

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 the 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).

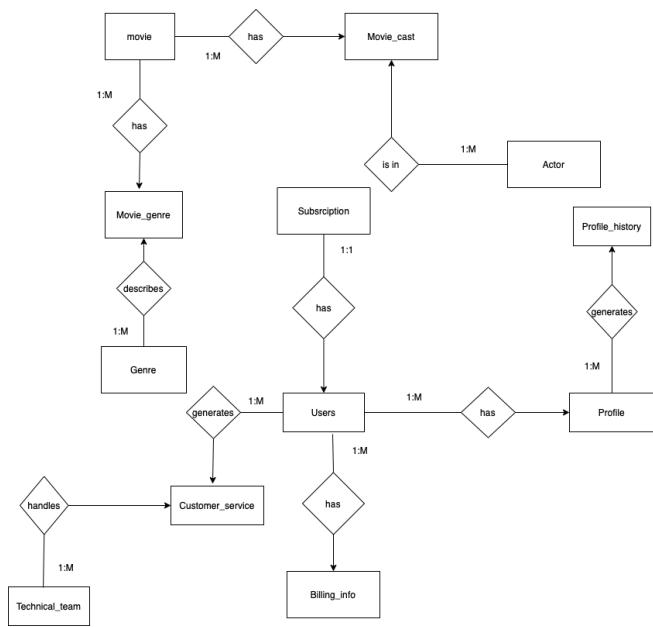
An RDBMS stores data in tables that can be linked depending on their relationship and shared information, which allows users of the SMD application to easily retrieve information from one or more tables with a single query. Our SMD application would need to enable many users/administrators to simultaneously 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:

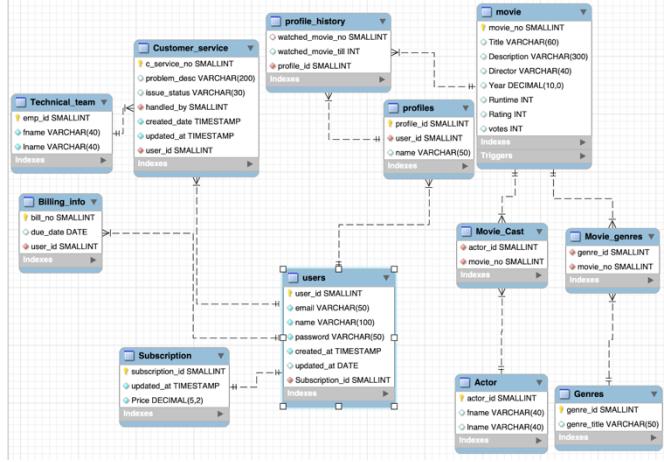
- The application will not have a movie-recommendation functionality.
- Users will not be able to fetch movies based on movie language.
- Users will not be able to fetch movies based on latest additions.
- Users cannot rate movies.

CONCEPTUAL DATABASE DESIGN



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

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



- 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

```

```

1 • SELECT
2     profile_history.profile_id,
3     movie.title AS 'Movie Title',
4     watched_movie_till AS 'Watched Until (sec)'
5   FROM
6     profile_history,
7     movie
8   WHERE
9     profile_history.watched_movie_no = movie.movie_no
10    AND profile_id = 409
11 ORDER BY movie.title
12

```

Result Grid

profile_id	Movie Title	Watched Until (sec)
409	Gravity	19
409	It's Only the End of the World	29
409	The Hunger Games: Mockingjay - Part 2	59

Action Output

Time	Action	Response	Duration / Fetch Time
15:25:06	SELECT profile_history.prof...	3 row(s) returned	0.084 sec / 0.000013...

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

```

```

1 • SELECT
2     movie.Title AS 'Movie Title',
3     CONCAT(Actor.fname, ' ', Actor.lname) AS 'Actor'
4   FROM
5     Actor,
6     Movie_Cast,
7     movie
8   WHERE
9     Actor.lname = 'Schwarzenegger'
10    AND Actor.actor_id = Movie_Cast.actor_id
11    AND Movie_Cast.movie_no = movie.movie_no
12 ORDER BY movie.Title
13

```

Result Grid

Movie Title	Actor
Escape Plan	Arnold Schwarzenegger
Terminator Genisys	Arnold Schwarzenegger

Action Output

Time	Action	Response	Duration / Fetch Time
15:26:32	SELECT movie.Title AS 'Mo... 2 row(s) returned		0.082 sec / 0.000020...

