SMD: Application Proposal & Database Implementation
DATA 225 2022, GROUP 4

Razan Dababo, Priya Khandelwal, and Maria Poulose
San Jose State University, razan.dababo, priya.khandelwal, mariapoulose@sjsu.edu

***Abstract* - In this report, we propose and implement a movie database using a relational database management system (RDBMS) design that would form the core of our movie-streaming application, SMD. We provide the schema model of our database, describe user functionalities, provide user-specific queries, implement triggers, procedures, and access privileges, and evaluate the SQL performance of our database.**

Index Terms - AWS, MySQL, RDBMS

INTRODUCTION

Problem Statement: Our proposed application, San Jose State University Movie Database (SMD), is a subscription-based streaming service that allows our members to watch movies without commercials on an internet-connected device. Our application will provide users a hassle-free experience to select from a large number of films based on various search criteria, such as rating, actor name, or genre, and access their history of watched movies. Our application will also allow managers and system administrators to grant access privileges to various users of the application and track information such as subscription details and customer service issues.

SOLUTION REQUIREMENTS

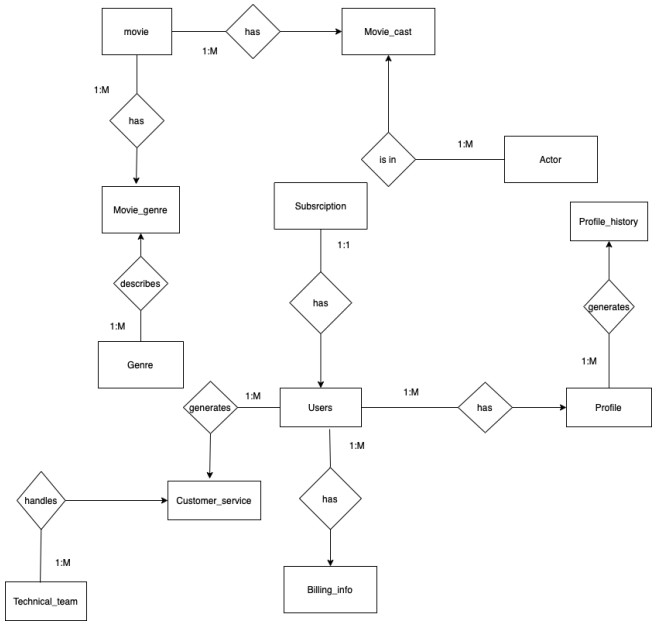
Given he large volume, structure, and interrelated nature of the data to be handled by our SMD application, the natural solution is to store and organize such data using a relational database management system (RDBMS).

An RDBMS stores data in tables that can be linked depending on their relationship and shared information, which allows users of the SMD application to easily retrieve information from one or more tables with a single query. Our SMD application would need to enable many users/administrators to simultaneously access/modify the database while maintaining data integrity, which RDBMS can accommodate with speedier response times. Our SMD application cannot afford to have any data inconsistencies; an RDBMS provides structured and consistent data, as it adheres to the ACID properties: Atomicity, Consistency, Isolation, and Durability). RDBMS design would also allow database administrators to enforce user authentication and access privileges and implement an efficient client/server protocol so clients on multiple remote hosts can access data concurrently. Overall, using an RDBMS would reduce our SMD application development time, as it would provide many of the solutions and features required by our application.

The scope of our application proposal has the following limitations:

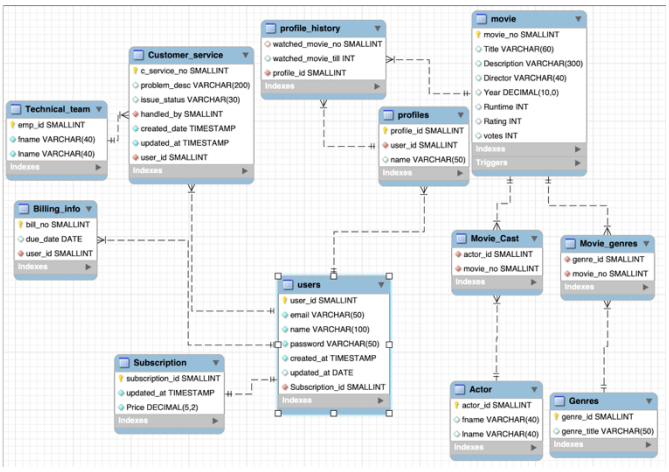
- The application will not have a movie-recommendation functionality.
- Users will not be able to fetch movies based on movie language.
- Users will not be able to fetch movies based on latest additions.
- Users cannot rate movies.

CONCEPTUAL DATABASE DESIGN



SMD database consist of 12 tables, as denoted in the figures below:

1. **The conceptual schema above details the interactions, relationships and cardinalities among the 12 tables of our SMD database.**



2. **The Entity Relationship (ER) diagram above details the attributes, participation, and cardinalities among the 12 tables that make up our SMD database.**

Functional Analysis

I. User Categories

Our application will cater to three types of users, as outlined below:

- Customers: users who will access the database to find information about movies. This group would search the database for common information about the movies, such as their actors or genres.
- Managers: users who will have unrestricted access to all tables and insert data into, update, or delete data from the database as needed.
- Administrators: users who will manage access privileges and upgrade/alter/program the database as needed.

II. Functional Components

Each customer account can have up to 5 profiles which have access to watch movies. Each account will automatically be billed a fixed-amount fee of $19.95 monthly. Users can report issues related to their account, and customer service issues are handled by the technical team, who have access to all customer information, including watch history, subscription and billing information, etc.

