Table of Contents

1.Case Study Selection and 3NF/ER Modeling

For this data warehouse implementation, we have selected a Movie Rental System as the source OLTP (Online Transaction Processing) system. The Movie Rental System handles a significant volume of transactions, encompassing movie rentals, streaming services, and customer subscriptions. It processes data across various platforms while capturing customer behaviors, movie performance metrics, and rental trends. Due to the system's extensive transactional data and complex relationships between entities, it serves as an ideal candidate for a data warehouse implementation.

The OLTP system is designed to manage detailed datasets, including information about movie titles, rental history, customer profiles, payment transactions, and inventory levels. These data points are crucial for deriving actionable insights. The Movie Rental System's structure supports analytics, allowing for the integration of data into a data warehouse. This enables the extraction of meaningful insights that can drive business decisions related to sales analysis, customer behavior, inventory optimization, and overall performance tracking.

1.1 OLTP Business Process

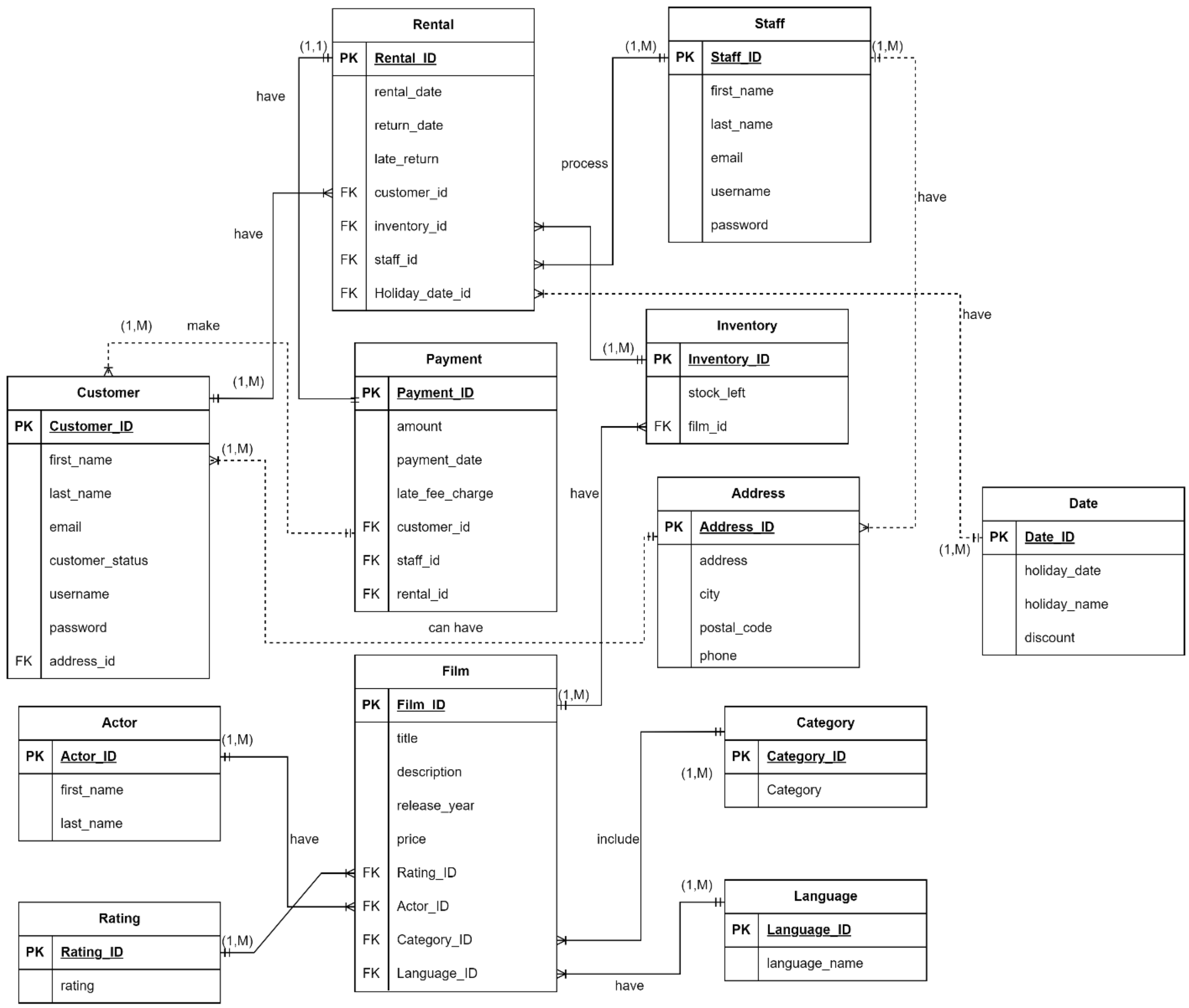
- **Address**
  - Stores customer address details, including city, postal code, and phone number.
  - Facilitates customer location management and shipping logistics.
- **Language**
  - Maintains data about languages for movies.
  - Used for filtering and categorizing films by language.
- **Rating**
  - Stores movie rating classifications.
  - Helps in parental control and content suitability for audiences.
- **Actor**
  - Tracks actors involved in movies, including their first and last names.
  - Enables detailed film metadata and search functionality.
- **Category**

- Manages movie categories.
- Supports browsing and filtering by genre or type.

