12 May 2022

Online Forum Database System

Pengxiang Huang, Han Linghu, Zihan Li, Boyi Chen

School of Data Science, The Chinese University of Hong Kong, Shenzhen

Abstract—A practical and robust online forum for programmer is of great significance and necessity for CUHKSZ programmers. In this report, we present our design and implementation for the online forum database system. Starting from the motivation, we continue to discuss the database structure, queries implementation. The database and GUI implantation will then be exhibited. Last, the data mining technique is utilized.

Index Terms—forum, database, E-R diagram, SQL, data mining, search engine.

I. Background & Introduction

It is common for many student-programmers from CUHKSZ to encounter many familiar programming questions in their projects or assignments. These questions mainly include searching online, emailing TAs or professors for help, uploading questions in class WeChat group, or making appointments with TAs or professors during their office hours. Searching online sometimes may not be efficient because the blogs or some guidance information may not directly answer the assignment question. Even worse, programmers need to spend much time filtering vast information. It becomes hard for them to get answers directly when the homework question is not relevant to the results on the website. Uploading questions on the class WeChat group could get detailed guidance and response. But the new WeChat group will be created every semester for other students who may encounter the same problems. The connection between students who have already taken this course with the students taking this course right now is broken in this way. Raising questions during office hours is not convenient for programmers to solve their questions immediately since they need to make an appointment and wait until that day comes. Therefore, our group would like to take the first step to change the current situation by providing an online Q&A platform with the support of an online forum database.

The target users of our database are programmers in a specific community who have similar knowledge backgrounds, for instance, CUHKSZ students who major in CS. By limiting the range of users to a particular group, we can ensure that the users will have similar problems related to their projects or assignments, saving them from filtering vast amounts of information. It would be easier for them to get help. The critical component of our project in the frontend is several blogs. The main body of the blog is the questions or problems users encounter during their programming procedure, which are stored in the database. To help users better explain their needs, we allow users to attach files and images to a particular blog. And files and images are stored in the remote cloud server, while the relationships between images and files are stored in the database. Additionally, we allow users to directly answer others' questions by leaving a comment under a certain blog, matching the answers and questions. If the user who raises the question is still confused after reading others' comments, he/she can also leave a comment under others' answers to get a further reply. To increase the performance of searching through the database, we made great efforts. We divide blogs into different groups, and each group also has many subgroups. So, the user can reduce the search range and get the most relevant blogs. Besides, we allow users to like and follow blogs or groups to find the information they want more easily next time.

To make our database more efficient and space-saving, we made efforts to do normalization on tables and introduce indexes in our project. Additionally, we built a fantastic UI in the frontend and robust backend server to hide the detailed implementation and manipulation of the database. So, users can focus on the Q&A procedure, and they don't need to worry about how to get the desired information from the database, as the functions carry out all the queries in the frontend and backend.

II. Database Structure Design

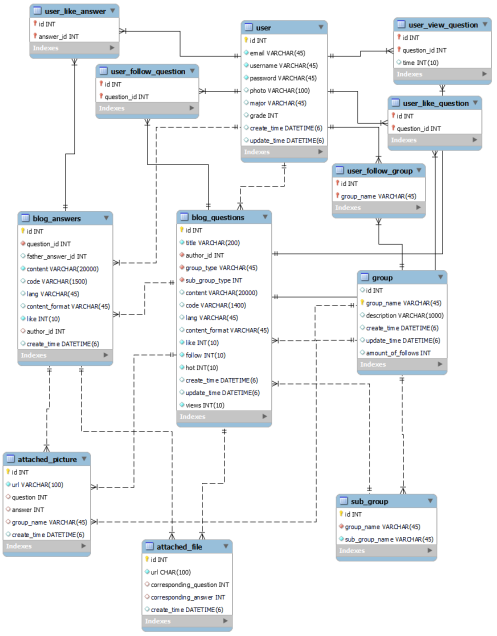
A. Requirement & Specification

The main component of our project is the online forum database. There are three main entities in the database, namely "*user*", "*blog_questions*" and "*blog_answers*". The "user" entity stores the account information about the user, including their name, email address, password and etc.,

which is used for identity authentication when the user enters the online forum. The core of our project is that users can post a blog to raise a question and get relevant answers from other users, which brings about the "*blog_questions*" entity and the "*blog_answers*" entity to become the key features. Those two entities store the questions and answers information about the blog, including content, *author_id*, etc. We also implement several relationship schemas to support the interaction between the user and the blog. For instance, the user can like a blog or follow a blog. With the purpose of making our database more efficient and space-saving, we spare efforts on doing normalization and applying indexes to our project.

B. Entity-Relation Diagram

The ER-diagram of our system contains 12 entities. The entities are *user, group*, *sub_group, blog_questions, blog_answers, attached_file, attached_picture, user_like_answer, user_like_question, user_follow_question, user_follow_group and user_view_question*.



User Anyone who wants to have access to our system needs an account, so the user entity has the related information of the user's account, such as username, password, email address, and the URL of the profile photo. In the process of identity authentication, the system will throw an exception if it detects that the identity information does not match the records stored in the user entity.

Group In our system, each blog has a related group, such as CSC3170, which is convenient for users to find the blogs in the corresponding group. Moreover, it serves as the filter conditions in the process of searching, which improve the efficiency of searching. The group entity has *group_name*, description, *create_time*, *update_time*, *amount_of_follows*. "*Group_name*" is designed to store the course number of the computer science courses offered in CUHKSZ, such as CSC3170 and CSC3150. "*Description"* is designed to store the detailed information of the course, such as Database System and Software Engineering. "*Create_time*" is designed to store the time of the creation of this group. "*Update_time*" is designed to store the time period of the update of the blogs belonging to this group. "*Amount_of_follows*" is designed to store the amounts of users who follow the group.

Sub_group In our project, each blog not only has a related group but also has a related subgroup, such as Assignment1. This can be served as the filter conditions in the process of searching, which improve the efficiency of searching. The *sub_group* entity has *group_name* and *sub_group_name*. "*Group_name*" is designed to store the course number of the computer science courses offered in CUHKSZ, such as CSC3170 and CSC3150. "*Sub_group_name*" is designed to store the assignments in each course, such as Assignment1 and Project.

