'Compiler', TRUE, '2018-01-01');

```

2. To get badges for a particular user -

```
    SELECT * FROM badges
    WHERE userid = 5;

```

3. Gold Badges : We assign gold badges to users who have atleast 50 accepted answers.

```
    with t1(personid , ansid) as
    (
        select a.OwnerUserId , a.Id
        from posts a
        where a.PostTypeId = 2
    ),
        t2(personid, ansid) as
    (
        select personid , ansid
        from t1 , posts b
        where ansid = b.AcceptedAnswerId
    ),
        t3(personid, countans) as
    (
        select personid, count(ansid)
        from t2
        group by personid
    )
        select personid from t3 where countans >= 50;


```

7.11.2 Comments

The comments table, as the name suggests, has all the comments for the posts. Each comment has a unique identifier (its id), the userid of the person who created this comment, the content of the comment, and the postid to which this comment was made.

This table will be only primarily be used for 3 purposes - To retrieve comments, to add comments on a post, or to edit comments.

The retrieve comments and add comments queries have been added on the questions page (as that is where we use them). The edit comments query will be similar to edit question query (by just replacing the typeofpostid)

7.11.3 PostHistory

The PostHistory table stores the information about the history of each posts, and any revisions made to it. Every edit is stored with 2 tuples, 1 recording the old value and another the new value (both these tuples are identified by the same revisionGUID). The posthistorytypeid parameter helps know what edit was made - i.e. was the edit made to the title, comment, question, answer etc. All this information is encoded.

We use this table to show history of each post to the user (Just like Piazza does). So, the only 2 operations this table supports is insert and fetch. The fetch query has been included in the "Question Revisions Page" section above and the insert query has been added in the "Edit Post Page" above.

7.11.4 PostLinks

This table contains posts that have some external links. These links can only be of 2 types, either to mark the post duplicate or to refer to some external post.

The following query is used to fetch all the linkedposts to the given post (This will also be used on the questions page while rendering the question) -

```
SELECT * FROM posts
WHERE id IN (SELECT relatedpostid FROM post_links WHERE postid = 5);
```

7.11.5 Posts

The Posts Table, again as the name suggests, has the data about all the posts.

Each post has a unique ID. The ownerUserID (or the ownerUsername, if the person who created the post was a guest) is stored. The columns lasteditoruserid (or lasteditorusername) are stored along with the lasteditdate to check for the updates to the post. Every post has a posttypeid attribute which indicates the type of post (question or answer only for our purposes). Each post may have an accepted answer and answer count (if it is a question), a score, a parentid (if it is an answer), viewcount, commentcount. Finally, it also has the content - the title, the body and the tags.

Most of the queries involving posts have been included above. Some other queries can be:

1. Fetch all (here, any 10) posts by a user

```
SELECT * FROM posts
WHERE owneruserid = 5 LIMIT 10;

```

2. Fetch all (here, any 10) posts by a user with a specific tag

```
SELECT * FROM posts
WHERE owneruserid = 5 AND tags LIKE  '%<compilers>%' LIMIT 10;

```

3. Fetch all posts that a user has voted on

```
SELECT * FROM posts
WHERE id IN (SELECT postid FROM votes WHERE userid = 5);

```

4. Fetch all posts that a user has voted on with a specific tag

```
SELECT * FROM posts
WHERE id IN (SELECT postid FROM votes WHERE userid = 5) AND tags LIKE '%<compilers>%';

```

5. Retrieve latest questions for a tag

```
SELECT * FROM posts
WHERE tags LIKE '%<compilers>%' AND posttypeid = 1
ORDER BY creationdate DESC LIMIT 10;


```

6. Retrieve all tags for a post

```
SELECT tags FROM posts WHERE id = 5;

```

7. Unanswered questions of the user ordered by creation date

```
select *
from posts p
where p.OwnerUserId = 5 and p.PostTypeId = 1 and (p.AnswerCount = 0 or p.AnswerCount =
NULL)
order by p.CreationDate desc;

```

8. Returns the number of accepted answers for a user

```
with t1(ansid) as
(
    select Id
    from posts
    where OwnerUserId = 1 and PostTypeId = 2
)
    select count(ansid)
    from t1 , posts
    where ansid = AcceptedAnswerId and PostTypeId = 1;

```

7.11.6 Tags

The tags table contains the description of the tags that can be used in posts. This table maintains a tag ID, along with its tag name, and the number of posts it is a part of.

1. We can retrieve all information about any particular tag using its id.