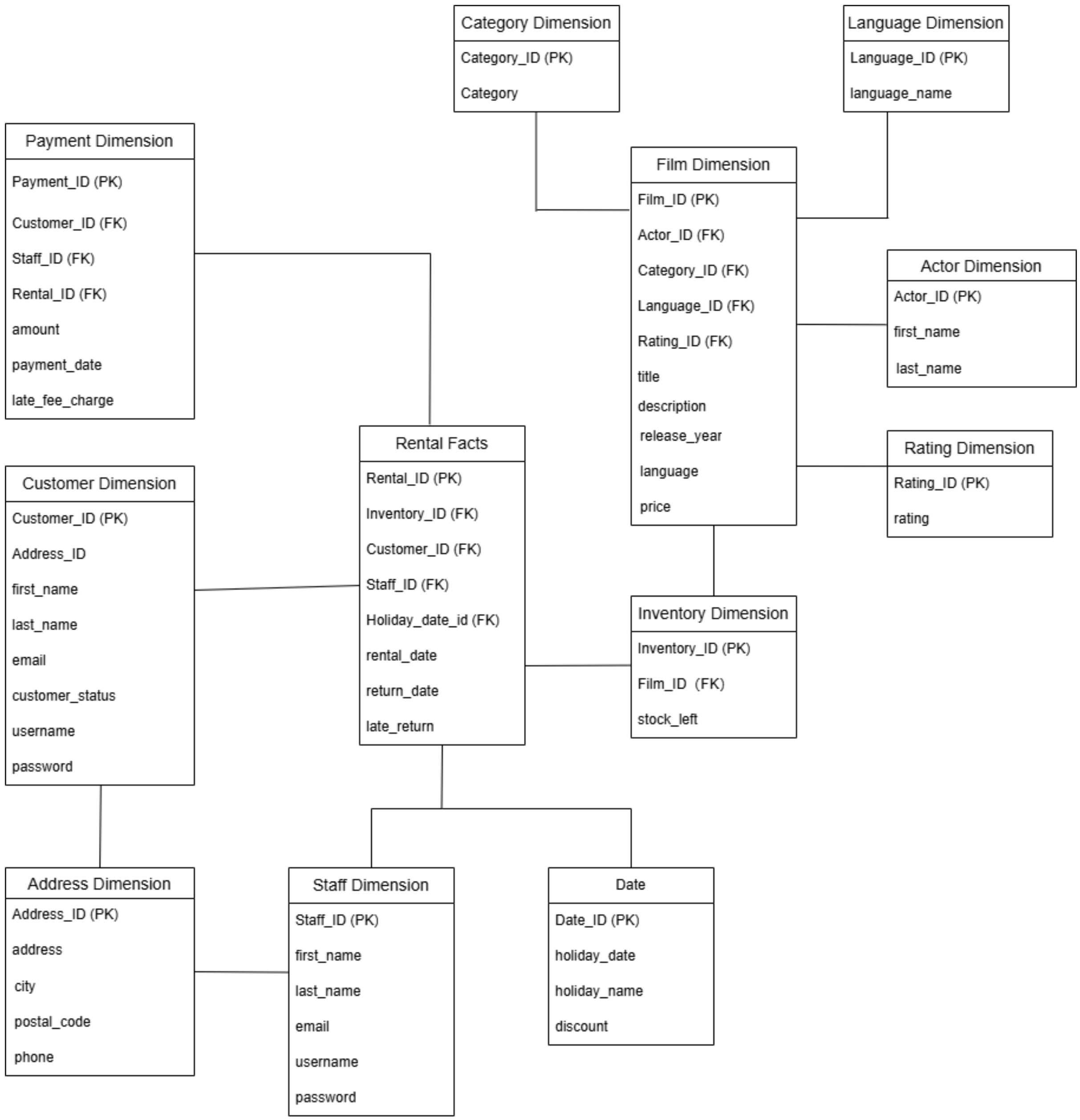
- **Film**
  - Core table for movie details, including title, description, release year, price, and associations with other entities.
  - Enables content discovery and metadata updates.
- **Inventory**
  - Tracks physical movie copies available in stock.
  - Includes stock levels and links to specific films for availability checks.
- **Customer**
  - Maintains customer profiles, including names, email, account status, and address.
  - Supports user account management and rental history tracking.
- **Staff**
  - Stores staff details such as names, email, and login credentials.
  - Supports internal operations and employee activity tracking.
- **Date**
  - Tracks holiday dates and special events.
  - Provides data for calculating discounts and holiday-specific rentals.
- **Rental**
  - Manages transactions where customers rent movies.
  - Tracks rental and return dates, inventory items rented, and any late return details.
- **Payment**
  - Records payment transactions, including amounts, dates, and customer information.
  - Manages late fee charges and links payments to rentals.

1.2 Entity-Relationship Diagram (ERD), Conceptual schema (choose star schema or snowflake schema)

ERD Diagram:



Snowflake schema:



2.Dimensional Model Design

2.1 Select the business process to build the dimensional model.

2.1.1 Selected Business Process: Rental Transactions

The process of Rental Transactions is selected for the dimensional model. As the business relates with the movie rental business, this process includes activities like renting of films by customers, inventory control, staff participation, and payment transactions processing. This is an important process in an organization operating in movie rental business and thus constructing a dimensional model best suits these processes. Based on the data collected from these transactions, better and reasoned decisions will be made using this model.

2.1.2 Justification for Selecting Rental Transactions

The model best depicts the concept of movie rental transactions as they are the key revenue transactions in the movie rental business. Following the practice of business revenue generation, these kinds of operations and strategies become very useful. Rental transactions encompass a lot of information including information about the customer, rental and return hours, amount paid and available stock of films. For this reason, adept serializing of such data provides a lot for the business as it broadens the conception of business organization and the main indicators of efficiency. For instance, it helps in developing a brand strategy, an evaluation of films sale, stock management, sales, and even labor productivity indicators. In addition, the source system is already reporting on such detail inclusive of customers, stores' rents, and payments, which are the basis for this dimensional model. In that case, focus is not lost on the issues of the relevance of the data the model is based on thereby making business model more practical.

2.2 Declare the grain of the fact table.

2.2.1 Grain of the Rental_Fact Table

The grain of the Rental_Fact table is defined as a single rental transaction, where each row represents one rental event. The level of detail captured includes:

- The customer who rented the films.
- The film(s) rented in the transaction.
- The rental and return dates.
- The staff member processing the rental.
- The inventory item(s) rented.
- Payment details and applied discounts.

