

COL362 - Project - Milestone 3

Databoys - Tanish Gupta, Rishi Jain, Daksh Khandelwal

30th April 2023

Contents

1	Introduction	3
2	Welcome Page	3
2.1	Application Front End Design	3
2.2	Demonstration Scenario	3
2.3	Overall Architecture	4
2.4	Supported Transactions	4
2.4.1	Query 1	4
2.4.2	Query 2	5
2.4.3	Query 3	6
2.4.4	Query 4	6
2.4.5	Query 5	7
3	Login and Signup page	8
3.1	Users	8
3.1.1	Application Front End Design	8
3.1.2	Demonstration Scenario	8
3.1.3	Overall Architecture	9
3.1.4	Supported Transactions	10
3.2	Admin	11
3.2.1	Application Front End Design	11
3.2.2	Demonstration Scenario	11
3.2.3	Overall Architecture	11
3.2.4	Supported Transactions	11
4	Question Page	12
4.1	Application Front End Design	13
4.2	Demonstration Scenario	13
4.3	Overall Architecture	13
4.4	Supported Transactions	15
4.4.1	Query 1	15
4.4.2	Query 2	16
4.4.3	Query 3	16
4.4.4	Query 4	16
5	Tags page	16
5.1	Application fronted	16
5.2	Demonstration Scenario	17
5.3	Overall Architecture	17
5.4	Supported Transactions	17
5.4.1	Query 1	17
5.4.2	Query 2	17

6	Admin Homepage	18
6.1	Application Front End Design	18
6.2	Demonstration Scenario	18
6.3	Overall Architecture	18
6.4	Supported Transactions	18
6.4.1	Query 1	18
6.4.2	Query 2	19
7	Profile page	19
7.1	Application Front End Design	19
7.2	Demonstration Scenario	19
7.3	Overall Architecture	19
7.4	Supported Transactions	19
7.4.1	Query 1	19
7.4.2	Query 2	20
8	Users page	20
8.1	Application Frontend	20
8.2	Demonstration Scenario	20
8.3	Overall Architecture	20
8.4	Supported Transactions	21
8.4.1	Query 1	21
8.4.2	Query 2	21
8.4.3	Query 3	21
9	New Question page	22
9.1	Application Front End Design	22
9.2	Demonstration Scenario	22
9.3	Overall Architecture	22
9.4	Supported Transactions	23
9.4.1	Query 1	23
10	Other Queries	23
10.1	Badges	23
10.2	Comments	24
10.3	PostHistory	24
10.4	PostLinks	24
10.5	Posts	24
10.6	Tags	26
10.7	Users	26
10.8	Votes	26
10.9	Person_follows_person	27
10.10	Person_follows_post	28
10.11	Person_like_tag	28
10.12	Admins Table	28
11	Changes in Data Dump	28
11.1	Ownership issue	28
11.2	Unique ID issue	29
12	ER Diagram	29
13	Relational Schema	30
14	Work Coordination	30
15	Conclusion	31

1 Introduction

We develop a clone of StackExchange (specifically the CS domain - the original can be found [here](#)). In the third, and the final milestone, we write the frontend part of our application (using HTML, CSS and JS), backend using Flask (Python), and connect both of them to deliver a full web application. We mainly focus on the database component of the project - using data-dumps, databases on server, how to execute queries using a backend framework etc. Hence, we present only a simplistic frontend design.

In this report, we discuss the various aspects of our project including the functionality and the user experience with our application. We present this report in different sections according to the different perspective of the three developers. The frontend part, the backend part, and the linking part - the pages and the queries are included above from all three viewpoints. For every page, we describe the frontend design in the "frontend design" subsection, the linking between frontend and backend in the demonstration scenario - which exactly shows what happens after a particular activity, and finally the backend design in the overall architecture - which shows how the requests from frontend are received and how are the responses from backend being sent.

2 Welcome Page

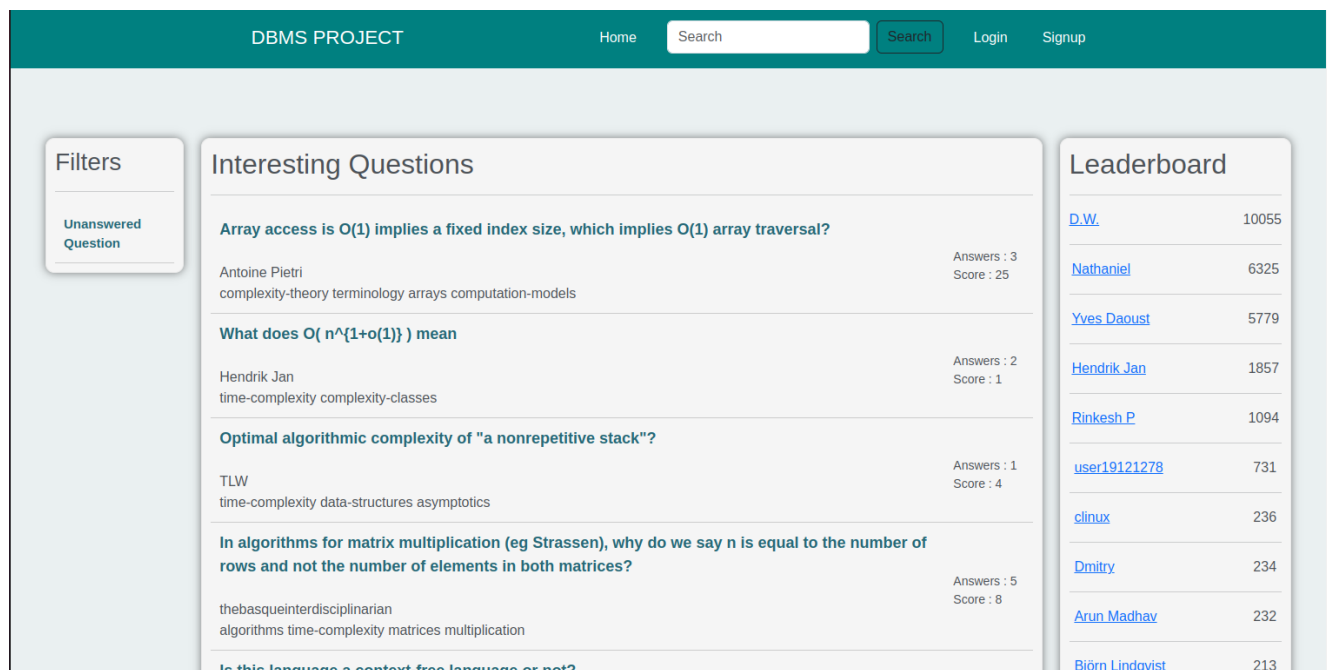


Figure 1: Welcome page

2.1 Application Front End Design

This page consists of interesting answered questions, unanswered questions from fellow users, votes associated with a post, and a leaderboard of the users.

2.2 Demonstration Scenario

This is the homepage for all users irrespective of they are logged in or not. It contains the following features:

- **Interesting Questions** - This section contains the questions which are most viewed and most voted. On clicking the question, the user is redirected to the question page.
- **Filter for Friend Questions** - This section contains the questions which are asked by the users who have answered the questions asked by the logged in user most number of times. On clicking the filter Questions button, the user is redirected to the questions page with the filter applied.

- **LeaderBoard** - This section contains the top users with highest LeaderBoard score. On clicking the user, the user is redirected to the user page of that particular user.
- **Login and Signup** - These buttons are used to login and signup respectively. When a user is logged in, the login button is replaced by the profile button. On clicking the profile button, the user is redirected to the profile page. If a user is already logged in, trying to access the login/signup button with redirect the user to the homepage.