Table Structure of SMD Database

Below we include the SQL statements that we used to create the tables for our database. These statements include constraints and primary/foreign key information.

```
CREATE TABLE IF NOT EXISTS `SMD`.`Actor` (
  `actor_id` SMALLINT NOT NULL AUTO_INCREMENT,
  `fname` VARCHAR(40) NULL DEFAULT NULL,
  `lname` VARCHAR(40) NULL DEFAULT NULL,
  PRIMARY KEY (`actor_id`))
```

```
CREATE TABLE IF NOT EXISTS `SMD`.`Subscription` (
  `subscription_id` SMALLINT NOT NULL
AUTO_INCREMENT,
  `updated_at` TIMESTAMP NOT NULL DEFAULT
CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  `Price` DECIMAL(5,2) NOT NULL,
  PRIMARY KEY (`subscription_id`))
```

```
CREATE TABLE IF NOT EXISTS `SMD`.`users` (
  `user_id` SMALLINT NOT NULL AUTO_INCREMENT,
  `email` VARCHAR(50) NOT NULL,
  `name` VARCHAR(100) NOT NULL,
  `password` VARCHAR(50) NOT NULL,
  `created_at` TIMESTAMP NOT NULL,
  `updated_at` DATE NULL DEFAULT NULL,
  `Subscription_id` SMALLINT NOT NULL,
  PRIMARY KEY (`user_id`),
  INDEX `fk_users_Subscription1_idx` (`Subscription_id` ASC)
VISIBLE,
  CONSTRAINT `fk_users_Subscription1`
    FOREIGN KEY (`Subscription_id`)
    REFERENCES `SMD`.`Subscription` (`subscription_id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
```

```
CREATE TABLE `Billing_info` (
  `bill_no` smallint NOT NULL AUTO_INCREMENT,
  `due_date` timestamp NULL DEFAULT NULL,
  `user_id` smallint NOT NULL,
  PRIMARY KEY (`bill_no`),
  KEY `fk_Billing_info_user1_idx` (`user_id`),
  CONSTRAINT `fk_Billing_info_user1` FOREIGN KEY
(`user_id`) REFERENCES `users` (`user_id`))
```

```
CREATE TABLE IF NOT EXISTS `SMD`.`Technical_team` (
  `emp_id` SMALLINT NOT NULL,
  `fname` VARCHAR(40) NOT NULL,
  `lname` VARCHAR(40) NOT NULL,
  PRIMARY KEY (`emp_id`))
```

```
CREATE TABLE IF NOT EXISTS `SMD`.`Customer_service` (
  `c_service_no` SMALLINT NOT NULL AUTO_INCREMENT,
  `problem_desc` VARCHAR(200) NULL DEFAULT NULL,
  `issue_status` VARCHAR(30) NULL DEFAULT NULL,
  `handled_by` SMALLINT NOT NULL,
```

```
  `created_date` TIMESTAMP NOT NULL,
  `updated_at` TIMESTAMP NOT NULL DEFAULT
CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  `user_id` SMALLINT NOT NULL,
  PRIMARY KEY (`c_service_no`),
  INDEX `handled_by` (`handled_by` ASC) VISIBLE,
  INDEX `fk_Customer_service_user1_idx` (`user_id` ASC)
VISIBLE,
  CONSTRAINT `customer_service_ibfk_3`
    FOREIGN KEY (`handled_by`)
    REFERENCES `SMD`.`Technical_team` (`emp_id`),
  CONSTRAINT `fk_Customer_service_user1`
    FOREIGN KEY (`user_id`)
    REFERENCES `SMD`.`users` (`user_id`))


CREATE TABLE IF NOT EXISTS `SMD`.`movie` (
  `movie_no` SMALLINT NOT NULL AUTO_INCREMENT,
  `Title` VARCHAR(60) NULL DEFAULT NULL,
  `Description` VARCHAR(300) NULL DEFAULT NULL,
  `Director` VARCHAR(40) NULL DEFAULT NULL,
  `Year` DECIMAL(10,0) NULL DEFAULT NULL,
  `Runtime` INT NULL DEFAULT NULL,
  `Rating` INT NULL DEFAULT NULL,
  `votes` INT NULL DEFAULT NULL,
  PRIMARY KEY (`movie_no`))


CREATE TABLE IF NOT EXISTS `SMD`.`Genres` (
  `genre_id` SMALLINT NOT NULL AUTO_INCREMENT,
  `genre_title` VARCHAR(50) NULL DEFAULT NULL,
  PRIMARY KEY (`genre_id`))


CREATE TABLE IF NOT EXISTS `SMD`.`Movie_Cast` (
  `actor_id` SMALLINT NOT NULL,
  `movie_no` SMALLINT NOT NULL,
  INDEX `movie_no` (`movie_no` ASC) VISIBLE,
  INDEX `actor_id` (`actor_id` ASC) VISIBLE,
  CONSTRAINT `movie_cast_ibfk_1`
    FOREIGN KEY (`movie_no`)
    REFERENCES `SMD`.`movie` (`movie_no`),
  CONSTRAINT `movie_cast_ibfk_2`
    FOREIGN KEY (`actor_id`)
    REFERENCES `SMD`.`Actor` (`actor_id`))


CREATE TABLE IF NOT EXISTS `SMD`.`Movie_genres` (
  `genre_id` SMALLINT NOT NULL,
  `movie_no` SMALLINT NOT NULL,
  INDEX `movie_no` (`movie_no` ASC) VISIBLE,
  INDEX `genre_id` (`genre_id` ASC) VISIBLE,
  CONSTRAINT `movie_genres_ibfk_1`
    FOREIGN KEY (`movie_no`)
    REFERENCES `SMD`.`movie` (`movie_no`),
  CONSTRAINT `movie_genres_ibfk_2`
    FOREIGN KEY (`genre_id`)
    REFERENCES `SMD`.`Genres` (`genre_id`))


CREATE TABLE IF NOT EXISTS `SMD`.`profiles` (
  `profile_id` SMALLINT NOT NULL AUTO_INCREMENT,
  `user_id` SMALLINT NOT NULL,
  `name` VARCHAR(50) NULL DEFAULT NULL,
  PRIMARY KEY (`profile_id`),
  INDEX `user_id` (`user_id` ASC) VISIBLE,
  CONSTRAINT `profile_ibfk_1`
    FOREIGN KEY (`user_id`)
    REFERENCES `SMD`.`users` (`user_id`))


CREATE TABLE IF NOT EXISTS `SMD`.`profile_history` (
  `watched_movie_no` SMALLINT NULL DEFAULT NULL,
  `watched_movie_till` INT NULL DEFAULT NULL,
  `profile_id` SMALLINT NOT NULL,
  INDEX `watched_movie_no` (`watched_movie_no` ASC)
VISIBLE,
  INDEX `fk_profile_history_profile1_idx` (`profile_id` ASC)
VISIBLE,
  CONSTRAINT `fk_profile_history_profile1`
    FOREIGN KEY (`profile_id`)
    REFERENCES `SMD`.`profiles` (`profile_id`),
  CONSTRAINT `profile_history_ibfk_1`
    FOREIGN KEY (`watched_movie_no`)
    REFERENCES `SMD`.`movie` (`movie_no`))
```

