Media Streaming Service

Database implementation

Link to github repo: https://github.com/jainam2907/Media-Streaming-DB

- By Query Masters

1. Project Objective and Real-World Relevance

Project Objective:

The project aims to model a relational database schema for a media streaming service that is similar to Netflix has projected a tracking of customers, contents, streaming events, and invoices that would include possible operations and queries related to the business model of the service.

The system should understand and store the following fundamental aspects:

- Account details, plans, and payments of a customer.
- Streaming events, which are customer activities that include watching a movie or a show, ratings and time.
- Detailed content like title, genre, cost-per-stream, and distribution information.
- Invoices with a few related subscription and payment plans of the customers.

The objective is to organize into a structure that allows the expanded interpretation of this data while minimizing redundancy and improving operational retrieval for multiple analytical and operational tasks.

Real-World Relevance

There are many companies in the media streaming business, including Netflix, Amazon Prime, Hulu, and others, this project is strongly relevant to all such companies. For such type of companies, data management is essential for doing business, analyzing user engagement, and tracing finances.

Business use cases of this schema:

- **Customer Engagement:** Tracking viewing behavior such as content preference, viewing patterns, and ratings makes it possible for a streaming service to know how to optimize content recommendations, experience, and satisfaction.

- **Financial Analytics:** Invoice data assists the firm in tracking the financial performance across different plans, including revenues earned per plan and country, and examining how cost-effective the content acquisitions are.
- **Content Strategy:** The company can introduce content, acquire content, or retire content by looking at views and ratings and analyzing the trends because of the performance metrics.

This schema coupled with the insights from the queries can help decision makers in deciding how to optimize operations for better profitability with superior user experience.

2. Detailed Overview of the Relational Schema:

Customer table:

This table contains information about the customers who use the streaming service.

Column Name	Data Type	Description
customer_id	INTEGER	Unique identifier for each customer (Primary Key).
first_name	VARCHAR(15)	First name of the customer.
last_name	VARCHAR(15)	Last name of the customer.
email_id	VARCHAR(32)	Email address of the customer.
phone_number	VARCHAR(12)	Contact number of the customer.
address	VARCHAR(100)	Customer’s physical address

state	VARCHAR(19)	Customer's state or province.
country	VARCHAR(13)	Customer's country.
sign_up_date	VARCHAR(11)	The date the customer signed up.
plan	VARCHAR(8)	The subscription plan the customer has selected (e.g., Basic, Premium, Standard).
total_payments	DECIMAL(10,2)	Total payments made by the customer

Streams table:

The streams table tracks individual streaming sessions, including content watched, customer details, and viewing times.

Column Name	Data Type	Description
stream_id	INTEGER	Unique identifier for each stream (Primary Key).
customer_id	INTEGER	Foreign key to the customer table, identifies the customer making the stream.
content_id	INTEGER	Foreign key to the content table, identifies the content being streamed.

content_title	VARCHAR(90)	Title of the content being streamed
genre	VARCHAR(29)	Genre of the content being streamed
stream_date	DATE	The date when the content was streamed.
stream_time	VARCHAR(50)	The exact time when the stream occurred.
stream_duration	INTEGER	The duration of the stream in minutes.
stream_rating	INTEGER	Rating given by the customer for the content streamed.

Content table:

The content table stores data on all the content available for streaming, including metadata such as title, genre, cost, and distributor.

Column Name	**Data Type**	**Description**
content_id	INTEGER	Unique identifier for each content item (Primary Key).

title	VARCHAR(90)	Title of the content (e.g., movie, TV show).
genre	VARCHAR(29)	Genre of the content (e.g., Drama, Comedy, Action).
duration	INTEGER	Duration of the content in minutes (e.g., length of a movie or an episode).
cost_per_stream	DECIMAL(6,2)	The cost of streaming the content per customer.
release_date	DATE	The release date of the content.
distributor	VARCHAR(36)	Distributor of the content (e.g., Netflix, Amazon).
stream_id	INTEGER	Reference to the streams table.
customer_id	INTEGER	Reference to the customer table.

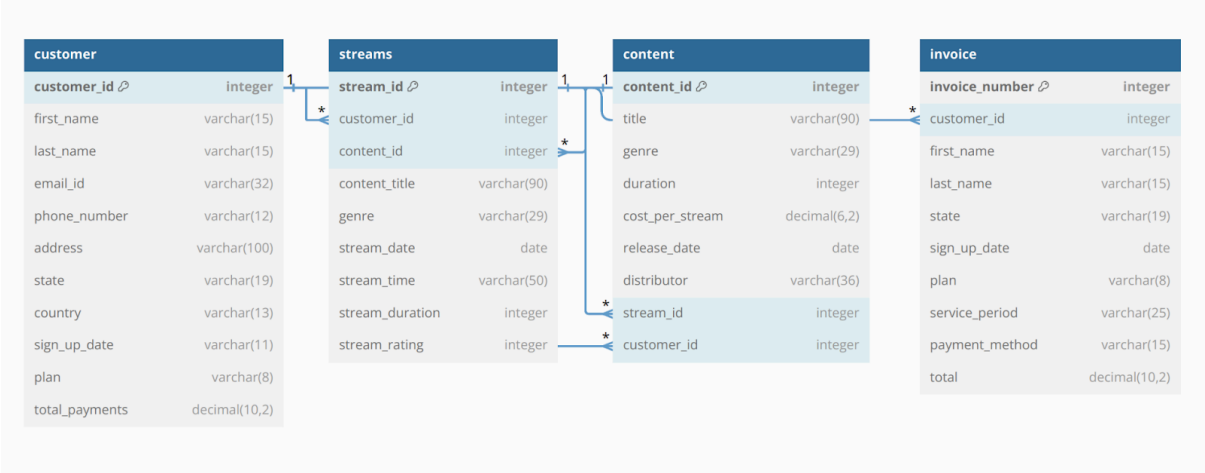
Invoice table:

The invoice table contains information about customer subscriptions, payment methods, and service plans.

Column Name	**Data Type**	**Description**
invoice_number	INTEGER	Unique identifier for each invoice (Primary Key).

customer_id	INTEGER	Foreign key to the customer table, identifies the customer associated with the invoice.
first_name	VARCHAR(15)	First name of the customer
last_name	VARCHAR(15)	Last name of the customer
state	VARCHAR(19)	State of the customer
sign_up_date	DATE	The sign-up date of the customer
plan	VARCHAR(8)	The subscription plan (e.g., Basic, Premium, Standard).
service_period	VARCHAR(25)	The duration of the service period for the invoice.
payment_method	VARCHAR(15)	The method of payment used by the customer (e.g., Credit Card, PayPal).
total	DECIMAL(10,2)	The total amount charged for the invoice.

ER diagram:



1. customer to streams:

- **Relation:** A customer can have multiple streams, e.g. a customer may be able to watch many content pieces; a stream can be associated with one customer only.
- **Cardinity:** One-to-Many(1:N) customer to streams.
    - **Explanation:** A single customer can have many stream records, but each stream record belongs to only that customer.