```
 SELECT * FROM Tags
 WHERE id = 5;

```

2. We can also add a new tag that we wish to see in the table.

```
INSERT INTO
Tags (Id, TagName, Count, ExcerptPostId, WikiPostId)
VALUES (0, 'test', 1, 1, 1);

```

3. We can return the most popular tags by using the count attribute.

```

SELECT * FROM Tags
ORDER BY Count DESC LIMIT 50;


```

7.11.7 Users

The users table hosts the information about the users of the platform. The userID is a unique identifier in this table. It also has an accountID attribute (which is not very relevant for our purposes, but we anyhow descibe it). Every user has a different ID on each of the stackexchange platforms, but has a common accountID across all platforms to uniquely identify the user. This table also has the reputation of users, which we use in one of the queries above. Each user has some upvotes and downvotes, a website URL, and an about me section. The creationdate of the user account and the last access date to the platform are also stored (updated when the user logs in).

Most of the queries involving users have been included above. We mention here some for the sake of completeness:

1. Fetch users that have the maximum reputation. (This is similar to a codeforces feature where we can see the Top Contributors on the side bar.

```
SELECT * FROM users
ORDER BY reputation
DESC LIMIT 100;

```

2. Fetch and update user profiles. These queries have been used above.

7.11.8 Votes

The votes table maintains all the votes to the posts. It has a postid attribute to refer to the post this vote belongs to. The votetypeid parameter can take can integer value, which means the following -

- 1 = AcceptedByOriginator
- 2 = UpMod (AKA upvote)

- 3 = DownMod (AKA downvote)
- 4 = Offensive
- 5 = Favorite (AKA bookmark; UserId will also be populated) feature removed after October 2022 / replaced by Saves
- 6 = Close (effective 2013-06-25: Close votes are only stored in table: PostHistory)
- 7 = Reopen
- 8 = BountyStart (UserId and BountyAmount will also be populated)
- 9 = BountyClose (BountyAmount will also be populated)
- 10 = Deletion
- 11 = Undeletion
- 12 = Spam
- 15 = ModeratorReview (i.e., a moderator looking at a flagged post)
- 16 = ApproveEditSuggestion

Some extra queries can be -

1. Fetch all posts that a user has voted on

```
    SELECT * FROM posts
    WHERE id IN (SELECT postid FROM votes WHERE userid = 5);

```

2. Retrieve all posts that a user has voted on with a specific tag

```
    SELECT * FROM posts
    WHERE id IN (SELECT postid FROM votes WHERE userid = 5) AND tags LIKE '%<compilers>%';

```

We also created some tables ourselves to add more functionality. The details and use of these tables are listed next:

7.11.9 Person_follows_person

This table contains the information about who follows whom.

```
    CREATE TABLE Person_follows_person
    (
        person_id INT,
        person_id_followed INT,
        PRIMARY KEY (person_id, person_id_followed),
        FOREIGN KEY (person_id) REFERENCES users(id),
        FOREIGN KEY (person_id_followed) REFERENCES users(id)
    );
```

We use this table to show questions asked by people you follow in your own feed. We also recommend new people to follow (using some approach that has been already described above). Though we don't include this, but this feature could also be extended to implement a chat feature.

We can also write some basic queries:

1. Retrieve the list of all people that the user follows

```

    SELECT * FROM Person_follows_person
    WHERE person_id = 5;


```

2. Retrieve the number of people I am being followed by

```
1
2     SELECT COUNT(*) FROM Person_follows_person
3     WHERE person_id_followed = 5;
4
5
```

7.11.10 Person_follows_post

This table contains the information about the posts that are being followed by a user. This information can be used to send notification to a user when a new activity is seen on that post (We do not implement this notification feature).