2.3 Overall Architecture

The welcome page for the user (the home page) has a few elements

1. First, it has the interesting questions - which we described in the previous milestone. Every user - irrespective of whether logged in or not, has access to this page.
2. This page also shows the leaderboard according to the score of all the users.

```

1  @app.route('/')
2  def welcome():
3      one_year_ago = datetime.now() - relativedelta(years=1)
4      cursor.execute(queries["welcome"], (one_year_ago,))
5      interesting_answered_questions = cursor.fetchall()
6
7      cursor.execute(queries["leaderboard"], (one_year_ago, one_year_ago, one_year_ago,
8      ↪ one_year_ago, ))
9      leaderboard = cursor.fetchall()
10
11     return render_template('welcome.html',
12     ↪ interesting_answered_questions=interesting_answered_questions, leaderboard=leaderboard,
13     ↪ title = "Interesting Questions")
14

```

3. There is also an option to checkout the unanswered questions. We implemented this in our query using some people known to the user - so we require the user to be logged in to access this page.

```

1  @app.route('/unanswered_questions')
2  @login_required
3  def unanswered_questions():
4
5      cursor.execute(queries["unanswered_questions"], (current_user.id,))
6      unanswered_questions = cursor.fetchall()
7
8      one_year_ago = datetime.now() - relativedelta(years=1)
9      cursor.execute(queries["leaderboard"], (one_year_ago, one_year_ago, one_year_ago,
10     ↪ one_year_ago, ))
11     leaderboard = cursor.fetchall()
12
13     return render_template('welcome.html', unanswered_questions=unanswered_questions,
14     ↪ leaderboard=leaderboard, title = "Unanswered Questions")
15

```

2.4 Supported Transactions

2.4.1 Query 1

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

```

1 WITH RelevantPosts
2 AS ( SELECT Id , Score , OwnerUserId
3 FROM posts
4 WHERE PostTypeId = 1 AND CreationDate > :date AND AcceptedAnswerId IS NOT NULL
5 ),
6
7 upvoteCount AS
8 (
9 SELECT RP.Id , RP.Score , RP.OwnerUserId , Count(*) AS upvoteCount
10 FROM RelevantPosts AS RP
11 LEFT OUTER JOIN (
12     SELECT PostId , VoteTypeId
13     FROM votes
14     WHERE VoteTypeId = 2) AS V
15 ON RP.Id = V.PostId
16 GROUP BY RP.Id , RP.score , RP.owneruserid
17 ),
18
19 downvoteCount AS
20 (
21 SELECT RP.Id , RP.Score , RP.OwnerUserId , Count(*) AS downvoteCount
22 FROM RelevantPosts AS RP
23 LEFT OUTER JOIN (
24     SELECT PostId , VoteTypeId
25     FROM votes
26     WHERE VoteTypeId = 3) AS V
27 ON RP.Id = V.PostId
28 GROUP BY RP.Id , RP.score , RP.owneruserid
29 ),
30
31 AllVotes AS(
32 SELECT upvoteCount.Id , upvoteCount.Score , upvoteCount.OwnerUserId , upvoteCount.upvoteCount ,
33 downvoteCount.downvoteCount
34 FROM upvoteCount , downvoteCount
35 WHERE upvoteCount.Id = downvoteCount.Id
36 )
37
38 SELECT AllVotes.Id , users.Reputation/10 + AllVotes.upvoteCount*10 + AllVotes.downvoteCount*10
39 + AllVotes.Score*50 AS totalScore
40 FROM AllVotes , users
41 WHERE AllVotes.OwnerUserId = users.Id
42 ORDER BY totalScore DESC
43 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 total Score is: $\text{Reputation}/10 + \text{upvoteCount} * 10 + \text{downvoteCount} * 10 + \text{Score} * 50$.

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.

2.4.2 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:

```

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

```

```

9      )
10
11      SELECT Id FROM posts WHERE OwnerUserId in (SELECT OwnerUserId FROM helpers) AND PostTypeId = 1
12      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.

2.4.3 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:

```

1  WITH upvoteCount AS (
2  SELECT Count(*) AS upvoteCount
3  FROM votes
4  WHERE PostId = :postId AND VoteTypeId = 2
5  ),
6
7  downvoteCount AS (
8  SELECT Count(*) AS downvoteCount
9  FROM votes
10 WHERE PostId = :postId AND VoteTypeId = 3
11 )
12
13 SELECT upvoteCount, downvoteCount
14 FROM upvoteCount, downvoteCount;

```

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

2.4.4 Query 4

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

```

1  WITH acceptedAnswerCount AS (
2  SELECT post2.OwnerUserId, COUNT(post2.OwnerUserId) AS answerCount
3  FROM (
4  SELECT ParentId, Id, OwnerUserId FROM posts WHERE PostTypeId = 2 AND CreationDate > '
2019-01-01') AS post2
5  INNER JOIN (SELECT Id, AcceptedAnswerId, OwnerUserId FROM posts WHERE PostTypeId = 1) AS post1
6  ON post2.ParentId = post1.Id WHERE post2.Id = post1.AcceptedAnswerId
7  GROUP BY post2.OwnerUserId
8  ),
9
10 answerCount AS (
11 SELECT post2.OwnerUserId, COUNT(post2.OwnerUserId) AS answerCount
12 FROM Posts AS post2 WHERE post2.PostTypeId = 2 AND post2.CreationDate > '2019-01-01 '
13 GROUP BY post2.OwnerUserId
14 ),
15
16 questionCount AS (
17 SELECT OwnerUserId, COUNT(OwnerUserId) AS questionCount
18 FROM posts
19 WHERE PostTypeId = 1 AND CreationDate > '2019-01-01 '
20 GROUP BY OwnerUserId
21 ),
22
23 commentCount AS (
24 SELECT UserId, COUNT(UserId) AS commentCount
25 FROM comments
26 WHERE CreationDate > '2019-01-01 '
27 GROUP BY UserId
28 ),
29
30 totalScore AS (
31 SELECT acceptedAnswerCount.OwnerUserId, acceptedAnswerCount.answerCount*40 + answerCount.
answerCount*10 + questionCount.questionCount*5 + commentCount.commentCount AS totalScore

```

```

32 FROM acceptedAnswerCount, answerCount, questionCount, commentCount
33 WHERE acceptedAnswerCount.OwnerUserId = answerCount.OwnerUserId AND acceptedAnswerCount.
   OwnerUserId = questionCount.OwnerUserId AND acceptedAnswerCount.OwnerUserId = commentCount.
   UserId
34 ORDER BY totalScore DESC
35 )
36
37 SELECT DisplayName, Id, totalScore
38 FROM users
39 INNER JOIN totalScore
40 ON users.Id = totalScore.OwnerUserId
41 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: $\text{acceptedAnswerCount} * 40 + \text{answerCount} * 10 + \text{questionCount} * 5 + \text{commentCount}$.

2.4.5 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.

```

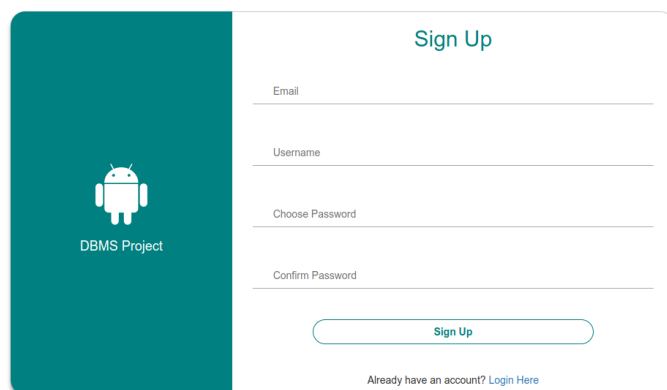
1  WITH RelevantPosts
2  AS ( SELECT Id, Score, OwnerUserId
3  FROM posts
4  WHERE PostTypeId = 1 AND CreationDate > '2019-01-01' AND AcceptedAnswerId IS NULL
5  ),
6
7  upvoteCount AS
8  (
9  SELECT RP.Id, RP.Score, RP.OwnerUserId, Count(*) AS upvoteCount
10 FROM RelevantPosts AS RP
11 LEFT OUTER JOIN (
12     SELECT PostId, VoteTypeId
13     FROM votes
14     WHERE VoteTypeId = 2) AS V
15 ON RP.Id = V.PostId
16 GROUP BY RP.Id, RP.score, RP.owneruserid
17 ),
18
19 downvoteCount AS
20 (
21 SELECT RP.Id, RP.Score, RP.OwnerUserId, Count(*) AS downvoteCount
22 FROM RelevantPosts AS RP
23 LEFT OUTER JOIN (
24     SELECT PostId, VoteTypeId
25     FROM votes
26     WHERE VoteTypeId = 3) AS V
27 ON RP.Id = V.PostId
28 GROUP BY RP.Id, RP.score, RP.owneruserid
29 ),
30
31 AllVotes AS(
32 SELECT upvoteCount.Id, upvoteCount.Score, upvoteCount.OwnerUserId, upvoteCount.upvoteCount,
   downvoteCount.downvoteCount
33 FROM upvoteCount, downvoteCount
34 WHERE upvoteCount.Id = downvoteCount.Id
35 )
36
37 SELECT AllVotes.Id, users.Reputation/10 + AllVotes.upvoteCount*10 + AllVotes.downvoteCount*10
   + AllVotes.Score*50 AS totalScore
38 FROM AllVotes, users
39 WHERE AllVotes.OwnerUserId = users.Id

```