QUERIES

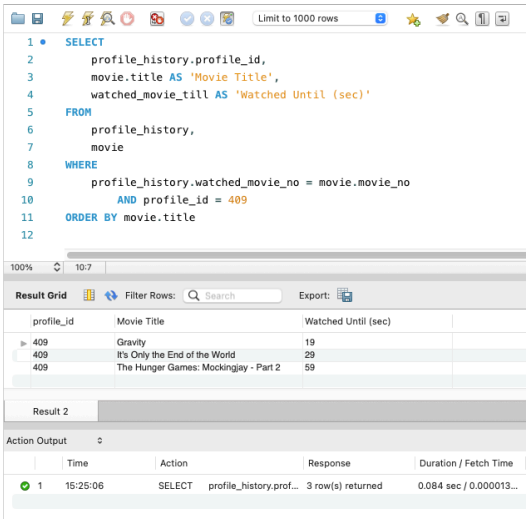
Below, we include examples of the queries that the users of the SMD application might find useful, along with query code, outputs, and performance (fetch time).

I. Customer Queries (CUS)

The following queries are specifically designed from a customer's perspective:

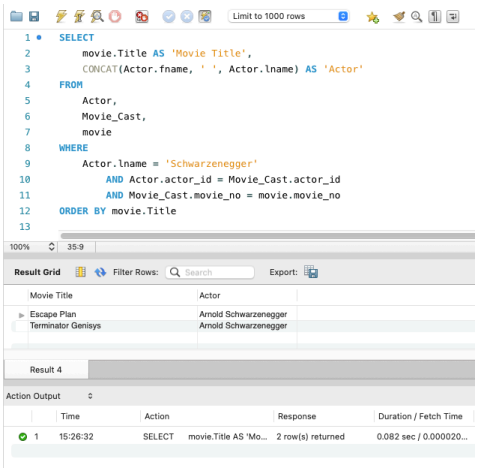
CUS1) This query helps customer to check how many movies were watched by a particular profile id (e.g. profile_id = 409) in that customer's account and how much of the movie they watched (watched until).

```
SELECT
   profile_history.profile_id,
   movie.title AS 'Movie Title',
   watched_movie_till AS 'Watched Until (sec)'
FROM
   profile_history,
   movie
WHERE
   profile_history.watched_movie_no = movie.movie_no
      AND profile_id = 409
ORDER BY movie.title
```



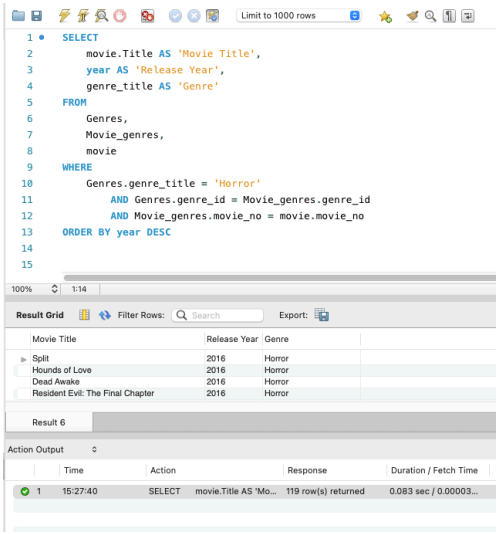
CUS2) Allow customer to fetch movie(s) based on a given actor (e.g. 'Schwarzenegger').

```
SELECT
   movie.Title AS 'Movie Title',
   CONCAT(Actor.fname, ' ', Actor.lname) AS 'Actor'
FROM
   Actor,
   Movie_Cast,
   movie
WHERE
   Actor.lname = 'Schwarzenegger'
      AND Actor.actor_id = Movie_Cast.actor_id
      AND Movie_Cast.movie_no = movie.movie_no
ORDER BY movie.Title
```