Blog_questions The entity is of great significance in our online forum system. It has title, *author_id*, *group_type, sub_group_type*, content, code, like, follow, hot, *create_time*, *update_time*, views. "*Title*" is designed to store the title of the blog. "*Author_id"* is designed to store the author of the blog, which is used to determine the authority of the deletion of the blog. "*Group_type*" is designed to store the course number to which the blog belongs, such as CSC3170 and CSC3150. "*Sub_group_type"* is designed to store the corresponding assignment to which the blog belongs, such as Assignment1 and Project. "*Content*" is designed to store the content of the blog. "*Code*" is designed to store the code of the blog since our system provides an online code editor and compiler. "*Like*" is designed to store the amounts of users who like the blog. "*Follow*" is designed to store the amounts of users who follow the blog. "*Hot*" is designed to store the popularity value of the blog, which is used to determine the order of the blogs displayed in the "*Hot Blogs*" part. The value is highly related to likes, favors, views, and the creation time. The newer the questions posted, the larger the likes, favors, and views, and the higher the popularity value. "*Create_time*" is designed to store the time period of the creation of this blog. "*Update_time*" is designed to store the time period of the update of the answers belonging to this blog. "*Views*" is designed to store how many times users have viewed the blog.

Blog_answers The entity is of great significance in our online forum system. It has *question_id, father_answer_id,* content, like, *author_id*, and *create_time*. "*Question_id"* is designed to store the blog question to which the answer belongs. "*Father_answer_id*" is designed to store the blog answers to which the solution belongs. Only one of the values between "*Question_id"* and "*Father_answer_id"* is not null since the answer can only reply to the question or the solution. "*Content*" is designed to store the content of the answer. "*like*" is designed to store the amounts of users

who like the answer. "*Author_id*" is designed to store the author of the answer. "*Create_time*" is designed to store the time period of the creation of this answer.

Attached_file Some blogs or answers may have attached files, which will be stored in this entity. The entity has URL, *corresponding_question*, *corresponding_answer*, and *create_time.* "URL" is designed to store the URL of the file. "*Corresponding_question*" is designed to store the blog question to which the file belongs. "*Corresponding_answer*" is designed to store the blog answer to which the file belongs. Only one of the values between "*Corresponding_question*" and "*Corresponding_answer*" is not null since the file can only belong to the question or the answer. "*Create_time*" is designed to store the time period of the creation of this file.

Attached_picture Some blogs or answers may have attached pictures, which will be stored in this entity. The entity has URL, question, answer, group_name, and create_time. "*URL*" is designed to store the URL of the picture. "*Question*" is designed to store the blog question to which the picture belongs. "*Answer*" is designed to store the blog answer to which the picture belongs. "*Group_name*" is designed to store the group to which the picture belongs. Only one of the values among "*Question*", "*Answer*" and "*Group_name*" is not null since the picture can only belong to the question, the answer, or the group.

User_like_answer In our online forum system, users can click the button to like the answer so that those information will be stored in this entity. The entity has id and *answer_id*. "*Id"* is designed to store the id of the user. "*Answer*_id*"* is designed to store the id of the answer. Therefore, each record means the user like the answer.

User_like_question In our online forum system, users can click the button to like the question so that that information will be stored in this entity. The entity has id and *question_id*. "*Id"* is designed to store the id of the user. "*Question_id"* is designed to store the id of the question. Therefore, each record means the user like the question.

User_follow_question In our online forum system, users can click the button to follow the question so that that information will be stored in this entity. The entity has id and *question_id*. *"Id"* is designed to store the id of the user. "*Question_id*" is designed to store the id of the question. Therefore, each record means the user follows the question.

User_follow_group In our online forum system, users can click the button to follow the group so that those information will be stored in this entity. The entity has an id and *group_name*. "Id" is designed to store the id of the user. "*Group_name*" is designed to store the name of the corresponding group. Therefore, each record means the user follows the group.

User_view_question In our online forum system, users can click the blog to view the detailed question so that those information will be stored in this entity. The entity has id, *question_id*, and time. "*Id"* is designed to store the id of the user. "*Question_id*" is designed to store the id of the question. "*Time*" is designed to store how many times the user views the question.

C. Schema & Normalization

In our project, we spare a lot of effort on normalization. At first, we intend to reach the first normal form, so we reconstruct our database. For example, "Table 1" shows our initial group table design with a tuple of subgroup names. Therefore, we split the group table into group table (Table 2) and sub group table (Table 3). In this case, the first normal form is achieved.

id	group_name	sub_group_name	description	create_time	update_time	amount_of_follows
1	CSC3170	{Assignment1, Assignment2, …}	Database System	2022-04-30	null	100

Table 1

id	group_name	description	create_time	update_time	Amount_of_follows
1	CSC3170	Database System	2020-04-30	Null	100

Table 2

id	group_name	sub_group_name
1	CSC3170	Assignment1
2	CSC3170	Assignment2

Table 3

Based on the first normal form, we also try to reach the second and third normal form. As a result, we create a unique id for each table to enable all nonprime attributes to be fully functionally dependent on the primary key (id). Apparently, there do not exist nonprime attributes in our tables transitively dependent on the primary key (id). Therefore, the second normal form and third normal form are also implemented. As a result, the relational schema is shown below:

user(id, email, username, password, photo, major, grade, create_time, update_time)
group(id, group_name, description, create_time, update_time, amount_of_follows)
sub_group(id, group_name, sub_group_name)
blog_questions(id, title, author_id, group_type, sub_group_type, content, code, lang, content_format, like, follow, hot, create_time, update_time, views)
blog_answers(id, question_id, father_answer_id, content, code, lang, content_format, like, author_id, create_time)
attached_file(id, url, corresponding_question, corresponding_answer, create_time)
attached_picture(id, url, question, answer, group_name, create_time)
user_like_answer(id, answer_id)
user_like_quesition(id, question_id)

user_follow_question(id, question_id)
user_follow_group(id, group_name)
user_view_question(id, question_id, time)

Some interpretations about foreign keys referencing:

1. “group_name” in sub_group refers to “group_name” in group: each sub group belongs to a group.

