**Project Title - Discussion Forum**

Team Members: Chaitanya Kintali   
Amulya Prodduturi

Prashanth Minkuri

Di Wu

## ***PROJECT PROPOSAL***

**Content, Scope and Objectives**

**Objective:**

The objective of this project is to build a web based discussion forum which acts as a platform where Users can post their queries, read about different topics, answer and solve the questions posted. Each post/query can be tagged with a category. A user can choose the categories that he is interested in and will be shown all the posts that belong to that category. Once a query is solved, the user who posted that query can mark it done and close the thread. A leaderboard is maintained to track the contributions made by the user to the forum. An admin can have the privileges to delete a post, ban a user and monitor the forum.

## ***PROJECT ENVIRONMENT***

Languages used :

|  |  |
| --- | --- |
| Frontend | HTML5, CSS3, Bootstrap4, Javascript |
| Middleware | Java(1.8) and Spring Boot(2.0) |
| Database | MySql (5.5) |

## 

## ***HIGH LEVEL REQUIREMENTS***

### **Initial user roles**

|  |  |
| --- | --- |
| **User Role** | **Description** |
| Admin | Admin monitors the forum. |
| Student (user) | Users can post their queries and answer the queries already posted. |

### **Initial user story descriptions**

|  |  |
| --- | --- |
| **Story ID** | **Story description** |
| US1 | As a User, I want to create an account. |
| US2 | As a User, I want to log in/ log out from my account. |
| US3 | As a User, I want to choose topic(s) that I am interested in, so that I can easily find the related queries. |
| US4 | As a User, I can post queries. |
| US5 | As a User, I can answer queries. |
| US6 | As a User, I can comment on others’ comments. |
| US7 | As a User, I can tag several categories to queries. |
| US8 | As a User, I can report queries and comments to Admin |
| US9 | As an Admin, I want to create a topic to classify the queries. |
| US10 | As an Admin, I can delete/hide the queries and answers posted in the forum. |

**Note**: User can be referred to both Student and Admin

## 

## ***HIGH LEVEL CONCEPTUAL DESIGN***

Entities:

**ROLE**

Attributes:

Role\_id (single-valued)

Role\_Name (single-valued)

**User**

Attributes:

User\_ID (single-valued)

User\_Name(single-valued)

Role\_id (single-valued)

Date\_of\_birth [composite, single-valued]

addr\_line1 [composite]

addr\_line2 [composite]

city (single-valued)

state (single-valued)

zip\_code (single-valued)

**Queries**

Query\_ID (single-valued)

Query\_Desc [composite]

Posted\_Date (single-valued)

User\_ID (single-valued)

Resolved (single-valued)

isActive (single-valued)

**Query\_detail**

Querydetail\_id(single-valued)

Query\_ID (single-valued)

Category\_ID (single-valued)

**Comments**

Comment\_ID(single-valued)

Comment\_Desc [composite]

Comment\_Date [composite]

User\_ID(single-valued)

Query\_Id (single-valued)

Commented\_On (single-valued)

Comment\_desc [composite]

**Categories**

Category\_ID (single-valued)

Category\_name (single-valued)

**Leaderboard**

User\_id(single-valued)

User\_name (single-valued)

Points(single-valued)

**Message**

MessageId (single-valued)

SenderId (single-valued)

ReciverId(single-valued)

Message [composite]

IsRead bool, (single-valued)

Relationships:

**User** has **role**

Cardinality: Many to One

Participation:

User has Partial participation

Role has Total participation

**User** posts **Queries**

Cardinality: One to many

Participation:

User has partial participation

Queries has total participation

**User** answers **Comments**

Cardinality: One to many

Participation:

User has partial participation

Comments has total participation

**Query** has **Comments**

Cardinality: One to many

Participation:

Query has partial participation

Comments has total participation

**Query** has **Categories**

Cardinality: many to many

Participation:

Query has Total participation

Categories has partial participation

**User** hide/delete **Queries**

Cardinality: One to many

Participation:

User has partial participation

Queries has total participation

**User** sends **Message**

Cardinality : One to many

Participation:

User has partial participation

Message has total participation

## ***LOGICAL DESIGN WITH NORMAL FORM IDENTIFICATION***

Table : **Role**

Columns:

Role\_id (PK)

Role\_Name (single-valued)

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: Role\_id

**Justification:** Role\_id is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

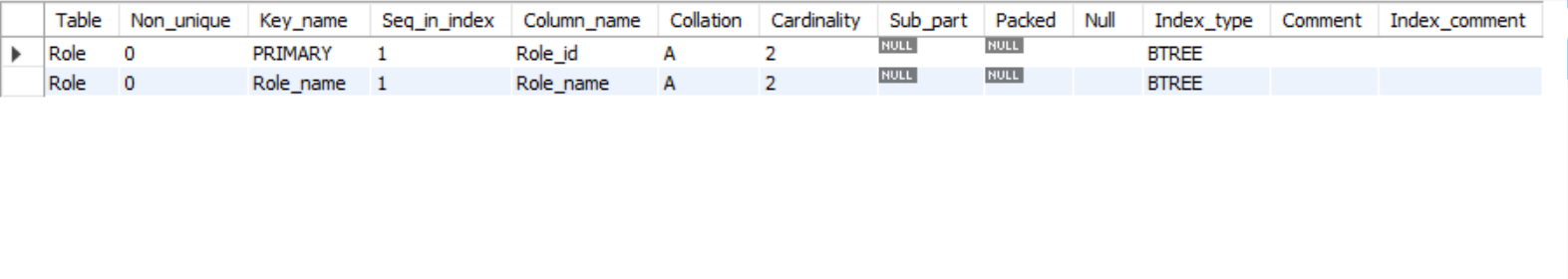


Table : **User**

Columns:

User\_ID (PK)

User\_Name

Role\_id [foreign key; references PK of **Role**]

Date\_of\_birth [composite, single-valued]

addr\_line1

addr\_line2

city

state

zipcode

**Highest Normalization: 2NF**

**Justification:**

1. The table is in 1NF

2. There are no partial dependencies

**Violations:**

Table contains transitive dependencies(A non-prime attribute determining the

non-prime attribute). Non-prime attribute Zip-code can be used to determine

State(non-prime attribute). Splitting the entity to make it to 3NF would result in performance overhead for joining the table for an attribute.

**Indexes**:

Index 1: clustered

Columns: User\_id

**Justification:** User\_id is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

Index 2: non-clustered

Columns: User\_name

**Justification:** It is very common that users search for details of some specific users through their names. So query are often used in some JOIN queries as condition. So we can create index for User\_name to increase the speed.

It has already been created automatically.

CREATE INDEX index\_UserUser\_id ON User (User\_name);

SHOW INDEX FROM User;

Index 3: non-clustered

Columns: addr\_line1;addr\_line2;city

**Justification:** It is very common that users search for details of some specific users through their address. So query are often used in some JOIN queries as condition. So we can create index for address to increase the speed.

It has already been created automatically.

CREATE INDEX index\_UserAddress ON User (addr\_line1,addr\_line2,city);

SHOW INDEX FROM User;

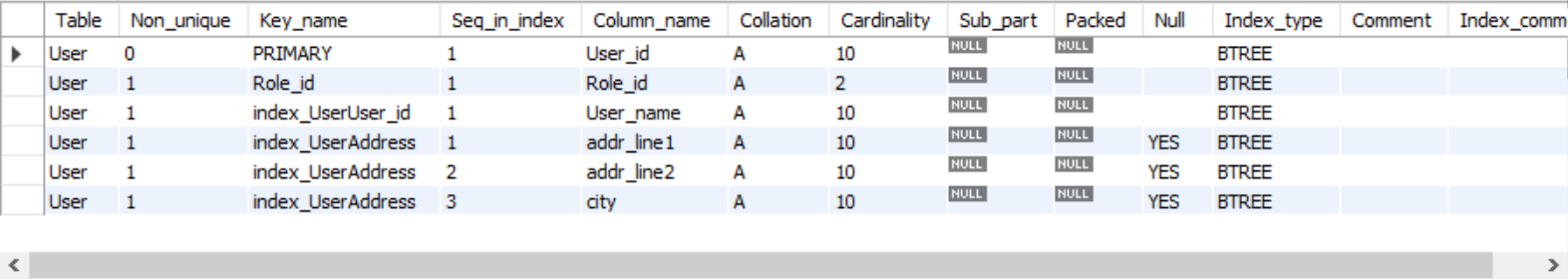


Table : **Queries**

Columns:

Query\_ID (PK)

Query\_Desc [composite]

Posted\_Date (single-valued)

User\_ID [foreign key; references PK of **User**]

Resolved (single-valued)

isActive (single-valued)

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: Query\_ID

**Justification:** Query\_ID is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

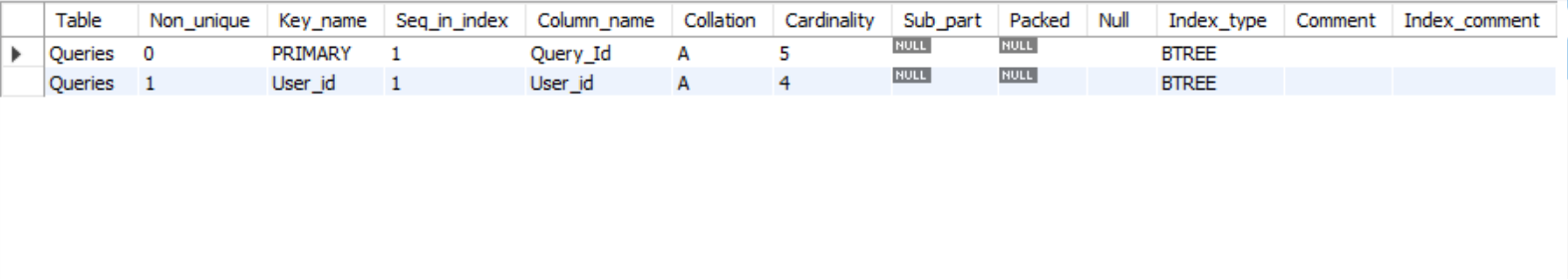
Index 2: non-clustered

Columns: User\_ID

**Justification:** It is very common that users search for Query of some specific user. So User\_ID are often used in some JOIN queries as condition. So we can create index for User\_ID to increase the speed.

It has already been created automatically.

SHOW INDEX FROM Queries;



**Cross reference table:**

Table : **Query\_detail**

Columns:

Querydetail\_id (PK)

Query\_ID [foreign key; references PK of **Queries**]

Category\_ID [foreign key; references PK of **Categories**]

**Cross reference table justification:** Query\_detail table is created to represent the many to many relationship between Queries and Categories

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: Query\_detail\_id

**Justification:** Query\_detail\_id is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

Index 2: non-clustered

Columns: Query\_ID

**Justification:** It is very common that users search for details of some specific query. So query are often used in some JOIN queries as condition. So we can create index for Query\_ID to increase the speed.

It has already been created automatically.

Index 3: non-clustered

Columns: Category\_ID

**Justification:** It is very common that users search for details of some specific query through category. So Category\_ID are often used in some JOIN queries as condition. So we can create index for Category\_ID to increase the speed.

It has already been created automatically.

SHOW INDEX FROM Query\_detail;

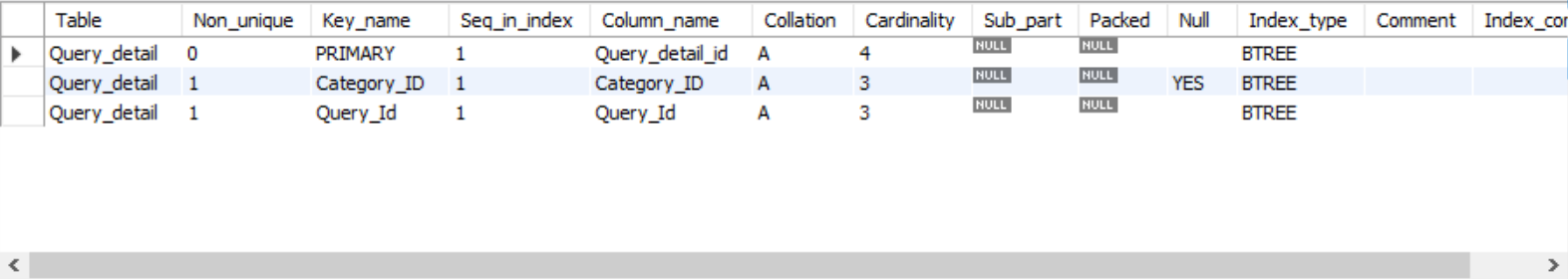


Table : **Comments**

Columns:

Comment\_ID (PK)

Comment\_Desc

Comment\_Date [composite]

User\_ID [foreign key; references PK of **User**]

Query\_Id [foreign key; references PK of **Queries**]

Commented\_On

Comment\_desc

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies

**Indexes**:

Index 1: clustered

Columns: Comment\_ID

**Justification:** Comment\_ID is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

Index 2: non-clustered

Columns: User\_id

**Justification:** It is very common that users search for comments of some specific user. So we can create index for user\_id to increase the speed.