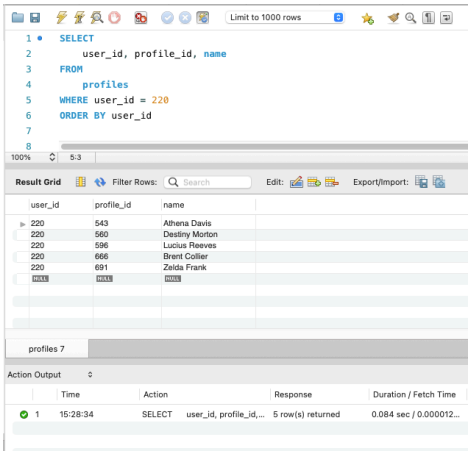
CUS3) Allow customer to fetch movie(s) based on a given genre (e.g. 'Horror').

```
SELECT
   movie.Title AS 'Movie Title',
   year as 'Release Year',
   genre_title as 'Genre'
FROM
   Genres,
   Movie_genres,
   movie
WHERE
   Genres.genre_title = 'Horror'
      AND Genres.genre_id = Movie_genres.genre_id
      AND Movie_genres.movie_no = movie.movie_no
ORDER BY year desc
```



CUS4) Allow customer to fetch details of profiles linked to their account.

```
SELECT
   user_id, profile_id, name
FROM
   profiles
WHERE user_id = 220
ORDER BY user_id
```



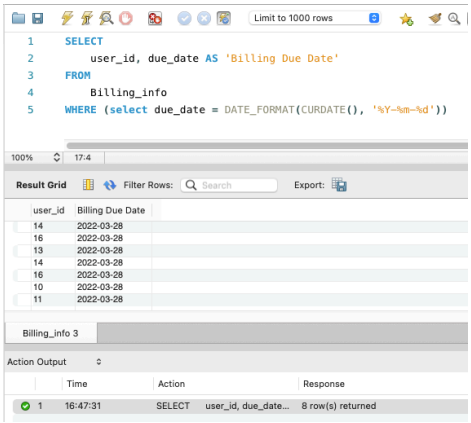
I. Manager Queries (MA)

These queries are specifically designed from the manager's perspective:

MA1) Allow manager to fetch customers who have billing due date at the current date. This helps

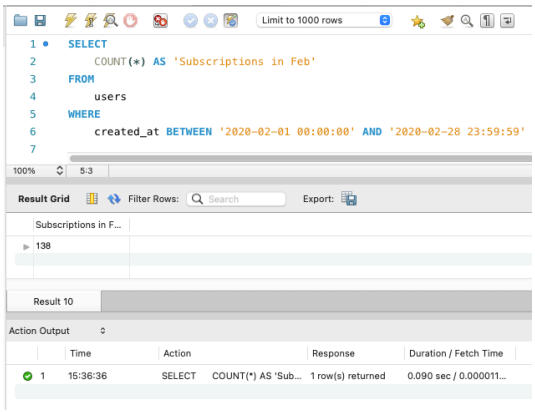
managers to know exactly how much subscription revenue they are going to receive on present day.

```
SELECT
   user_id, due_date AS 'Billing Due Date'
FROM
   Billing_info
WHERE (select due_date = DATE_FORMAT(CURDATE(), '%Y-%m-%d'))
```



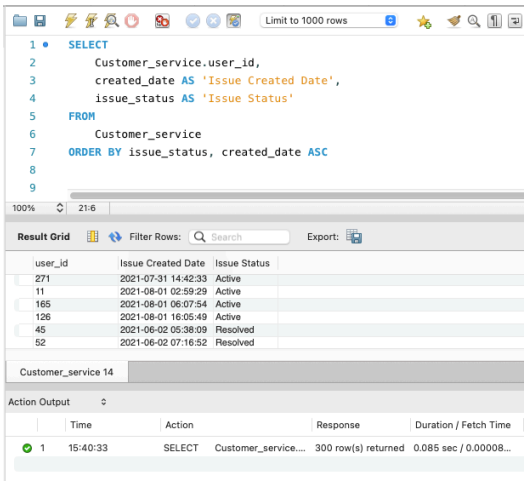
MA2) Allow manager to fetch details of monthly subscription numbers. This query helps managers to get the count of new customers they got for the SMD application in a given month (e.g. February, 2020).

```
SELECT
   COUNT(*) AS 'Subscriptions in Feb'
FROM
   users
WHERE
   created_at BETWEEN '2020-02-01 00:00:00' AND '2020-02-28 23:59:59'
```



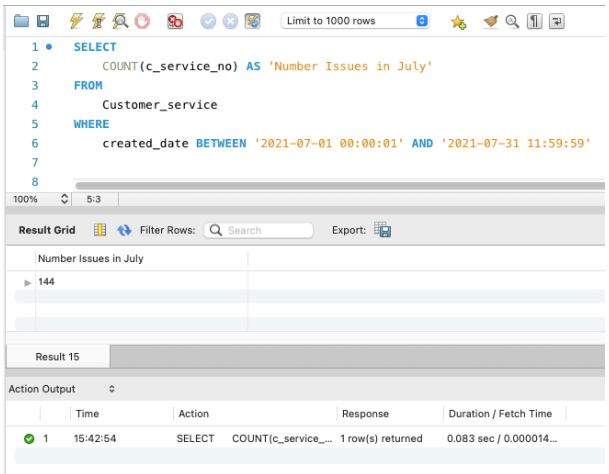
MA3) Allow manager to fetch details of customer service issue status. This query helps managers to track how many customer service problems are in active or resolved status.

```
SELECT
   Customer_service.user_id,
   created_date AS 'Issue Created Date',
   issue_status AS 'Issue Status'
FROM
   Customer_service
ORDER BY issue_status, created_date ASC
```



MA4) Allow manager to fetch total number of problems faced by users in a month (e.g. July, 2021).