2. “author_id” in blog_questions refers to “id” in user: each blog belongs to a user.

3. “group_type” in blog_questions refers to “group_name” in group: each blog belongs to a group.

4. “sub_group_type” in blog_questions refers to “sub_group_name” in sub_group: each blog belongs to a sub group.

5. “question_id” in blog_answers refers to “id” in blog_questions: each answer belongs to a blog.

6. “author_id” in blog_answers refers to “id” in user: each answer belongs to a user.

7. “corresponding_question” in attached_file refers to “id” in blog_questions, and “corresponding_answer” in attached_file refers to “id” in blog_answers: each file belongs to a question or an answer.

8. “question” in attached_picture refers to “id” in blog_questions, “answer” in attached_picture refers to “id” in blog_answers, and “group_name” in attached_picture refers to “group_name” in group: each picture belongs to a question, an answer or a group.

D. Index & Hashing

a) Brief Introduction

To introduce index in our project to speed up some frequently used queries. We first search on the internet to know some main kinds of indexes:

1. T-Tree Index

2. B-Tree Index

3. Full-Text Index

4. Spatial Index

5. Hash Index

After taking the structure of our database into consideration, we decided to focus on the hash index and B-Tree index, which can fulfill our requirements.

b) Hash Index

The first kind of index we thought of is the hash index. The main advantage of hash index is the fact that hashes can be much smaller than the indexed value itself, which can be space-saving. And the query speed is relatively fast for exact lookups if there are not many conflicts.

But after searching online and reading some formal documentation about MySQL, we found that hash index also has disadvantages. The main problem is that this index type can’t be used to do anything other than a simple lookup. So its usage is limited. What's worse, the hash index is only supported by the "Memory" engine, and the hashed index tables are all stored in the main memory, so the data is temporary and volatile. When the database restarts or encounters some system failure, the data are all gone. But the requirement of our database is that it stores all the questions and answers so users can quickly solve their problems by simply searching the database or posting a blog to get some help. So the hash index does not satisfy our project’s requirement and was not adopted.

c) B-Tree Index

The B-Tree index is the most commonly used index, and it can speed up queries that match a full value. It uses the B-Tree structure to store the index, so the average searching time is reduced to log(h) (h is the height of the tree). Compared to the all-table scanning of a simple query without an index, the improvement is vast.

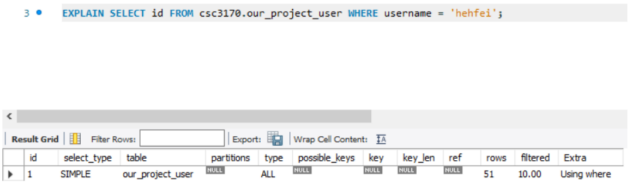
Considering the trade-off between search performance and system cost, we add indices to some frequently queried columns in certain tables, e.g., the "username" column in the "User" table. To better understand the importance of index, we use the "explain" syntax, the meaning of some keywords is listed below:

1. type: information about join type.

2. possible_keys & key: indexes that could be used for the particular query and the index chosen by the optimizer as the most efficient one.

3. rows: an estimate of how many rows the query will scan.

4. extra: prints additional information relevant to how the query will be executed.

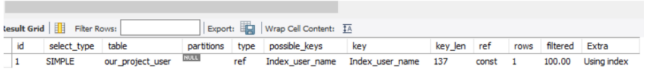


Before introducing index

Before we add index to the table, when we want to get the id of a user with a specific name, the whole table is scanned, and around 51 rows are scanned. (If the table grows larger, this number will also grow), so the speed is

very slow.

```
ALTER TABLE csc3170.our_project_user ADD INDEX Index_user_name (username);
-- DROP INDEX Index_user_name on csc3170.our_project_user;
EXPLAIN SELECT id FROM csc3170.our_project_user WHERE username = 'hehfei';
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	our_project_user	NULL	ref	Index_user_name	Index_user_name	137	const	1	100.00	Using index

After introducing index

But after we introduce the B-Tree index, the query will take advantage of the index, and the number of rows to be scanned is significantly reduced.

The following part is where we introduce indexes with some explanation.

1. index on "username"

The 'username' attribute of the "user" table is frequently queried when the user register for an account or he/she wants to log in, so an index is assigned to it

```
if request.method == 'POST':
    username = request.POST['username']
    email = request.POST['email']
    password = request.POST['password']

    #check user name. Multi-thread considerration.
    old_users = User.objects.filter(username=username)
    if (old_users):# if exists
        data['isRegister'] = 0
        return HttpResponse(json.dumps(data), content_type='application/json')
```

register in the backend

The above figure shows the register procedure; the server will check if the given 'username' is already in the database.

```
  `update_time` datetime(6) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `Index_user_name` (`username`)
) ENGINE=InnoDB AUTO_INCREMENT=78 DEFAULT CHARSET=utf8mb3;
```

assigning index to 'username'

2. index on "group_name"

The 'group_name' of the 'sub_group' table is frequently queried when the user wants to get the groups, he/she follows.

```
group_names = user_follow_group.objects.filter(id = userid).values()

data = {}
for i in range(0, len(group_names)):
    group_name = group_names[i]['group_name']

    group = Group.objects.filter(group_name=group_name).values()[0]
```

search group name

The above figure shows the procedure that the server returns the group_name of sub_groups the user follows.

```
  PRIMARY KEY (`id`),
  KEY `group_NAME_idx` (`group_name`),
  CONSTRAINT `group_NAME` FOREIGN KEY (`group_name`) REFERENCES `Our_project_group`
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8mb3;
```

group_name

3. index on "'sub_group_type"

The 'sub_group_type' attribute of the "blog_questions" table is frequently queried during the range limited searching procedure when the user wants to search for blogs in the specific sub-group.

```
questions = Blog_Questions.objects.filter(group_type = group_name, sub_group_type = sub_group_id).order_by('-hot').values()
```

search group type

```
  PRIMARY KEY (`id`),
  KEY `sub_group_name_idx` (`sub_group_type`),
  KEY `email_idx` (`author_id`),
  KEY `group_type_idx` (`group_type`),
  CONSTRAINT `author_id` FOREIGN KEY (`author_id`)
```

group_type

4. index on "author_id"

The 'author_id' attribute of the "blog_answer" table is frequently queried when showing the answers to the blog.