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

```

```

1 • SELECT
2     movie.Title AS 'Movie Title',
3     year AS 'Release Year',
4     genre_title AS 'Genre'
5   FROM
6     Genres,
7     Movie_genres,
8     movie
9   WHERE
10    Genres.genre_title = 'Horror'
11    AND Genres.genre_id = Movie_genres.genre_id
12    AND Movie_genres.movie_no = movie.movie_no
13 ORDER BY year DESC
14
15

```

Result Grid

Movie Title	Release Year	Genre
Split	2016	Horror
Hounds of Love	2016	Horror
Dead Awake	2016	Horror
Resident Evil: The Final Chapter	2016	Horror

Action Output

Time	Action	Response	Duration / Fetch Time
15:27:40	SELECT movie.Title AS 'Mo... 119 row(s) returned		0.083 sec / 0.00003...

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

```

```

1 • SELECT
2     user_id, profile_id, name
3   FROM
4     profiles
5   WHERE user_id = 220
6   ORDER BY user_id
7
8

```

Result Grid

user_id	profile_id	name
220	543	Athena Davis
220	560	Destiny Morton
220	596	Lucilia Reeves
220	666	Brenna Pollard
220	691	Zelida Frank

Action Output

Time	Action	Response	Duration / Fetch Time
15:28:34	SELECT user_id, profile_id,... 5 row(s) returned		0.084 sec / 0.000012...

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'))
```

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query is:

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

The result grid displays the following data:

user_id	Billing Due Date
14	2022-03-28
16	2022-03-28
15	2022-03-28
14	2022-03-28
16	2022-03-28
10	2022-03-28
11	2022-03-28

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'
```

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query is:

```
1 • SELECT
2     COUNT(*) AS 'Subscriptions in Feb'
3 FROM
4     users
5 WHERE
6     created_at BETWEEN '2020-02-01 00:00:00' AND '2020-02-28 23:59:59'
7
```

The result grid displays the following data:

Subscriptions in F...
138

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
```

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query is:

```
1 • SELECT
2     Customer_service.user_id,
3     created_date AS 'Issue Created Date',
4     issue_status AS 'Issue Status'
5 FROM
6     Customer_service
7 ORDER BY issue_status, created_date ASC
8
9
```

The result grid displays the following data:

user_id	Issue Created Date	Issue Status
271	2021-07-31 14:42:33	Active
11	2021-08-01 02:59:29	Active
165	2021-08-01 06:07:54	Active
126	2021-08-01 16:05:49	Active
45	2021-06-02 05:38:09	Resolved
52	2021-06-02 07:16:52	Resolved

Action Output:

Time	Action	Response	Duration / Fetch Time
16:47:31	SELECT	Customer_service...	300 row(s) returned 0.085 sec / 0.00008...

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'
```

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query is:

```
1 • SELECT
2     COUNT(c_service_no) AS 'Number Issues in July'
3 FROM
4     Customer_service
5 WHERE
6     created_date BETWEEN '2021-07-01 00:00:01' AND '2021-07-31 11:59:59'
7
8
```

The result grid displays the following data:

Number Issues in July
144

Action Output:

Time	Action	Response	Duration / Fetch Time
16:42:54	SELECT	COUNT(c_service_no) AS 'Number Issues in July'	1 row(s) returned 0.083 sec / 0.000014...