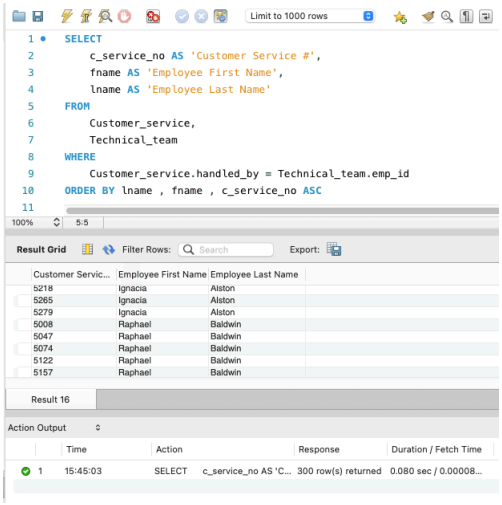
```
SELECT
   COUNT(c_service_no) AS 'Number Issues in July'
FROM
   Customer_service
WHERE
   created_date BETWEEN '2021-07-01 00:00:01' AND '2021-07-31 11:59:59'
```



MA5) Allow manager to fetch list of which employees handled which customer service issue.

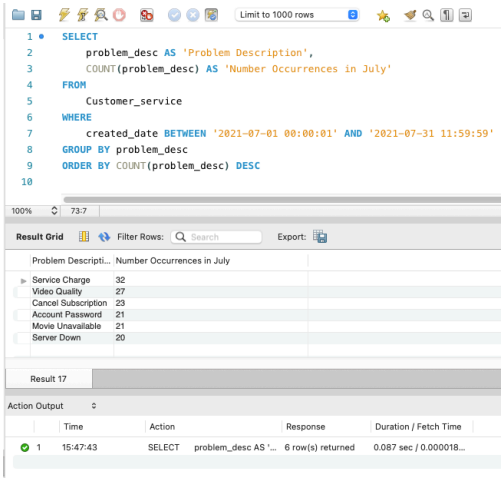
```
SELECT
   c_service_no AS 'Customer Service #',
   fname AS 'Employee First Name',
   lname AS 'Employee Last Name'
```

```
FROM
   Customer_service,
   Technical_team
WHERE
   Customer_service.handled_by = Technical_team.emp_id
ORDER BY lname , fname , c_service_no ASC
```



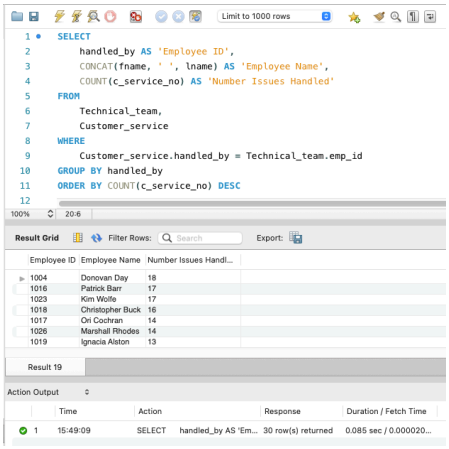
MA6) Allow manager to fetch customer service issue counts by category for a given month (e.g. July, 2021).

```
SELECT
   problem_desc AS 'Problem Description',
   COUNT(problem_desc) AS 'Number Occurrences in July'
FROM
   Customer_service
WHERE
      created_date BETWEEN '2021-07-01 00:00:01' AND '2021-07-31 11:59:59'
GROUP BY problem_desc
ORDER BY COUNT(problem_desc) DESC
```



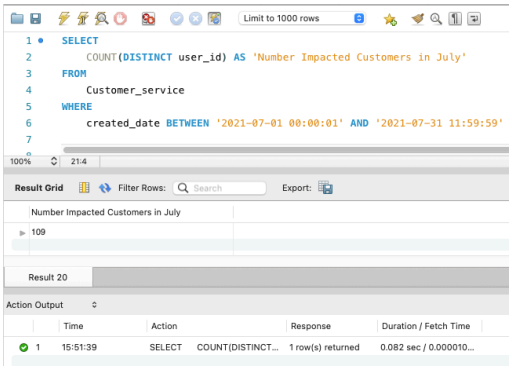
MA7) Allow manager to fetch which employee handled the most customer service issues overall.

```
SELECT
   handled_by AS 'Employee ID',
   CONCAT(fname, ' ', lname) AS 'Employee Name',
   COUNT(c_service_no) AS 'Number Issues Handled'
FROM
   Technical_team,
   Customer_service
WHERE
   Customer_service.handled_by = Technical_team.emp_id
GROUP BY handled_by
ORDER BY COUNT(c_service_no) DESC
```



MA8) Allow manager to fetch number of impacted customers in a given month (e.g. July, 2021).