```
# according to the author id, return the author's profile url and username
author_id = answer["author_id"]
author_name = User.objects.filter(id = author_id).values()[0]['username']
profile_url = User.objects.filter(id = author_id).values()[0]['photo']
```

get author id

The above figure shows the procedure of getting the information about the author of the answer.

```
PRIMARY KEY (`id`),
KEY `question_id_idx` (`question_id`), -- index
KEY `father_answer_id_idx` (`father_answer_id`),
KEY `id_of_author_idx` (`author_id`),
CONSTRAINT `id_of_author` FOREIGN KEY (`author_id`)
```

author id

III. SQL Function

To access different information in the database, we have to implement different queries. In this part, the data definition language and different ways to implement queries including direct SQL, SQL with host language, and Query Set will be detailly introduced.

A. Data Definition Language

As we learned in class, we have to use data definition language to convert our database design into physical schemas and tables. As the graph, there are mainly two parts in the DDL.

In the first part, we set all the necessary attributes and specify different types of attributes based on the logical schema. For instance, the attribute '*id*' in the *blog_question* table is an integer type. These operations will prevent data with false data types, which may destroy the table.

The second part is about integrity constraints. We have tried to specify all necessary integrity constraints to make these schemas and tables detailed and specific. Basically, these constraints contain primary keys, foreign keys, and keys that are not null or unique. For instance, the attribute '*id*' is with the integrity constraint NOT NULL, with means in the physical schema, null values are prohibited. On top of that, the instance below also shows that the attribute '*id*' is set as the primary key, and the '*author_id*' is referencing the attribute '*id*' in another table user. Also, we add necessary index constraints during the creation of the table, which has been detailly talked about in the previous part.

```sql
-- Table structure for table `Our_project_blog_questions`
DROP TABLE IF EXISTS `Our_project_blog_questions`;
CREATE TABLE `Our_project_blog_questions` (
  `id` int NOT NULL AUTO_INCREMENT,
  `title` varchar(200) NOT NULL,
  `author_id` int NOT NULL,
  `group_type` varchar(45) NOT NULL,
  `sub_group_type` int NOT NULL,
  `content` varchar(20000) DEFAULT NULL,
  `code` varchar(1400) DEFAULT NULL,
  `lang` varchar(45) DEFAULT NULL,
  `content_format` varchar(45) DEFAULT NULL,
  `like` int(10) unsigned zerofill NOT NULL,
  `follow` int(10) unsigned zerofill NOT NULL,
  `hot` int(10) unsigned zerofill NOT NULL,
  `create_time` datetime(6) DEFAULT NULL,
  `update_time` datetime(6) DEFAULT NULL,
  `views` int(10) unsigned zerofill NOT NULL,
  PRIMARY KEY (`id`),
  KEY `sub_group_name_idx` (`sub_group_type`),
  KEY `email_idx` (`author_id`),
  KEY `group_type_idx` (`group_type`),
  CONSTRAINT `author_id` FOREIGN KEY (`author_id`) REFERENCES `Our_project_user` (`id`),
  CONSTRAINT `group_type` FOREIGN KEY (`group_type`) REFERENCES `Our_project_group` (`group_name`),
  CONSTRAINT `sub_group_type` FOREIGN KEY (`sub_group_type`) REFERENCES `Our_project_sub_group` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb3;
```

Data Definition Language

B. Queries

We have used different ways to implement the queried on the basis of efficiency, including direct SQL, SQL with host language Python, and Query Set.

a) SQL

The easiest way to interact with the database is directly using the SQL language. This method is fast and with an in-time response. Usually, we prefer to use this way especially when we are testing our newly created tables and in the final Alpha test. As below, we are testing the insertion operation on the table *user_like_question*. In this way, data can be directly inserted without other procedures like through the standard user interface.

```sql
LOCK TABLES `Our_project_user_like_answer` WRITE;
INSERT INTO `Our_project_user_like_answer` VALUES (4,1),(5,1),(2,2),(5,2),(3,3),(1,4),(3,4),(4,4);
UNLOCK TABLES;
```

Direct SQL

Also, we are using the SQL to test the table *"sub_group"* by inserting the new mock data: sub-group Project in group CSC3170. Apparently, there are some integrity constraints in some tables, such as the sub_group. To directly use SQL to interact with the database, we must temporarily disable these constraints. So, in the sample, we use ALTER TABLE … DISABLE KEYS to disable the constraints. After the success of insertion, we enable these constraints again using ALTER TABLE … ENABLE KEYS.

```sql
LOCK TABLES `Our_project_sub_group` WRITE;
ALTER TABLE `Our_project_sub_group` DISABLE KEYS ;
INSERT INTO `Our_project_sub_group` VALUES (1,'CSC3170','Project');
ALTER TABLE `Our_project_sub_group` ENABLE KEYS ;
UNLOCK TABLES;
```

Direct SQL

b) SQL with Host Language

The problem with the SQL language is that direct SQL is not a Turing machine equivalent language. Therefore, it might be easy to access the data during the creation stage, but it is hard for us to efficiently analyze and process the data using direct SQL. We tried to solve this question with the host language Python, specifically using the pymysql package. The basic logic of SQL with a host language is intuitive. We get the data and input data as part of the host language in order to analyze and process these data.

As below, we are connecting to the database CSC3170 and implementing deletion in the table user. This operation is equivalent to the DELETE FROM operation in the direct SQL.

```python
#1. connect to the DB
conn = pymysql.connect(host="[redacted]", port=3306, user="[redacted]", passwd="[redacted]", db="CSC3170", charset="utf8")
cursor = conn.cursor()

#2. command
deleteId = 3
sql = "Delete from our_project_user where id=%s"
cursor.execute(sql, deleteId)
#3. commit
cursor.commit()

#4. close conection
cursor.close()
conn.close()
```

Delete with the host language

Below illustrates the update operation using the host language. Specifically, a new user named 'hands' with password '123123' is updated into the table user. It is equivalent to the UPDATE SET operation in the direct SQL.