```
40 ORDER BY totalScore DESC
41 LIMIT 100;
```

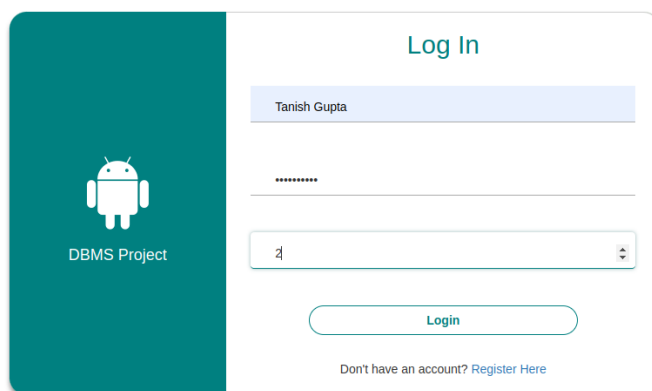
3 Login and Signup page

3.1 Users



The sign up form features a teal sidebar on the left with the Android logo and 'DBMS Project' text. The main white area is titled 'Sign Up' and contains four input fields: 'Email', 'Username', 'Choose Password', and 'Confirm Password'. A teal 'Sign Up' button is positioned below the fields, and a link 'Already have an account? Login Here' is at the bottom.

Figure 2: User Sign up



The login form has a teal sidebar with the Android logo and 'DBMS Project' text. The main white area is titled 'Log In' and includes a text input for 'Tanish Gupta', a password field with masked characters, and a dropdown menu showing '2'. A teal 'Login' button is at the bottom, with a link 'Don't have an account? Register Here' below it.

Figure 3: User login

3.1.1 Application Front End Design

The user is allowed to login or signup depending on whether his/her account exists already or not. For new users we ask the email, username and password. For already existing users - we allow them to login using username and account id or password.

3.1.2 Demonstration Scenario

Login Page :-

This page is used to login to the application. User can enter the username and password to login. If the user is not

registered, he/she can click on the register button to register. On clicking the register button, the user is redirected to the signup page. Also if user doesn't remember the password, he/she can login using his userid. If password doesn't match, it will show an error message.

Signup Page :-

This page is used to register to the application. User can enter the username, email and password to register.. User has to enter display name, about me, location, website url and password to register. If the user is already registered, he/she can click on the login button to login. On clicking the login button, the user is redirected to the login page. On signup entry is added to the user table and the user is redirected to the login page.

3.1.3 Overall Architecture

The login page can only be accessed by users who are not already logged in. If a user is already logged in, we redirect him to the home page of our application. We check this using the flask-login package, which lets us know if the current user is already authenticated.

For the login and signup forms, we use flask-wtf module.

```
1 @app.route('/login.html', methods=['GET', 'POST'])
2 @app.route('/login', methods=['GET', 'POST'])
3 def login():
4
5     if current_user.is_authenticated:
6         return redirect(url_for('welcome'))
7     form = LoginForm()
8
9     if form.validate_on_submit():
10         print("Here")
11         cursor.execute(queries["login_check"], (form.username.data, form.accid.data,
12         ↪ form.password.data,))
13         result = cursor.fetchone()
14
15         if result is None:
16             return redirect(url_for('login')), 401
17         else:
18
19             cursor.execute(queries["login"], (form.username.data, form.accid.data,
20             ↪ form.password.data,))
21             conn.commit()
22
23             #get userid
24             cursor.execute(queries["get_userid"], (form.username.data, form.accid.data,
25             ↪ form.password.data,))
26             userid = cursor.fetchone()[0]
27
28             user = User(userid, form.username.data, form.password.data, form.accid.data)
29             login_user(user)
30             next_page = request.args.get('next')
31             if not next_page or url_parse(next_page).netloc != '':
32                 next_page = url_for('welcome')
33             return redirect(next_page)
34
35     return render_template('login.html', title='Sign In', form=form)
```

Note that we have a next argument here. This argument is added by flask redirect. This is because it may happen that we have redirected the user to the login page when (s)he was trying to access some page that can only be accessed by a logged in user. Therefore, if the login here is successful, we redirect the user to the page (s)he was trying to access. If no such page exists, we simply redirect the user to the home page.

Logging out a user is simple, since we are using the flask-login module. It works in the following manner -

```

1 @app.route('/logout')
2 @login_required
3 def logout():
4     logout_user()
5     return redirect(url_for('welcome'))

```

Ofcourse, to log out a user - (s)he needs to be logged in already. After logging out the user, we redirect to the home page.

The signup page also has a similar functionality.

```

1 @app.route('/signup', methods=['GET', 'POST'])
2 def signup():
3
4     if current_user.is_authenticated:
5         return redirect(url_for('welcome'))
6
7     form = SignupForm()
8
9     if form.validate_on_submit():
10         try:
11             cursor.execute(queries["signup"], (form.displayname.data, form.aboutme.data,
12             ↪ form.location.data, form.websiteurl.data, form.password.data,))
13             conn.commit()
14
15             return redirect(url_for('login'))
16         except Exception as e:
17             print(e)
18             return redirect(url_for('signup'))
19     return render_template('signup.html', title='Sign Up', form=form)

```

We try to sign up the user, and if any error occurs (for example, if the user already exists) - we return the error.

3.1.4 Supported Transactions

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.

```

1 SELECT * FROM users WHERE DisplayName = %s and (AccountId = %s or Password = %s);
2
3 UPDATE users
4 SET LastAccessDate = CURRENT_DATE
5 WHERE DisplayName = %s and (AccountId = %s or Password = %s);

```

For signup, we use -

```

1 INSERT INTO
2 users (DisplayName, AboutMe, Location, WebsiteUrl, Views, UpVotes, DownVotes, CreationDate,
3 LastAccessDate, Reputation, Password)
VALUES (%s, %s, %s, %s, 0, 0, 0, CURRENT_DATE, CURRENT_DATE, 1, %s);

```

3.2 Admin

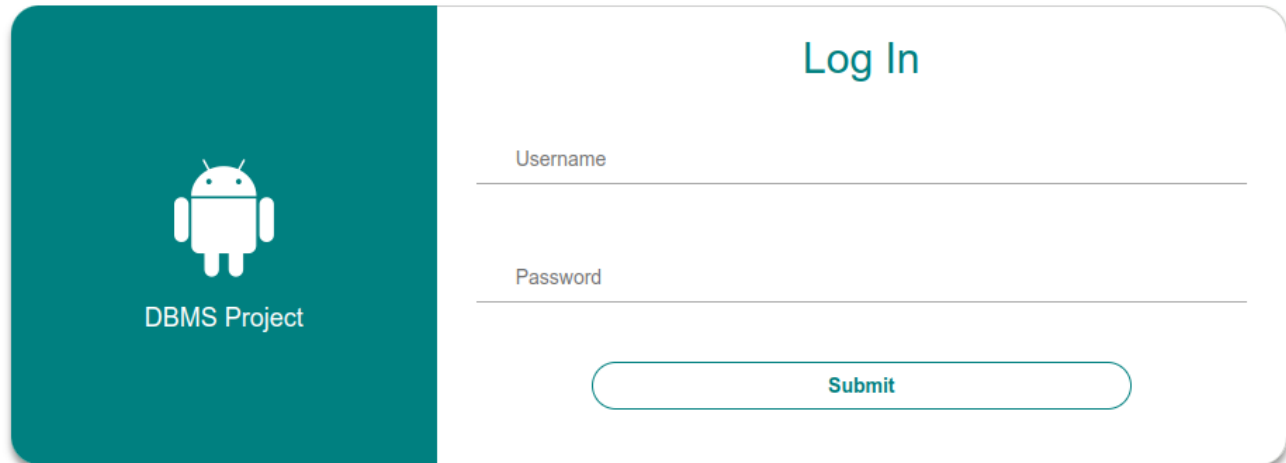
The image shows a login interface for an application. On the left, there is a teal-colored square containing a white Android robot icon and the text "DBMS Project" below it. To the right of this square is a white rounded rectangle. Inside this rectangle, the text "Log In" is displayed in teal at the top right. Below it, there are two input fields: the first is labeled "Username" and the second is labeled "Password". Both labels are in a small, light gray font. At the bottom of the white rectangle, there is a teal-colored rounded button with the word "Submit" in white text.

Figure 4: Admin login

3.2.1 Application Front End Design

For admins, we don't permit the signup functionality. Admin login is allowed and supported by our application - as shown below.

3.2.2 Demonstration Scenario

Similar to the user login page. Admin can login using their Id and password. If password doesn't match, it will show an error message.

3.2.3 Overall Architecture

The login page for the admin will be similar to the normal login page for users. As mentioned, we do not allow sign ups for admins.

3.2.4 Supported Transactions