```
1 CREATE TABLE Person_follows_post
2 (
3     person_id INT,
4     post_id INT,
5     PRIMARY KEY (person_id, post_id),
6     FOREIGN KEY (person_id) REFERENCES users(id),
7     FOREIGN KEY (post_id) REFERENCES posts(id)
8 );
```

7.11.11 Person_like_tag

The person_like_tag table contains the information about the tags a particular user is interested in. This information can be used to personalize his/her feed, or to send him/her notifications when a new question on that tag is asked, or to also recommend him/her new people to follow who have the same interests.

```
1 CREATE TABLE Person_like_tag
2 (
3     person_id INT,
4     tag_id INT,
5     PRIMARY KEY (person_id, tag_id),
6     FOREIGN KEY (person_id) REFERENCES users(id),
7     FOREIGN KEY (tag_id) REFERENCES tags(id)
8 );
```

The query to recommend people has been implemented above. The feed personalization thing is also done above. We do not implement the notification part for the purposes of this project.

7.12 Admins Table

We have created a table to store the information about the moderators of the platform.

```
1  DROP TABLE IF EXISTS admins;
2  CREATE TABLE admins (
3
4      Id INT NOT NULL PRIMARY KEY,
5      DisplayName VARCHAR(255) NOT NULL,
6      WebsiteUrl VARCHAR(255),
7      CreationDate TIMESTAMP NOT NULL,
8      LastAccessDate TIMESTAMP NOT NULL,
9      Password VARCHAR(255) NOT NULL
10
11 );
```

We will also give the privilege to the user to write own SQL queries and run them on our database (Just like the original stackexchange does). There will be heuristics to determine this too - for example, we need to make sure that no data breach happens etc. But all this can be handled by the BackEnd, and this milestone doesn't need to care about that.

8 Index Choices