Index 3: non-clustered

Columns: Commented\_On

**Justification:** It is very common that users search for comments of some specific queries. So we can create index for attribute Commented\_On to increase the speed.

Create INDEX index\_CommentUser\_ID ON Comments(User\_id);

Create INDEX index\_Comment Commented\_On ON Comments(Commented\_On);

SHOW INDEX FROM Comments;

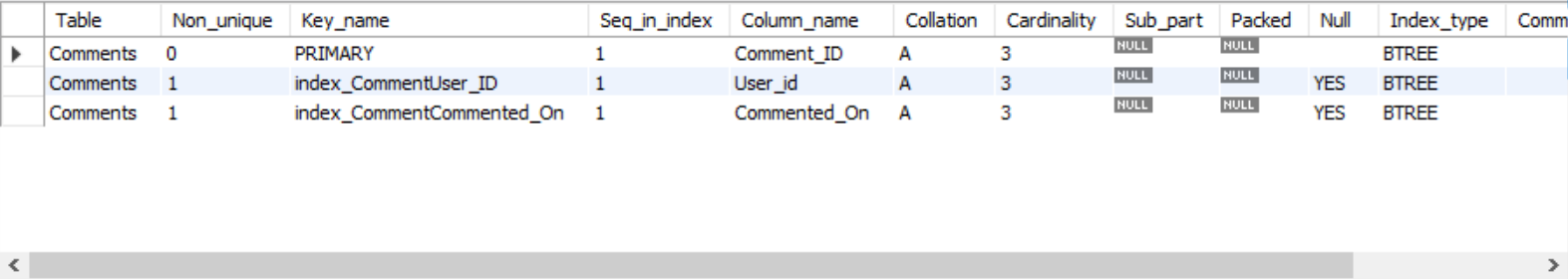


Table : **Categories**

Columns:

Category\_ID (PK)

Category\_name

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: Category\_ID

**Justification:** Category\_ID is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries. It has already been created automatically.

Index 2: non-clustered

Columns: Category\_name

**Justification:** It is very common to search for a specific category according to its name. So we can create an index for them so that we can increase the speed when doing queries.

CREATE INDEX index\_Categoryname ON Categories(Category\_name);

SHOW INDEX FROM Categories;

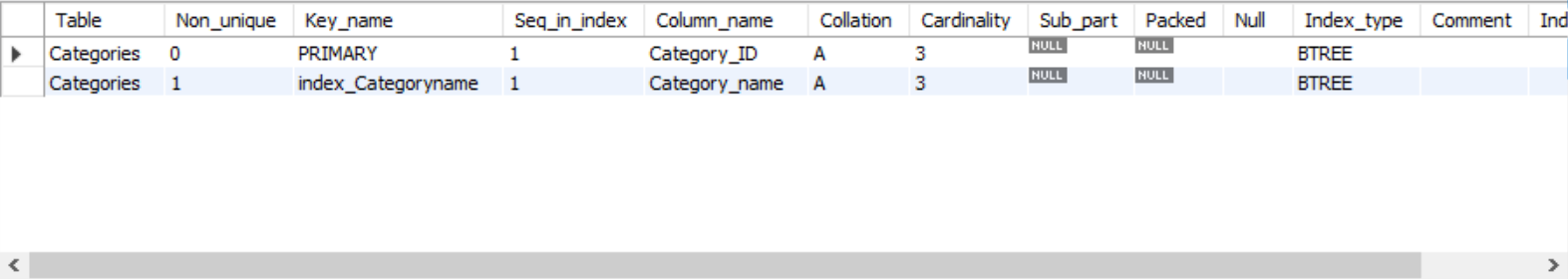


Table : **Leaderboard**

Columns:

leaderrank (PK)

User\_id [foreign key; references PK of **User**]

Points (single-valued)

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: User\_id

**Justification:** User\_id is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

Index 2: non-clustered

Columns: User\_Name

**Justification:** It is very common that users search for leaderrank information of some specific user. So we can create index for User\_name to increase the speed.

CREATE INDEX index\_LeaderboardUser\_id ON Leaderboard (User\_Name);

SHOW INDEX FROM Leaderboard;

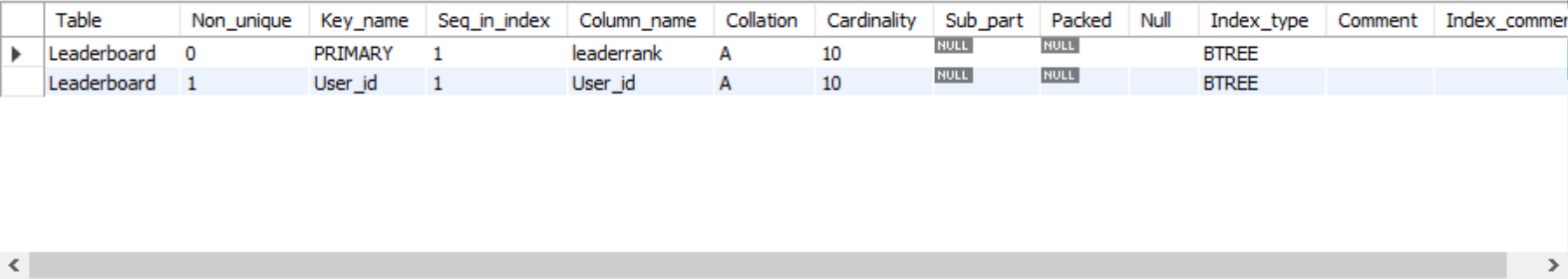


Table : **Message**

Columns:

MessageId (PK)

SenderId [foreign key; references PK of **User**]

ReciverId [foreign key; references PK of **User**]

Message [composite]

IsRead bool, [single-valued]

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: MessageID

**Justification:** MessageID is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

Index 2: non-clustered

Columns: SenderID

**Justification:** It is very common that users search for message of some specific sender. So senderID are often used in some JOIN queries as condition. So we can create index for senderID to increase the speed.

It has already been created automatically.

Index 3: non-clustered

Columns: ReceiverID

**Justification:** It is very common that users search for message of some specific receiver. So receiverID are often used in some JOIN queries as condition. So we can create index for receiverID to increase the speed.

It has already been created automatically.

SHOW INDEX FROM Message;

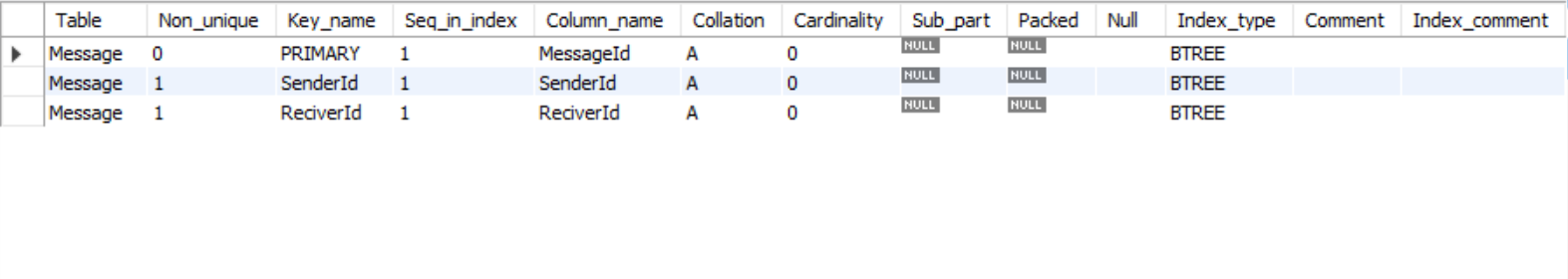


Table : **Query\_AuditBackup**

Columns:

Query\_ID (PK)

Query\_Desc [composite]

Posted\_Date (single-valued)

User\_ID [foreign key; references PK of **User**]

Resolved (single-valued)

isActive (single-valued)

modified\_timestamp

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: Query\_ID

**Justification:** Query\_ID is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

Index 2: non-clustered

Columns: User\_ID

**Justification:** It is very common that users search for details of some specific users through their ID. So query are often used in some JOIN queries as condition. So we can create index for User\_ID to increase the speed.

It has already been created automatically.

SHOW INDEX FROM Query\_AuditBackup;

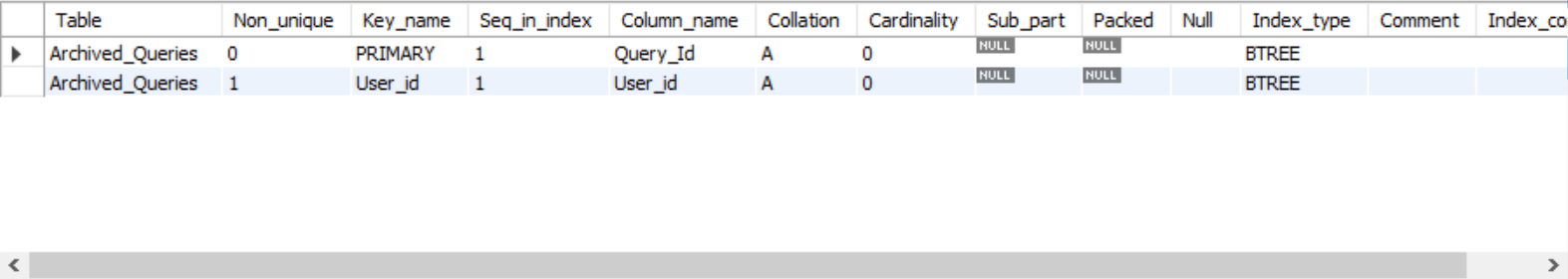


Table : **Backup\_Queries**

Columns:

Query\_ID (PK)

Query\_Desc [composite]

Posted\_Date (single-valued)

User\_ID [foreign key; references PK of **User**]

Resolved (single-valued)

isActive (single-valued)

**Highest Normalization: 4NF**

**Justification:**

1. The table is in BCNF

2. The table does not have multi-valued dependencies.

**Indexes**:

Index 1: clustered

Columns: Query\_ID

**Justification:** Query\_ID is the primary key of this table, and it is necessary to have at least one unique attribute in a table. We are using the order of this clustered index to store our records and also using this order to scan all the record when doing any queries.

It has already been created automatically.

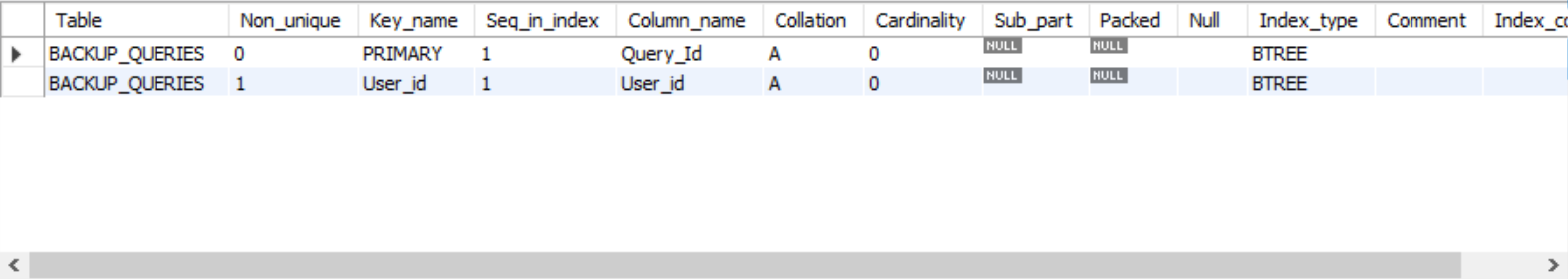
Index 2: non-clustered

Columns: User\_ID

**Justification:** It is very common that users search for details of some specific users through their ID. So query are often used in some JOIN queries as condition. So we can create index for User\_ID to increase the speed.

It has already been created automatically.