```
1  SELECT * FROM admin WHERE DisplayName = %s and Password = %s ;
2
3  UPDATE admin
4  SET LastAccessDate = CURRENT_DATE
5  WHERE DisplayName = %s and Password = %s ;
```

4 Question Page

DBMS PROJECT

Home

Search

Search

Login

Signup

Optimal algorithmic complexity of "a nonrepetitive stack"?

I'm wondering about the optimal complexity - or at the very least, some way of achieving non-terrible complexity - of a particular stack variant, that I'm calling a 'nonrepetitive stack'.

A nonrepetitive stack is like an ordinary stack, except that a push that would result in a repeated subsequence fails, not updating the stack but instead returning the location of the subsequence that would be repeated. (To disambiguate, because "repeated subsequence" can mean multiple things: I mean multiple contiguous copies of contiguous subsequences. If you treat the stack as a string, something matching `.*(.)\1.*`.)

(Assume the usual model, e.g. comparing two individual items for equality is $O(1)$.)

The completely naive approach would be to check the entire stack for any repeated subsequences after each push, and undo the push and fail if one is found. Each check, and thus push, appears to be $O(n^3)$ in the current size of the stack, worst-case.

We can do somewhat better by instead noting that the stack can never actually include any repeated substrings (as you cannot introduce a repeated substring by popping from a stack, and pushes that would introduce a repeated substring instead fail), and so a push only needs to check potential repeats that include the top of stack. This gets us down to $O(n^2)$ in the size of the stack per push.

Some (terrible) Python pseudocode for this approach, to illustrate (again: this code is just to illustrate. Please don't focus on the exact code.)

```
def push(s, x):
    s.append(x)
    for i in range(1, len(s)//2 + 1):
        # this comparison is _not_ O(1) time.
        if s[-2*i:-1] == s[-i:]:
            ret = s[-i:], len(s)-2*i, len(s)-i
            s.pop()
            return True, *ret
    return True, *ret
```

Related Questions

[equivalency of some facts in \$\mathcal{O}\$ notation](#)
algorithms time-complexity data-structures asymptotics discrete-mathematics

[Binary Search Complexity](#)
algorithms time-complexity data-structures asymptotics

[Is there a data structure that can find the kth smallest in constant time with logarithmic add and delete operations?](#)
algorithms data-structures time-complexity asymptotics

[Comparing two algorithms for all-pairs shortest paths](#)
algorithms graphs time-complexity data-structures asymptotics

[Optimal lookup complexity when requiring insertion complexity to be at most \$\mathcal{O}\(\log n\)\$?](#)
algorithms data-structures time-complexity asymptotics

Figure 5: Question Page

element and their short-circuits, which works out to $\mathcal{O}(n)$. [i.e. I know that CPython's array comparison with slices I showed doesn't actually work that way.] Best-case for an unsuccessful push is, of course, $\mathcal{O}(1)$.)

Is there a way of doing better here? In particular, is it possible to achieve (amortized) worst-case complexity for pushes that is sublinear in the size of the stack?

Upvote

Downvote

score : 4

Comments

Brainstorming: Can you get anywhere by storing the set of substrings in a [trie](https://en.wikipedia.org/wiki/Trie)? (reading out symbols in the stack starting from the top of stack and proceeding to bottom of stack) Would a dynamic suffix array be useful?

[755](#) 2022-09-30 05:21:50

I've been thinking / attempting to use `_something_` involving storing suffixes or substrings; the major difficulty I keep running into is that "every" suffix `_changes_` every time you add /remove a character. I haven't been able to figure out a way of handling this efficiently, either with a standard suffix tree (or suffix array) or in general.

[19696](#) 2022-09-30 06:54:47

An arbitrary subsequence can occur an arbitrary number of times in the stack without being repetitive. E.g. ``ABaABbABcABdABeABfABgABhABi...``. So even assuming one somehow stores and updates the set of substrings efficiently, one goes to look up ``AB`` and gets $O(n)$ matches that one must somehow sort through. There very well might be a way to get an overall speedup even so; I have as of yet been unable to find one.

[19696](#) 2022-09-30 07:07:49

There are apparently sophisticated algorithms to maintain a suffix array even as the underlying string is changing, with polylog time per update to the string. So if a dynamic suffix array would help, there might be solutions for how to update it. That said, I can't quite tell how to use a suffix array, even if there were a way to update it efficiently.

[755](#) 2022-09-30 07:48:18

Answers

Related Questions

[Most efficient known priority queue for inserts](#)
data-structures time-complexity priority-queues

[Lock-free, constant update-time concurrent tree data-structures?](#)
reference-request data-structures time-complexity concurrency search-trees

Figure 6: Comments on a question



Figure 7: Answers to a question

4.1 Application Front End Design

Every question has a unique question page - which has the question, the answers, and comments (if any). On the right-hand side, we also display the related questions to the current question (on the basis of tags). This page is visible to all the users even those who haven't logged in yet. On the top bar - we provide options for login, signup, and search for a new question.

4.2 Demonstration Scenario

The question page contains the question, answers and comments on the question. It also contains the related questions on the right side. The question page contains the following features:

- **Question** - This section contains the question asked by the user.
- **Related Questions** - This section contains the questions which are related to the current question. On clicking the question, the user is redirected to the question page for that question.
- **Answers** - This section contains the answers for the current question. User can comment on the answer. On clicking the upvote or downvote button, the user can upvote or downvote the answer. User can also view the profile of the user who has answered the question by clicking on the username (userID).
- **Comments** - Each answer can have multiple comments. User can comment on the answer. User can also view the profile of the user who has commented on the answer by clicking on the username.

4.3 Overall Architecture

We need to present many things simultaneously on a question page. Firstly, we need to present all the information for the question - the title, the body, and the user who asked this question. Then we present any comments if they are present to this question. If there exists an accepted answer for this question, we present that next. All other answers follow. Each answer may also contain comments. Then, we also show the similar questions based on the tags they match. All this is fetched through the DB, and sent to the frontend.

```
1 @app.route('/question')
2 def question():
3     qid = request.args.get('id')
```

```

4     if qid == None:
5         return render_template('404.html'), 404
6     #convert qid to int
7     qid = int(qid)
8     cursor.execute(queries["similar_questions"].format(qid, qid))
9     similar_questions = cursor.fetchall()
10    cursor.execute(queries["question"], (qid, ))
11    question = cursor.fetchone()
12
13    if(question == None):
14        return render_template('404.html'), 404
15
16    cursor.execute(queries["update_viewcount"], (qid, ))
17    conn.commit()
18
19    acceptedanswerid = question[1]
20    acceptedanswer = None
21    aaid = -1
22    comments_acceptedanswer = []
23    if acceptedanswerid != None:
24        cursor.execute(queries["accepted_answer"], (acceptedanswerid, ))
25        acceptedanswer = cursor.fetchone()
26        aaid = acceptedanswer[0]
27        aaid = int(aaid)
28        cursor.execute(queries["comments"], (aaid, ))
29        comments_acceptedanswer = cursor.fetchall()
30
31    cursor.execute(queries["answers"], (qid, aaid))
32    answers = cursor.fetchall()
33
34    cursor.execute(queries["comments"], (qid, ))
35    comments_question = cursor.fetchall()
36
37    for i in range(len(comments_question)):
38        comments_question[i] = (comments_question[i][0], comments_question[i][1],
39                                ↪ comments_question[i][2], comments_question[i][3].strftime("%Y-%m-%d %H:%M:%S"))
40
41    for i in range(len(answers)):
42        cursor.execute(queries["comments"], (answers[i][0], ))
43        comments_answer = cursor.fetchall()
44        answers[i] = (answers[i], comments_answer)
45
46    return render_template('question.html', question=question, acceptedanswer=acceptedanswer,
47                            ↪ answers=answers, comments_question=comments_question,
48                            ↪ comments_acceptedanswer=comments_acceptedanswer, similar_questions=similar_questions)

```

We also allow the user to upvote and downvote a question (ofcourse, this is allowed only if (s)he is logged in). For the original CS stackexchange website, they allow multiple votes by users - there are other restrictions, for example, no more than 40 votes per day, voting allowed only if reputation is greater than a fixed value. We simply check the last vote by the user, and update the vote accordingly.