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

```

```

1 • SELECT
2   c_service_no AS 'Customer Service #',
3   fname AS 'Employee First Name',
4   lname AS 'Employee Last Name'
5   FROM
6     Customer_service,
7     Technical_team
8   WHERE
9     Customer_service.handled_by = Technical_team.emp_id
10    ORDER BY lname , fname , c_service_no ASC
11
100% 5-5 | Result Grid Filter Rows: Search Export:
Customer Service... Employee First Name Employee Last Name
5218 Ignacia Alston
5265 Ignacia Alston
5279 Ignacia Alston
5008 Raphael Baldwin
5047 Raphael Baldwin
5074 Raphael Baldwin
5122 Raphael Baldwin
5157 Raphael Baldwin
Result 16
Action Output
Time Action Response Duration / Fetch Time
1 15:45:03 SELECT c_service_no AS 'C... 300 row(s) returned 0.080 sec / 0.00008...

```

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

```

```

1 • SELECT
2   problem_desc AS 'Problem Description',
3   COUNT(problem_desc) AS 'Number Occurrences in July'
4   FROM
5     Customer_service
6   WHERE
7     created_date BETWEEN '2021-07-01 00:00:01' AND '2021-07-31 11:59:59'
8   GROUP BY problem_desc
9   ORDER BY COUNT(problem_desc) DESC
10
100% 73-7 | Result Grid Filter Rows: Search Export:
Problem Description... Number Occurrences in July
> Service Charge 35
Video Quality 27
Cancel Subscription 23
Account Password 21
Movie Unavailable 21
Server Down 20
Result 17
Action Output
Time Action Response Duration / Fetch Time
1 15:47:43 SELECT problem_desc AS '... 6 row(s) returned 0.087 sec / 0.000018...

```

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

```

```

1 • SELECT
2   handled_by AS 'Employee ID',
3   CONCAT(fname, ' ', lname) AS 'Employee Name',
4   COUNT(c_service_no) AS 'Number Issues Handled'
5   FROM
6     Technical_team,
7     Customer_service
8   WHERE
9     Customer_service.handled_by = Technical_team.emp_id
10    GROUP BY handled_by
11    ORDER BY COUNT(c_service_no) DESC
12
100% 20-6 | Result Grid Filter Rows: Search Export:
Employee ID Employee Name Number Issues Hand...
> 1004 Donovan Day 18
1016 Patrick Barr 17
1023 Michael Scott 17
1018 Christopher Buck 16
1017 Ori Cochran 14
1026 Marshall Rhodes 14
1019 Ignacia Alston 13
Result 19
Action Output
Time Action Response Duration / Fetch Time
1 15:49:09 SELECT handled_by AS 'Em... 30 row(s) returned 0.085 sec / 0.000020...

```

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'

```

```

1 • SELECT
2   COUNT(DISTINCT user_id) AS 'Number Impacted Customers in July'
3   FROM
4     Customer_service
5   WHERE
6     created_date BETWEEN '2021-07-01 00:00:01' AND '2021-07-31 11:59:59'
7
100% 21-4 | Result Grid Filter Rows: Search Export:
Number Impacted Customers in July
> 109
Result 20
Action Output
Time Action Response Duration / Fetch Time
1 15:51:39 SELECT COUNT(DISTINCT... 1 row(s) returned 0.082 sec / 0.000010...

```

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

```

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

```

1 • SELECT
2     c_service_no AS 'Customer Service #',
3     issue_status AS 'Problem Status',
4     updated_at
5   FROM
6     customer_service
7  WHERE
8    issue_status = 'Resolved'
9  ORDER BY c_service_no
10

```

Result Grid

Customer Service #	Problem Status	updated_at
5001	Resolved	2021-06-22 11:05:37
5003	Resolved	2021-06-27 06:07:18
5005	Resolved	2021-07-03 13:21:36
5006	Resolved	2021-06-17 19:50:34
5007	Resolved	2021-07-13 18:54:13
5008	Resolved	2021-06-17 09:51:50

Action Output

Time	Action	Response	Duration / Fetch Time
15:53:56	SELECT c_service_no AS 'C...' 143 row(s) returned	0.080 sec / 0.00006...	