```python
#1. connect to the DB
conn = pymysql.connect(host="[redacted]", port=3306, user="[redacted]", passwd="[redacted]", db="CSC3170", charset="utf8")
cursor = conn.cursor()

#2. command
newUsername = 'hands'
newpassword = '123123'
sql = "update our_project_user set username=%s and password=%s"%(newUsername, newpassword)
cursor.execute(sql)
#3. commit
cursor.commit()

#4. close conection
cursor.close()
conn.close()
```

Update with the host language

The advantage of the better ability in analyzing and processing using the host language is demonstrated. The instance is part of the inverted index in the data mining process. Briefly speaking, the word index is getting from the database, and these data are normalized and directly put in the analysis procedure. If you are interested in the data mining part, please refer to the individual data mining part for more information.

```python
conn = pymysql.connect(host='175.[redacted]84', port=3306, user='[redacted]', passwd='[redacted]', db='[redacted]', charset='utf8'
cursor = conn.cursor()
Answer_T = []
Answer_C = []
blockNumberTotallist = []
lenList = len(question)
searchIndex = 1
for item in question:
    if item[0].isalpha:
        sql = 'select * from Our_project_{} where WORDS = "{}";'.format(item[0].upper(), item)
        cursor.execute(sql)
        a=cursor.fetchall()
        # print(a)
        # print(len(a))
    else:
        sql = 'select * from Our_project_{} where WORDS = "{}";'.format('OTHERS', item)
        cursor.execute(sql)
        a=cursor.fetchall()

    for i in range(len(a)):
        if a[i][1] not in blockNumberTotalList: blockNumberTotalList.append(a[i][1])
        Ttemp1 = [0]*(lenList+1)
        Ctemp1 = [0]*(lenList+1)
        Ttemp1[0] = a[i][1]
        Ttemp1[searchIndex] = a[i][2]
        Ctemp1[0] = a[i][1]
        Ctemp1[searchIndex] = a[i][3]
        Answer_T.append(Ttemp1)
        Answer_C.append(Ctemp1)
    searchIndex += 1
searchIndex -= 1
conn.close()
```

Data Process with Python

c) Query Set

Within the Django framework, there is another efficient way for us to manipulate the data. And it is the query set. The logic of the query set is similar to SQL, which means it is not a Turing machine equivalent language either, and it is used for data accessing only. What is more, the query set is in a totally different gramma from SQL, making it a little bit more challenging to use for SQL users.

Considering that in some cases, it is more efficient for us to use the Query Set, such as interact with some models within the Django system, we also implemented some data manipulation with the help of Query Set.

The below figure is about finding the titles of the blocks in the database. It is obvious that the result can also be set directly in the analysis process within the host language. Like in this instance, all the titles are extracted and uppercased.

```
titleDB = Blog_Questions.objects.values('title')
for title in titleDB:
    DBlist.append(title['title'].upper())
print(DBlist)
```

Query Set: Select

The instance below uses the query set to create a new user in the database with a username, password, and email inserted. This instance can also be implemented by the direct SQL and SQL with pymysql in Python.