2. customer to content (via content.customer_id):

- **Relation:** A customer could have zero or more contents attached to him/her (contents viewed by the customer) while each content record has a corresponding customer.
- **Cardinality:** One(1)-To-Many Customer to Content.
    - **Explanatory note:** It reveals a redundancy in the schema that a customer may be related to many contents but each content record stores the ID of the customer.

3. streams to content:

- **Relation:** Every stream is associated with a piece of content and indicates that a customer streams specific content. However, it is also possible that multiple streams are related to the same content (as many customers can stream the same content).
- **Cardinality:** Many-to-one (N:1) from streams to content.
    - **Explanation:** Although many stream entries can be linked to similar pieces of content, each stream entry refers to only one content.

4. streams to customer:

- **Relation:** A Stream knows an identified customer while a customer can exist with several streams.
- **Cardinality:** Many-to-one (N:1) from stream to customer.
- **Explanation:** Multiple stream records can exist for a single customer, but a stream is always associated with a customer.

5. invoice to customer:

- **Relation:** There exists a connection between an invoice and a customer whereby one customer can possess many invoices owing to many service periods or transactions.
- **Cardinality:** It is a one-to-many (1:N) relationship with respect to customer invoices.
- **Explanation:** Although a customer may hold multiple invoices, each invoice will relate singularly to that particular customer.

Summary of the Relations and Cardinalities:

- **customer** → **streams**: One-to-many (1:N)
- **customer** → **content**: One-to-many (1:N) (but this is a redundancy)
- **streams** → **content:** Many-to-one (N:1)
- **streams** → **customer**: Many-to-one (N:1)
- **invoice** → **customer**: One-to-many (1:N)

Normalization:

Table: customer

```
CREATE TABLE customer (
   customer_id     INTEGER PRIMARY KEY,
   first_name      VARCHAR(15),
   last_name       VARCHAR(15),
   email_id        VARCHAR(32),
   phone_number    VARCHAR(12),
   state           VARCHAR(19),
   country         VARCHAR(13),
   sign_up_date    DATE,
   plan            VARCHAR(8), -- Redundant
   total_payments  DECIMAL(10,2) -- Redundant
);
```

1NF (First Normal Form)

- **Conditions:** Each cell must have an atomic value and should be unique in each row.
- **Already in 1NF:** Both the conditions of 1NF have been satisfied; all attributes are atomic and have unique rows.

2NF (Second Normal Form)

- **Conditions:** eliminate all possible dependencies (non-prime attributes are dependent on the entire primary key).
- **Analysis:** plan and total_payments depend on the primary key (customer_id).
- **Solution:** Moving plan and total_payments into extra tables.

Normalized Tables:

```
CREATE TABLE customer (
   customer_id     INTEGER PRIMARY KEY,
   first_name      VARCHAR(15),
   last_name       VARCHAR(15),
   email_id        VARCHAR(32),
   phone_number    VARCHAR(12),
   state           VARCHAR(19),
   country         VARCHAR(13),
   sign_up_date    DATE
);

CREATE TABLE customer_plan (
   customer_id     INTEGER PRIMARY KEY,
   plan            VARCHAR(8),
   total_payments  DECIMAL(10,2),
   FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

3NF (Third Normal Form)

- **Conditions:** Eliminate transitive dependencies (non-prime attributes depending only on primary key).
- **Analysis:** There are no transitive dependencies after the alterations from 2NF.
- **Result:** The Table is in 3NF

BCNF (Boyce-Codd Normal Form)

- **Conditions:** Functional dependencies are to be ensured to have a superkey.
- **Analysis:** customer_id is the superkey, and relies upon it all the non-prime attributes.
- **Result:** The Table is in BCNF.

Table: streams

Initial Schema

```
CREATE TABLE streams (
   stream_id       INT PRIMARY KEY,
   customer_id     INT,
   content_id      INT,
   content_title   VARCHAR(90), -- Redundant
   genre           VARCHAR(29), -- Redundant
   stream_date     DATE,
   stream_time     VARCHAR(50),
   stream_duration INT,
   stream_rating   INT
);
```

1NF

- **Already in 1NF**: All values are atomic and rows are unique.

2NF

- **Problem**: content_title and genre are attributes of content, not streams, and do not depend on the whole primary key.
- **Solution**: Move content_title and genre to the content table.

Normalized Table:

```
CREATE TABLE streams (
   stream_id       INT PRIMARY KEY,
   customer_id     INT,
   content_id      INT,
   stream_date     DATE,
   stream_time     VARCHAR(50),
```

```
   stream_duration INT,
   stream_rating   INT,
   FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
   FOREIGN KEY (content_id) REFERENCES content(content_id)
);
```

3NF

- **Analysis**: No transitive dependencies exist in the table after 2NF.
- **Result**: Table is in 3NF.

BCNF

- **Analysis**: primary key stream_id determines all attributes, and no functional dependencies violate BCNF.
- **Result**: Table is in BCNF.

Table: content

Initial Schema

```
CREATE TABLE content (
   content_id       INTEGER PRIMARY KEY,
   title            VARCHAR(90),
   genre            VARCHAR(29),
   duration         INTEGER,
   cost_per_stream  DECIMAL(6,2),
   release_date     DATE,
   distributor      VARCHAR(36),
   stream_id        INT, -- Redundant
   customer_id      INT -- Redundant
);
```

1NF

- **Already in 1NF**: Values are atomic, and rows are unique.

2NF

- **Problem**: stream_id and customer_id are attributes of streams, not content.
- **Solution**: Remove these columns.

Normalized Table:

```
CREATE TABLE content (
   content_id       INTEGER PRIMARY KEY,
   title            VARCHAR(90),
   genre            VARCHAR(29),
   duration         INTEGER,
   cost_per_stream  DECIMAL(6,2),
   release_date     DATE,
   distributor      VARCHAR(36)
);
```

3NF

- **Analysis**: No transitive dependencies exist in the table after 2NF.
- **Result**: Table is in 3NF.

BCNF

- **Analysis**: The primary key content_id determines all attributes, and no functional dependencies violate BCNF.
- **Result**: Table is in BCNF.

Table: invoice

Initial Schema

```
CREATE TABLE invoice (
   invoice_number   INTEGER PRIMARY KEY,
   customer_id      INTEGER,
   first_name       VARCHAR(15), -- Redundant
   last_name        VARCHAR(15), -- Redundant
   state            VARCHAR(19), -- Redundant
   sign_up_date     DATE, -- Redundant
   plan             VARCHAR(8),
   service_period   VARCHAR(25),
   payment_method   VARCHAR(15),
   total            DECIMAL(10,2)
```

```
);
```

1NF

- **Already in 1NF**: Values are atomic, and rows are unique.

2NF

- **Problem**: first_name, last_name, state, and sign_up_date are attributes of customer, not invoice.
- **Solution**: Remove these columns and reference the customer table.

Normalized Table:

```
CREATE TABLE invoice (
   invoice_number   INTEGER PRIMARY KEY,
   customer_id      INTEGER,
   plan             VARCHAR(8),
   service_period   VARCHAR(25),
   payment_method   VARCHAR(15),
   total            DECIMAL(10,2),
   FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

3NF

- **Analysis**: No transitive dependencies exist in the table after 2NF.
- **Result**: Table is in 3NF.