This granular level of detail ensures comprehensive tracking of individual rental transactions.

In the Rental_Fact table, the grain designates the level of detail at which each row is populated and, in this case, corresponds to the single rental transaction. Each record in the table depicts the occurrence of a discrete event in the rental history: an action by a customer who rents a movie, an action by a worker renting out basic film inventory, and the worker action of processing the transaction. Each of these events is associated with vital attributes that include rentals and returns of the rented movies, the customer renting the films, a specific film that was rented out, and the inventory item. Moreover, the rental fact is associated with any payment made, that is inclusive of any pending charge for the delayed return of the rented movie. The fact table, the sales revenue notes, may also include any promotional sales during major holidays or events, for example a New Years Promotion.

2.2.2 Rental_Fact Table

Column Name	Description	Example
Rental_ID (PK)	Unique identifier for each rental transaction. This is the primary key of the table and helps differentiate individual rental events.	R0001
Customer_ID (FK)	A foreign key that links to the **Customer** table. It identifies the customer who rented the movie.	C0001
Inventory_ID (FK)	A foreign key that links to the **Inventory** table. It identifies the specific inventory item (e.g., a specific copy of a film) rented by the customer.	I0001
Staff_ID (FK)	A foreign key that links to the **Staff** table. It identifies the staff member who processed the rental transaction.	S0001
Rental_Date	The date and time when the rental occurred.	2024-06-01 10:00:00
Return_Date	The date and time when the movie is returned by the customer.	2024-06-05 09:00:00
Late_Return	An integer that tracks whether the customer returned the movie late (e.g., the number of days late). This helps calculate potential late fees.	5
Holiday_Date	A foreign key to the **Date** table, which may indicate if the rental occurred on a holiday or during a special promotion, such as a Christmas sale.	H001

```sql
CREATE TABLE Rental (
    Rental_ID VARCHAR(5) NOT NULL,
    Rental_Date TIMESTAMP NOT NULL,
    Return_Date TIMESTAMP NOT NULL,
    Customer_ID VARCHAR(5) NOT NULL,
    Inventory_ID VARCHAR(5) NOT NULL,
    Staff_ID VARCHAR(5) NOT NULL,
    Late_Return INT DEFAULT NULL,
    Holiday_Date VARCHAR(5) DEFAULT NULL,
    CONSTRAINT rental_pk PRIMARY KEY (Rental_ID),
    CONSTRAINT rental_fk_customer FOREIGN KEY (Customer_ID) REFERENCES Customer (Customer_ID),
    CONSTRAINT rental_fk_inventory FOREIGN KEY (Inventory_ID) REFERENCES Inventory (Inventory_ID),
    CONSTRAINT rental_fk_staff FOREIGN KEY (Staff_ID) REFERENCES Staff (Staff_ID),
    CONSTRAINT rental_fk_date FOREIGN KEY (Holiday_Date) REFERENCES Date (Date_ID)
);
```

2.3 Identify dimensions and attributes.

2.3.1 Dimension Identification with Relevant Attributes

1. Customer Dimension

   The Customer Dimension captures key information about the customers renting films. Each customer is uniquely identified by a Customer_ID, which links to personal details such as their name, email, and account status. This table also links to the Address table, enabling the company to know the customer's address.

Column Name	Description	Example
Customer_ID (PK)	Unique identifier for each customer.	C0001
First_Name	Customer's first name.	John
Last_name	Customer's last name.	Deo
Email	Customer's email address.	john.doe@example.com
Customer_status	Indicates whether the customer is active (TRUE) or inactive (FALSE).	TRUE
Username	Customer's username for logging into the system.	johndoe
Password	Customer's password for logging into the system.	password123
Address_ID (FK)	Foreign key linking to the Address table, representing the customer's address.	E0001

```sql
CREATE TABLE Customer (
    Customer_ID VARCHAR(5) NOT NULL,
    First_Name VARCHAR(50) NOT NULL,
    Last_Name VARCHAR(50) NOT NULL,
    Email VARCHAR(100),
    Customer_Status BOOLEAN DEFAULT TRUE,
    Username VARCHAR(50) NOT NULL UNIQUE,
    Password VARCHAR(255) NOT NULL,
    Address_ID VARCHAR(5) NOT NULL,
    CONSTRAINT customer_pk PRIMARY KEY (Customer_ID),
    CONSTRAINT customer_fk_address FOREIGN KEY (Address_ID) REFERENCES Address (Address_ID)
);
```

2. Staff Dimension

   The Staff Dimension contains information about the employees who process rental transactions. Staff members are identified by their Staff_ID.

Column Name	Description	Example
Staff_ID (PK)	Unique identifier for each staff member.	S0001
First_Name	Staff member's first name.	Ahmad
Last_Name	Staff member's last name.	Rahman
Email	Staff member's email address.	ahmad.rahman@example.com
Username	Staff member's unique username for accessing the system.	ahmadr
Password	Staff member's unique password for accessing the system.	password123

```sql
CREATE TABLE Staff (
    Staff_ID VARCHAR(5) NOT NULL,
    First_Name VARCHAR(50) NOT NULL,
    Last_Name VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE,
    Username VARCHAR(50) NOT NULL UNIQUE,
    Password VARCHAR(255) NOT NULL,
    CONSTRAINT staff_pk PRIMARY KEY (Staff_ID)
);
```

3. Inventory Dimension

   The Inventory Dimension represents the physical copies of films available for rent. Each Inventory_ID uniquely identifies a specific copy of a film.

Column Name	Description	Example
Inventory_ID (PK)	Unique identifier for each inventory item.	I0001
Film_ID (FK)	Foreign key linking to the **Film** table, representing the film in the inventory.	F0001
Stock_Left	Number of items remaining in stock for the film.	15

```sql
CREATE TABLE Inventory (
    Inventory_ID VARCHAR(5) NOT NULL,
    Stock_Left INT NOT NULL CHECK (Stock_Left >= 0),
    Film_ID VARCHAR(5) NOT NULL,
    CONSTRAINT inventory_pk PRIMARY KEY (Inventory_ID),
    CONSTRAINT inventory_fk_film FOREIGN KEY (Film_ID) REFERENCES Film(Film_ID)
);
```

4. Payment Dimension

The Payment Dimension captures payment details associated with each rental transaction. It includes the payment amount, date, customer, staff, and rental details, along with any late fee charges if applicable.