```

1  @app.route('/upvote', methods=['POST'])
2  @login_required
3  def upvote():
4
5      pid = request.args.get('id')
6      if pid == None:
7          return render_template('404.html'), 404
8      #convert pid to int
9      pid = int(pid)
10

```

```

11     cursor.execute(queries["check_vote"], (current_user.id, pid, ))
12     result = cursor.fetchall()
13     if len(result) == 0:
14         cursor.execute(queries["upvote"], (pid, current_user.id, ))
15         conn.commit()
16         cursor.execute(queries["increase_score"], (pid, ))
17         conn.commit()
18     else:
19         if result[len(result) - 1][1] == 3:
20             cursor.execute(queries["delete_vote"], (result[len(result) - 1][0],))
21             conn.commit()
22             cursor.execute(queries["increase_score"], (pid, ))
23             conn.commit()
24
25     return redirect(url_for('question', id=pid))
26
27
28
29
30 @app.route('/downvote', methods=['POST'])
31 @login_required
32 def downvote():
33
34     pid = request.args.get('id')
35     if pid == None:
36         return render_template('404.html'), 404
37     #convert pid to int
38     pid = int(pid)
39
40     cursor.execute(queries["check_vote"], (current_user.id, pid, ))
41     result = cursor.fetchall()
42     if len(result) == 0:
43         cursor.execute(queries["downvote"], (pid, current_user.id, ))
44         conn.commit()
45         cursor.execute(queries["decrease_score"], (pid, ))
46         conn.commit()
47     else:
48         if result[len(result) - 1][1] == 2:
49             cursor.execute(queries["delete_vote"], (result[len(result) - 1][0],))
50             conn.commit()
51             cursor.execute(queries["decrease_score"], (pid, ))
52             conn.commit()
53     return redirect(url_for('question', id=pid))

```

4.4 Supported Transactions

4.4.1 Query 1

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

```

1  WITH tagsForGivenQuestion AS (
2  SELECT *
3  FROM (
4      SELECT regexp_split_to_table(tags, '[><]') AS tag
5      FROM posts
6      WHERE Id = :questionId
7  ) AS tags WHERE tag != '';
8  )
9  SELECT posts.Id, count(*) AS count
10 FROM posts, tagsForGivenQuestion
11 WHERE posts.Tags LIKE '%' || tagsForGivenQuestion.tag || '%'
12 AND posts.Id != :questionId
13 GROUP BY posts.Id
14 ORDER BY count DESC

```

15 `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.

4.4.2 Query 2

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

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

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

4.4.3 Query 3

We will need to update the viewcount of the question when we visit it.

```
1 UPDATE posts
2 SET viewcount = viewcount + 1
3 WHERE Id = %s;
```

4.4.4 Query 4

To get all other details that need to be displayed -

```
1 --Accepted Answer
2 SELECT id, score, body FROM posts WHERE Id = %s;
3
4 --All Other Answers
5 SELECT id, score, body FROM posts WHERE ParentId = %s AND PostTypeId = 2 AND id != %s;
6
7 --Comments
8 SELECT id, userid, text, creationdate from comments WHERE PostId = %s;
```

5 Tags page

The screenshot shows a web interface titled "Tags page". It features a section "Most popular tags after a particular date" with a date input field (placeholder "dd/mm/yyyy") and a "Fetch Tags" button. Below this is a table of popular tags. To the right, there is another section titled "Most popular tags" with its own table.

TagName	Id	Count
cplusplus	2	100

TagName	Id	Count
compiler	1	1

Figure 8: Tags page

5.1 Application fronted

User can select tags for which they want to search Question. On selecting a Tag it will get added to the tag search list and on clicking search button, questions corresponding to those tags will be displayed. On clicking a question, user is redirected to Question Page. (We plan to integrate the CSS and improve design that is consistent with other pages before the demo).

5.2 Demonstration Scenario

User can either enter a date, or can search for tags without date. Upon clicking the search button, questions are shown corresponding to those tags. User can click any of those questions to get redirected to the corresponding question page.

5.3 Overall Architecture

For the tags page, we present the most popular tags. If we get a request argument, we process it to find the most popular tags only after that date. If we do not get any date as an argument, we simply find out the most popular tags in our database.

The corresponding flask implementation -

```
1 @app.route('/popular-tags')
2 def popular_tags():
3
4     date = request.args.get('dates')
5     if date:
6         cursor.execute(queries["popular_tags_by_date"], (date, ))
7     else:
8         cursor.execute(queries["popular_tags"])
9
10    popular_tags = cursor.fetchall()
11    return render_template('popular_tags.html', popular_tags=popular_tags)
12
```

5.4 Supported Transactions

5.4.1 Query 1

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

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

5.4.2 Query 2

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

```
1 With RelevantPosts AS (
2     SELECT Id, Tags
3     FROM posts
4     WHERE PostTypeId = 1 AND CreationDate > :date
5 )
6 SELECT TagName, tags.Id, Count(*) AS Count
7 FROM RelevantPosts, tags
8 WHERE RelevantPosts.Tags LIKE '%' || tags.TagName || '%'
9 GROUP BY TagName, tags.Id
10 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.

6 Admin Homepage

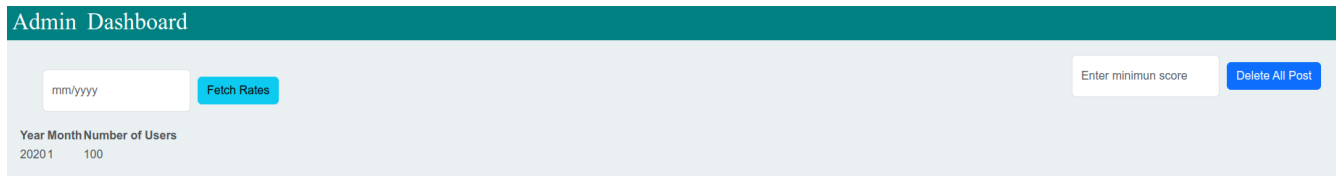


Figure 9: Admin Dashboard

6.1 Application Front End Design

Admin can enter month and year and check user signup rates for that particular time period. Admin is also delete all posts with score less than a given **minimum score**. (We plan to integrate the CSS and improve design that is consistent with other pages before the demo).

6.2 Demonstration Scenario

This page is visible only to the admin. From this page admin can find the number of users registered in a particular month. Just enter mm/yy and click Fetch Rates. On the right side admin can delete posts which are not appropriate based on net votes. Just enter the minimum net votes and click Delete All Posts. On clicking the delete button, all the posts with net votes less than the entered value will be deleted.

6.3 Overall Architecture

Admin Homepage will contain the user statistics.

```
1 @app.route('/admin')
2 def admin():
3     cursor.execute(queries["admin"])
4     admin = cursor.fetchall()
5     return render_template('admin.html', admin=admin)
6
```

Admin also has the option to delete the posts where a specific condition is met.

```
1 @app.route('/admin/delete-posts')
2 def delete_posts():
3     cursor.execute(queries["admin_delete_posts"])
4     conn.commit()
5     return render_template('admin.html')
6
7
```

6.4 Supported Transactions

6.4.1 Query 1

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

```
1 select Year(CreationDate) year ,
2        Month(CreationDate) month ,
3        count(*) users
4 from Users
5 group by Year(CreationDate), Month(CreationDate)
6 order by Year(CreationDate), Month(CreationDate);
```

6.4.2 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 -

```
1  DELETE *
2  FROM posts
3  WHERE score <= -5;
4
```

7 Profile page

7.1 Application Front End Design

This will display basic information about the user, the history of his/her posts and allow options to answer a question, ask a question, or search for a post.