```sql
-- Create relational indexes for Stack Overflow
CREATE INDEX IX_UserId ON Badges(UserId);
CREATE INDEX IX_PostIdComment ON Comments(PostId);
CREATE INDEX IX_PostIdLinks ON post_links(PostId);
CREATE INDEX IX_Posts ON Posts(Id);
CREATE INDEX IX_PostIdVotes ON Votes(PostId);
CREATE INDEX IX_PostIdHistory ON Post_History(PostId);
CREATE INDEX IX_TagNameInd  ON Tags(TagName);
CREATE INDEX IX_PLT ON Person_like_tag (person_id);
CREATE INDEX IX_PersonFollowsPost ON Person_follows_post (person_id);
CREATE INDEX IX_PersonFollowsPerson ON Person_follows_person (person_id_followed);
```

8.1 Badges Table

For badges table it is best to index on userId. This is because of the fact that whenever we want to access a badge we search for it using the user on which badge was awarded.

8.2 Comments Table

For comments table it is best to index on postId. Similar to badges comment are also accessed for a post.

8.3 PostLinks Table

For postlinks table it is best to index on postId and relatedPostId. This is because of the fact that whenever we want to access a postlink we search for it using the post on which postlink was created.

8.4 Posts Table

In most of the Queries we join other tables with post table based on the postId. Therefore it is best to index on postId.

8.5 Votes Table

For votes table it is best to index on postId. This is because of the fact that whenever we want to access a vote we search for it using the post on which vote was cast.

8.6 Tags Table

Tags table is only used to find the tags for a post. So it is best to index on TagName because we can directly get the tagname from the post and then search for it in the tags table.

8.7 Posts Table

In most of the Queries we join other tables with post table based on the postId. Therefore it is best to index on postId.

8.8 PostHistory Table

For posthistory table it is best to index on postId. This is because of the fact that whenever we want to access a posthistory we search for it using the post on which posthistory was created.

8.9 Person_like_tag Table

For *person_like_tag* table it is best to index on *person_id*. In most of the queries we search for the tags liked by a person. Therefore it is best to index on *person_id*.

8.10 Person_follows_post Table

For *person_follows_post* table it is best to index on *person_id*. In most of the queries we search for the posts followed by a person. Therefore it is best to index on *person_id*.

8.11 Person_follows_person Table

For *person_follows_person* table it is best to index on *person_id_followed*. In most of the queries we search for the persons followed by a person. Therefore it is best to index on *person_id_followed*.

9 Database Size and Performance Analysis

Table Name	Number of tuples (rows)	Number of attributes (columns)	Size (in MB)
Badges	171919	6	9.6
Comments	187994	8	50.4
PostHistory	343014	10	245.3
PostLinks	14330	5	0.651
Posts	100484	22	136.1
Tags	670	5	0.018
Users	130823	12	22.4
Votes	394951	6	16.2

Table 1: Database Size

Data Dump Size = 491.6 MB

Page Name	Query Number	Time taken (without index)	Time taken (with index) (in ms)
Login and Signup	1	68.096	50.263
Login and Signup	2	10.021	8.307
Questions	1	799.997	790.375
Questions	2	3.089	2.987
Tags	1	2.049	1.988
Tags	2	5665.367	5602.89
Welcome	1	1200.53	496.226
Welcome	2	295.161	292.984
Welcome	3	50.748	0.273
Welcome	4	979.983	982.632
Welcome	5	1742.118	1705.584
Profile	1	8.728	0.902
Profile	2	10.176	1.533
Users	1	37.820	38.696
Users	3	2.078	15.260
Questions Revision	1	216.574	12.312
Edit Post	1	13.115	8.806
Edit Post	2	10.323	2.615
Admin Homepage	1	142.631	135.635
Admin Homepage	2	65.826	43.048

Table 2: Performance Analysis

We include the analysis on our local system. This is because, we didn't have the access to modify the table on the server, and thus couldn't create indices.

```
group_9=> CREATE INDEX IX_UsersId ON Users(Id);
ERROR:  must be owner of table users
group_9=>
```

Figure 1: Permission denied error

We got similar error when we tried to access the table through python script.

The problem lies in the fact that the owner of table is not us.