Column Name	Description	Example
Payment_ID (PK)	Unique identifier for each payment.	P0001
Payment_Date	The date the payment was made.	2024-01-01 10:00:00
Amount	Total payment amount.	4.99
Customer_ID (FK)	Foreign key linking to the customer dimension.	C0001
Staff_ID (FK)	Foreign key linking to the staff dimension.	S0001
Rental_ID (FK)	Foreign key linking to the rental fact table.	R0001
Late_Fee_Charge	Any extra charges incurred due to a late return.	6.00

```sql
CREATE TABLE Payment (
    Payment_ID SERIAL NOT NULL,
    Amount DECIMAL(10, 2) NOT NULL CHECK (Amount >= 0),
    Payment_Date TIMESTAMP NOT NULL,
    Customer_ID VARCHAR(5) NOT NULL,
    Staff_ID VARCHAR(5) NOT NULL,
    Rental_ID VARCHAR(5) NOT NULL,
    Late_Fee_Charge DECIMAL(10, 2) DEFAULT NULL,
    CONSTRAINT payment_pk PRIMARY KEY (Payment_ID),
    CONSTRAINT payment_fk_customer FOREIGN KEY (Customer_ID) REFERENCES Customer (Customer_ID),
    CONSTRAINT payment_fk_staff FOREIGN KEY (Staff_ID) REFERENCES Staff (Staff_ID),
    CONSTRAINT payment_fk_rental FOREIGN KEY (Rental_ID) REFERENCES Rental (Rental_ID)
);
```

5. Address Dimension

The Address Dimension stores the details of the customers' addresses. It links the customer's location data such as street address, city, postal code, and phone number.

Column Name	Description	Example
Address_ID (PK)	Unique identifier for each address.	E0001
Address	The physical address of the customer.	123 Elm Street
City	City name of the address.	Kuala Lumpur
Postal_Code	Postal code of the address.	50450
Phone	Customer's phone number.	03-1234-5678

```sql
CREATE TABLE Address (
    Address_ID VARCHAR(5) NOT NULL,
    Address VARCHAR(255) NOT NULL,
    City VARCHAR(100) NOT NULL,
    Postal_Code VARCHAR(20),
    Phone VARCHAR(20),
    CONSTRAINT address_pk PRIMARY KEY (Address_ID)
);
```

6. Film Dimension

The Film Dimension stores details about the films available for rent. This includes basic film information like the Film_ID, the Title, and its Release_Year. It also links to the Rating, Actor, Category, and Language dimensions to provide a richer dataset.

Column Name	Description	Example

Film_ID (PK)	Unique identifier for each film.	F0001
Title	The title of the film.	Inception
Description	A short description of the film.	A mind-bending thriller
Release_Year	The year the film was released.	2010
Price	The price of renting the film.	4.99
Rating_ID (FK)	Foreign key linking to the Rating table, which specifies the film's rating.	R001
Actor_ID (FK)	Foreign key linking to the Actor table, indicating the actor(s) in the film.	A0047
Category_ID (FK)	Foreign key linking to the Category table, specifying the genre/category of the film.	G001
Language_ID (FK)	Foreign key linking to the Language table, specifying the film's language.	L001

```
CREATE TABLE Film (
    Film_ID VARCHAR(5) NOT NULL,
    Title VARCHAR(255) NOT NULL,
    Description TEXT,
    Release_Year INT CHECK (Release_Year >= 1900 AND Release_Year <= EXTRACT(YEAR FROM CURRENT_DATE)),
    Price DECIMAL(10, 2) NOT NULL CHECK (Price >= 0),
    Rating_ID VARCHAR(5) NOT NULL,
    Actor_ID VARCHAR(5) NOT NULL,
    Category_ID VARCHAR(5) NOT NULL,
    Language_ID VARCHAR(5) NOT NULL,
    CONSTRAINT film_pk PRIMARY KEY (Film_ID),
    CONSTRAINT film_fk_rating FOREIGN KEY (Rating_ID) REFERENCES Rating(Rating_ID),
    CONSTRAINT film_fk_actor FOREIGN KEY (Actor_ID) REFERENCES Actor(Actor_ID),
    CONSTRAINT film_fk_category FOREIGN KEY (Category_ID) REFERENCES Category(Category_ID),
    CONSTRAINT film_fk_language FOREIGN KEY (Language_ID) REFERENCES Language(Language_ID)
);
```

7. Actor Dimension

The Actor Dimension stores information about the actors involved in films.

Column Name	Description	Example
Actor_ID (PK)	Unique identifier for each actor.	A0001
First_Name	Actor's first name.	Leonardo
Last_Name	Actor's last name.	DiCaprio

```
CREATE TABLE Actor (
    Actor_ID VARCHAR(5) NOT NULL,
    First_Name VARCHAR(50) NOT NULL,
    Last_Name VARCHAR(50) NOT NULL,
    CONSTRAINT actor_pk PRIMARY KEY (Actor_ID)
);
```

8. Category Dimension

The Category Dimension defines the genre or type of film available for rent.

Column Name	Description	Example
Category_ID (PK)	Unique identifier for each category.	G001
Category	The genre/category of the film (e.g., Action, Comedy, etc.).	Drama

```
CREATE TABLE Category (
    Category_ID VARCHAR(5) NOT NULL,
    Category VARCHAR(50) NOT NULL,
    CONSTRAINT category_pk PRIMARY KEY (Category_ID)
);
```

9. Language Dimension

The Language Dimension stores the languages in which films are available.