```

68 • SET SQL_SAFE_UPDATES = 0;
69 • drop EVENT Automated_billing_event;
70 • DROP EVENT IF EXISTS Automated_billing_event;
71 • SET SQL_SAFE_UPDATES = 0;
72 • CREATE EVENT IF NOT EXISTS Automated_billing_event
73   ON SCHEDULE EVERY 24 hour
74   DO
75     # Inserts all values retrieved in the inner query
76     insert into SMD.Billing_info(due_date, user_id)
77     # Inner query finds info of all users whose
78     # updated_at + 30 days == current_date
79     select CURDATE(), user_id
80     from users
81     where DATE_ADD(updated_at, INTERVAL 30 DAY) = CURDATE();
82
83     # Updates the updated_at field of users table whose
84     # updated_at + 30 days == current_date
85 • update users set users.updated_at = curdate()
86   where DATE_ADD(updated_at, INTERVAL 30 DAY) = CURDATE();
87
88 • select * from users;
89 • select * from Billing_info;

```

Action Output

Time	Action	Response	Duration / Fetch Time
16:42:25	DROP EVENT IF EXISTS Automated_billing_event	0 row(s) affected	0.085 sec
16:42:32	CREATE EVENT IF NOT EXISTS Automated_billing_ev...	0 row(s) affected	0.085 sec
16:42:32	update users set users.updated_at = curdate() wh...	2 row(s) affected...	0.083 sec
16:42:42	select * from Billing_info LIMIT 0, 1000	15 row(s) returned	0.078 sec / 0.000054...

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 retrieved 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:

```

89 • select * from Billing_info;
90
91
00% 1:190 | 1 error found
Result Grid
Action Output


| bill_no | due_date   | user_id |
|---------|------------|---------|
| 7301    | 2022-03-26 | 1       |
| 7302    | 2022-03-26 | 6       |
| 7303    | 2022-03-26 | 15      |
| 7304    | 2022-03-26 | 34      |
| 7305    | 2022-03-26 | 50      |
| 7306    | 2022-03-26 | 62      |
| 7307    | 2022-03-26 | 81      |


Billing_info 100
Action Output


| Time     | Action                                              | Response             | Duration / Fetch Time   |
|----------|-----------------------------------------------------|----------------------|-------------------------|
| 16:42:32 | update users set users.updated_at = curdate() wh... | 2 row(s) affected... | 0.083 sec               |
| 16:42:42 | select * from Billing_info LIMIT 0, 1000            | 15 row(s) returned   | 0.078 sec / 0.000054... |
| 16:46:53 | SET SQL_SAFE_UPDATES = 1                            | 0 row(s) affected    | 0.080 sec               |
| 16:51:14 | select * from Billing_info LIMIT 0, 1000            | 15 row(s) returned   | 0.078 sec / 0.000020... |


```

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):

The screenshot shows the MySQL Workbench interface. At the top, there's a status bar with icons and the text "Limit to 1000 rows". Below it is a toolbar with various icons. The main area has a title bar "1 • insert into movie(Runtime) values[20]". Underneath is a "Result Grid" table with columns "Rating" and "Title". The data shows ratings from 9 to 9 for movies like Interstellar, The Dark Knight, etc. Below the grid is a "Result 101" section with a table for "Action Output" showing one row inserted with a duration of 0.081 sec. A message in the response column says: "Error Code: 1644. Check constraint on Runtime in table movie failed. Runtime too short.".

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):

The screenshot shows the MySQL Workbench interface. At the top, there's a status bar with icons and the text "Limit to 1000 rows". Below it is a toolbar with various icons. The main area has a title bar "1 • insert into movie(rating) values[2]". Underneath is a "Result Grid" table with columns "Rating" and "Title". The data shows two rows with ratings 187 and 191 respectively. Below the grid is a "Result 1" section with a table for "Action Output" showing one row inserted with a duration of 0.082 sec. A message in the response column says: "Error Code: 1644. Check constraint on Rating in table movie failed. Rating too low".

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):

The screenshot shows the MySQL Workbench interface. At the top, there's a status bar with icons and the text "100% 1:98 2 errors found". Below it is a toolbar with various icons. The main area has a title bar "97 • CALL film_rating();". Underneath is a "Result Grid" table with columns "Rating" and "Title". The data shows ratings from 9 to 9 for movies like Interstellar, The Dark Knight, etc. Below the grid is a "Result 101" section with a table for "Action Output" showing one row returned with a duration of 0.079 sec. A message in the response column says: "12 row(s) returned".

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):

The screenshot shows the MySQL Workbench interface. At the top, there's a status bar with icons and the text "100% 1:98 2 errors found". Below it is a toolbar with various icons. The main area has a title bar "97 • CALL film_runtime();". Underneath is a "Result Grid" table with columns "Runtime" and "Title". The data shows two rows with runtimes 187 and 191 respectively. Below the grid is a "Result 1" section with a table for "Action Output" showing one row returned with a duration of 0.079 sec. A message in the response column says: "2 row(s) returned".

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):