BCNF

- **Analysis**: The primary key invoice_number determines all attributes, and no functional dependencies violate BCNF.
- **Result**: Table is in BCNF.

Final schema:

Table: customer

```
CREATE TABLE customer (
   customer_id     INTEGER PRIMARY KEY,
   first_name      VARCHAR(15),
   last_name       VARCHAR(15),
   email_id        VARCHAR(32),
   phone_number    VARCHAR(12),
   state           VARCHAR(19),
   country         VARCHAR(13),
   sign_up_date    DATE
);
```

```
mysql> CREATE TABLE customer (
    ->    customer_id     INTEGER PRIMARY KEY AUTO_INCREMENT,
    ->    first_name      VARCHAR(15),
    ->    last_name       VARCHAR(15),
    ->    email_id        VARCHAR(32),
    ->    phone_number    VARCHAR(12),
    ->    state           VARCHAR(19),
    ->    country         VARCHAR(13),
    ->    sign_up_date    DATE
    -> );
Query OK, 0 rows affected (0.03 sec)
```

Table: customer_plan

```
CREATE TABLE customer_plan (
   customer_id     INTEGER PRIMARY KEY,
   plan            VARCHAR(8),
   total_payments  DECIMAL(10,2),
   FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

```
mysql> CREATE TABLE customer_plan (
    ->    customer_id     INTEGER PRIMARY KEY,
    ->    plan            VARCHAR(8),
    ->    total_payments  DECIMAL(10,2),
    ->    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
    -> );
Query OK, 0 rows affected (0.03 sec)
```

Table: streams

```
CREATE TABLE streams (
   stream_id       INTEGER PRIMARY KEY,
   customer_id     INTEGER,
   content_id      INTEGER,
   stream_date     DATE,
   stream_time     VARCHAR(50),
   stream_duration INTEGER,
   stream_rating   INTEGER,
   FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
   FOREIGN KEY (content_id) REFERENCES content(content_id)
);
```

```
mysql> CREATE TABLE streams (
    ->    stream_id       INTEGER PRIMARY KEY AUTO_INCREMENT,
    ->    customer_id     INTEGER,
    ->    content_id      INTEGER,
    ->    stream_date     DATE,
    ->    stream_time     VARCHAR(50),
    ->    stream_duration INTEGER,
    ->    stream_rating   INTEGER,
    ->    FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
    ->    FOREIGN KEY (content_id) REFERENCES content(content_id)
    -> );
Query OK, 0 rows affected (0.04 sec)
```

Table: content

```
CREATE TABLE content (
   content_id       INTEGER PRIMARY KEY,
   title            VARCHAR(90),
   genre            VARCHAR(29),
   duration         INTEGER,
   cost_per_stream  DECIMAL(6,2),
   release_date     DATE,
   distributor      VARCHAR(36)
);
```

```
mysql> CREATE TABLE content (
    ->     content_id       INTEGER PRIMARY KEY AUTO_INCREMENT,
    ->     title            VARCHAR(90),
    ->     genre            VARCHAR(29),
    ->     duration         INTEGER,
    ->     cost_per_stream  DECIMAL(6,2),
    ->     release_date     DATE,
    ->     distributor      VARCHAR(36)
    -> );
Query OK, 0 rows affected (0.03 sec)
```

Table: invoice

```
CREATE TABLE invoice (
   invoice_number   INTEGER PRIMARY KEY,
   customer_id      INTEGER,
   plan             VARCHAR(8),
   service_period   VARCHAR(25),
   payment_method   VARCHAR(15),
   total            DECIMAL(10,2),
   FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

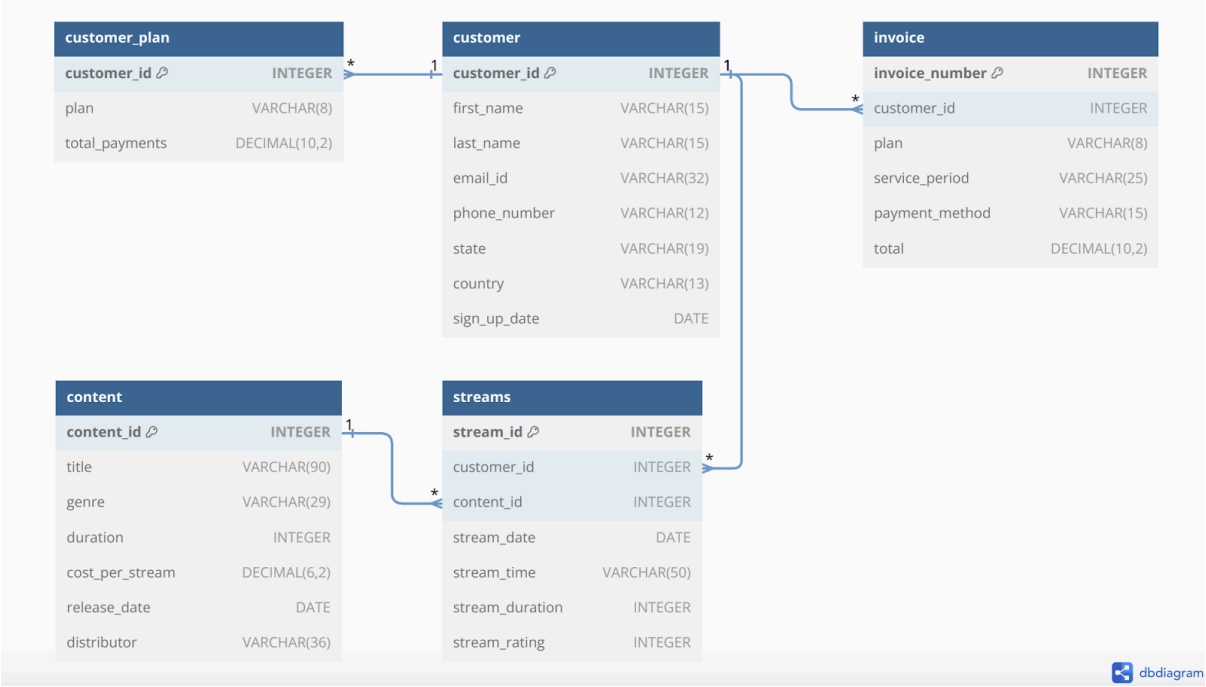
```
mysql> CREATE TABLE invoice (
    ->     invoice_number   INTEGER PRIMARY KEY AUTO_INCREMENT,
    ->     customer_id      INTEGER,
    ->     plan             VARCHAR(8),
    ->     service_period   VARCHAR(25),
    ->     payment_method   VARCHAR(15),
    ->     total            DECIMAL(10,2),
    ->     FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
    -> );
Query OK, 0 rows affected (0.03 sec)
```

Explanation of Changes

1. Customer Table
    - Removed redundant columns plan and total_payments and created a separate customer_plan table.
2. Streams Table
    - Removed redundant content_title and genre attributes and linked content_id with the content table.
3. Content Table

- Removed stream_id and customer_id as they do not belong to content.