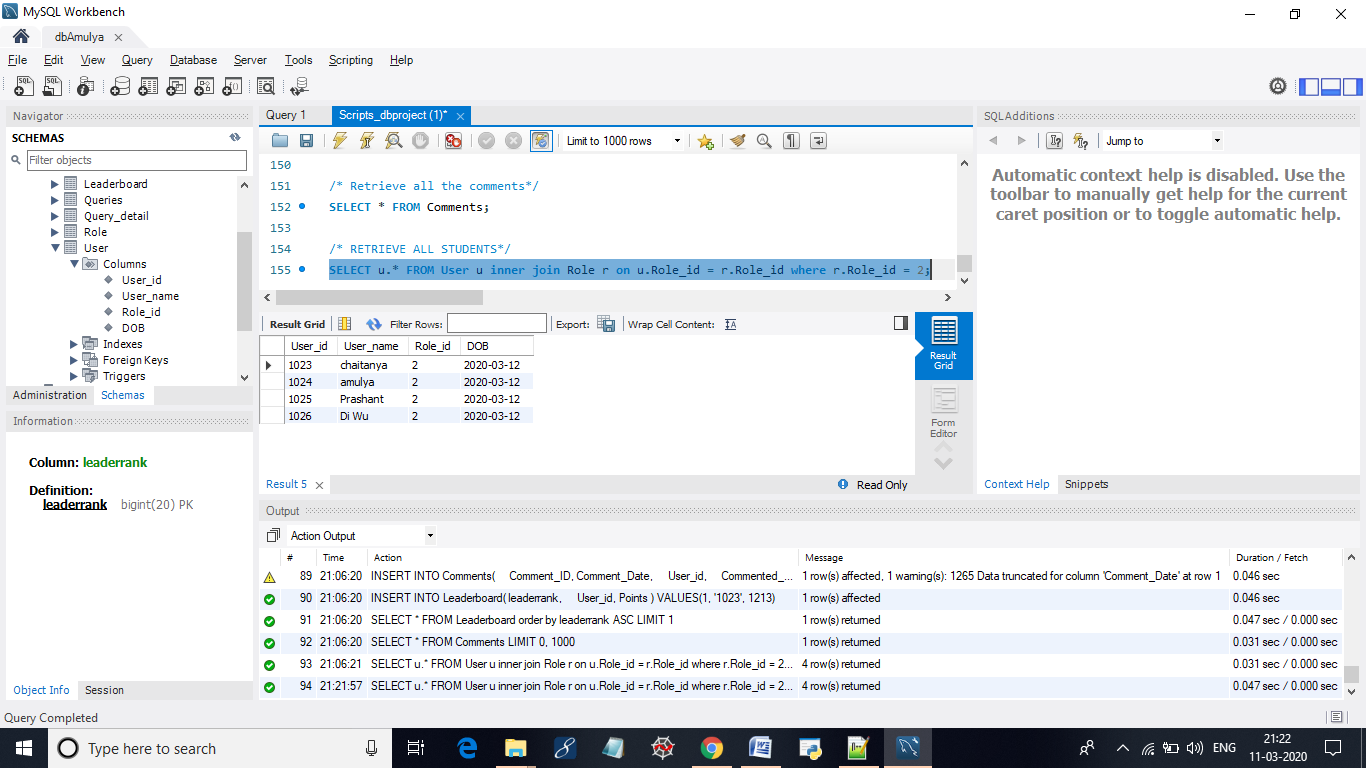
SHOW INDEX FROM BACKUP\_QUERIES;



## ***SQL QUERIES***

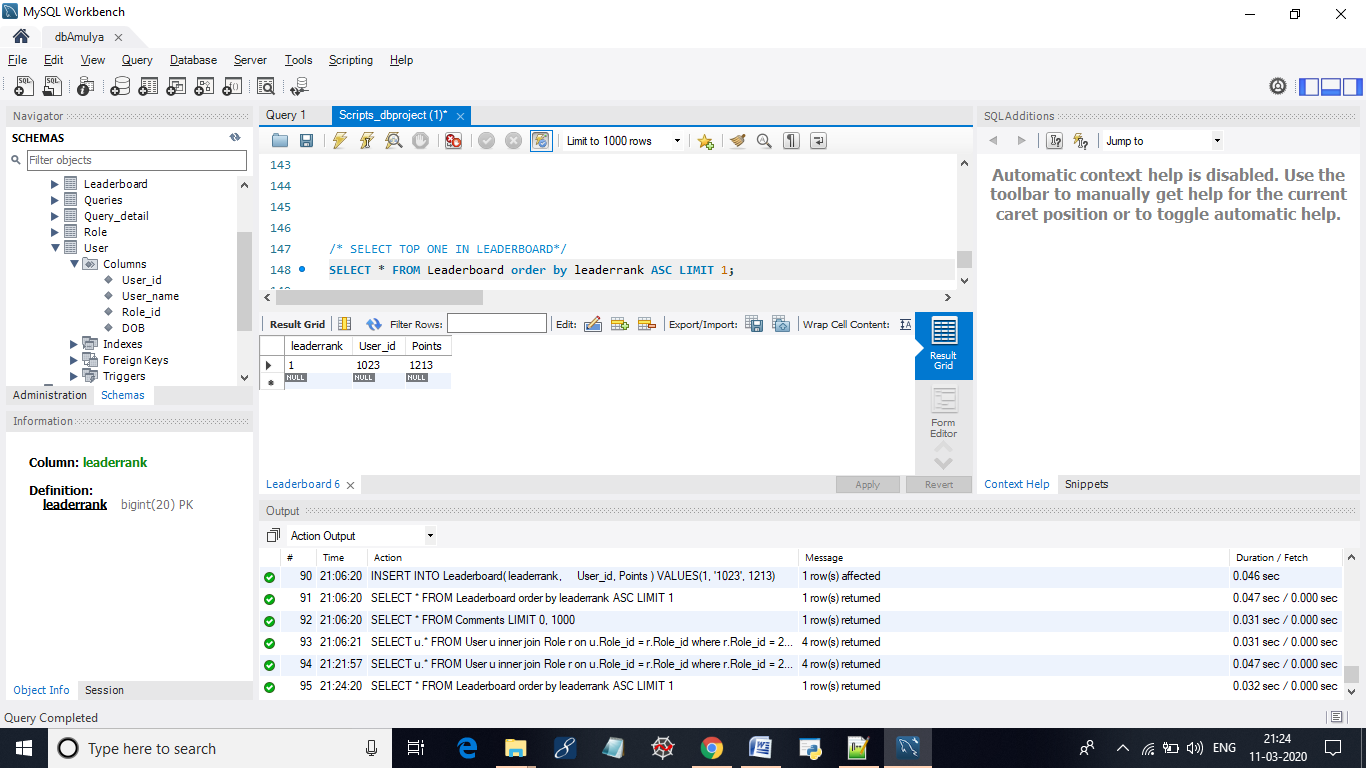
**1. Query to retrieve all students:**

SELECT u.\* FROM User u inner join Role r on u.Role\_id = r.Role\_id where r.Role\_id = 2;



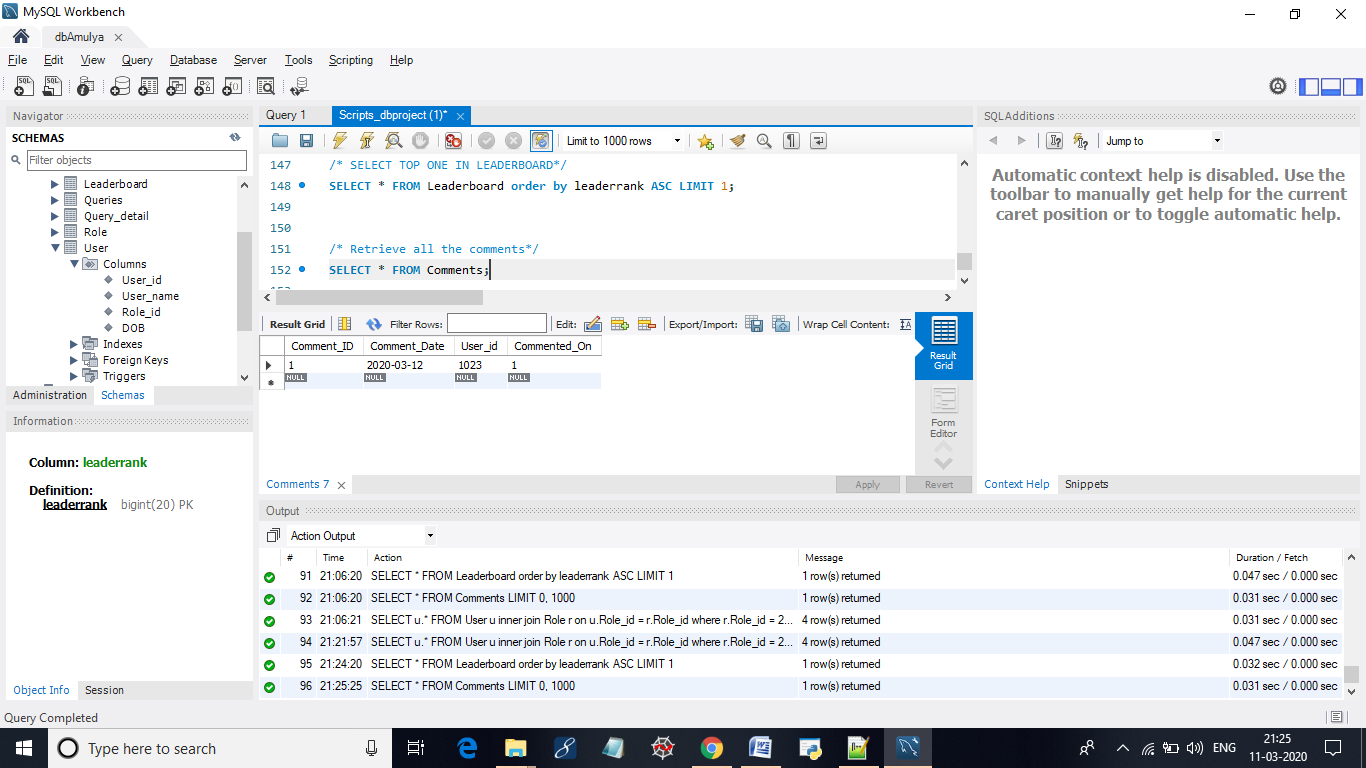
**2. Query to select top one in Leaderboard:**

SELECT \* FROM Leaderboard order by leaderrank ASC LIMIT 1;

****

**3. Query to retrieve all comments:**

SELECT \* FROM Comments;



**4. Display all the queries which belong to both computerscience and Mathematics**

select Query\_id, Query\_desc from Queries where Query\_id in

( select Query\_id from Query\_detail inner join Categories on

Query\_detail.Category\_ID = Categories.Category\_ID

where Category\_name = "computer science")