```
94 • CALL active_problem_customer_service()
95
00% 1:95 | 1 error found
Result Grid Filter Rows: Search Export:
Customer Service # Problem Status
5010 Active
5011 Active
5013 Active
5014 Active
5016 Active
5017 Active
5019 Active
Result 106
Action Output
Time Action Response Duration / Fetch Time
1 17:07:30 CALL active_problem_customer_service() 157 row(s) returned 0.079 sec / 0.000057...
```

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):

```
Result Grid Filter Rows: Search Export:
user_id profile_id
2 542
3 416
3 751
5 620
5 657
6 507
6 615
7 705
8 453
9 445
10 774
Result 1
Action Output
Time Action Response Duration / Fetch Time
1 17:20:07 call SMD.profiles_per_account() 400 row(s) returned 0.080 sec / 0.00009...
```

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.

```
46 • SHOW GRANTS FOR 'customer1';
47
48
49
0% 29:46
Result Grid Filter Rows: Search Export:
Grants for customer1@%
▶ GRANT USAGE ON *.* TO `customer1`@`%`  

GRANT SELECT ON `SMD`.`Billing_info` TO `customer1`@`%`  

GRANT SELECT ON `SMD`.`Customer_service` TO `customer1`@`%`  

GRANT SELECT ON `SMD`.`movie` TO `customer1`@`%`  

GRANT SELECT ON `SMD`.`profile_history` TO `customer1`@`%`  

GRANT SELECT ON `SMD`.`profiles` TO `customer1`@`%`  

GRANT SELECT, INSERT ON `SMD`.`users` TO `customer1`@`%`
```

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

```
46 • SHOW GRANTS FOR 'manager1';
47
48
0% 1:47
Result Grid Filter Rows: Search Export:
Grants for manager1@%
▶ GRANT USAGE ON *.* TO `manager1`@`%`  

GRANT SELECT, INSERT ON `SMD`.* TO `manager1`@`%`
```

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	BL0B
2022-03-27 21:23:42.763880	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BL0B
2022-03-27 21:23:41.763822	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BL0B
2022-03-27 21:23:40.763766	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BL0B
2022-03-27 21:23:39.763701	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BL0B
2022-03-27 21:23:38.763693	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BL0B
2022-03-27 21:23:37.763593	rdsadmin[rdsadmin] @ localhost []	9	2091084759	Execute	BL0B
2022-03-27 21:23:37.715551	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BL0B
2022-03-27 21:23:37.715163	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BL0B
2022-03-27 21:23:37.696447	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BL0B
2022-03-27 21:23:37.696123	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	10	2091084759	Query	BL0B