Column Name	Description	Example
Language_ID (PK)	Unique identifier for each language.	L003
Language	Name of the language (e.g., English, Malay, etc.).	English

```
CREATE TABLE Language (
    Language_ID VARCHAR(5) NOT NULL,
    Language_Name VARCHAR(50) NOT NULL,
    CONSTRAINT language_pk PRIMARY KEY (Language_ID)
);
```

10. Rating Dimension

The Rating Dimension stores film rating information, helping to analyze rental patterns based on film ratings.

Column Name	Description	Example
Rating_ID (PK)	Unique identifier for each rating.	R001
Rating	Rating description (e.g., G, PG-13, R).	G

```
CREATE TABLE Rating (
    Rating_ID VARCHAR(5) NOT NULL,
    Rating VARCHAR(20) NOT NULL,
    CONSTRAINT rating_pk PRIMARY KEY (Rating_ID)
);
```

11. Date Dimension

The Date Dimension is crucial for time-related analysis. It stores details about specific dates, particularly holidays, and any promotions linked to those holidays. This helps to understand how certain dates or promotions impact rental behavior.

Column Name	Description	Example
Date_ID(PK)	Unique identifier for the date.	H001
Holiday_Date	The date of the public holiday.	2024-01-01
Holiday_Name	Name of the holiday. (e.g., New Year's Day).	New Years Day
Discount	Any discount applied during a holiday or special event. Discount given in decimal.	0.10

```
CREATE TABLE Date (
    Date_ID VARCHAR(5) NOT NULL,
    holiday_date DATE,
    holiday_name VARCHAR(100),
    discount DECIMAL(5, 2),
    CONSTRAINT date_pk PRIMARY KEY (Date_ID)
);
```

2.4 Define facts

In this section, we define the numeric facts that will be measured and stored in the Rental_Fact table. These facts represent key metrics from rental transactions, which will be used to assess business performance and support data-driven decision-making. The grain of the fact table is defined as a single rental transaction, which includes details of the rented film, rental fees, any applicable late fees, and customer interactions.

2.4.1 Numeric Facts to be Measured

1. Total Amount
   The **Total Amount** is the sum of the rental amount and any additional fees, including late fees and discounts. This fact represents the total revenue generated from the rental transaction and is essential for calculating overall business performance.

   Total Amount =Rental Amount+Late Fee Charge−Discount Amount

2. Late Fee Charge
   The **Late Fee Charge** is charged when a customer returns a movie after the due date. It is calculated based on the number of days the rental is overdue. This fact is useful for monitoring late returns and generating additional revenue from penalties.

$$\text{Late Fee Charge} = \text{Late Days} \times \text{Late Fee per Day}$$

3. Rental Quantity
   Rental Quantity represents the number of films rented in a given transaction. This fact is generally 1, as most customers rent a single film per transaction. However, in cases of multiple rentals, this fact reflects the total number of films rented.

$$\text{Rental Quantity} = \text{Number of Films Rented in Transaction}$$

4. Rental Duration
   The **Rental Duration** represents the total time a customer keeps a rented item, measured in days. This fact is important for understanding customer rental patterns, such as average rental length and frequency of late returns. By analyzing this metric, the business can assess whether customers are typically adhering to the rental time frame or if they tend to keep items longer than expected.

$$\text{Rental Duration (days)} = \text{Return Date} - \text{Rental Date}$$

5. Discount Amount
   The **Discount Amount** represents any discount applied to the rental transaction. Discounts may be offered during promotions or for customer loyalty programs. Tracking this fact helps analyze the effectiveness of marketing strategies and promotions.

$$\text{Discount Amount} = \text{Percentage Discount} \times \text{Rental Amount}$$

Column Name	Description	Example	Calculation
Late_Fee_Charge	The late fee charged due to delayed return of the rented film.	10.00	Late Fee = 5 days × RM 2.00/day = RM 10.00
Rental_Quantity	The number of films rented in the transaction.	3	Rental Quantity = 3

Rental_Duration	The total time period the customer kept the rental, measured in days.	6 days	Rental Duration = Return Date (Jan 7) - Rental Date (Jan 1) = 6 days
Total_Amount	The total amount charged for the rental transaction, including rental fees, late fees, and discounts.	13.99	Total Amount = RM 5.99 (Rental) + RM 10.00 (Late Fee) - RM 2.00 (Discount) = RM 13.99
Discount_Amount	The amount of discount applied to the rental transaction.	0.60	Discount = 10% × RM 5.99 = RM 0.60

2.4.2 Surrogate Key For Fact Table Row

The **Rental_ID** serves as a **surrogate key**, uniquely identifying each rental transaction. It helps to distinguish each transaction and ensures that the facts (Total Amount, Late Fee Amount, etc.) are linked to the correct rental event.

2.4.3 Additivity of Facts

The facts in the **Rental_Fact** table are additive, which means they can be summed up across different dimensions (e.g., date, customer, staff) to produce meaningful summaries. For example:

- **Total Amount** can be summed across different time periods to track total revenue.
- **Late Fee Charge** can be summed to see how much money the business is earning from late fees.
- **Rental Quantity** can be summed to see how many films were rented over a specific period.

Such additivity is significant as it allows the collection of the data and higher-level analysis or estimation like in computing total sales or total revenue that came from late fees or total rental activities by a particular customer. The facts defined in the Rental_Fact table such as Total Amount, Late fee charge, Rental Quantity, Rental Period, and Discount Amount constitute the operational indicators needed to measure and manage the performance of the movie rental enterprise. This enables the business to understand the breadth of revenue, level of customer spending, trends in movie rentals and other insights based on the facts of a single rental transaction.

3.ETL Process Implementation

3.1 Define the data sources and outline the ETL workflow.

Data Sources:

Our data is referring to the Sakila Database, which contains data related to film rentals, inventory and payments. Other than that, we also added custom table Date and attributes such as late fee, discount, stock left to the schema to enhance the functionality of the data warehouse.