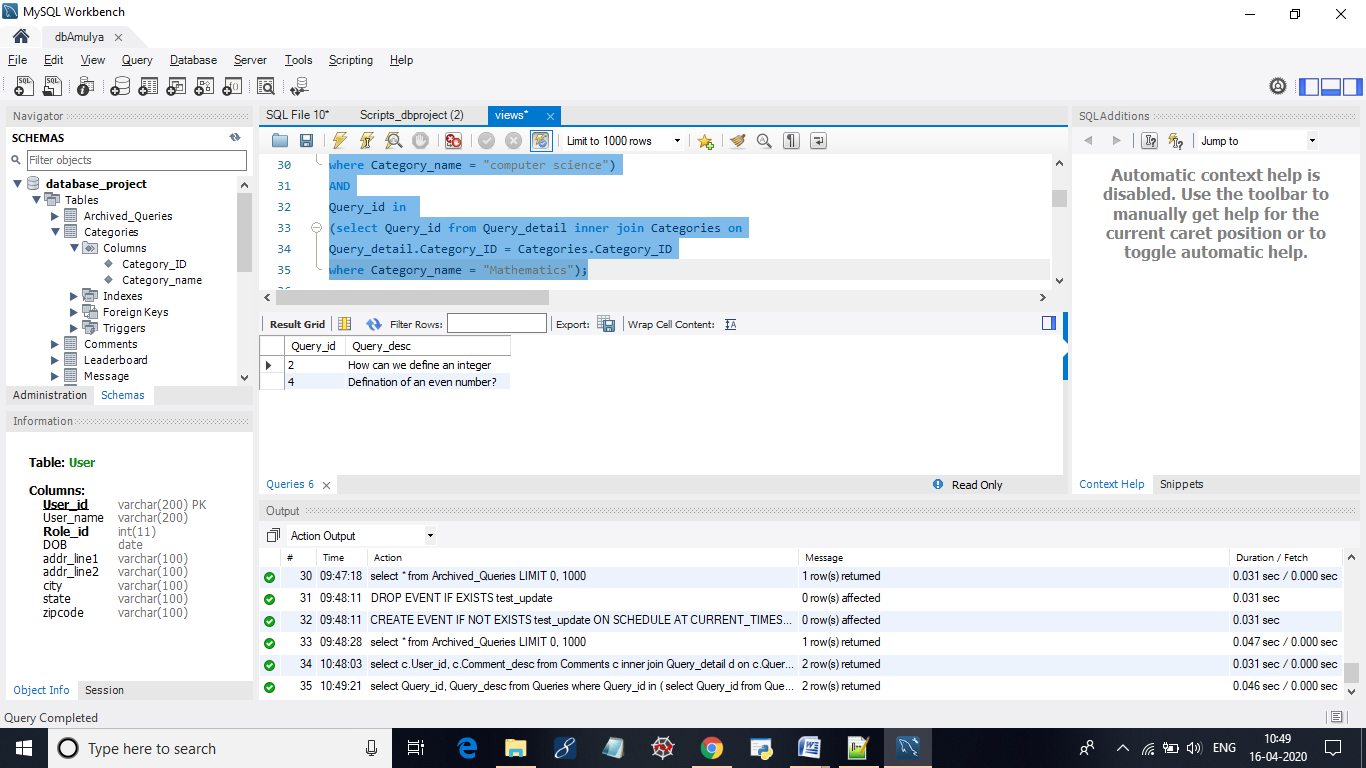
AND

Query\_id in

(select Query\_id from Query\_detail inner join Categories on

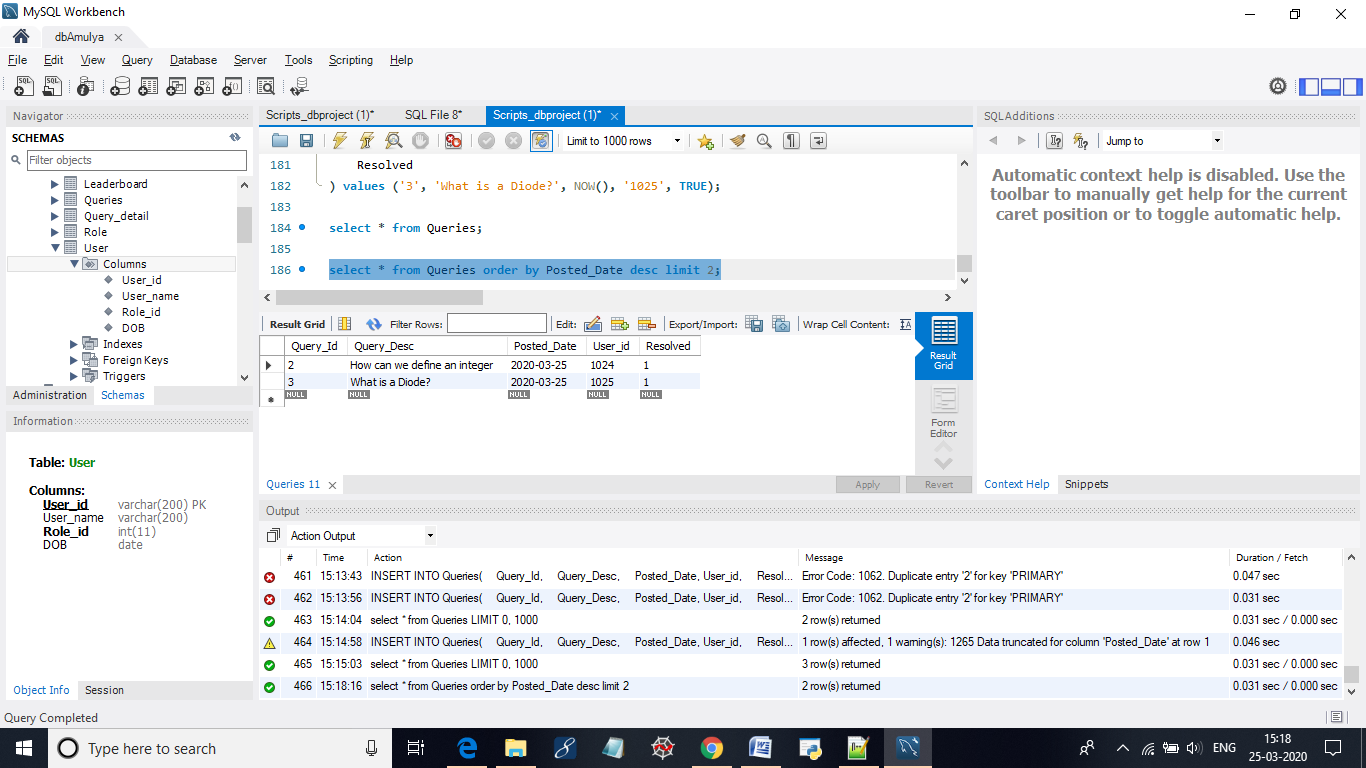
Query\_detail.Category\_ID = Categories.Category\_ID

where Category\_name = "Mathematics");



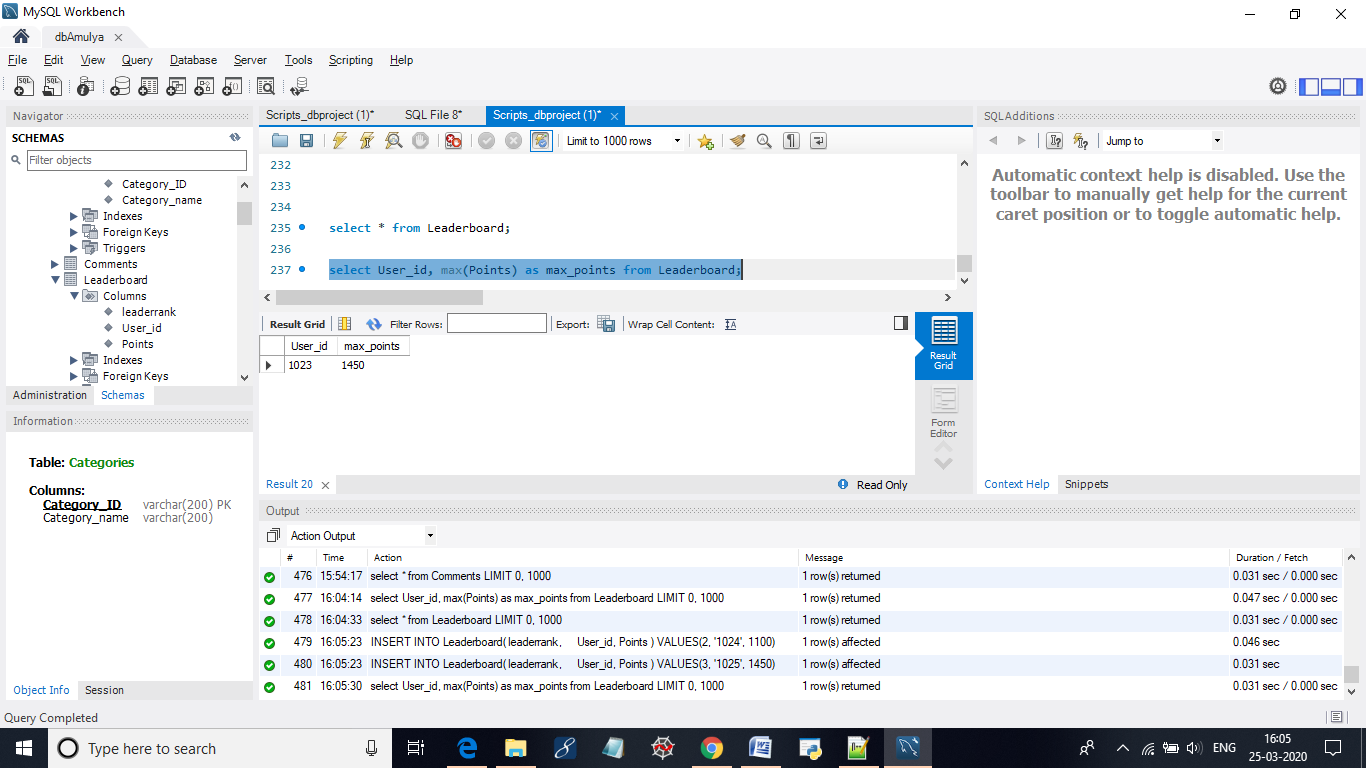
**5 .Query to select the two most recent queries posted by the users.**

select \* from Queries order by Posted\_Date desc limit 2;



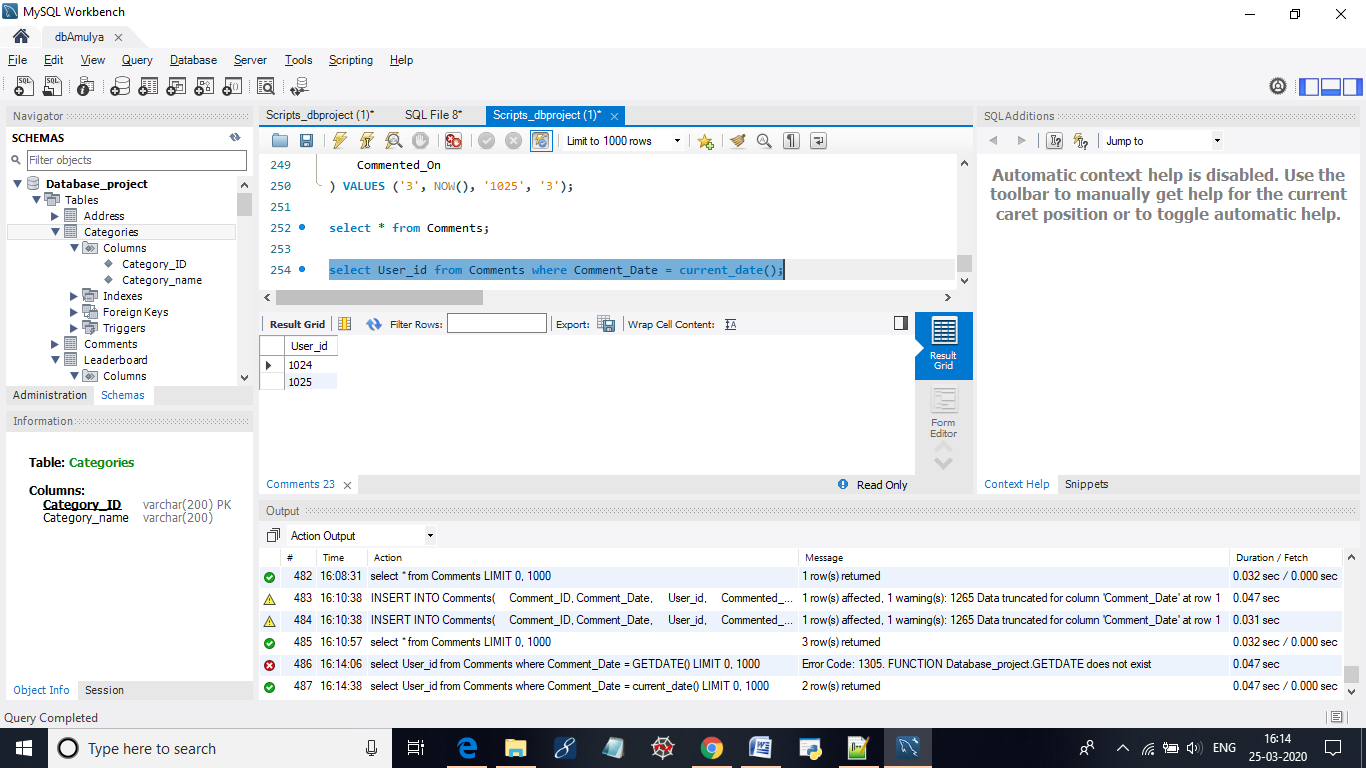
**6. Select the user who has scored maximum points**

select User\_id, max(Points) as max\_points from Leaderboard;



**7. Display all the users who has commented on the queries which are posted today(current date)**

select User\_id from Comments where Comment\_Date = current\_date();