4. Invoice Table
    - Removed first_name, last_name, state, and sign_up_date as they belong to the customer.



3. Queries and Optimization

Queries for different use cases:

Query 1 - To fetch customer engagement statistics (total streams and average ratings per customer)

```sql
SELECT s.customer_id,
       c.first_name,
       c.last_name,
       COUNT(s.stream_id) AS total_streams,
       AVG(s.stream_rating) AS avg_rating
FROM streams s
JOIN customer c ON s.customer_id = c.customer_id
GROUP BY s.customer_id, c.first_name, c.last_name
ORDER BY total_streams DESC;
```

```
mysql> SELECT s.customer_id,
    ->           c.first_name,
    ->           c.last_name,
    ->           COUNT(s.stream_id) AS total_streams,
    ->           AVG(s.stream_rating) AS avg_rating
    -> FROM streams s
    -> JOIN customer c ON s.customer_id = c.customer_id
    -> GROUP BY s.customer_id, c.first_name, c.last_name
    -> ORDER BY total_streams DESC;
+-------------+-----------------+----------------+---------------+------------+
| customer_id | first_name      | last_name      | total_streams | avg_rating |
+-------------+-----------------+----------------+---------------+------------+
|        7784 | FirstName_7784  | LastName_7784  |             7 |     4.0000 |
|        9569 | FirstName_9569  | LastName_9569  |             6 |     3.5000 |
|        3429 | FirstName_3429  | LastName_3429  |             6 |     3.6667 |
|        2127 | FirstName_2127  | LastName_2127  |             6 |     3.1667 |
|        7905 | FirstName_7905  | LastName_7905  |             6 |     2.8333 |
|         774 | FirstName_774   | LastName_774   |             5 |     3.2000 |
|        8164 | FirstName_8164  | LastName_8164  |             5 |     3.6000 |
|        8706 | FirstName_8706  | LastName_8706  |             5 |     2.8000 |
|        6867 | FirstName_6867  | LastName_6867  |             5 |     2.0000 |
|        4581 | FirstName_4581  | LastName_4581  |             5 |     3.4000 |
|        5777 | FirstName_5777  | LastName_5777  |             5 |     3.8000 |
|        9548 | FirstName_9548  | LastName_9548  |             5 |     2.8000 |
|        8599 | FirstName_8599  | LastName_8599  |             5 |     2.2000 |
|        5745 | FirstName_5745  | LastName_5745  |             5 |     3.4000 |
|        9845 | FirstName_9845  | LastName_9845  |             5 |     2.6000 |
|        4128 | FirstName_4128  | LastName_4128  |             5 |     2.4000 |
|        1217 | FirstName_1217  | LastName_1217  |             5 |     2.8000 |
```

Objective: Analyze customer activity, including the number of streams and their average ratings.

Use Case: Helps identify highly engaged customers and their satisfaction levels.

Query 2 - To identify the most popular content by total streams.

```sql
SELECT c.title,
       COUNT(s.stream_id) AS total_views
FROM streams s
JOIN content c ON s.content_id = c.content_id
GROUP BY c.title
ORDER BY total_views DESC
LIMIT 10;
```

```
mysql> SELECT c.title,
    ->           COUNT(s.stream_id) AS total_views
    -> FROM streams s
    -> JOIN content c ON s.content_id = c.content_id
    -> GROUP BY c.title
    -> ORDER BY total_views DESC
    -> LIMIT 10;
+--------------------+-------------+
| title              | total_views |
+--------------------+-------------+
| Content Title 7318 |           7 |
| Content Title 5281 |           7 |
| Content Title 395  |           7 |
| Content Title 5501 |           6 |
| Content Title 7222 |           6 |
| Content Title 5019 |           6 |
| Content Title 4465 |           6 |
| Content Title 1903 |           5 |
| Content Title 684  |           5 |
| Content Title 2100 |           5 |
+--------------------+-------------+
10 rows in set (0.02 sec)
```

Objective: Find the most-streamed content titles.

Use Case: Assists in strategic decisions about content retention or promotion

Query 3 - To calculate total revenue generated per subscription plan:

```sql
SELECT cp.plan,
       SUM(i.total) AS total_revenue
```

```
FROM customer_plan cp
JOIN invoice i ON cp.customer_id = i.customer_id
GROUP BY cp.plan
ORDER BY total_revenue DESC;
```

```
mysql> SELECT cp.plan,
    ->           SUM(i.total) AS total_revenue
    -> FROM customer_plan cp
    -> JOIN invoice i ON cp.customer_id = i.customer_id
    -> GROUP BY cp.plan
    -> ORDER BY total_revenue DESC;
+----------+---------------+
| plan     | total_revenue |
+----------+---------------+
| Premium  |     343382.71 |
| Standard |     332709.32 |
| Basic    |     330495.74 |
+----------+---------------+
3 rows in set (0.06 sec)
```

Objective: Determine the revenue contribution of each subscription plan.

Use Case: Evaluates the financial performance of different plans.

Query 4 - To identify content profitability (revenue vs. cost):

```
SELECT c.title,
       SUM(i.total) AS total_revenue,
       (c.cost_per_stream * COUNT(s.stream_id)) AS total_cost
FROM content c
JOIN streams s ON c.content_id = s.content_id
JOIN invoice i ON s.customer_id = i.customer_id
GROUP BY c.title, c.cost_per_stream
HAVING total_revenue > total_cost
ORDER BY total_revenue DESC
LIMIT 10;
```

```
mysql> SELECT c.title,
    ->           SUM(i.total) AS total_revenue,
    ->           (c.cost_per_stream * COUNT(s.stream_id)) AS total_cost
    -> FROM content c
    -> JOIN streams s ON c.content_id = s.content_id
    -> JOIN invoice i ON s.customer_id = i.customer_id
    -> GROUP BY c.title, c.cost_per_stream
    -> HAVING total_revenue > total_cost
    -> ORDER BY total_revenue DESC
    -> LIMIT 10;
+-------------------+---------------+------------+
| title             | total_revenue | total_cost |
+-------------------+---------------+------------+
| Content Title 9662 |       1337.22 |      39.00 |
| Content Title 4821 |       1143.06 |       9.18 |
| Content Title 25   |       1063.61 |      20.70 |
| Content Title 8788 |       1048.80 |      39.36 |
| Content Title 9845 |       1014.77 |      37.73 |
| Content Title 8505 |        995.82 |      68.72 |
| Content Title 1944 |        963.47 |      63.00 |
| Content Title 1961 |        950.54 |      42.42 |
| Content Title 5393 |        938.35 |      42.80 |
| Content Title 4989 |        936.23 |      50.96 |
+-------------------+---------------+------------+
10 rows in set (0.10 sec)
```

Objective: Compare the revenue generated by content with its total cost.