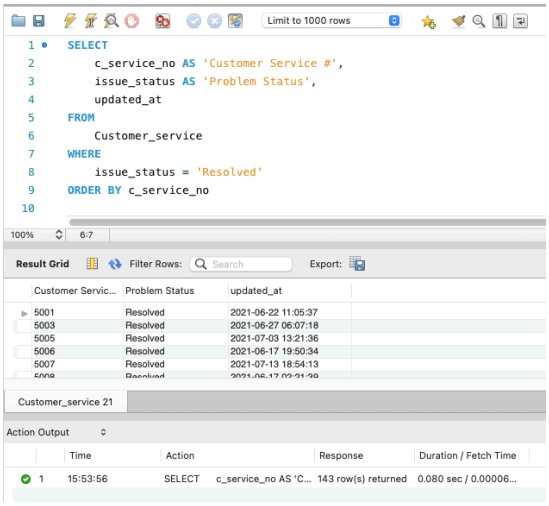
```
SELECT
   COUNT(DISTINCT user_id) AS 'Number Impacted Customers in July'
FROM
   Customer_service
WHERE
   created_date BETWEEN '2021-07-01 00:00:01' AND '2021-07-31 11:59:59'
```



MA9) Allow manager to fetch when a particular customer service issue was resolved

```
SELECT
   c_service_no AS 'Customer Service #',
   issue_status AS 'Problem Status',
   updated_at
FROM
   Customer_service
WHERE
   issue_status = 'Resolved'
```

ORDER BY c_service_no



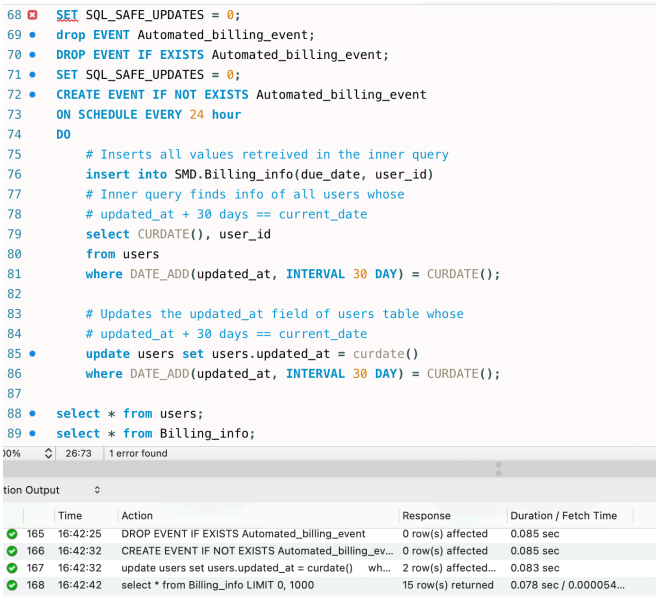
EVENT SCHEDULER FOR AUTOMATED BILLING

The billing system is automated so that monthly statements can be generated for all valid customers. In the SMD billing system, every 24 hours the billing statement is generated for the bills that are due that day and the due date is inserted in the Billing_info table.

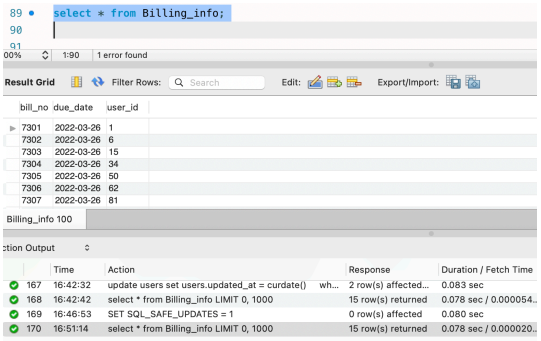
```
CREATE EVENT IF NOT EXISTS Automated_billing_event
ON SCHEDULE EVERY 24 hour
DO
        # Inserts all values retreived in the inner query
   insert into SMD.Billing_info(due_date, user_id)
   # Inner query finds info of all users whose
   # updated_at + 30 days == current_date
        select CURDATE(), user_id
   from users
       where DATE_ADD(updated_at, INTERVAL 30 DAY) =
CURDATE();

   # Updates the updated_at field of users table whose
   # updated_at + 30 days == current_date
        update users set users.updated_at = curdate()
   where DATE_ADD(updated_at, INTERVAL 30 DAY) =
CURDATE();
```

Output and Performance (Fetch Time):



After running the above event, the Billing_info table would be updated as follows:



TRIGGERS

We have implemented the triggers below, which fire when an event occurs in a database.

Trig1) Trigger before inserting row in movie table; if we try to insert runtime below 25, then this helps to give a warning, 'Check constraint on Runtime in table movie failed. Runtime too short.' Trigger code:

```
DELIMITER $$
CREATE trigger movie_runtime before insert on movie
for each row
BEGIN
  if new.Runtime <= 25 then
  signal sqlstate '42000'
  set message_text = 'Check constraint on Runtime in table movie
failed. Runtime too short';
  end if;
END $$
DELIMITER ;
```

Trigger output before insert and performance (fetch time):

Trig2) Trigger before inserting row in movie table; if we try to insert rating below 2, then this helps to give a warning, 'Check constraint on Rating in table movie failed. Rating too low.' Trigger code:

```
DELIMITER $$
CREATE trigger new_movie_rating before insert ON movie
for each row
BEGIN
  if (new.Rating < 3)  then
            signal sqlstate '42000'
            set message_text = 'Check constraint on Rating in table
movie failed. Rating too low';
            end if;
END $$
DELIMITER ;
```

Trigger firing and performance (fetch time):