ETL workflow:

The Sakila database was built using MySQL, while our project uses openGauss, which relies on SQL. Therefore, we cannot import the Sakila database directly. In addition, since we have customized the tables and attributes to better suit our needs, it is not appropriate to import data directly from the Sakila database. Instead, we designed the data warehouse using the snowflake schema to structure the data into fact tables and dimension tables for efficiency and scalability.

To populate the schema, we created our own data values using the Sakila database as a reference. This ensured that the data met our needs and was fully compatible with our design. After determining the table attributes and data types, we merged all the SQL codes for creating the tables into the *Ass-schema.opengauss_corrected.sql* file. The data to be inserted into each table is organized in the *Ass-data.opengauss.sql* file. After all the code was prepared, both files were imported into a database named *ass* for further processing.

3.2 Include data cleaning, transformation rules, and data quality checks.

Data Quality Checks

Identify duplicate records in rows that should be unique.

SELECT Language_Name, **COUNT(*)**

FROM Language

GROUP BY Language_Name

HAVING COUNT(*) > 1;

```
 language_name | count
---------------+-------
 Russian       |     2
 Japanese      |     2
(2 rows)
```

SELECT Rating, **COUNT(*)**

FROM Rating

GROUP BY Rating

HAVING COUNT(*) > 1;

```
 rating | count
--------+-------
(0 rows)
```

SELECT CONCAT(first_name, ‘ ‘, last_name) **AS** actor_name,

COUNT(*) AS name_count

FROM Actor

GROUP BY first_name, last_name

HAVING COUNT(*) > 1;

```
 actor_name | name_count
------------+------------
(0 rows)
```

SELECT Category, **COUNT(*)**

FROM Category

GROUP BY Category

HAVING COUNT(*) > 1;

```
 category | count
----------+-------
(0 rows)
```

SELECT Title, **COUNT(*)**

FROM Film

GROUP BY Title

HAVING COUNT(*) > 1;

```
                 title                 | count
---------------------------------------+-------
 The Wolf of Wall Street               |     2
 Inception                             |     2
 Interstellar                          |     2
 Deadpool                              |     2
 The Great Gatsby                      |     2
 Finding Nemo                          |     2
 The Dark Knight                       |     2
 The Incredibles                       |     2
 The Hobbit: An Unexpected Journey     |     2
 The Avengers                          |     2
 Coco                                  |     2
 Frozen                                |     2
 The Dark Knight Rises                 |     2
 The Lion King                         |     2
 Shrek                                 |     2
 Mad Max: Fury Road                    |     2
 Zootopia                              |     2
 Avatar                                |     2
(18 rows)
```

Check Invalid Data

SELECT Price **FROM** Film

WHERE Price <= 0;

```
 price
-------
(0 rows)
```

Data Cleaning

Display all rows from the table that have duplicate records in rows that should be unique.

SELECT Language_ID, Language_Name

FROM (

SELECT Language_ID, Language_Name,

COUNT(*) OVER (**PARTITION BY** Language_Name) **AS** duplicate_count

FROM Language

) subquery

WHERE duplicate_count > 1;

```
 language_id | language_name
-------------+---------------
 L008        | Japanese
 L019        | Japanese
 L020        | Russian
 L007        | Russian
(4 rows)
```

SELECT Film_ID, Title

FROM (

SELECT Film_ID, Title, COUNT(*) **OVER** (**PARTITION BY** Title)

AS duplicate_count

FROM Film

) subquery

WHERE duplicate_count > 1

```
 film_id |              title
---------+----------------------------------
 F0020   | Avatar
 F0098   | Avatar
 F0065   | Coco
 F0050   | Coco
 F0023   | Deadpool
 F0096   | Deadpool
 F0030   | Finding Nemo
 F0088   | Finding Nemo
 F0083   | Frozen
 F0009   | Frozen
 F0001   | Inception
 F0100   | Inception
 F0015   | Interstellar
 F0078   | Interstellar
 F0055   | Mad Max: Fury Road
 F0036   | Mad Max: Fury Road
 F0087   | Shrek
 F0025   | Shrek
 F0082   | The Avengers
 F0024   | The Avengers
 F0003   | The Dark Knight
 F0099   | The Dark Knight
 F0081   | The Dark Knight Rises
 F0027   | The Dark Knight Rises
 F0037   | The Great Gatsby
 F0061   | The Great Gatsby
 F0091   | The Hobbit: An Unexpected Journey
 F0026   | The Hobbit: An Unexpected Journey
 F0089   | The Incredibles
 F0035   | The Incredibles
 F0012   | The Lion King
 F0080   | The Lion King
 F0033   | The Wolf of Wall Street
 F0052   | The Wolf of Wall Street
 F0031   | Zootopia
 F0063   | Zootopia
(36 rows)
```

Revise duplicate rows

UPDATE Language

SET Language_Name = 'French'

WHERE Language_ID = 'L019';

UPDATE Language

SET Language_Name = 'Italian'

WHERE Language_ID = 'L020';

UPDATE Film

SET Title = 'Persepolis', Language_ID = 'L019'

WHERE Film_ID = 'F0087';

UPDATE Film

SET Title = 'The Diving Bell and the Butterfly', Language_ID = 'L019'

WHERE Film_ID = 'F0055';

UPDATE Film

SET Title = 'Belle de Jour', Language_ID = 'L019'

WHERE Film_ID = 'F0078';

UPDATE Film

SET Title = 'The Artist', Language_ID = 'L019'

WHERE Film_ID = 'F0100';

UPDATE Film

SET Title = 'A Prophet', Language_ID = 'L019'

WHERE Film_ID = 'F0083';

UPDATE Film

SET Title = 'La Haine', Language_ID = 'L019'

WHERE Film_ID = 'F0088';

UPDATE Film

SET Title = 'Blue Is The Warmest Color', Language_ID = 'L019'

WHERE Film_ID = 'F0096';