```
try: #Multi-thread consideration
#insert data
    user = User.objects.create(username=username, password=password_m, email=email)
except Exception as e:
    print('--create user error%s'%(e))
    data['isRegister'] = 0
    return HttpResponse(json.dumps(data), content_type='application/json')
```

Query Set: Insert

Above all, three different ways to query have been introduced. The direct SQL is fast and efficient in testing. The SQL with host language Python is much easier to implement different operations, including both data accessing and data processing. Even though it is in another gramma for query set, it is still quite effective in accessing and analyzing data. With different purposes and considerations, different query ways are chosen and implemented.

C. Concurrency and multi-threads issues

It is essential to consider the concurrency and multi-threads issues in the query part because inappropriate operations will lead to data inconsistency or even break the database down. Hence, we have tried to consider different concurrency issues in different cases.

For the direct SQL, as the samples in first section, tables are locked and unlocked before and after being manipulated. In this way, the concurrency and the data consistency are protected.

For SQL with host language and the query set, we also try to promise the protect the database with multi-threads consideration. We prohibited creating one user multiple times at the same time, which will surely damage the database.

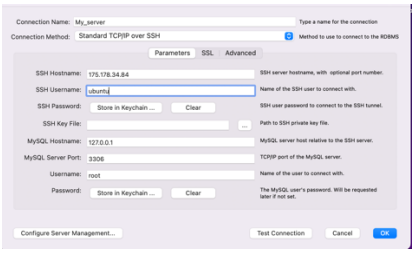
IV. Database System Implementation

A. Database Instance Deployment

For the connivence of database management among group members, instead of building up the database system on one team member's personal computer, we choose to purchase a remote cloud server and configure the mysql database for project usage.

We purchase a Tencent Cloud Server, which is deployed in Guangzhou using MySQL 8.0 version. The configuration of the database is four CPU cores with 4GB memory and 100GB storage.

Database can be connected via MySql workbench with the below information.



The database implementation is based on mysql and python, as introduced above. The frontend webpage deisgn is based on Vue frameworkd. Also, we use nigix to mount and agent our webserver, which provide a domain name http://www.cuhksz-stackoverflow.cn/. It is more convenient for normal user to register and check our webpage without building the program in their local computers.

B. Data Generation

To better test the function of the database, our group also populated our database with an abundant amount of real data. As the question blogs in forum is the major data information, we collect those data from two different method. One is from the developer perspective, the other is from the real user perspective.

For the developer, our group member collects the data from many sources. We collect the questions from wechat group, other forums, and websites and write them into blog. The work is time consuming since we need to write and generate the data all by hand. And we also run a simple web crawl to get some question blog from the website stack-overflow, and use those information to generate a blog automatically.

To make our database more relevant and friendy to real user, we also collect the information from another user. Before we build the completed program, we let some user register the web account and post their questions that encountered recently. Those users will generate data that is

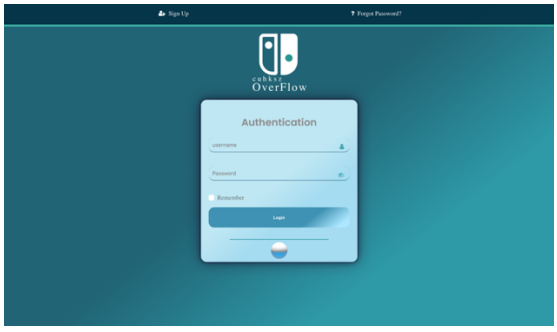
more specific to their course project which is meet the requirment of our goal of this forum.

C. Graphic User Interface Design

In this part, the graphic User Interface of online forum will be introduced. As mentioned, the system is implemented based on Vue framwork which provide embedded html, css and javascript. Since this framwork is suitable for building a quick and agile website, the style of the system is simple and clear. The main color style of our webpage is baby blue with many corlorful button embedded. The whole frontend is pretty user-friend and wisely organized. there are 5 modules totally in our system. Here is the briefly introduction of those basic components with screenshots.

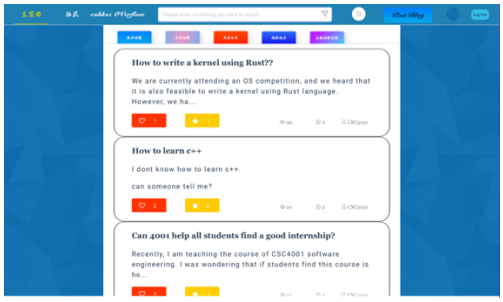
a) Login & Register

This module is used for collecting the user information and store the information in database server. Also, when user login to our webpage, the query of retrieve the corresponding username and password to match the input of user will be triggered. Hence, based on this module, the user will be required to do the authentication with his own username.



b) User Home Page

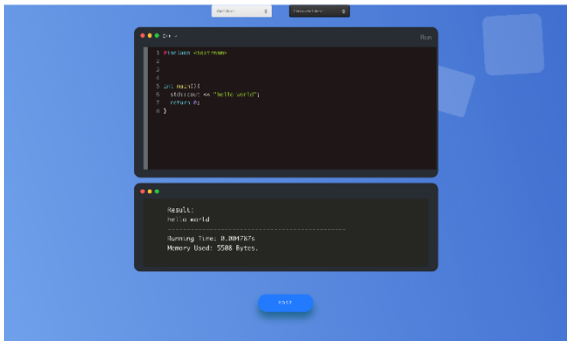
This module is the main page for user to execute serval operations on our webpage. The webpage will show the catalog for user to see the servel blogs. For instance, there are 5 catalog contains "hot blogs", "my blogs", "unsolved blogs" and so on. Those catalof show different orders of all blogs in our database. This page also include many buttons for user to jump to anthor operations.



c) Post Blog

This module is used for user to write their blog content. The users are required to type the blog title and content, partition, and their sub partitions in order to classify. The content input will be finished in a rich editor. The rich editor supports many forms of input and file uploading. Users could upload their file or just insert a file link. Our server will receive this file and store it on the server. Users could use the rich editor to create the blog forms they prefer. And all the blogs on the home page will be decoded as the same as they posted on this webpage. users are also allowed to write their code as a supplement material.

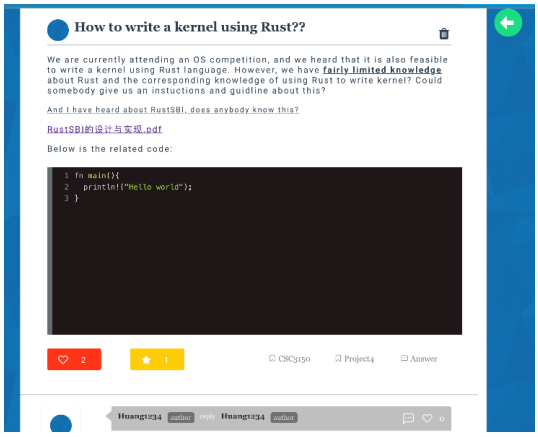
Users could also run the code to see the output. Typically, Users are allowed to write the code without environmental configuration. They could also write code on an iPad or a phone. The online compiler supports many languages, including C, C++, Python, Rust… The running time and memory used will be shown on the webpage just as an open Jude system. The code will be highlighted due to different language, different language has a different highlighted method, and they will also be used here to improve user coding feeling.



d) Blog Page

This module is used for showing the content of each blog. When users go into the blog page, they could see the blog's partition and its sub partition and click the like or followed button to support this blog. The user is allowed to reply to poster on the bottom of this blog, which provide an efficient way to communicate and solve the problem stated in blog.

The users are allowed to delete their own blog, which will trigger the query to delete the blog. The user only has the access to delete their own blog. They cannot delete the blog that is not posted by them.