Use Case: Identifies highly profitable content to prioritize similar acquisitions.

Query 5 - To analyze the popularity of content genres across countries.

```sql
SELECT cu.country,
       co.genre,
       SUM(s.stream_id) AS genre_views
FROM streams s
JOIN customer cu ON s.customer_id = cu.customer_id
JOIN content co ON s.content_id = co.content_id
GROUP BY cu.country, co.genre
ORDER BY cu.country, genre_views DESC;
```

```
mysql> SELECT cu.country,
    ->           co.genre,
    ->           SUM(s.stream_id) AS genre_views
    -> FROM streams s
    -> JOIN customer cu ON s.customer_id = cu.customer_id
    -> JOIN content co ON s.content_id = co.content_id
    -> GROUP BY cu.country, co.genre
    -> ORDER BY cu.country, genre_views DESC;
+---------+-------------+-------------+
| country | genre       | genre_views |
+---------+-------------+-------------+
| USA     | Action      |    10545964 |
| USA     | Horror      |    10039307 |
| USA     | Drama       |     9913444 |
| USA     | Documentary |     9800052 |
| USA     | Comedy      |     9716234 |
+---------+-------------+-------------+
5 rows in set (0.04 sec)
```

Objective: Understand the demand for specific genres in different regions.

Use Case: Supports regional content strategy planning.

Analysis of Query Performance before and after optimization:

Query optimization for fetching customer engagement statistics

```sql
SELECT s.customer_id,
       c.first_name,
       c.last_name,
       COUNT(s.stream_id) AS total_streams,
       AVG(s.stream_rating) AS avg_rating
FROM streams s
JOIN customer c ON s.customer_id = c.customer_id
GROUP BY s.customer_id, c.first_name, c.last_name;
```

```
mysql> SELECT s.customer_id,
    ->           c.first_name,
    ->           c.last_name,
    ->           COUNT(s.stream_id) AS total_streams,
    ->           AVG(s.stream_rating) AS avg_rating
    -> FROM streams s
    -> JOIN customer c ON s.customer_id = c.customer_id
    -> GROUP BY s.customer_id, c.first_name, c.last_name;
+-------------+-----------------+----------------+---------------+------------+
| customer_id | first_name      | last_name      | total_streams | avg_rating |
+-------------+-----------------+----------------+---------------+------------+
|         774 | FirstName_774   | LastName_774   |             5 |     3.2000 |
|        2649 | FirstName_2649  | LastName_2649  |             1 |     3.0000 |
|        6225 | FirstName_6225  | LastName_6225  |             2 |     5.0000 |
|        6497 | FirstName_6497  | LastName_6497  |             2 |     2.0000 |
|         312 | FirstName_312   | LastName_312   |             4 |     2.5000 |
|        7977 | FirstName_7977  | LastName_7977  |             3 |     3.0000 |
|        9793 | FirstName_9793  | LastName_9793  |             4 |     4.0000 |
|        6234 | FirstName_6234  | LastName_6234  |             2 |     3.5000 |
|        6763 | FirstName_6763  | LastName_6763  |             2 |     2.0000 |
|        2824 | FirstName_2824  | LastName_2824  |             3 |     2.3333 |
|          54 | FirstName_54    | LastName_54    |             4 |     4.2500 |
|        2726 | FirstName_2726  | LastName_2726  |             3 |     4.0000 |
|        4670 | FirstName_4670  | LastName_4670  |             2 |     3.0000 |
```

To check the performance, we run the following `EXPLAIN` on this query

```sql
EXPLAIN SELECT s.customer_id,
               c.first_name,
               c.last_name,
               COUNT(s.stream_id) AS total_streams,
               AVG(s.stream_rating) AS avg_rating
FROM streams s
JOIN customer c ON s.customer_id = c.customer_id
GROUP BY s.customer_id, c.first_name, c.last_name;
```

```
mysql> EXPLAIN SELECT s.customer_id,
    ->                c.first_name,
    ->                c.last_name,
    ->                COUNT(s.stream_id) AS total_streams,
    ->                AVG(s.stream_rating) AS avg_rating
    -> FROM streams s
    -> JOIN customer c ON s.customer_id = c.customer_id
    -> GROUP BY s.customer_id, c.first_name, c.last_name;
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------------------+------+----------+-----------------------------------------------
----------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref                         | rows | filtered | Extra
          |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------------------+------+----------+-----------------------------------------------
----------+
|  1 | SIMPLE      | s     | NULL       | ALL    | customer_id   | NULL    | NULL    | NULL                        | 9688 |   100.00 | Using where; Using
temporary |
|  1 | SIMPLE      | c     | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | media_streaming.s.customer_id |    1 |   100.00 | NULL
          |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------------------+------+----------+-----------------------------------------------
----------+
2 rows in set, 1 warning (0.01 sec)
```

Adding Indices to `streams.customer_ID`

```sql
CREATE INDEX idx_streams_customer_id ON streams(customer_id);
```

```
mysql> CREATE INDEX idx_streams_customer_id ON streams(customer_id);
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Adding Indices to `customers.customer_ID`

```sql
 CREATE INDEX idx_customer_id ON customer(customer_id);
```

```
mysql> CREATE INDEX idx_customer_id ON customer(customer_id);
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query Optimization for Most Popular Content

```sql
SELECT co.title,
       COUNT(s.stream_id) AS total_views
FROM streams s
JOIN content co ON s.content_id = co.content_id
GROUP BY co.title;
```

```
mysql> SELECT co.title,
    ->           COUNT(s.stream_id) AS total_views
    -> FROM streams s
    -> JOIN content co ON s.content_id = co.content_id
    -> GROUP BY co.title;
+---------------------+-------------+
| title               | total_views |
+---------------------+-------------+
| Content Title 1     |           3 |
| Content Title 3     |           1 |
| Content Title 4     |           3 |
| Content Title 5     |           2 |
| Content Title 8     |           1 |
| Content Title 10    |           1 |
| Content Title 12    |           2 |
| Content Title 15    |           2 |
| Content Title 16    |           2 |
| Content Title 18    |           1 |
| Content Title 19    |           2 |
| Content Title 21    |           2 |
| Content Title 22    |           2 |
| Content Title 23    |           2 |
| Content Title 25    |           5 |
| Content Title 27    |           2 |
```

```sql
EXPLAIN SELECT co.title,
               COUNT(s.stream_id) AS total_views
FROM streams s
JOIN content co ON s.content_id = co.content_id
GROUP BY co.title;
```

```
mysql> EXPLAIN SELECT co.title,
    ->                COUNT(s.stream_id) AS total_views
    -> FROM streams s
    -> JOIN content co ON s.content_id = co.content_id
    -> GROUP BY co.title;
+----+-------------+-------+------------+--------+---------------+------------+---------+-----------------------------+------+----------+-----------------------------------------------
-------------------------+
| id | select_type | table | partitions | type   | possible_keys | key        | key_len | ref                         | rows | filtered | Extra
                         |
+----+-------------+-------+------------+--------+---------------+------------+---------+-----------------------------+------+----------+-----------------------------------------------
-------------------------+
|  1 | SIMPLE      | s     | NULL       | index  | content_id    | content_id | 5       | NULL                        | 9688 |   100.00 | Using where; Usin
g index; Using temporary |
|  1 | SIMPLE      | co    | NULL       | eq_ref | PRIMARY       | PRIMARY    | 4       | media_streaming.s.content_id |    1 |   100.00 | NULL
                         |
+----+-------------+-------+------------+--------+---------------+------------+---------+-----------------------------+------+----------+-----------------------------------------------
-------------------------+
2 rows in set, 1 warning (0.00 sec)
```

```sql
CREATE INDEX idx_streams_content_id ON streams(content_id);
```

```
mysql> CREATE INDEX idx_streams_content_id ON streams(content_id);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```sql
CREATE INDEX idx_content_id ON content(content_id);
```

```
mysql> CREATE INDEX idx_content_id ON content(content_id);
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Optimizing:

Composite Indexes:

```sql
SELECT s.customer_id,
       c.first_name,
       c.last_name,
       COUNT(s.stream_id) AS total_streams,
       AVG(s.stream_rating) AS avg_rating
FROM streams s
JOIN customer c ON s.customer_id = c.customer_id
GROUP BY s.customer_id, c.first_name, c.last_name;
```

```
mysql> SELECT s.customer_id,
    ->        c.first_name,
    ->        c.last_name,
    ->        COUNT(s.stream_id) AS total_streams,
    ->        AVG(s.stream_rating) AS avg_rating
    -> FROM streams s
    -> JOIN customer c ON s.customer_id = c.customer_id
    -> GROUP BY s.customer_id, c.first_name, c.last_name;
+-------------+----------------+---------------+---------------+------------+
| customer_id | first_name     | last_name     | total_streams | avg_rating |
+-------------+----------------+---------------+---------------+------------+
|         774 | FirstName_774  | LastName_774  |             5 |     3.2000 |
|        2649 | FirstName_2649 | LastName_2649 |             1 |     3.0000 |
|        6225 | FirstName_6225 | LastName_6225 |             2 |     5.0000 |
|        6497 | FirstName_6497 | LastName_6497 |             2 |     2.0000 |
|         312 | FirstName_312  | LastName_312  |             4 |     2.5000 |
|        7977 | FirstName_7977 | LastName_7977 |             3 |     3.0000 |
|        9793 | FirstName_9793 | LastName_9793 |             4 |     4.0000 |
|        6234 | FirstName_6234 | LastName_6234 |             2 |     3.5000 |
|        6763 | FirstName_6763 | LastName_6763 |             2 |     2.0000 |
|        2824 | FirstName_2824 | LastName_2824 |             3 |     2.3333 |
|          54 | FirstName_54   | LastName_54   |             4 |     4.2500 |
|        2726 | FirstName_2726 | LastName_2726 |             3 |     4.0000 |
|        4670 | FirstName_4670 | LastName_4670 |             2 |     3.0000 |
|        8996 | FirstName_8996 | LastName_8996 |             1 |     5.0000 |
|        1331 | FirstName_1331 | LastName_1331 |             1 |     5.0000 |
```

```
CREATE INDEX idx_streams_customer_id_rating ON streams(customer_id, stream_rating);
```

```
mysql> CREATE INDEX idx_streams_customer_id_rating ON streams(customer_id, stream_rating);
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Covering index

```
SELECT co.title, COUNT(s.stream_id) AS total_views
FROM streams s
JOIN content co ON s.content_id = co.content_id
GROUP BY co.title;
```

```
mysql> SELECT co.title, COUNT(s.stream_id) AS total_views
    -> FROM streams s
    -> JOIN content co ON s.content_id = co.content_id
    -> GROUP BY co.title;
+--------------------+-------------+
| title              | total_views |
+--------------------+-------------+
| Content Title 1    |           3 |
| Content Title 3    |           1 |
| Content Title 4    |           3 |
| Content Title 5    |           2 |
| Content Title 8    |           1 |
| Content Title 10   |           1 |
| Content Title 12   |           2 |
| Content Title 15   |           2 |
| Content Title 16   |           2 |
| Content Title 18   |           1 |
| Content Title 19   |           2 |
| Content Title 21   |           2 |
| Content Title 22   |           2 |
```

```sql
CREATE INDEX idx_covering_streams ON streams(content_id, stream_id);
```

```
mysql> CREATE INDEX idx_covering_streams ON streams(content_id, stream_id);
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Query Caching

Using redis or mysql query cache

```sql
SELECT co.title,
       COUNT(s.stream_id) AS total_views
FROM streams s
JOIN content co ON s.content_id = co.content_id
GROUP BY co.title;
```

```
mysql> SELECT co.title,
    ->        COUNT(s.stream_id) AS total_views
    -> FROM streams s
    -> JOIN content co ON s.content_id = co.content_id
    -> GROUP BY co.title;
+---------------------+-------------+
| title               | total_views |
+---------------------+-------------+
| Content Title 1     |           3 |
| Content Title 3     |           1 |
| Content Title 4     |           3 |
| Content Title 5     |           2 |
| Content Title 8     |           1 |
| Content Title 10    |           1 |
| Content Title 12    |           2 |
| Content Title 15    |           2 |
| Content Title 16    |           2 |
| Content Title 18    |           1 |
| Content Title 19    |           2 |
| Content Title 21    |           2 |
| Content Title 22    |           2 |
| Content Title 23    |           2 |
| Content Title 25    |           5 |
```

Optimizing group by and order by

```sql
SELECT co.title, COUNT(s.stream_id) AS total_views
FROM streams s
JOIN content co ON s.content_id = co.content_id
GROUP BY co.title
ORDER BY total_views DESC;
```

```
mysql> SELECT co.title, COUNT(s.stream_id) AS total_views
    -> FROM streams s
    -> JOIN content co ON s.content_id = co.content_id
    -> GROUP BY co.title
    -> ORDER BY total_views DESC;
+--------------------+-------------+
| title              | total_views |
+--------------------+-------------+
| Content Title 395  |           7 |
| Content Title 5281 |           7 |
| Content Title 7318 |           7 |
| Content Title 4465 |           6 |
| Content Title 5019 |           6 |
| Content Title 5501 |           6 |
| Content Title 7222 |           6 |
| Content Title 25   |           5 |
| Content Title 684  |           5 |
| Content Title 722  |           5 |
| Content Title 792  |           5 |
| Content Title 976  |           5 |
| Content Title 1153 |           5 |
| Content Title 1171 |           5 |
| Content Title 1903 |           5 |
| Content Title 2100 |           5 |
| Content Title 3709 |           5 |
| Content Title 3899 |           5 |
| Content Title 4051 |           5 |
| Content Title 4117 |           5 |
| Content Title 4144 |           5 |
| Content Title 4408 |           5 |
```

```
CREATE INDEX idx_covering_streams ON streams(content_id, stream_id);
```

Using query hints

If MySQL chooses a suboptimal query plan, use **hints** to guide the optimizer.