```

1 select * from mysql.general_log order by event_time desc
2 LIMIT 0, 1000

```

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.

Statement Analysis

Query	Fl	Exe...	Er	W	Total Time (us)	Max Time (us)	Avg Time (us)
SHOW CREATE PROCEDURE 'SMD' . 'profiles...	1	0	0	0	263.21	263.21	263.21
SHOW GRANTS FOR ?	2	0	0	0	300.51	169.03	150.25
SHOW CREATE PROCEDURE 'SMD' . 'film_rat...	2	0	0	0	306.68	158.28	153.34
SELECT `c_service_no` AS T , `issue_status` ...	1	0	0	0	350.53	350.53	350.53
SELECT `profile_history` . `profile_id` , `Title` ...	1	0	0	0	355.46	355.46	355.46
EXPLAIN `Movie`	1	0	0	0	363.16	363.16	363.16
SELECT `profile_history` . `profile_id` , `Title` ...	1	0	0	0	363.81	363.81	363.81
EXPLAIN `Movie`	1	1	0	0	365.95	365.95	365.95
INSERT INTO `movie` ('Runtime') VALUES (?)	1	1	0	0	375.48	375.48	375.48
SET NAMES ? COLLATE ?	4	0	0	0	398.71	105.81	99.68
SELECT * FROM `mysql` . `slow_query_log` O...	1	1	0	0	399.16	399.16	399.16
SET @@SESSION . `autocommit` = OFF	4	0	0	0	406.23	158.21	101.56
SHOW PLUGINS	*	1	0	0	445.13	445.13	445.13
SHOW CHARSET	*	1	0	0	447.23	447.23	447.23
SHOW CREATE PROCEDURE 'SMD' . 'active_...	1	0	0	0	470.08	470.08	470.08
SET `SQL_SAFE_UPDATES` = ?	7	0	0	0	508.54	95.64	72.65
SELECT DISTINCTROW `problem_desc` AS ? ...	*	1	0	0	577.92	577.92	577.92
SELECT * FROM 'SMD' . `users` LIMIT ?, ...	*	1	0	0	636.52	636.52	636.52
SELECT COUNT (DISTINCTROW `user_id`) A...	*	1	0	0	639.66	639.66	639.66
SELECT DISTINCTROW `problem_desc` AS ? ...	*	1	0	0	669.85	669.85	669.85

Statements in Highest 5 Percent by Runtime

Query	Full T...	Executed (#)	Errors (#)	Warnings (#)	Total Time (u...)	Maximum Ti...	Avg Time (u...)
SELECT * FROM `mysql` . `general_log` ORDER...	*	2	0	0	143540.10	78940.69	71770.05
SHOW GLOBAL VARIABLES LIKE ?	*	349	0	0	2217724...	24018.48	63546.49
SELECT * FROM 'SMD' . `movie` LIMIT ?, ...	*	4	0	0	173380.69	169749.28	43345.17
SELECT * FROM `sys` . `x\$statements_with_r...	*	2	0	0	85417.85	49525.16	42708.92
SELECT * FROM `sys` . `x\$memory_by_thread...	*	1	0	0	41639.04	41639.04	41639.04
SELECT * FROM `mysql` . `general_log` ORDER...	*	5	0	0	162384.67	59983.26	36476.93
SELECT COUNT (*) FROM `mysql` . `rds_hist...	*	1745	0	0	604053630...	123868.10	34615.26
FLUSH LOGS	*	350	0	0	11920683...	149018.14	34059.09
SELECT * FROM `mysql` . `general_log` LIMIT...	*	6	0	0	167316.95	155810.39	27886.16
SELECT `LOGFILE_GROUP_NAME` , `FILE_LNAM...	*	1	0	0	27234.18	27234.18	27234.18
SELECT * FROM `sys` . `schema_object_overv...	*	1	0	0	14526.09	14526.09	14526.09
CALL `mysql` . `rds_rotate_general_log`	*	1	0	0	14090.03	14090.03	14090.03
SELECT * FROM `sys` . `x\$user_summary`	*	2	0	0	25427.26	14264.82	12713.63
GRANT SELECT ON 'SMD' . `Customer_servic...	*	1	0	0	10799.28	10799.28	10799.28
SELECT COUNT (*) FROM `mysql` . `rds_repl...	*	1745	0	0	17973888...	89915.53	10300.22
CREATE PROCEDURE `active_problem_custo...	*	1	0	0	8989.28	8989.28	8989.28

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].