7.2 Demonstration Scenario

From this page User can ask a question. On clicking Ask a question user is redirected to ask a question page. On clicking answer a question, user is redirected to welcome page. User can also search on the basis of tags for a question.

7.3 Overall Architecture

For a user to check her/his profile, (s)he must be, obviously, logged in. We ensure this using the `login_required` decorator of the flask-login module.

```
1  @app.route('/profile')
2  @login_required
3  def profile():
4
5      userid = request.args.get('userid')
6      if userid:
7          cursor.execute(queries["profile"], (userid, ))
8          profile = cursor.fetchall()
9          return render_template('profile.html', profile=profile)
10     else:
11         pass #For now, here we are just doing nothing. In future, we will extract the ID from the
12             ↪ login data stored with us.
13         return render_template('profile.html')
```

7.4 Supported Transactions

7.4.1 Query 1

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

```
1  SELECT * FROM users
2  WHERE users.id = 5;
3
```

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).

7.4.2 Query 2

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

```
1 UPDATE users
2 SET
3     DisplayName = 'myName',
4     Location = 'newLocation',
5     websiteUrl = 'myUrl',
6     aboutMe = 'This is me :)'
7 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.

8 Users page

Other users of the platform																
Users with maximum reputation	Admins	User Recommendation														
<table><tr><th>DisplayName</th><th>id</th><th>Reputation</th></tr><tr><td>john</td><td>1</td><td>100</td></tr></table>	DisplayName	id	Reputation	john	1	100	<table><tr><th>DisplayName</th><th>id</th></tr><tr><td>sam</td><td>2</td></tr></table>	DisplayName	id	sam	2	<table><tr><th>DisplayName</th><th>id</th></tr><tr><td>ben</td><td>1</td></tr></table>	DisplayName	id	ben	1
DisplayName	id	Reputation														
john	1	100														
DisplayName	id															
sam	2															
DisplayName	id															
ben	1															

Figure 10: Other users page

8.1 Application Frontend

This page presents the user page. We represent users with maximum reputation, or users that are admin and users that are recommended. (We plan to integrate the CSS and improve design that is consistent with other pages before the demo).

8.2 Demonstration Scenario

This page just displays the name, id, reputation, name of admins, and recommended person that is having matching interests with the user. Clicking on each user will redirect to that particular page.

8.3 Overall Architecture

As promised in the last milestone, we plan to show users according to different criterion - according to their reputation, users which are admins, and some users which we recommend them to follow.

```
1 @app.route('/users')
2 @login_required
3 def users():
4     tab = request.args.get('tab')
5     if tab == 'admin':
6         cursor.execute(queries["users_by_admin"])
7     elif tab == 'recommended':
8         cursor.execute(queries["users_by_recommended"])
9     else:
10        cursor.execute(queries["users_by_reputation"])
11
12    users = cursor.fetchall()
13    return render_template('users.html', users=users)
```

We also present a User Page, where we show the specific profiles of users. This page can be accessed in many ways - for example, clicking on username on the leaderboard on homepage, or clicking on the userIDs in the comments to a question, or even just directly sending an ID as request argument to the backend, and getting user profiles.

```

1 @app.route('/user')
2 def user():
3     userid = request.args.get('id')
4     if userid == None:
5         return redirect(url_for('profile'))
6     #convert userid to int
7     userid = int(userid)
8     cursor.execute(queries["get_user"], (userid, ))
9     user = cursor.fetchone()
10    if(user == None):
11        return render_template('404.html'), 404
12    cursor.execute(queries["update_user_views"], (userid, ))
13    conn.commit()
14    return render_template('user.html', user=user)

```

8.4 Supported Transactions

8.4.1 Query 1

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

```

1
2 SELECT * FROM users
3 ORDER BY reputation
4 DESC LIMIT 50;
5
6

```

8.4.2 Query 2

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

```

1
2 SELECT * FROM admins LIMIT 50;
3
4

```

8.4.3 Query 3

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

```

1
2 WITH v1 AS
3 (
4     SELECT person_id, tag_id FROM Person-like-tag WHERE person_id = 5
5 ),
6
7 v2 AS
8 (
9     SELECT person_id, tag_id FROM Person-like-tag WHERE person_id != 5 AND tag_id IN (
10    SELECT tag_id FROM v1)
11 ),
12
13 v3 AS
14 (
15    SELECT DISTINCT person_id FROM v2 WHERE person_id NOT IN (SELECT person_id_followed
16    FROM Person-follows-person WHERE person_id = 5)
17 ),
18
19 v4 AS
20 (
21    SELECT id, lastaccessdate FROM users WHERE id IN (SELECT person_id FROM v3) ORDER BY
22    lastaccessdate DESC LIMIT 10
23 ),
24
25 v5 AS

```

```

23  (
24      SELECT id FROM v4
25  )
26
27  SELECT * FROM users
28  WHERE id IN (SELECT id FROM v5);
29
30

```

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).

9 New Question page

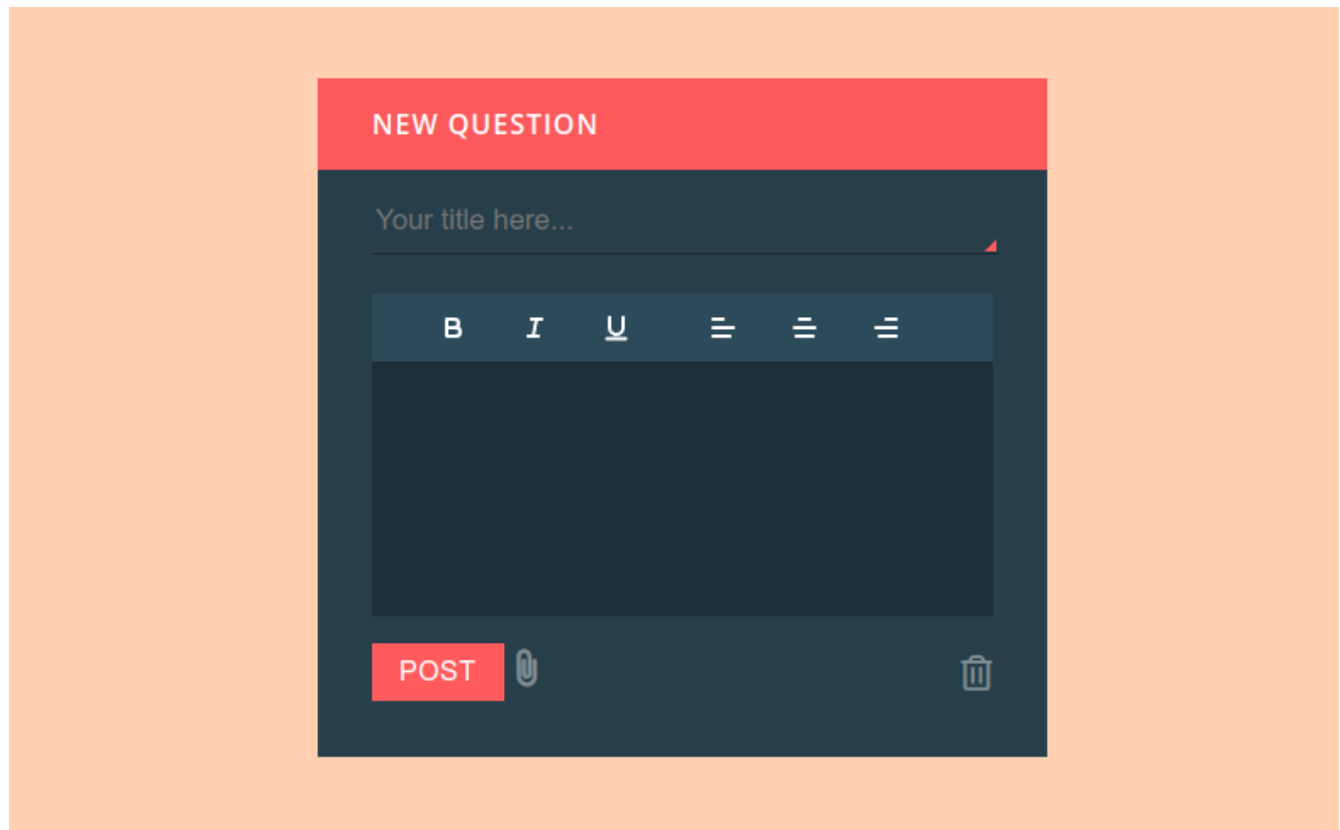


Figure 11: New Question

9.1 Application Front End Design

This page allows logged-in users to add a new question.

9.2 Demonstration Scenario

This page is used to ask a new question. User can enter the title and description of the question. User can also add tags to the question. On clicking the submit button, the question is added to the question table and the user is redirected to the question page for that question.

9.3 Overall Architecture

To post new questions to the backend, we simply add a new entry to the database with required details.