```
SELECT c.title, COUNT(s.stream_id) AS total_views
FROM streams s
JOIN content c USE INDEX (idx_content_id) ON s.content_id =
c.content_id
GROUP BY c.title;
```

```
mysql> SELECT c.title, COUNT(s.stream_id) AS total_views
    -> FROM streams s
    -> JOIN content c USE INDEX (idx_content_id) ON s.content_id = c.content_id
    -> GROUP BY c.title;
+--------------------+-------------+
| title              | total_views |
+--------------------+-------------+
| Content Title 1    |           3 |
| Content Title 3    |           1 |
| Content Title 4    |           3 |
| Content Title 5    |           2 |
| Content Title 8    |           1 |
| Content Title 10   |           1 |
| Content Title 12   |           2 |
| Content Title 15   |           2 |
| Content Title 16   |           2 |
| Content Title 18   |           1 |
| Content Title 19   |           2 |
| Content Title 21   |           2 |
| Content Title 22   |           2 |
```

Avoid SELECT * Queries
Select only the columns you need to minimize data transfer and processing time.

```sql
SELECT stream_id, content_id, stream_date, stream_rating
FROM streams
WHERE customer_id = 101;
```

```
mysql> SELECT stream_id, content_id, stream_date, stream_rating
    -> FROM streams
    -> WHERE customer_id = 101;
+-----------+------------+-------------+---------------+
| stream_id | content_id | stream_date | stream_rating |
+-----------+------------+-------------+---------------+
|       195 |        230 | 2023-11-21  |             3 |
|      5530 |       8945 | 1994-09-04  |             5 |
+-----------+------------+-------------+---------------+
2 rows in set (0.00 sec)
```

4. Transaction and Concurrency Handling

- Atomicity: A transaction must be all or nothing. If one part of the transaction fails, the entire transaction is rolled back.

- Consistency: A transaction must take the database from one valid state to another, maintaining all the rules, constraints, and triggers.
- Isolation: Transactions are isolated from each other. The partial results of one transaction should not be visible to other transactions.
- Durability: If a transaction commits, its changes are durable in the presence of a crash.

1. Transaction Demonstration (ACID Compliance)

Transaction Scenario:

- When a customer makes a payment for an invoice, the system updates the `invoice` table and the `customer_plan` table (e.g., updating the `total_payments` column).

SQL Transaction for ACID Compliance

```sql
START TRANSACTION;




UPDATE customer_plan

SET total_payments = total_payments + 100.00

WHERE customer_id = 1;




INSERT INTO invoice (customer_id, plan, service_period, payment_method, total)

VALUES (1, 'Premium', '2024-01', 'Credit Card', 100.00);




COMMIT;
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> UPDATE customer_plan
    -> SET total_payments = total_payments + 100.00
    -> WHERE customer_id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql>
mysql> INSERT INTO invoice (customer_id, plan, service_period, payment_method, total)
    -> VALUES (1, 'Premium', '2024-01', 'Credit Card', 100.00);
Query OK, 1 row affected (0.00 sec)

mysql>
mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)
```

Atomicity:

If any part of this transaction fails-say, because of a constraint violation, such as trying to update a non-existent customer-then none of the changes will take effect, leaving the database in a consistent state.

Consistency:

The database maintains consistency by updating the total_payments field and inserting the invoice, keeping the customer_plan and invoice tables in a valid state.

Isolation:

While this transaction is running, other transactions should not see intermediate results. For example, if a second transaction tries to read or modify the customer_plan or invoice tables, it should either wait for the transaction to complete or see a consistent state based on the isolation level.

Durability:

Once the transaction is committed, the changes are permanent, even if the system crashes immediately after the commit.

2. Simulating Concurrency Issues and Solving with Isolation Levels

In multi-user environments, concurrency issues may occur when more than one transaction tries to access the same data for modification simultaneously. We will

simulate some common concurrency problems and handle them by using appropriate isolation levels.

Common Concurrency Issues:

Lost Update: When two transactions update the same data simultaneously; updates from one transaction get lost.
Temporary Inconsistency (Dirty Read): One transaction is reading data being modified by some other transaction.

Non-repeatable Read: One transaction reads a value and, when it reads again, it has been changed by another transaction. Phantom Read: A transaction reads rows based on a condition where the condition may be rendered different by some other inserting or deleting rows.

Isolation Levels:

Read Uncommitted (Lowest Isolation Level): Transactions can read uncommitted data, known as dirty reads, which might introduce temporary inconsistencies.

Read Committed: One transaction can only read those data which are committed. It prevents dirty reads but allows non-repeatable reads.

Repeatable Read: It prevents dirty reads and non-repeatable reads, but it may still allow phantom reads.

Serializable (Highest Isolation Level): Avoids dirty reads, non-repeatable reads, and phantom reads by using locks on the data.

Scenario 1: Lost Update

Two transactions try to update the same customer's payment amount concurrently:

- **Transaction A:**

```
START TRANSACTION;

UPDATE customer_plan

SET total_payments = total_payments + 100.00
```

```
WHERE customer_id = 1;
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE customer_plan
    -> SET total_payments = total_payments + 100.00
    -> WHERE customer_id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

- **Transaction B:**

```
START TRANSACTION;

UPDATE customer_plan

SET total_payments = total_payments + 150.00

WHERE customer_id = 1;
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE customer_plan
    -> SET total_payments = total_payments + 150.00
    -> WHERE customer_id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from customer_plan
    -> LIMIT 1;
+-------------+----------+----------------+
| customer_id | plan     | total_payments |
+-------------+----------+----------------+
|           2 | Standard |         360.83 |
+-------------+----------+----------------+
1 row in set (0.00 sec)
```

If **Transaction A** commits firstly, then **Transaction B** also commits, the update of Transaction A is lost, this is a Lost Update problem.

Solution: Using Serializable Isolation Level

To avoid this, we set the isolation level to Serializable that will prohibit both transactions to modify the data at the same time:

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;




START TRANSACTION;

UPDATE customer_plan SET total_payments = total_payments + 100.00 WHERE
customer_id = 1;

COMMIT;
```

```sql
START TRANSACTION;

UPDATE customer_plan SET total_payments = total_payments + 150.00 WHERE
customer_id = 1;

COMMIT;
```

```
mysql> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> -- Transaction A
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE customer_plan
    -> SET total_payments = total_payments + 100.00
    -> WHERE customer_id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE customer_plan
    -> SET total_payments = total_payments + 150.00
    -> WHERE customer_id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Here, **Transaction B** will be blocked until **Transaction A** commits or rolls back, ensuring that both transactions cannot modify the same data concurrently.