UPDATE Film

SET Title = 'The Intouchables', Language_ID = 'L019'

WHERE Film_ID = 'F0050';

UPDATE Film

SET Title = 'Amelie', Language_ID = 'L019'

WHERE Film_ID = 'F0098';

UPDATE Film

SET Title = 'Life is Beautiful', Language_ID = 'L020'

WHERE Film_ID = 'F0082';

UPDATE Film

SET Title = 'Cinema Paradiso', Language_ID = 'L020'

WHERE Film_ID = 'F0099';

UPDATE Film

SET Title = 'The Great Beauty', Language_ID = 'L020'

WHERE Film_ID = 'F0081';

UPDATE Film

SET Title = 'The Bicycle Thief', Language_ID = 'L020'

WHERE Film_ID = 'F0061';

UPDATE Film

SET Title = 'Gomorrah', Language_ID = 'L020'

WHERE Film_ID = 'F0091';

UPDATE Film

SET Title = 'A Fistful of Dollars', Language_ID = 'L020'

WHERE Film_ID = 'F0089';

UPDATE Film

SET Title = 'The Postman', Language_ID = 'L020'

WHERE Film_ID = 'F0080';

UPDATE Film

SET Title = 'The Conformist', Language_ID = 'L020'

WHERE Film_ID = 'F0052';

UPDATE Film

SET Title = 'Mediterraneo', Language_ID = 'L020'

WHERE Film_ID = 'F0063';

```
ass=# UPDATE Language
ass-# SET Language_Name = 'French'
ass-# WHERE Language_ID = 'L019';
UPDATE 1
ass=# UPDATE Language
SET Language_Name = 'Italian'
WHERE Language_ID = 'L020';
UPDATE 1
```

```
ass=# UPDATE Film
SET Title = 'Persepolis', Language_ID = 'L019'
WHERE Film_ID = 'F0087';
UPDATE 1
ass=# UPDATE Film
SET Title = 'The Diving Bell and the Butterfly', Language_ID = 'L019'
WHERE Film_ID = 'F0055';
UPDATE 1
ass=# UPDATE Film
SET Title = 'Belle de Jour', Language_ID = 'L019'
WHERE Film_ID = 'F0078';
UPDATE 1
```

```
ass=# UPDATE Film
SET Title = 'A Prophet', Language_ID = 'L019'
WHERE Film_ID = 'F0083';
UPDATE 1
ass=# UPDATE Film
SET Title = 'La Haine', Language_ID = 'L019'
WHERE Film_ID = 'F0088';
UPDATE 1
ass=# UPDATE Film
SET Title = 'Blue Is The Warmest Color', Language_ID = 'L019'
WHERE Film_ID = 'F0096';
UPDATE 1
ass=# UPDATE Film
SET Title = 'The Intouchables', Language_ID = 'L019'
WHERE Film_ID = 'F0050';
UPDATE 1
ass=# UPDATE Film
SET Title = 'Amelie', Language_ID = 'L019'
WHERE Film_ID = 'F0098';
UPDATE 1
```

```
ass=# UPDATE Film
SET Title = 'The Artist', Language_ID = 'L019'
WHERE Film_ID = 'F0100';
UPDATE 1
```

```
ass=# UPDATE Film
SET Title = 'Life is Beautiful', Language_ID = 'L020'
WHERE Film_ID = 'F0082';
UPDATE 1
ass=# UPDATE Film
SET Title = 'Cinema Paradiso', Language_ID = 'L020'
WHERE Film_ID = 'F0099';
UPDATE 1
ass=# UPDATE Film
SET Title = 'The Great Beauty', Language_ID = 'L020'
WHERE Film_ID = 'F0081';
UPDATE 1
ass=# UPDATE Film
SET Title = 'The Bicycle Thief', Language_ID = 'L020'
WHERE Film_ID = 'F0061';
UPDATE 1
ass=# UPDATE Film
SET Title = 'Gomorrah', Language_ID = 'L020'
WHERE Film_ID = 'F0091';
UPDATE 1
ass=# UPDATE Film
SET Title = 'A Fistful of Dollars', Language_ID = 'L020'
WHERE Film_ID = 'F0089';
UPDATE 1
ass=# UPDATE Film
SET Title = 'The Postman', Language_ID = 'L020'
WHERE Film_ID = 'F0080';
UPDATE 1
ass=# UPDATE Film
SET Title = 'The Conformist', Language_ID = 'L020'
WHERE Film_ID = 'F0052';
UPDATE 1
ass=# UPDATE Film
SET Title = 'Mediterraneo', Language_ID = 'L020'
WHERE Film_ID = 'F0063';
UPDATE 1
```

No duplicate rows after update the table

```
ass=# SELECT Language_ID, Language_Name
FROM (
SELECT Language_ID, Language_Name, COUNT(*) OVER (PARTITION BY Language_Name) AS duplicate_count
FROM Language
) subquery
WHERE duplicate_count > 1;
 language_id | language_name
-------------+---------------
(0 rows)
```

```
ass=# SELECT Film_ID, Title
FROM (
SELECT Film_ID, Title, COUNT(*) OVER (PARTITION BY Title) AS duplicate_count
FROM Film
) subquery
WHERE duplicate_count > 1;
 film_id | title
---------+-------
(0 rows)
```

Transformation rules:

- Revise the rows in the Language table that have duplicate Language_name but keep the one with the lowest Language_ID.
- Modify the rows in the Film table that have duplicate Title, but keep the one with the lowest Film_ID.
- stock_left in Inventory table will decrease by 1 when 1 copy of the film has been rented out. When stock_left becomes less than 0, an exception that indicates that the film is currently out of stock will be raised.
- When a rental duration exceeds 7 days, the Late_Return column which is initially default as null will updated with the number of late days.
- On holidays, a discount is applied to the rental amount based on the holiday discount percentage in the Date table.
- Each rental is inserted into the Payment table with the calculated amount. Additional late fee is charged and recorded under Late_Fee_Charge.