e) Searching

In this module, user is allowed to search the relevant blogs by typing the keyword. The detailed implementation will be introduced with the graphic result in the data mining part.

V. Data Mining

A. Motivation and target

Originally, for the blog searching function, we simply apply String match method which simply divide the sentence into pieces of words without any special tokenize, and then search every word separately to match all the blogs in our database to determine whether they are relevant. This methods is practical when the question string is small and the database is relatively small at the same time, since the complexity for this design is O(M * N), where M represent the number of words in user's searching sentence, and N represent all of the words in forum database. The performance becomes worse when the database grows, which means more users post their blog on our forum. This search method will search every word in an equal status and do not identify the key word in the search sentence, which makes it search every word in the same procedure repeat in the database. Each search operation will iterate all of the data blocks that store the information of blog, which generate a huge IO time and significantly reduce the search efficiency.

Due to this limitation, our group decide to create a search engine that could reduce the IO overhead and improve the searching speed. The Searching engine mainly can implement the following functions:

Identify the key word in the search sentence and recognize the patterns in a sentence. The search engine is able to identify the key word and filter stop word to reduce the useless searching procedure. It only uses the useful key word to map the corresponding result in database.

Search engine should create and maintain a data structure to replace the string match method, which take advantage in insertion but have drawback in searching. The Search engine should find the corresponding record in data block in a short time.

The Search engine should maintain an order algorithm, which should sort all the blogs that have been classified into relevant. The search engine should compare the similarity between the original searching question with all of records has been found, and finally return those blogs in a sorted way to user interface.

B. Method

To design such an engine, we need to tokenize the key word in a sentence and construct a special data structure to maintain a short search runtime. So, we divide this procedure into 3 parts via below procedure:

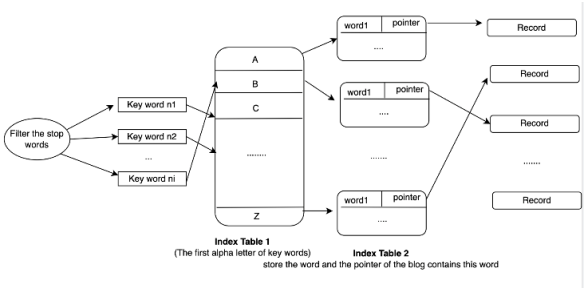
a) Tokenize the key word in a search sentence.

In this procedure, the search engine will use Natural Processing Language method to be training a model which can identify the stop word to filter. For instance, the search engine is responsible to filter the meaning less word such as "my", "the", "an", as which do not play a role in the searching process. This procedure will also count the key word occurrence times. The more occurrence times in a word, the more weight it will have in the searching process. Based on the weight, the search engine is able to compare the similarity of search questions and the blogs in database. More importantly, the search engine should transfer different tense word into the same. Since the search engine identify the word based on its meaning, the word has different tense should be treated equally. For instance, the word "make" is equal to word "made", and the word "built" should be treated as "build".

b) Construct an inverted index table for searching.

Originally, the searching method is to match every word with all data content in database, which is inefficient since it need compare each pair of word even if they are not relevant. This method is good for insertion but not friendly for searching. In our way, the search engine will construct an inverted index table which is a two-level index. The first index is table that store the alpha letter from A to Z, which represent the first letter of a word. The search engine will identify the first letter of the keyword and find the corresponding pointer in first index table which point to the second index table. In The second index table, it stores all the words that have occurred in past blogs that have the same first letter. For example, the first second index table store all the words that have occurred in past blogs, such as "abandon", "abort" and so on. Notice that those words are required to be meaningful, the word like "a", "an" will not be shown in this table, which could speed up the searching process. Hence, in this way, every time when user post a blog, the search engine should spend the extra overhead to spilt and filter the blog content then update both the first index and second index table. Similarly, after user delete its own blog, search engine should update and maintain the index table at the same time. Based on the two-index table,

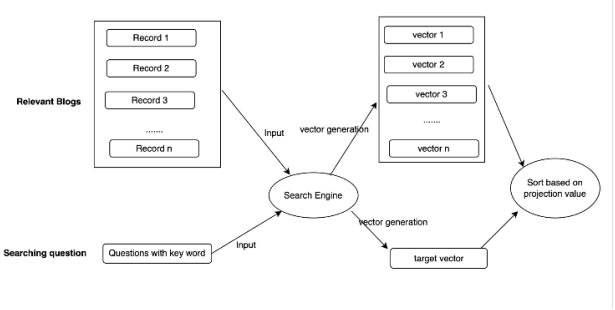
the key word will find the corresponding blog contains this word directly instead of scanning through the whole database. In this design, it takes time in insertion and deletion but efficient in searching runtime. Based on the tradeoff, we choose to implement this search engine, below is the demonstrative figure:



c) Compare the similarity between the question and corresponding result in database.

After we retrieves the corresponding result in database, we need to sort those blogs based on the similarity. The search engine is responsible to design an algorithm that could sort those blogs in an order. In our design, we reference the TF-IDF method, which construct the words into vectors and use projection to compare the similarity. Our search engine will create a word vector for the searching question and those corresponding blog and do the projection in pair to sort those result blog based on projection value. Notice that the vector creation process will consider the word weight. The word that occurs in searching sentence for many times should have the high priority than those words occur less. The word occur in title should have the higher priority than those occur in content. The search engine should also take care of it when doing the vector creation and projection. Below is the demonstrative figure:

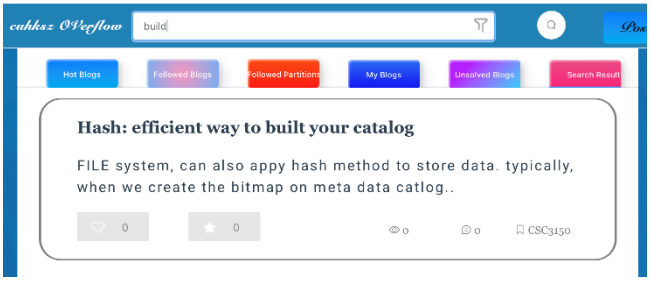
C. Outcome Analysis



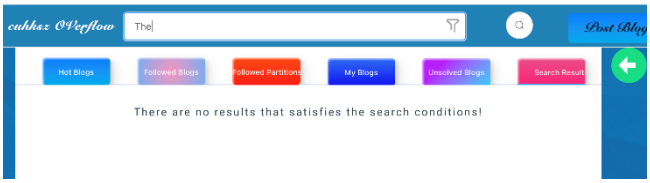
Based on the search engine design, the experiment mainly tests three parts. The first part is the tokenize of the search engine. It tests whether the search engine could transfer the tense, filter the stop words. The second parr is going to test the performance of the searching process, which is to examine its searching speed and use precision-recall method to examine its precision. The third part is to test the similarity sortation, which aim to examine the sortation algorithm accuracy.