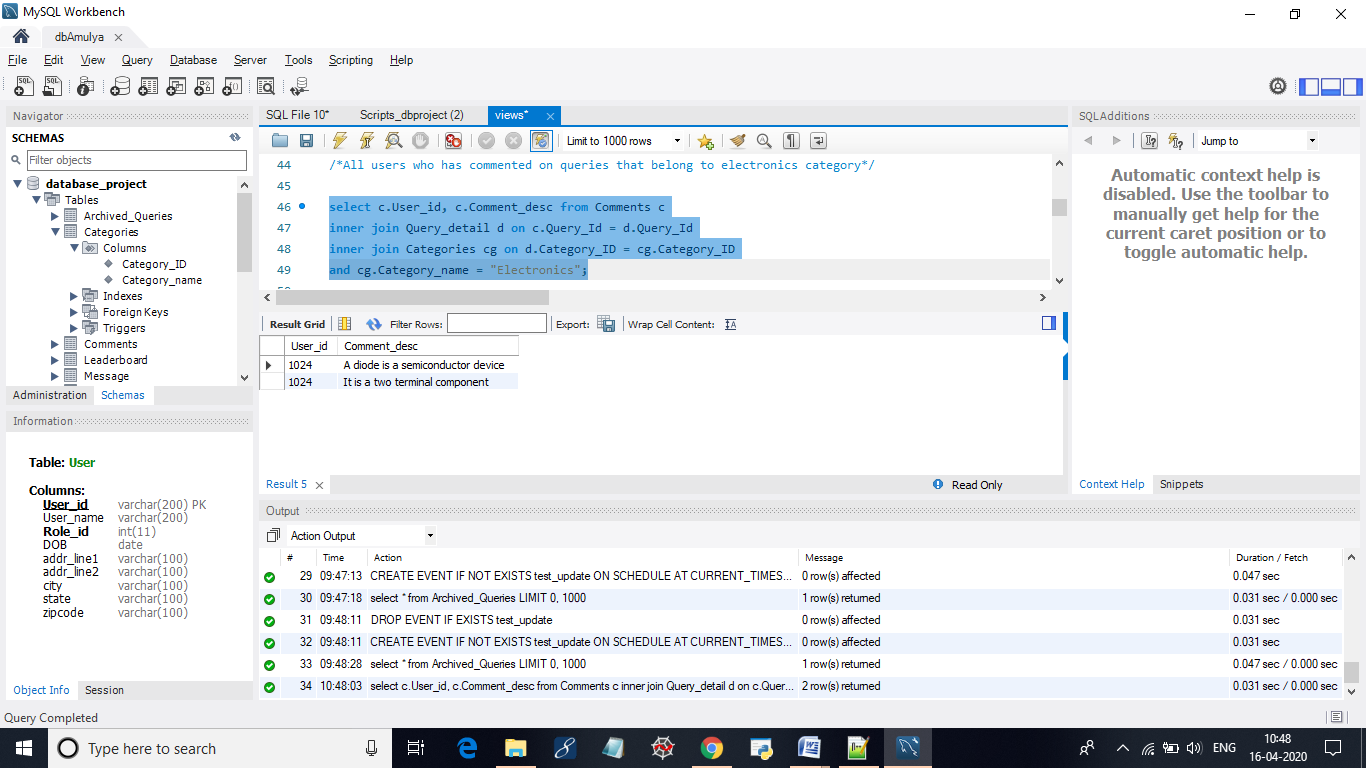
**8. Display all users who has commented on queries that belong to electronics category**

select c.User\_id, c.Comment\_desc from Comments c

inner join Query\_detail d on c.Query\_Id = d.Query\_Id

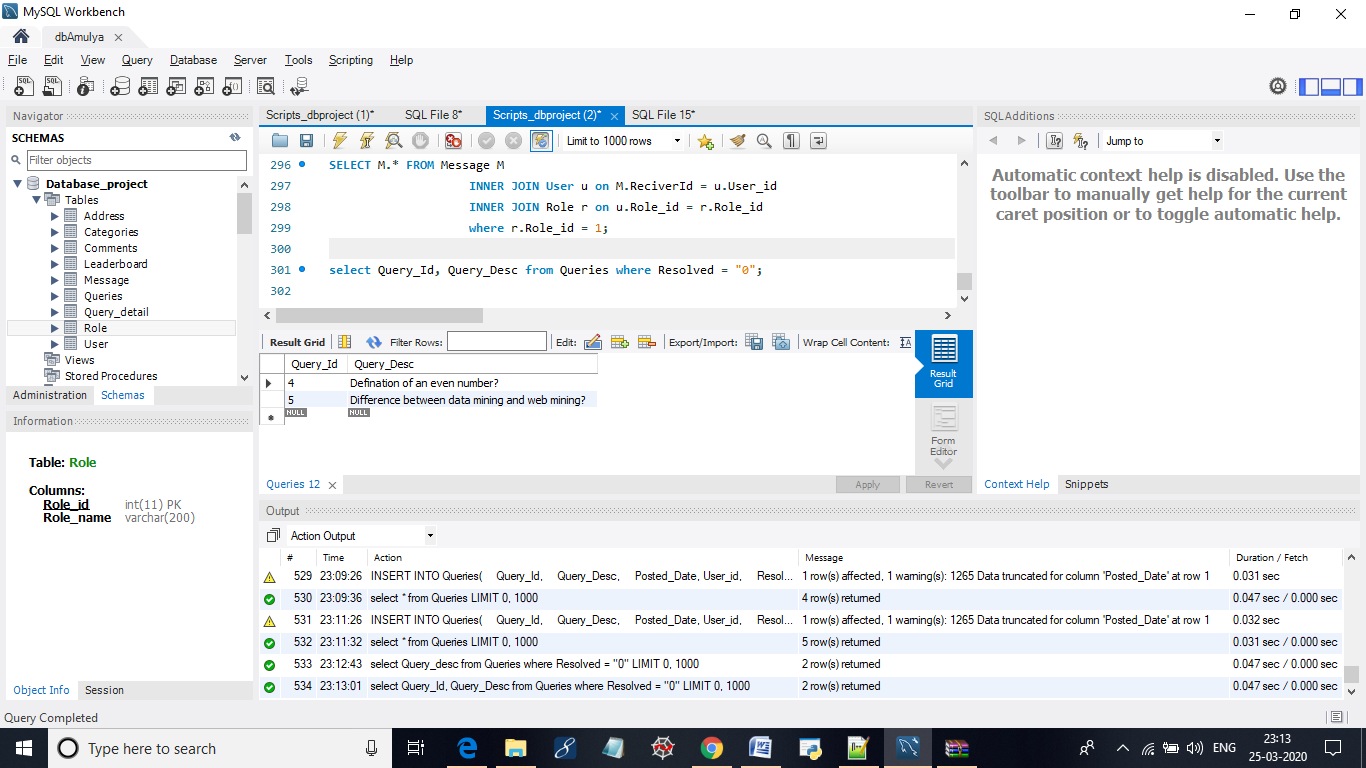
inner join Categories cg on d.Category\_ID = cg.Category\_ID

and cg.Category\_name = "Electronics";



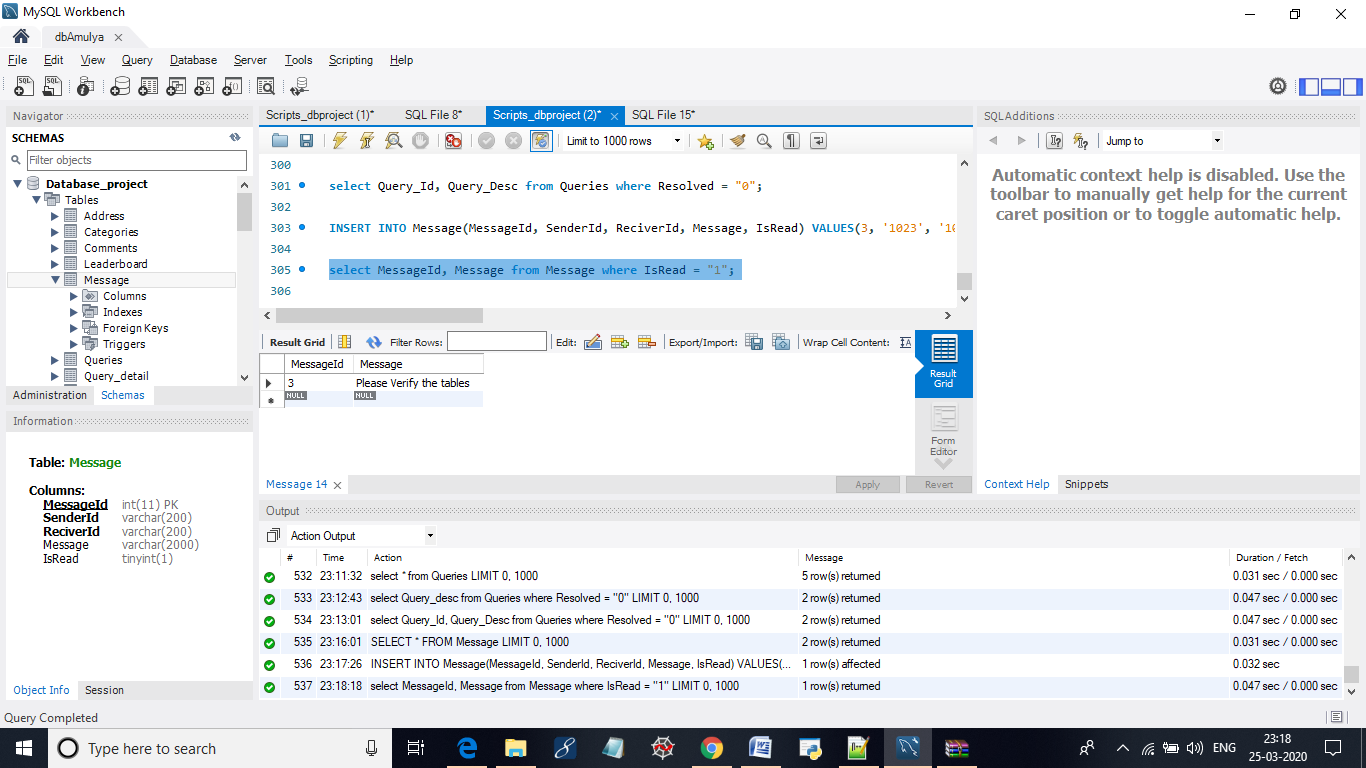
**9. Select all the Queries which are not resolved by the users**

select Query\_Id, Query\_Desc from Queries where Resolved = "0";



**10. Find the list of all messages which are read by Admin**

select MessageId, Message from Message where IsRead = "1";



## ***VIEWS AND STORED PROGRAMS:***

**View:** student

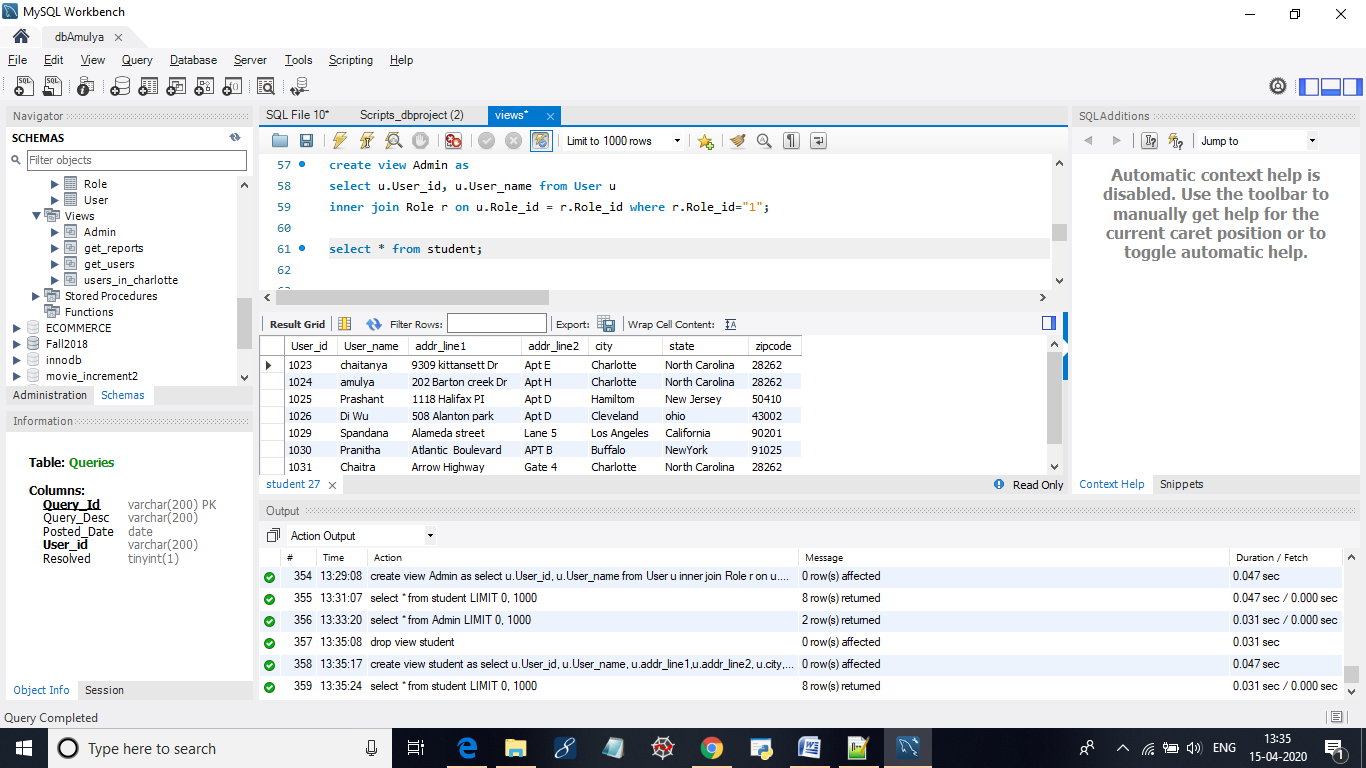
Goal: This view is created to get the list of all the students in the system

create view student as

select u.User\_id, u.User\_name, u.addr\_line1,

u.addr\_line2, u.city, u.state, u.zipcode from User u

inner join Role r on u.Role\_id = r.Role\_id where r.Role\_id="2";