```

1 @app.route('/postquestion', methods=['GET', 'POST'])
2 @login_required
3 def postquestion():
4     form = PostQuestionForm()
5
6     if form.validate_on_submit():
7         cursor.execute(queries["post_question"], (current_user.id, form.title.data, form.body.data,
8             ↪ form.tags.data))
9         conn.commit()
10
11         cursor.execute(queries["get_question_id"], (current_user.id, form.title.data,
12             ↪ form.body.data, form.tags.data))
13         qid = cursor.fetchone()[0]
14         conn.commit()
15
16         return redirect(url_for('question', id=qid))
17
18     return render_template('postquestion.html', title='Post Question', form=form)

```

9.4 Supported Transactions

9.4.1 Query 1

```

1 DECLARE owneruserid INTEGER := 99
2         title TEXT := "sampletitle"
3         contentlicense TEXT := "CC BY-SA 4.0"
4         body TEXT := "samplebody"
5         tagsss := ["tag1", "tag2", "tag3", "tag4"]
6
7
8 FOR r IN tagsss:
9     UPDATE tags
10    SET Count = Count + 1
11    WHERE tags.TagName = r
12
13 INSERT INTO posts(OwnerUserId, LastEditorUserId, PostTypeId, AcceptedAnswerId, Score, ParentId,
14 ViewCount, AnswerCount, CommentCount, OwnerDisplayName, LastEditorDisplayName, Title, Tags,
15 ContentLicense, Body, FavoriteCount, CreationDate, CommunityOwnedDate, ClosedDate,
16 LastEditDate, LastActivityDate)
17 VALUES
18     (@owneruserid, @owneruserid, 1, NULL, 0, NULL, 0, 0, 0, (SELECT DisplayName from users where Id =
19     @owneruserid), (SELECT DisplayName from users where Id = @owneruserid), @title, @tagss,
20     @contentlicense, @body, NULL, CURRENTDATE, NULL, NULL, CURRENTDATE, CURRENTDATE );

```

10 Other Queries

10.1 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 -

```

1 INSERT INTO
2     badges (id, userid, class, name, tagbased, date)
3 VALUES
4     (1, 5, 1, 'Compiler', TRUE, '2018-01-01');
5

```

2. To get badges for a particular user -

```

1 SELECT * FROM badges
2 WHERE userid = 5;
3

```

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

```
1  with t1(personid , ansid) as
2  (
3      select a.OwnerUserId , a.Id
4      from posts a
5      where a.PostTypeId = 2
6  ),
7      t2(personid , ansid) as
8  (
9      select personid , ansid
10     from t1 , posts b
11     where ansid = b.AcceptedAnswerId
12  ),
13     t3(personid , countans) as
14  (
15     select personid , count(ansid)
16     from t2
17     group by personid
18  )
19     select personid from t3 where countans >= 50;
20
21
```

10.2 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)

10.3 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.

10.4 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) -

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

10.5 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

```
1 SELECT * FROM posts
2 WHERE owneruserid = 5 LIMIT 10;
3
```

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

```
1 SELECT * FROM posts
2 WHERE owneruserid = 5 AND tags LIKE '%<compilers>%' LIMIT 10;
3
```

3. Fetch all posts that a user has voted on

```
1 SELECT * FROM posts
2 WHERE id IN (SELECT postid FROM votes WHERE userid = 5);
3
```

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

```
1 SELECT * FROM posts
2 WHERE id IN (SELECT postid FROM votes WHERE userid = 5) AND tags LIKE '%<compilers>%';
3
```

5. Retrieve latest questions for a tag

```
1 SELECT * FROM posts
2 WHERE tags LIKE '%<compilers>%' AND posttypeid = 1
3 ORDER BY creationdate DESC LIMIT 10;
4
5
```

6. Retrieve all tags for a post

```
1 SELECT tags FROM posts WHERE id = 5;
2
```

7. Unanswered questions of the user ordered by creation date

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

8. Returns the number of accepted answers for a user

```
1 with t1(ansid) as
2 (
3     select Id
4     from posts
5     where OwnerUserId = 1 and PostTypeId = 2
6 )
7 select count(ansid)
8 from t1 , posts
9 where ansid = AcceptedAnswerId and PostTypeId = 1;
10
```

10.6 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.

```
1 SELECT * FROM Tags
2 WHERE id = 5;
3
```

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

```
1 INSERT INTO
2 Tags (Id, TagName, Count, ExcerptPostId, WikiPostId)
3 VALUES (0, 'test', 1, 1, 1);
4
```

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

```
1 SELECT * FROM Tags
2 ORDER BY Count DESC LIMIT 50;
3
4
5
```

10.7 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 describe 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 creation date 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.

```
1 SELECT * FROM users
2 ORDER BY reputation
3 DESC LIMIT 100;
4
```

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

10.8 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 an 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

```
1  SELECT * FROM posts
2  WHERE id IN (SELECT postid FROM votes WHERE userid = 5);
3
```

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

```
1  SELECT * FROM posts
2  WHERE id IN (SELECT postid FROM votes WHERE userid = 5) AND tags LIKE '%<compilers>%';
3
```

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

10.9 Person_follows_person

This table contains the information about who follows whom.

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

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

```
1
2  SELECT * FROM Person_follows_person
3  WHERE person_id = 5;
4
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
```

10.10 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 );
```

10.11 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.

10.12 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 );
```

11 Changes in Data Dump

We had to make a couple of changes to the database dump that we had uploaded earlier.

11.1 Ownership issue

Firstly, we had some permission issues during the last milestone. Precisely, the issue was because the dump we had uploaded had the owner of tables marked as "postgres" (the new local user I created to take the dump) - but while altering or updating the table, our username was "group_9", and so, we weren't able to make changes.

To fix this, as suggested by the TA, we re-uploaded the database dump with the changes we wished to see in our table. To ensure that the same problem doesn't happen again, I changed the owner of the tables to "group_9". To do this, I took the dump as explained. In the dump file thus created, I searched for the owner fields, and just updated them to the new owner. After this change, the database works fine now.

11.2 Unique ID issue

To insert values into our "users" table, "posts" table and "votes" table, we had to ensure that the ID that we assign to each new user, post and vote respectively should be unique. This could have either been done manually - by scanning the table and finding the maximum value of the ID, and then assigning $\text{max} + 1$ to the new object, but an elegant way to do this was to alter the table and tell POSTGRESQL to manage this on its own.

To achieve this, one of the methods is to use SERIAL Datatype - but we found out that this isn't compatible with PostgreSQL 12. So, we finally used the IDENTITY Datatype. We used the following command for this to happen -

```
1 ALTER TABLE table_name
2 ALTER COLUMN column_name
3 ADD GENERATED ALWAYS AS IDENTITY;
```

We also had to use this -