a) Tokenize Test

We use two demonstrative test case here to report the tokenize result. The first case is for the tense identification. The word has the same meaning but with different tense should be treat as the same. It means that when user search the question with one tense, the word has the same meaning with all the tense should be retrieved. For instance, here we search the word "build", the blog contain either "built" or "build" should be retrieved. Below is the demonstrative figure:



The second test case is for stop words filtration. The word that has no useful meaning should be ignored when searching. The stop words contain "my", "he", "the", "a", and so on. We create a stop words list, so every word in searching should be identify whether it is a stop word when searching. If the word is be classified as the stop word, the search engine should terminate the following retrieve part and return "no result found" to User Interface. Here comes the demonstrative figure:



The third test case is for case insensitive test. The search engine should retrieve the word in a case insensitive way, which means the words in upper letter or in lower letter won't change the retrieval result. Here comes the example when we search word "file", and the upper case "FILE" will be retrieved at the search runtime.

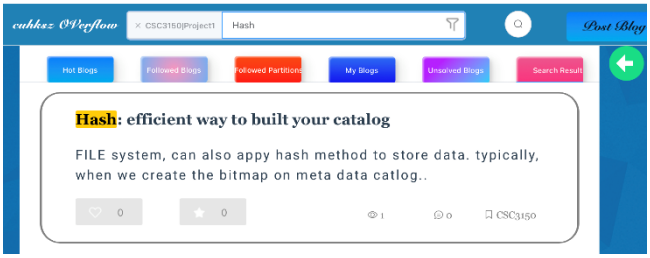
b) Searching speed and accuracy evaluation

Based on the design, we could drive the searching complexity is around O(1) for each word, since each word have the index directly interact with the record block. Hence the whole retrieval speed is O(N), where N represent the number of key words in the search sentence, which is significantly less the original one O(M*N). where M represent the number of words in database, which is a giant number. More importantly, The IO overhead will reduce significantly in searching runtime, Because the search engine only needs to execute one IO operation (read) after find the blog by index. Instead, the original one will execute more IO operations since it need to read all the

blogs located in data record. The performance of the searching speed will be improved dramatically.

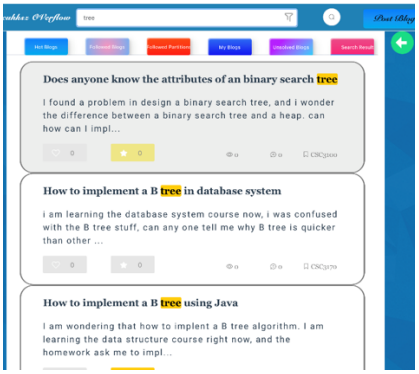
For the accuracy evaluation, our searching engine typically will retrieve all the documents that contains the keyword. Assuming that in our database design the blog is relevant if and only if the document contains the key word in the searching questions. This design is not robust enough since there is a possibility that the documents do not contain any key word but still relevant to the searching question. However, for the data in our database, this assumption is practical enough. In this way, since all the blog containing the key word will be retrieved, the search engine could achieve a high recall rate (the documents retrieved that are relevant in database/ all the documents that retrieved). However, this method sacrifices the precision rate. There is a limitation for our design which is the searching engine only support OR operations for the key word searching. Basically, each key word will be connected in an OR logical operations. The search engine does not support AND or XOR operations to narrow down the search filed. Hence, the precision rate will shrink in this design.

To cover the problem stated above, we design another filter to narrow down searching filed hence improve the searching precision. User are allowed to choose the partition and sub partition that they want to narrow down. Based on this filter, only the blogs that exactly in this partition will be retrieved. Hence search engine are able to narrow down the searching scope in order to increase the precision rate. Below is the demonstrative figure:

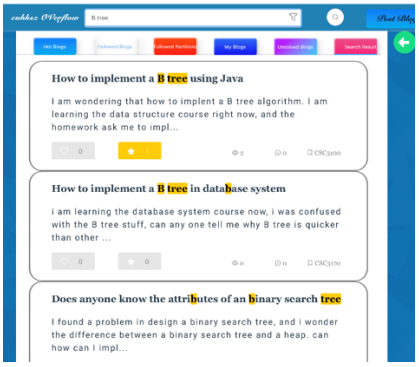


c) Similarity Sortation Algorithm

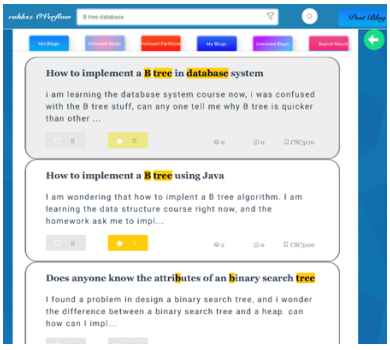
As we stated above, the search engine use OR logical operations in each key word retrieval. Hence, we need to similarity algorithm to sort those results. In the test case, the first searching word is "tree", then the engine will retrieve all the blogs that contains the word "tree". The most relevant one is the blog that contains the greatest number of "trees".



The second test case is "B tree", which will retrieve all blogs contain "B" or "tree". But the most relevant one is the blog contains both "B" and "tree", which shows the high similarity.



The third test case is "B tree database", which will retrieve the blogs contain either of three words. But the most relevant one is the one which contains all of the three words, as the figure shown, the order of blog is changed.



VI. Conclusion

In conclusion, our project is about implementation, optimization, and analysis of an online forum database.

For the E-R diagram and database design, we optimized all tables into 3NF form to avoid redundancy as much as we could. For queries, we use different ways, including direct SQL, SQL with Python, and Query Set to access and manipulate different data in an efficient way. For the index optimization, we designed a B tree searching method to increase our query speed in order to provide users a faster and more fluence searching experience. We also designed

and implemented a friendly user interface for users. For the data mining, we designed and implemented an effective search engine with machine learning NLP and inverted index tables.

VII. Contribution & Self-Evaluation

For the workload, all team members evenly contribute to this team project.

In this project, we build an online forum system which is beneficial to the programmers in CUHKSZ. From implementations and functions, the project has the following strengths:

1. Users have easy access to the webpage due to the cloud deployment and domain name system.

2. Inverted table is applied to the search engine, which improves the efficiency of searching.

3. A relative fancy UI improves the user experience when using the online forum system.

4. The system provides various relationships, for instance, user can like a question, like an answer, follow a question and follow a group.

To further improve our system, we analyzed some drawbacks. Our system can be robust enough by providing larger amounts of relationships, for instance, user can follow another user and user can send private messages to another user.

References

Explosion, I (2022, May, 3) spaCy (Version 3.2). URL: https://github.com/explosion/spaCy

Gidden T. (2012, Mar, 20) mysql-inverted-index. URL: https://github.com/tomgidden/mysql-inverted-index

Ksiazek K. (2015, Oct, 1) *An Overview of MySQL Database Indexing*. Accessed on: May 1, 2022. Available: https://severalnines.com/database-blog/overview-mysql-database-indexing

MySQL Modify and Delete Index. Accessed on May 1, 2022. Available: http://c.biancheng.net/view/2607.html