Scenario 2: Dirty Read

Suppose **Transaction A** updates a customer's payment but has not yet committed the change:

- **Transaction A:**

```
START TRANSACTION;

UPDATE customer_plan SET total_payments = total_payments + 100.00 WHERE
customer_id = 1;
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE customer_plan SET total_payments = total_payments + 100.00 WHE
RE customer_id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

-- Transaction is not committed yet.

- **Transaction B:**

```
START TRANSACTION;
```

```sql
SELECT * FROM customer_plan WHERE customer_id = 1;
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM customer_plan WHERE customer_id = 1;
Empty set (0.00 sec)
```

-- Reads uncommitted data.

Transaction B is reading uncommitted data, and this can lead to inconsistencies in results. It's a dirty read problem.

Solution: Using Read Committed Isolation Level

Setting the isolation level to **Read Committed** ensures that Transaction B sees only committed data:

```sql
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;



-- Transaction A:

START TRANSACTION;

UPDATE customer_plan SET total_payments = total_payments + 100.00 WHERE
customer_id = 1;

COMMIT;



-- Transaction B:

START TRANSACTION;

SELECT * FROM customer_plan WHERE customer_id = 1;
```

```
COMMIT;
```

```
mysql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
ERROR 1568 (25001): Transaction characteristics can't be changed while a tra
nsaction is in progress
mysql>
mysql>
mysql> -- Transaction A:
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE customer_plan SET total_payments = total_payments + 100.00 WHE
RE customer_id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> -- Transaction B:
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM customer_plan WHERE customer_id = 1;
Empty set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Now, **Transaction B** will not read uncommitted changes made by **Transaction A**.

Scenario 3: Phantom Read

Assume that **Transaction A** reads a set of invoices for a customer, and **Transaction B** inserts a new invoice for that customer:

- **Transaction A:**

```
START TRANSACTION;

SELECT * FROM invoice WHERE customer_id = 1;
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM invoice WHERE customer_id = 1;
+----------------+-------------+----------+----------------+----------------
+--------+
| invoice_number | customer_id | plan     | service_period | payment_method
| total  |
+----------------+-------------+----------+----------------+----------------
+--------+
|            143 |           1 | Standard | Period_9       | PayPal
| 117.81 |
|          10001 |           1 | Premium  | 2024-12        | Credit Card
|  99.99 |
|          10002 |           1 | Premium  | 2024-01        | Credit Card
| 100.00 |
+----------------+-------------+----------+----------------+----------------
+--------+
3 rows in set (0.00 sec)
```

-- Reads invoices for customer 1.

- **Transaction B:**

```
START TRANSACTION;

INSERT INTO invoice (customer_id, plan, service_period, payment_method, total)

VALUES (1, 'Premium', '2024-01', 'Credit Card', 100.00);

COMMIT;
```

```
mysql> INSERT INTO invoice (customer_id, plan, service_period, payment_metho
d, total)
    -> VALUES (1, 'Premium', '2024-01', 'Credit Card', 100.00);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)
```

This may result in transaction A not seeing the newly inserted row in its initial query, thus causing phantom reads.

Solution: Using Repeatable Read Isolation Level

By setting the isolation level to **Repeatable Read, Transaction A** is guaranteed that its view of the data set remains constant, even in the presence of concurrent data insertion by **Transaction B:**

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;



-- Transaction A:

START TRANSACTION;

SELECT * FROM invoice WHERE customer_id = 1;

-- Will not see newly inserted rows during the transaction.

COMMIT;



-- Transaction B:

START TRANSACTION;

INSERT INTO invoice (customer_id, plan, service_period, payment_method, total)

VALUES (1, 'Premium', '2024-01', 'Credit Card', 100.00);

COMMIT;
```

```
mysql> -- Transaction A:
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM invoice WHERE customer_id = 1;
+----------------+-------------+----------+----------------+----------------
+--------+
| invoice_number | customer_id | plan     | service_period | payment_method
| total  |
+----------------+-------------+----------+----------------+----------------
+--------+
|            143 |           1 | Standard | Period_9       | PayPal
| 117.81 |
|          10001 |           1 | Premium  | 2024-12        | Credit Card
|  99.99 |
|          10002 |           1 | Premium  | 2024-01        | Credit Card
| 100.00 |
|          10003 |           1 | Premium  | 2024-01        | Credit Card
| 100.00 |
+----------------+-------------+----------+----------------+----------------
+--------+
4 rows in set (0.00 sec)

mysql> -- Will not see newly inserted rows during the transaction.
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO invoice (customer_id, plan, service_period, payment_metho
d, total)
    -> VALUES (1, 'Premium', '2024-01', 'Credit Card', 100.00);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
```

Now, **Transaction A** will not see any new invoices that **Transaction B** inserts during its execution.

5. Backup and Recovery Plan

MySQL natively supports backup and recovery operations that enable database administrators to ensure data safety and quick recovery in case of failures. Tools and methods native to MySQL are listed below:

Native MySQL Backup Functionalities:

- **mysqldump:** A command-line utility that performs logical backups, which export database schemas and data into a portable format, such as SQL files.
- **MySQL Enterprise Backup:** This is an extended utility in MySQL Enterprise Edition for hot backups, which also include data files, logs, and configurations.
- **Binary Logs:** These allow for point-in-time recovery by logging all changes to the database since the last full backup.

Recovery Features:

- **Restoring from mysqldump Files**: It rebuilds the database from SQL scripts.
- **Point-in-Time Recovery:** This combines full backups with binary logs to restore the database at an exact moment.
- **Replication:** Native to MySQL, replication helps in maintaining a secondary copy of the database for failover and recovery.

Script to backup our media streaming database:

```
# Variables
BACKUP_DIR="/var/backups/media_streaming"
DB_USER="root"
DB_PASS="password"
DB_NAME="media_streaming"
DATE=$(date +"%Y%m%d_%H%M%S")
BACKUP_FILE="$BACKUP_DIR/${DB_NAME}_backup_$DATE.sql"

# Create backup directory if it does not exist
mkdir -p "$BACKUP_DIR"
```

```
# Perform the backup
mysqldump -u $DB_USER -p$DB_PASS $DB_NAME > "$BACKUP_FILE"

# Verify the backup
if [ $? -eq 0 ]; then
    echo "Backup successfully created: $BACKUP_FILE"
else
    echo "Backup failed!" >&2
    exit 1
fi
```

Script to recover our media streaming database:

```
# Variables
BACKUP_FILE="/var/backups/media_streaming/media_streaming_backup.sql"
DB_USER="root"
DB_PASS="password"
DB_NAME="media_streaming"

# Restore the database
mysql -u $DB_USER -p$DB_PASS $DB_NAME < "$BACKUP_FILE"

# Verify the restore
if [ $? -eq 0 ]; then
    echo "Database successfully restored from: $BACKUP_FILE"
else
    echo "Restore failed!" >&2
    exit 1
fi
```

With these built-in tools, MySQL makes it easy to create customized backup and recovery plans that cater to the specific needs of different applications and environments.