**View:** get\_reports

Goal: This view is created to display list of all the reports sent to admin

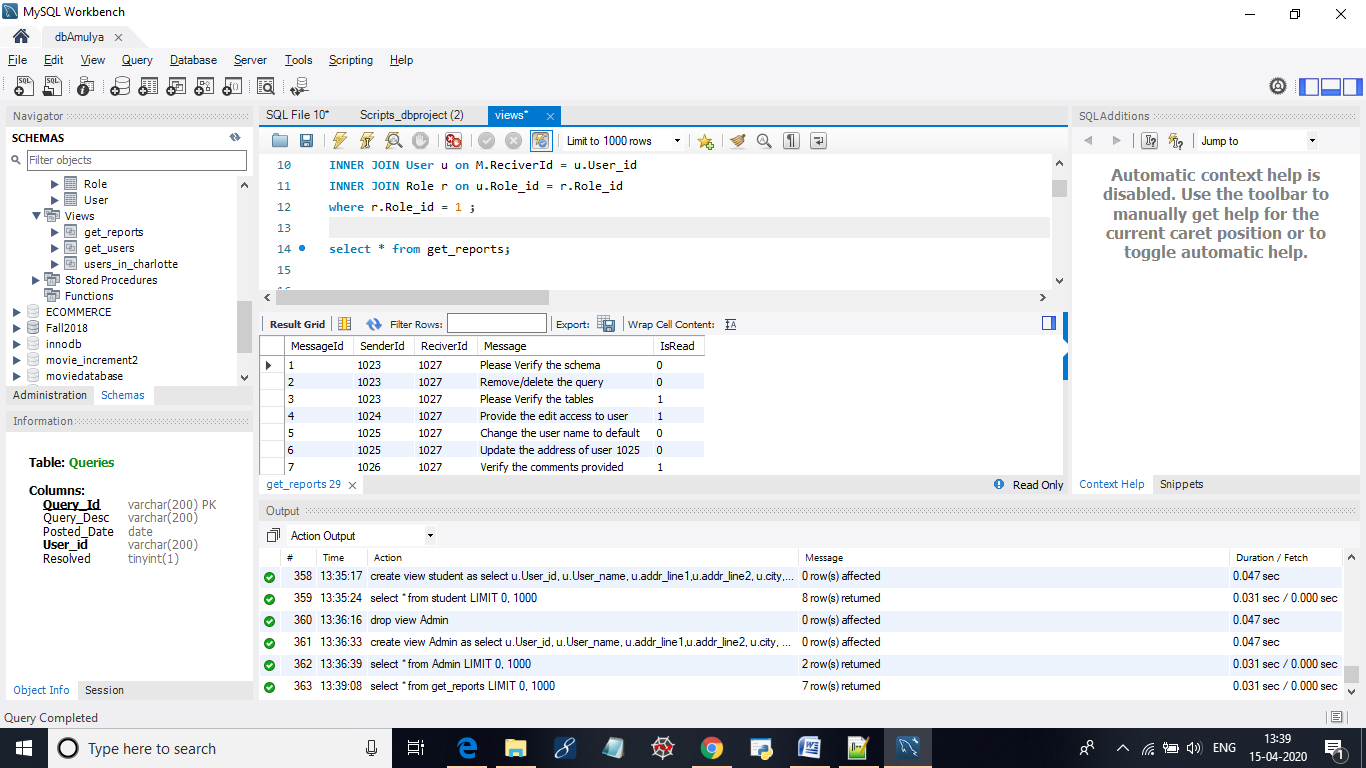
create view get\_reports as

SELECT M.\* FROM Message M

INNER JOIN User u on M.ReciverId = u.User\_id

INNER JOIN Role r on u.Role\_id = r.Role\_id

where r.Role\_id = 1 ;



**View:** get\_details

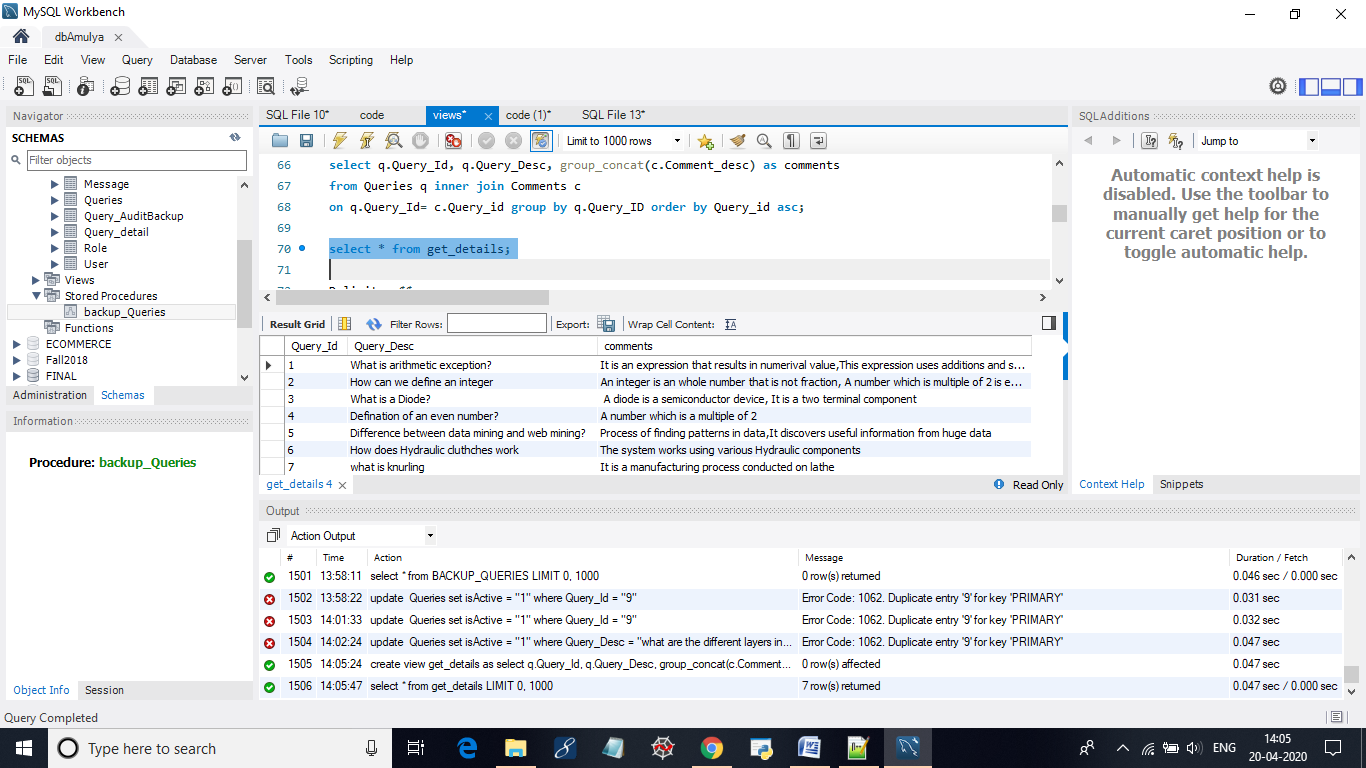
Goal: This view is created to display all the queries with their corresponding comments

create view get\_details as

select q.Query\_Id, q.Query\_Desc, group\_concat(c.Comment\_desc) as comments

from Queries q inner join Comments c

on q.Query\_Id= c.Query\_id group by q.Query\_ID order by Query\_id asc;



**View:** users\_in \_charlotte

This view is created to get all the users who belong to a particular location.

create view users\_in\_charlotte as

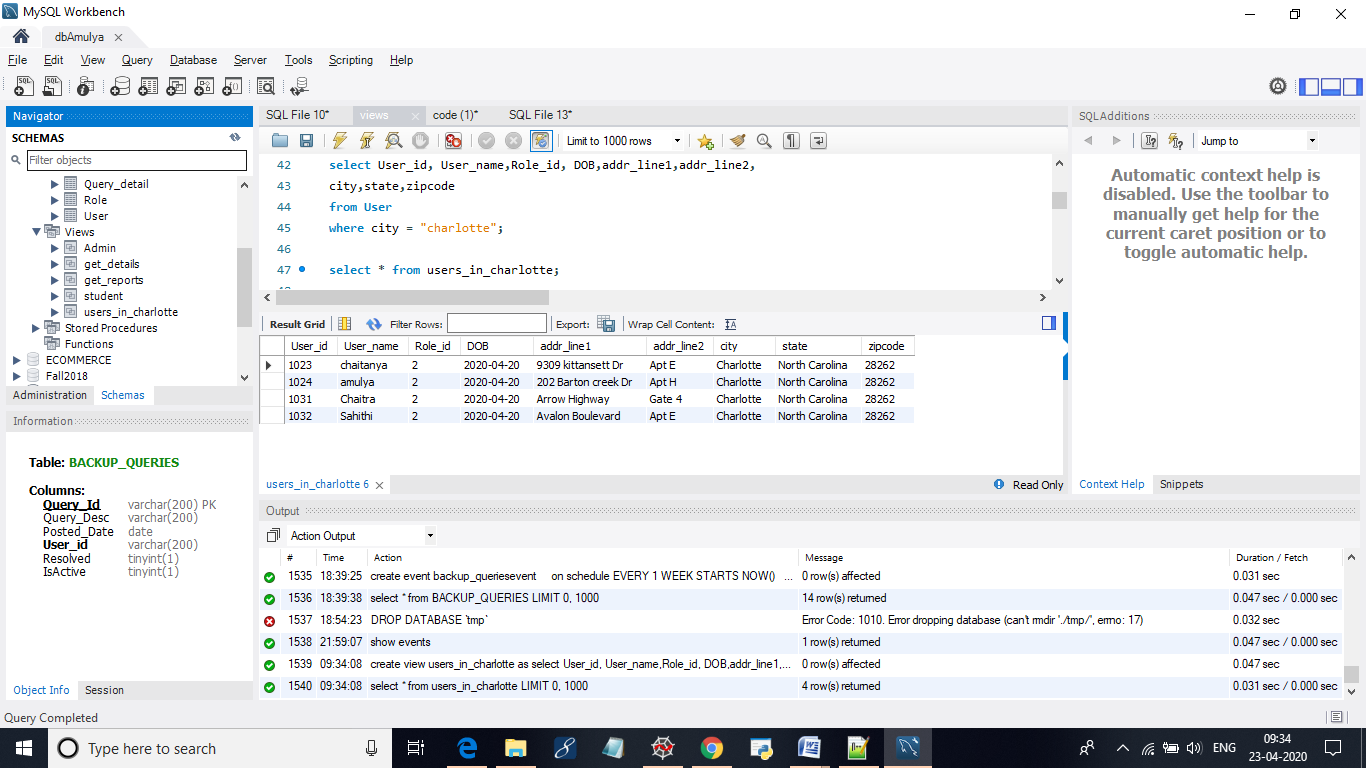
select u.User\_id, u.User\_name,u.Role\_id, u.DOB,u.addr\_line1,u.addr\_line2,

u.city,u.state,u.zipcode

from User u inner join Role r

on u.Role\_id = r.Role\_id

and u.city = "charlotte" and r.Role\_name = "student";



***PROCEDURES:***

**Procedure:** TopFiveActiveUsers()

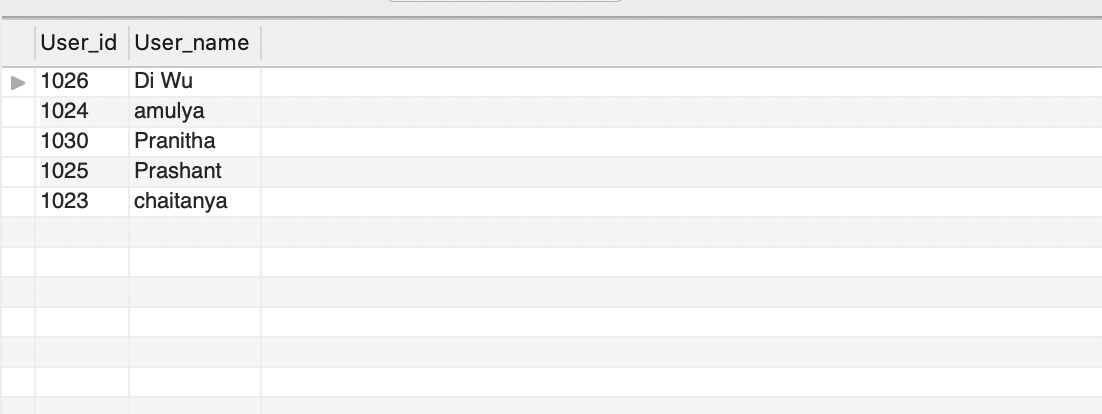
Goal: This procedure is created to retrieve the top five active users from the system.