```
1 SELECT setval('tablename_columnname_seq', (SELECT max(columnname) FROM tablename));
```

So, now, to insert a new record into the table, we do not need to specify the ID field. Instead, it automatically finds the maximum and assigns the next value.

12 ER Diagram

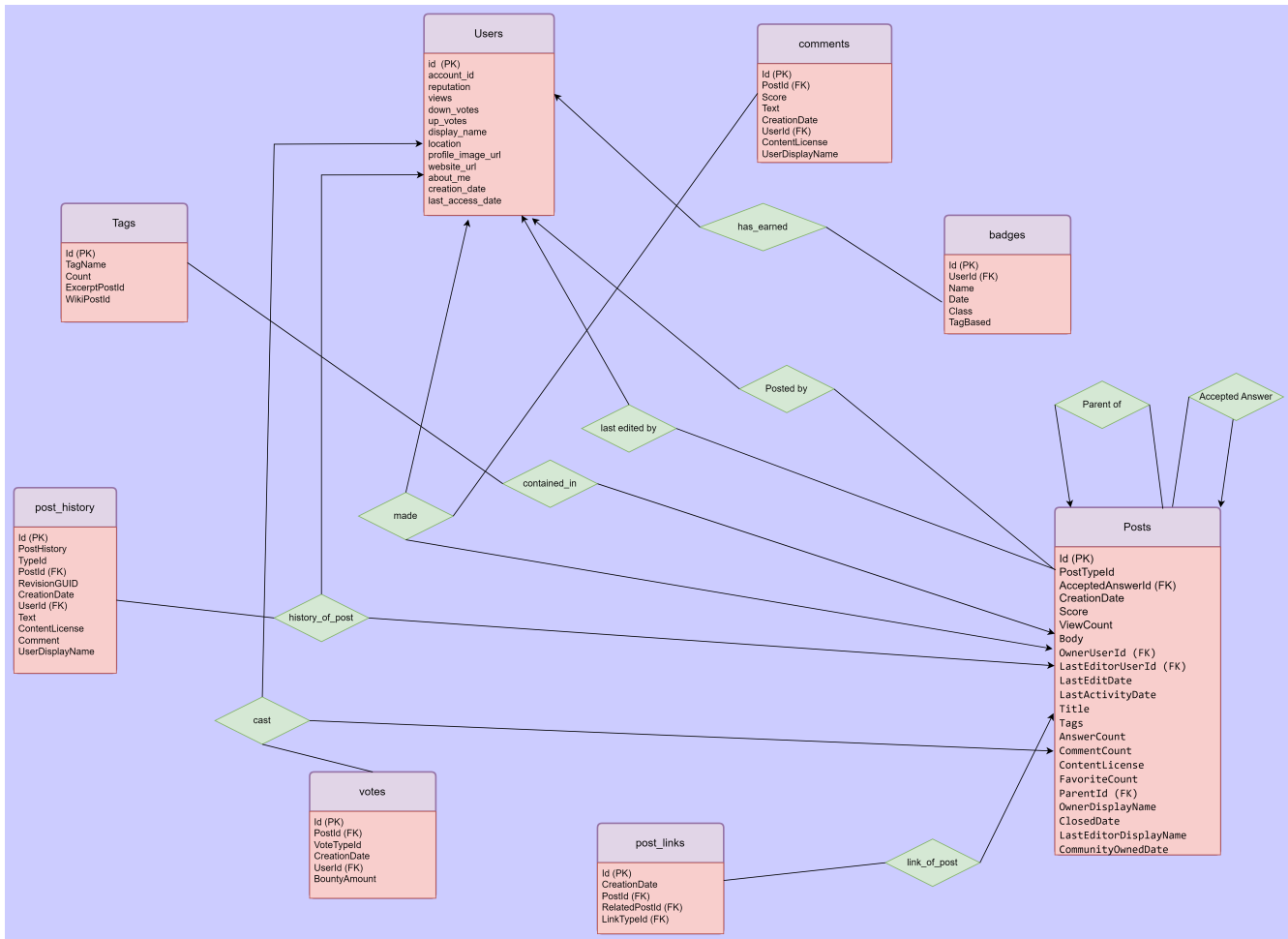


Figure 12: ER Diagram

13 Relational Schema

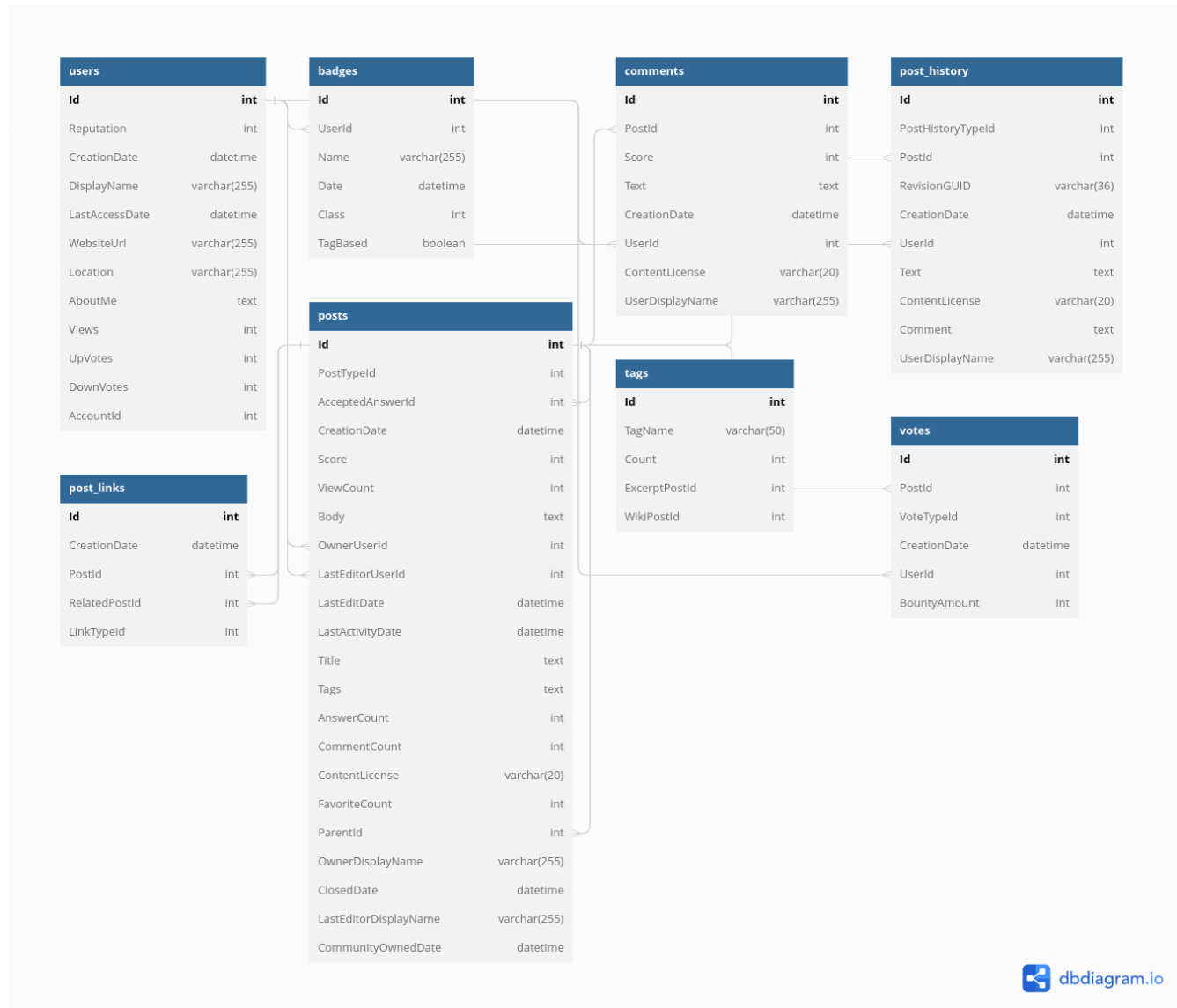


Figure 13: Relational Schema

14 Work Coordination

We planned to divide the work amongst ourselves so that we could work in parallel. One of us took most of the frontend part - the HTML, CSS and JS work. One other worked on Flask - the endpoints and queries to be executed when a particular endpoint is called. The third one mainly worked on the coordination part - linking everything on the frontend and the backend. This included ensuring that the data being returned by the backend is in the exact format as the frontend expects it to be, taking care of the functionality wherever user is required to be logged in before accessing a particular webpage etc.

We had discussions regarding some new functionality that could be added, (for example, we had not originally planned how to exactly use the search bar in the navigation bar - but then decided to use it for tags etc). Design, implementation details, constraints etc were also the things we brainstormed on together.

We used Visual Code Live Share feature to collaborate on our project. This helped us see our changes in the real time so that we could fix them as soon as we find some bugs. One of us (who had the VS Code running) also ran the flask server on localhost. This localhost was then tunnelled using ngrok so that all three of us could see the

website in real time.

15 Conclusion

For this final milestone, we present the complete web application that can be used as CS stackexchange. We deliver the frontend and the backend. There are some things missing that we plan to add during the next few days - add styling to some of our remaining HTML pages using CSS. We include all such pages in our final demo and not in this report. We plan to write more queries to present more data on our webpages - so that they look more informative.