STORED PROCEDURES

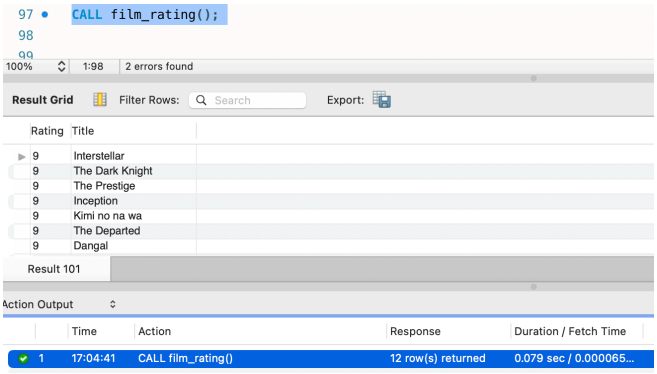
We created the below stored procedures, which are useful, prepared SQL queries that can be saved so that the query can be reused over and over again.

1. Display titles of movies having rating greater than 8. Procedure code:

```
DELIMITER $$
CREATE PROCEDURE film_rating()
BEGIN
  SELECT Rating, Title
  FROM movie
  WHERE Rating > 8;
END $$
DELIMITER ;

CALL film_rating();
```

Output and Performance (Fetch Time):

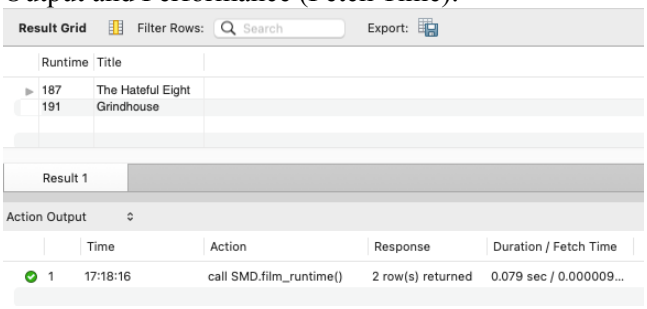


2. Display titles of movies having runtime greater than 180 minutes. Procedure code:

```
DELIMITER $$
CREATE PROCEDURE film_runtime()
BEGIN
  SELECT Runtime, Title
  FROM movie
  WHERE Runtime > 180;
END $$
DELIMITER ;

CALL film_runtime();
```

Output and Performance (Fetch Time):

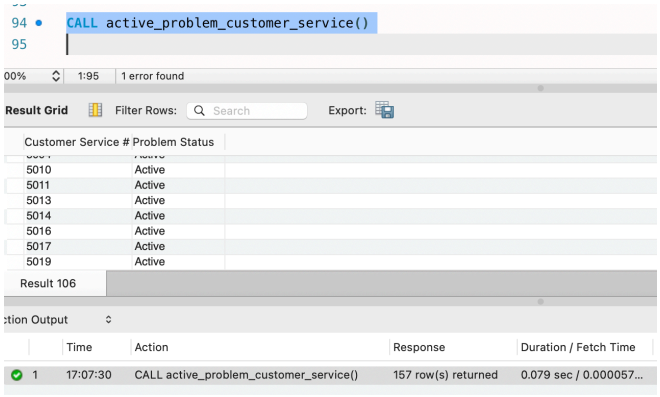


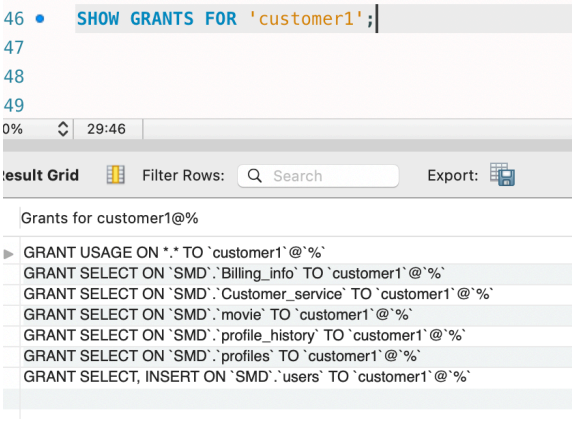
3. To display the customers who have active problem status. Procedure code:

```
delimiter //
create procedure active_problem_customer_service()
begin
SELECT
   c_service_no AS 'Customer Service #',
   issue_status AS 'Problem Status'
FROM
   Customer_service
WHERE
   issue_status = 'Active';
end//
delimiter //

CALL active_problem_customer_service()
```

Output and Performance (Fetch Time):





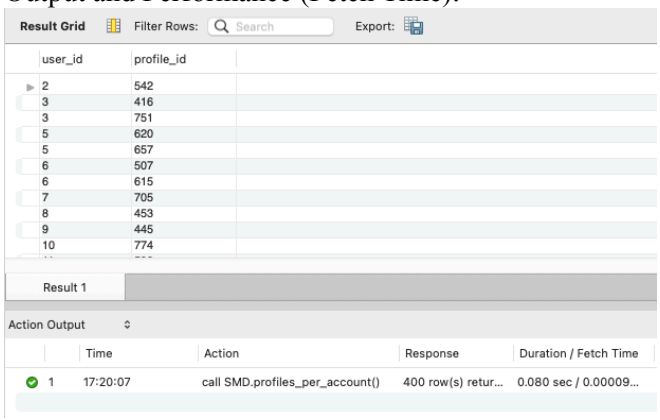
4. To display the profiles linked to specific customer account.