CREATE DEFINER=`pminkuri`@`%` PROCEDURE `TopFiveActiveUsers`()

BEGIN

select User\_id, User\_name from Comments inner join User using (User\_id) group by User\_id order by count(User\_id) desc limit 5;

END



**Procedure :** UserActivity()

Goal: This procedure is created to search for the recent activity done by a particular user.

CREATE DEFINER=`pminkuri`@`%` PROCEDURE `UserActivity`(in userid varchar(20))

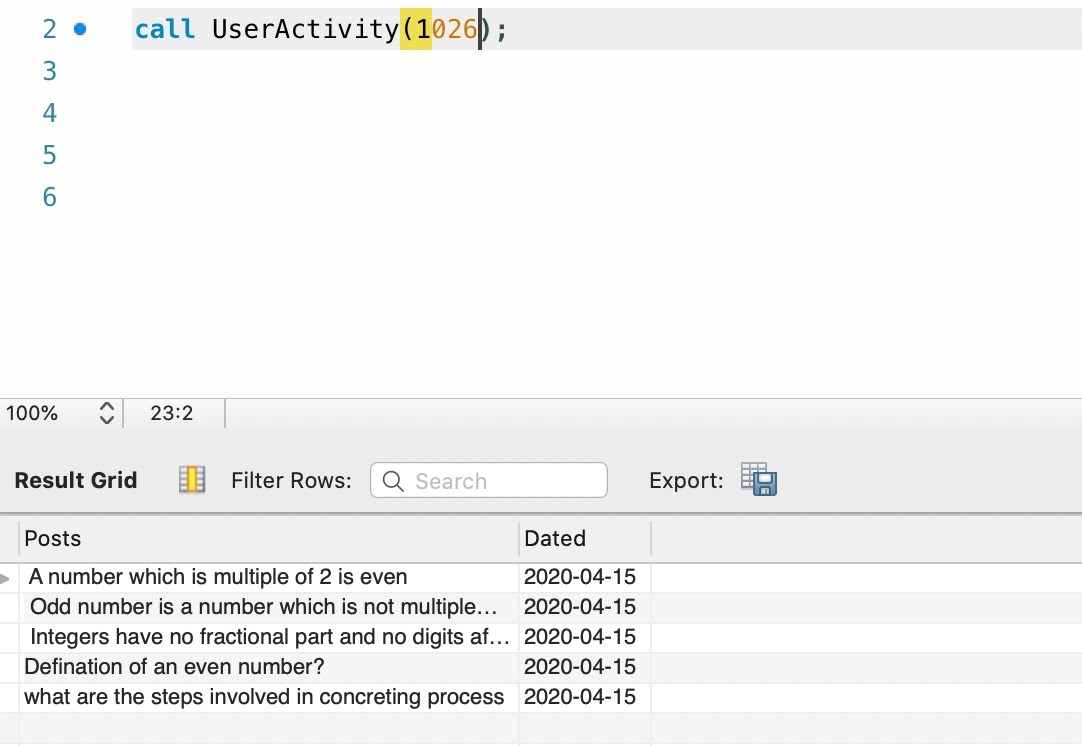
BEGIN

select Comment\_desc Posts ,Comment\_Date Dated from Comments where User\_id=userid

union

select Query\_Desc,Posted\_Date from Queries where User\_id=userid;

END



**Procedure:** ResolvedQueries()

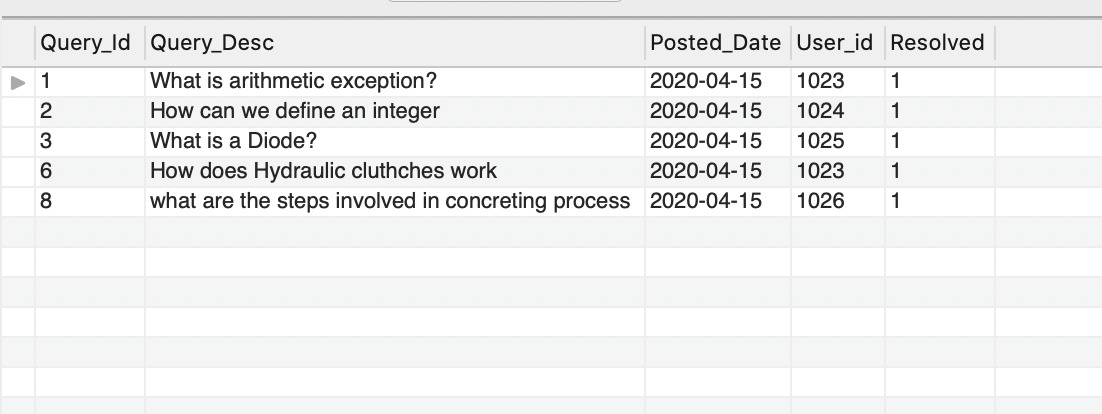
Goal: This procedure is created to display all the queries which are resolved by user

CREATE DEFINER=`pminkuri`@`%` PROCEDURE `ResolvedQueries`()

BEGIN

select \* from Queries where Resolved=true;

END



## ***TRIGGERS AND EVENTS:***

**1. Trigger to update points in the leaderboard when a user posts a new comment:**

DELIMITER $$

CREATE TRIGGER updateLeaderboard\_trig

AFTER INSERT ON `Comments` FOR EACH ROW

begin

UPDATE leaderboard

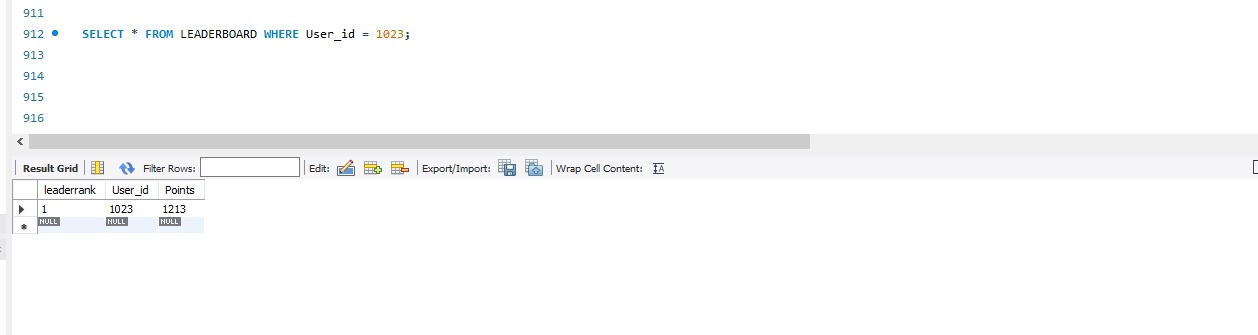
SET Points = Points + 1

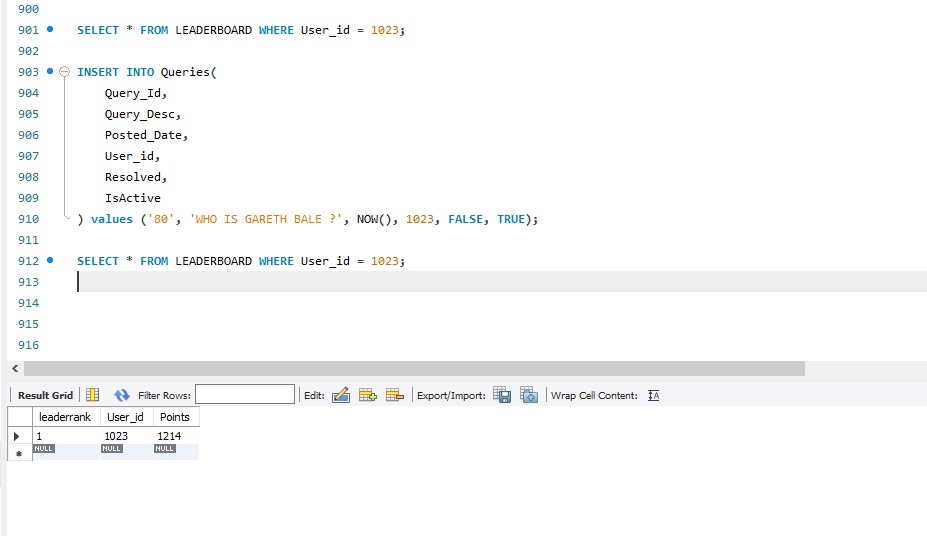
WHERE User\_id = NEW.User\_id;

END;

$$

DELIMITER ;





**2. Trigger to add new records in the audit table after deleting a query from Queries table:**

DELIMITER $$

CREATE TRIGGER addrecordsAuditQuery\_trig

AFTER DELETE ON `Queries` FOR EACH ROW

begin

INSERT INTO Query\_AuditBackup(

Query\_Id,

Query\_Desc,

Posted\_Date,

User\_id,

Resolved,

IsActive,

modified\_TIMESTAMP

) values (OLD.Query\_Id,

OLD.Query\_Desc,

OLD.Posted\_Date,

OLD.User\_id,

OLD.Resolved,

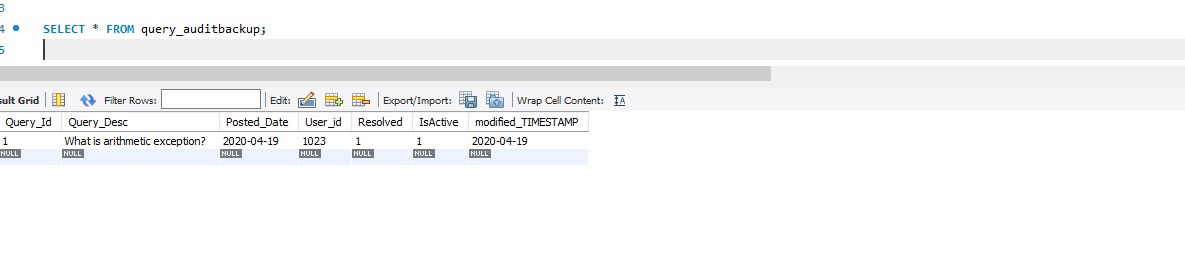
OLD.IsActive,

NOW());

END;

$$

DELIMITER ;



**3. Trigger to update records in the audit table:**

DELIMITER $$

CREATE TRIGGER updaterecordsAuditQuery\_trig

AFTER UPDATE ON `Queries` FOR EACH ROW

begin

INSERT INTO Query\_AuditBackup(

Query\_Id,

Query\_Desc,

Posted\_Date,

User\_id,

Resolved,

IsActive,

modified\_TIMESTAMP

) values (OLD.Query\_Id,

OLD.Query\_Desc,

OLD.Posted\_Date,

OLD.User\_id,

OLD.Resolved,

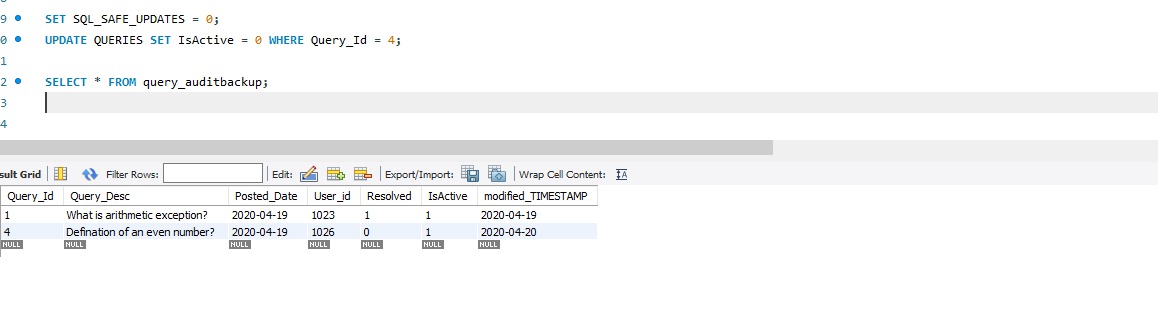
OLD.IsActive,

NOW());