```
group_9=> \dt
                List of relations
 Schema |          Name           | Type  |  Owner
--------+-------------------------+-------+----------
 public | admins                  | table | group_9
 public | badges                  | table | postgres
 public | comments                | table | postgres
 public | person_follows_person   | table | group_9
 public | person_follows_post     | table | group_9
 public | person_like_tag         | table | group_9
 public | post_history            | table | postgres
 public | post_links              | table | postgres
 public | posts                   | table | postgres
 public | tags                    | table | postgres
 public | users                   | table | postgres
 public | votes                   | table | postgres
(12 rows)
```

Figure 2: Owners

10 ER Diagram

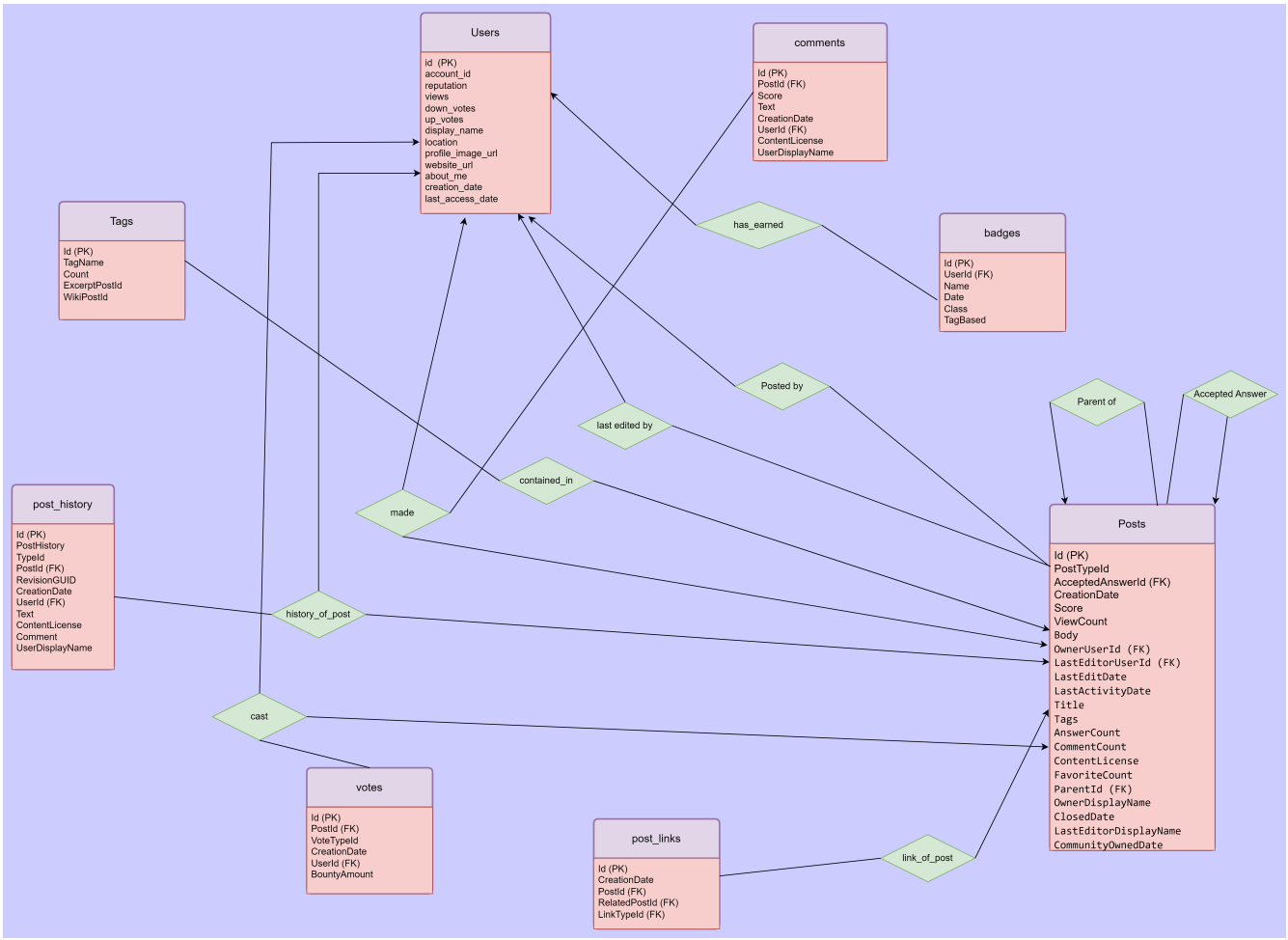

Figure 3: ER Diagram

11 Relational Schema

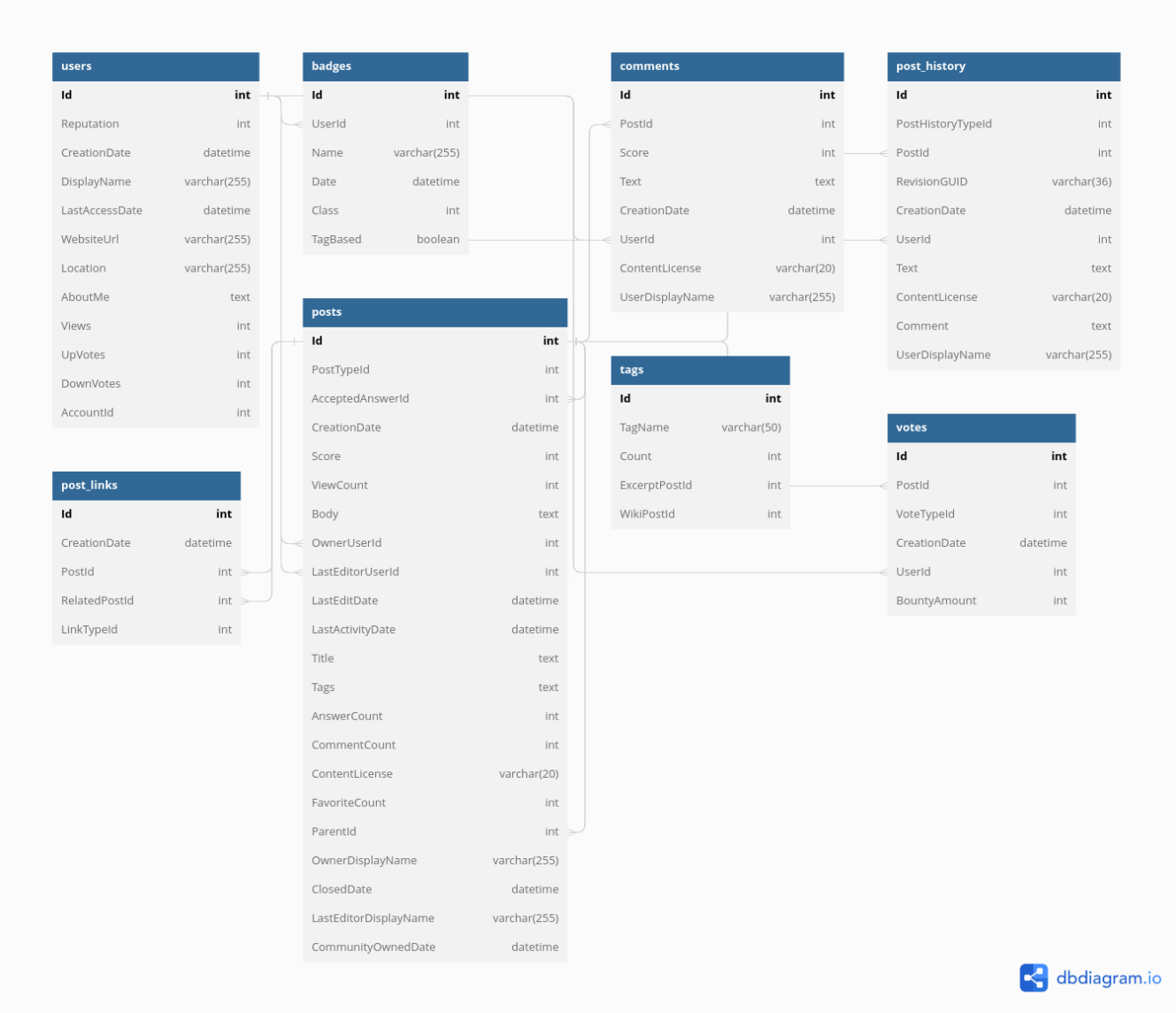

Figure 4: Relational Schema

12 Work Coordination

As the first step, we brainstormed on the features that we plan to offer on our platform. We designed the overall flow, the pages that we display to the users, and the data that will be displayed on these pages. Then, we distibuted the work amongst ourselves based on the pages. Each one of us took a fraction of pages, and wrote all the queries that will be used to render that page, and queries that will give all the functionality to that page.

After this, we came up with queries that could help improve the overall design. We include such queries in the "other basic queries" section above. These queries, though not as significant as other, add functionality and data to our application.

13 Conclusion

For this milestone, we plan the overall design of our platform i.e. what are the exact features that we propose to offer. Corresponding to all those features, we write SQL queries which shall help drive our application. These

queries will fetch and store (or update) the data from (in) our database, that will be displayed on (taken from) the platform. We defer the exact semantics of how the results of these queries will be cleaned up to be displayed on the webpage for the next milestone.