Procedure code:

```
DELIMITER $$
CREATE PROCEDURE profiles_per_account()
BEGIN
        SELECT
                profiles.user_id, profile_id
        FROM
                profiles,
                users
        WHERE
                users.user_id = profiles.user_id
        ORDER BY profiles.user_id;
END $$
DELIMITER $$

CALL profiles_per_account();
```

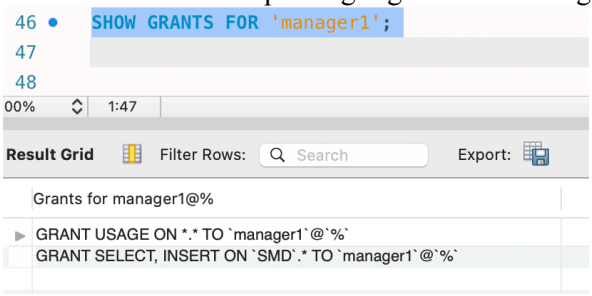
Output and Performance (Fetch Time):



Access Privileges

In SMD, customers will have limited table access. For example, they can sign up for an account in the system and watch movies. They would be able to check their billing information as well. In the image below, we summarize the access privileges granted to customers in our database.

In SMD, managers have access to all tables available in the database. In the image below, we summarize the access privileges granted to managers.



Logging

We have implemented logging to enable the MySQL function that logs each SQL query statement received from customers and managers, in order to get the time that query statement has submitted and runtime execution of the server, not to interfere with its runtime execution.

```
SET global general_log = 1;
SET global log_output = 'table';

select * from mysql.general_log order by event_time desc;
```

event_time	user_host	thread_id	server_id	command_type	argument
2022-03-27 21:23:43.718935	admin[admin] @ c-71-202-41-231.hsd1.ca.com...	12	2091084759	Query	BLOB
2022-03-27 21:23:42.763880	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BLOB
2022-03-27 21:23:41.763822	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BLOB
2022-03-27 21:23:40.763766	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BLOB
2022-03-27 21:23:39.763701	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BLOB
2022-03-27 21:23:38.763653	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BLOB
2022-03-27 21:23:37.763593	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BLOB
2022-03-27 21:23:37.715551	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BLOB
2022-03-27 21:23:37.715163	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BLOB
2022-03-27 21:23:37.696447	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BLOB
2022-03-27 21:23:37.696123	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BLOB

AWS CONNECTIVITY

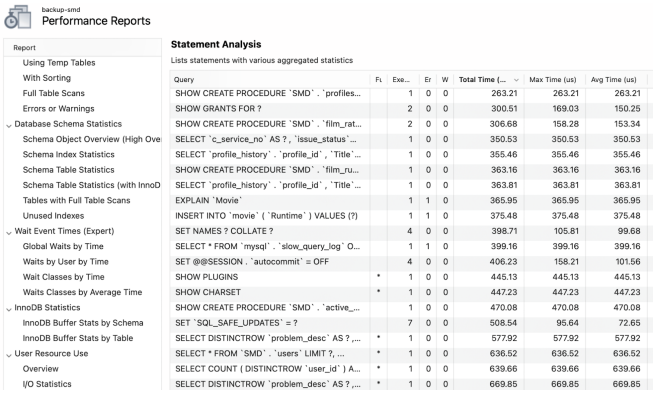
We used Amazon Web Services (AWS) Relational Database Service (RDS) instance to store data gathered and modified by our application. We connected to our MySQL database created on AWS with a python program in order to store data in a MySQL server hosted on AWS RDS.

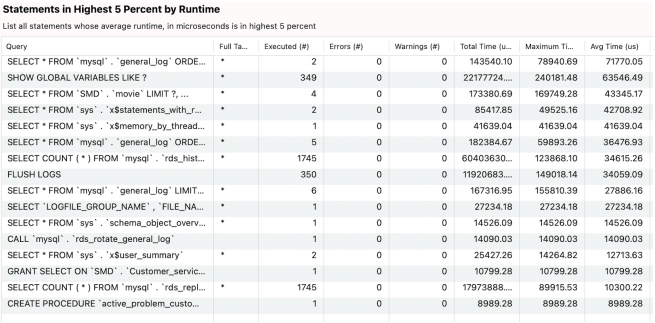
```
import mysql.connector
from getpass import getpass
from mysql.connector import connect, Error
try:
    with connect(host="smd-backup-25mar-2.cr5zcfvn0ztg.us-east-1.rds.amazonaws.com",
        user="admin",
        password="admin123",
    ) as connection:
        show_db_query = "SHOW TABLES"
        with connection.cursor() as cursor:
            cursor.execute('use SMD');
            cursor.execute(show_db_query)
            for db in cursor:
                print(db)
except Error as e:
    print(e)
```

```
('Actor',)
('Billing_info',)
('Customer_service',)
('Genres',)
('Movie_Cast',)
('Movie_genres',)
('Subscription',)
('Technical_team',)
('movie',)
('profile_history',)
('profiles',)
('users',)
```

SQL PERFORMANCE

Reviewing performance of the database is necessary to track statistics about server events and query execution. As seen in the fetch times for all the queries, triggers, and procedures above, and as seen in the performance report below, some queries have a longer fetch time than others and are more expensive than others.





REFERENCES

[1] Real Python, "Python and mysql database: A practical introduction," Real Python, 11-Dec-2020. [Online]. Available: https://realpython.com/python-mysql/. [Accessed: 19-Mar-2022].

[2] "Developer zone," MySQL. [Online]. Available: https://dev.mysql.com/. [Accessed: 16-Mar-2022].