END;

$$

DELIMITER ;



**/\*EVENTS\*/**

delimiter $$

create procedure backup\_Queries()

begin

insert into BACKUP\_QUERIES select \* from Queries;

end $$

delimiter ;

**-- Now, suposing that you want to execute this procedure every one week:**

delimiter $$

create event backup\_queriesevent

on schedule EVERY 1 WEEK STARTS NOW()

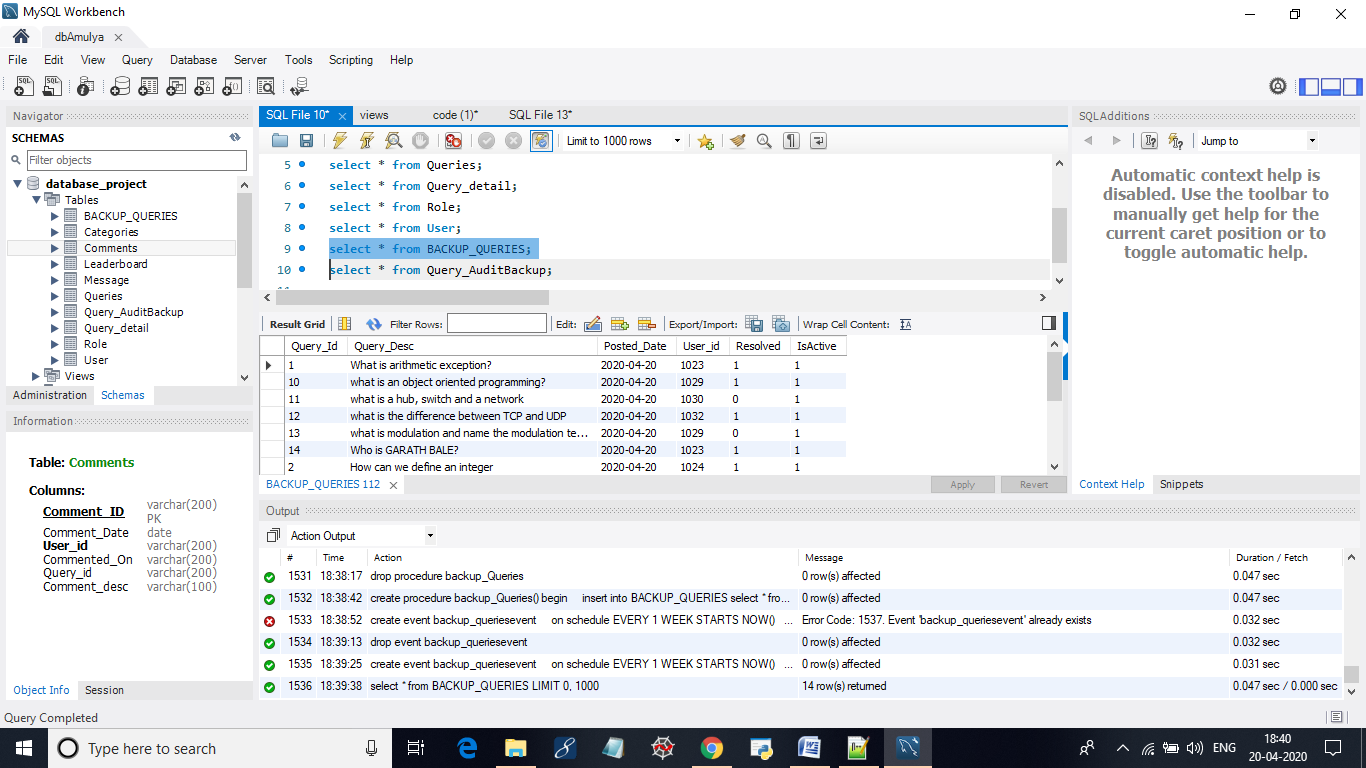
do

begin

call backup\_Queries();

end $$

delimiter ;

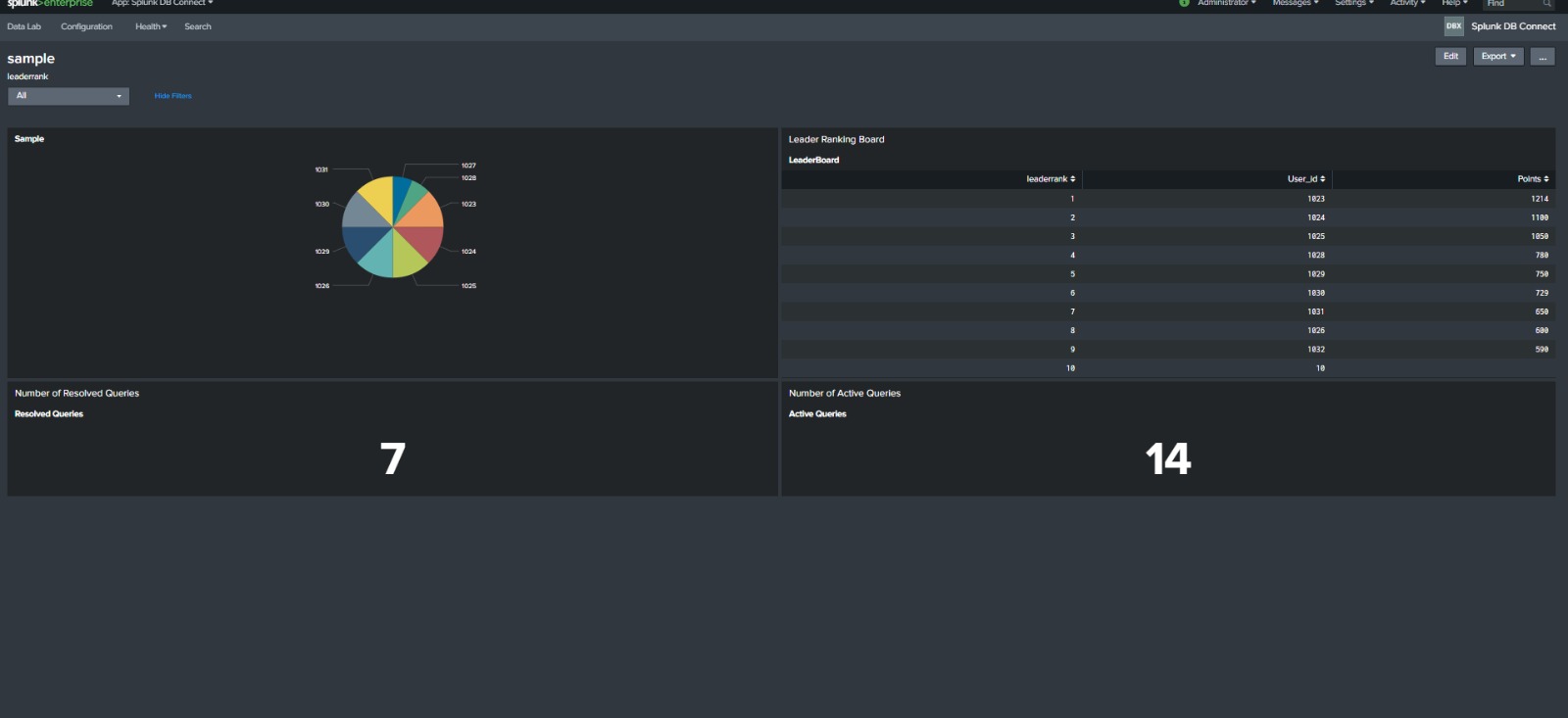


***SPLUNK:***

1. Created a dashboard which displays the points and rank for the corresponding user.

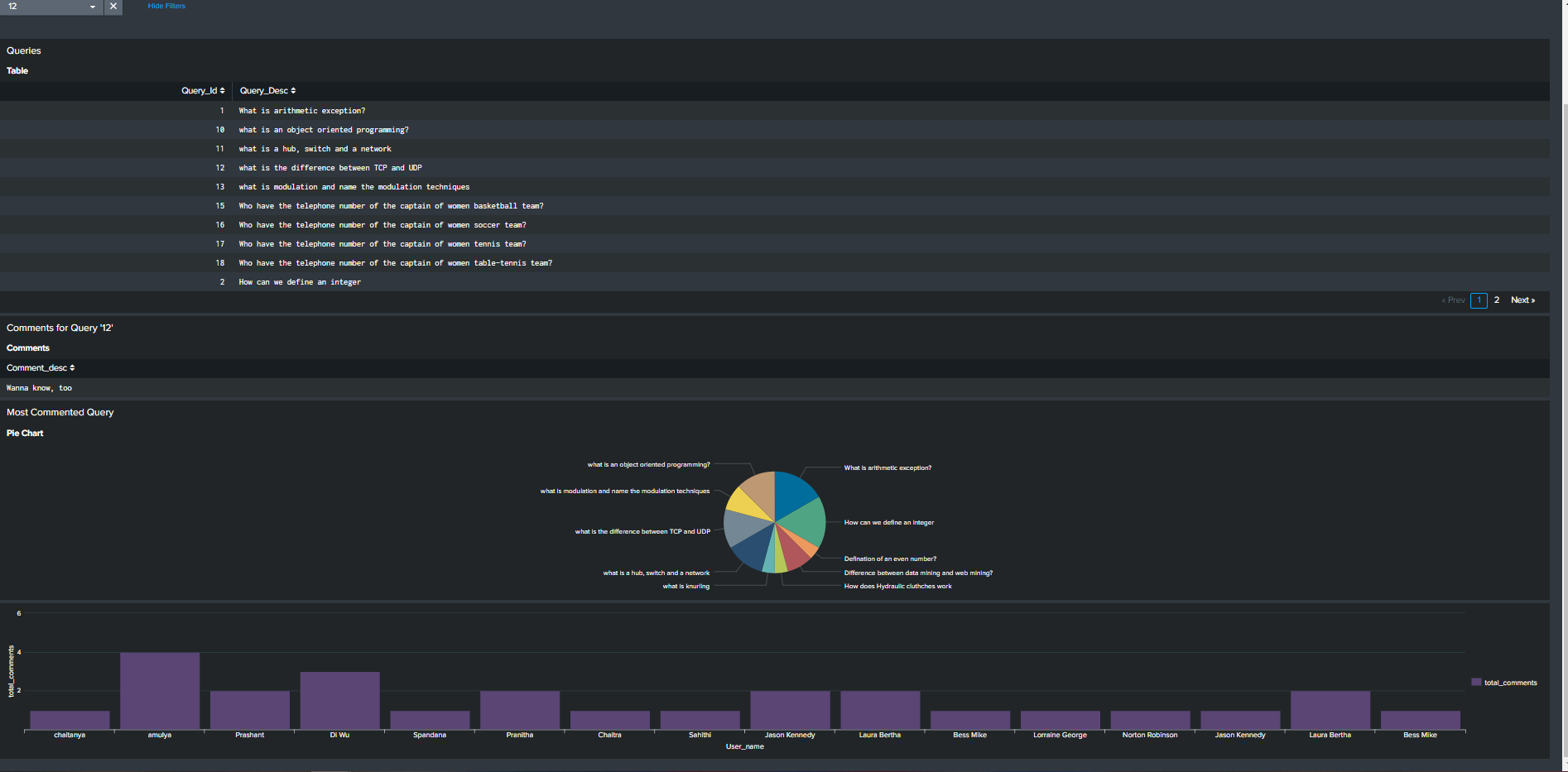
User can select the leader rank, and see the points assigned for the user.

The dashboard also has single valued panel to show the number of resolved queries and active .



2.Comments Dashboard

Comments Dashboard facilaites users to see all the queries for that id and retrieve the comments from query and pie char explaining the most commented query and user’s who contributed the most in the form of comments.



2. Category

Category dashboard displays all the queries for the selected category, Number of queries unresolved, resolved in that category, and a bar graph with user name with most query contribution for that category