3.3 Implement the ETL pipeline using openGauss to load data into the dimensional model.

The *scp* command copies the file *Ass-schema.opengauss_corrected.sql* and *Ass-data.opengauss.sql* from local machine to virtual environment, specifically to the */home/omm* directory of the server.

```
C:\Users\ling1>scp "C:\Users\ling1\Documents\Adv Database ass1\Ass-schema.opengauss_corrected.sql" root@192.168.1.40:/home/omm/

Authorized users only. All activities may be monitored and reported.
root@192.168.1.40's password:
Ass-schema.opengauss_corrected.sql                                                    100% 4504     2.2MB/s   00:00
```

```
C:\Users\ling1>scp "C:\Users\ling1\Documents\Adv Database ass1\Ass-data.opengauss.sql" root@192.168.1.40:/home/omm/

Authorized users only. All activities may be monitored and reported.
root@192.168.1.40's password:
Ass-data.opengauss.sql                     100%   80KB  13.0MB/s   00:00
```

After import sql files into virtual environment, logged into the remote server and switched to the omm user using *su – omm* command. Then, started the OpenGauss cluster by running *gs_om –t start* and accessed the OpenGauss database using *gsql –r –d postgres*. After that, create a database name *ass* for furthur operations.

```
C:\Users\ling1>ssh root@192.168.1.40

Authorized users only. All activities may be monitored and reported.
root@192.168.1.40's password:

Authorized users only. All activities may be monitored and reported.
Activate the web console with: systemctl enable --now cockpit.socket

Last login: Sun Dec 22 07:39:44 2024 from 192.168.1.10


Welcome to 5.10.0-191.0.0.104.oe2203sp3.x86_64

System information as of time:  Sun Dec 22 12:08:40 PM CST 2024

System load:    0.07
Processes:      113
Memory used:    17.3%
Swap used:      0%
Usage On:       40%
IP address:     192.168.1.40
IP address:     192.168.1.43
Users online:   2
```

```
[root@hjr ~]# su - omm
Last login: Sun Dec 22 07:39:51 CST 2024 on pts/0


Welcome to 5.10.0-191.0.0.104.oe2203sp3.x86_64

System information as of time:  Sun Dec 22 12:08:49 PM CST 2024

System load:    0.28
Processes:      115
Memory used:    17.4%
Swap used:      0%
Usage On:       40%
IP address:     192.168.1.40
IP address:     192.168.1.43
Users online:   2
To run a command as administrator(user "root"),use "sudo <command>".
[omm@hjr ~]$ gs_om -t start
Starting cluster.
=========================================
[SUCCESS] hjr:
[2024-12-22 12:09:06.853][155342][][gs_ctl]: gs_ctl started,datadir is /gaussdb/data/db1
[2024-12-22 12:09:06.860][155342][][gs_ctl]:  another server might be running; Please use the restart command
=========================================
Successfully started.
[omm@hjr ~]$ gsql -r -d postgres
gsql ((openGauss 5.0.1 build 33b035fd) compiled at 2023-12-15 19:51:49 commit 0 last mr  )
NOTICE : The password has been expired, please change the password.
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.
```

```
openGauss=# CREATE DATABASE ass;
CREATE DATABASE
openGauss=# \q
[omm@hjr ~]$ 
```

The command *gsql -d ass -f /home/omm/Ass-schema.opengauss_corrected.sql* connect to the ass database in OpenGauss and executes the SQL scripts located at /home/omm/Ass-schema. opengauss_corrected.sql to create tables in the database. Other than that, the command *gsql -d ass -f /home/omm/Ass-data.opengauss.sql* insert data of the tables into the database.

```
[omm@hjr ~]$ gsql -d ass -f /home/omm/Ass-schema.opengauss_corrected.sql
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:9: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "address_pk" for table "address"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:16: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "language_pk" for table "language"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:23: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "rating_pk" for table "rating"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:31: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "actor_pk" for table "actor"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:38: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "category_pk" for table "category"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:56: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "film_pk" for table "film"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:65: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "inventory_pk" for table "inventory"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:79: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "customer_pk" for table "customer"
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:79: NOTICE:  CREATE TABLE / UNIQUE will create implicit index "customer_username_key" for table "customer"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:90: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "staff_pk" for table "staff"
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:90: NOTICE:  CREATE TABLE / UNIQUE will create implicit index "staff_email_key" for table "staff"
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:90: NOTICE:  CREATE TABLE / UNIQUE will create implicit index "staff_username_key" for table "staff"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:98: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "date_pk" for table "date"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:115: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "rental_pk" for table "rental"
CREATE TABLE
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:130: NOTICE:  CREATE TABLE will create implicit sequence "payment_payment_id_seq" for serial column "payme
nt.payment_id"
gsql:/home/omm/Ass-schema.opengauss_corrected.sql:130: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "payment_pk" for table "payment"
CREATE TABLE
```

```
[omm@hjr ~]$ gsql -d ass -f /home/omm/Ass-data.opengauss.sql
INSERT 0 150
INSERT 0 20
INSERT 0 6
INSERT 0 107
INSERT 0 20
INSERT 0 110
INSERT 0 110
INSERT 0 150
INSERT 0 8
INSERT 0 310
INSERT 0 12
```

The ass database is now ready for future operations.

4.SQL Implementation

4.1 Stored Procedure

Address table

address_id	address	city	postal_code	phone

INSERT INTO Address (Address_ID, Address, City, Postal_Code, Phone) VALUES

('E0001', '123 Elm Street', 'Kuala Lumpur', '50450', '03-1234-5678');

address_id	address	city	postal_code	phone
E0001	123 Elm Street	Kuala Lumpur	50450	03-1234-5678
E0002	456 Oak Avenue	Penang	10450	04-9876-5432
E0003	789 Pine Road	Johor Bahru	80000	07-2345-6789
E0004	101 Maple Drive	Ipoh	31400	05-3456-7890
E0005	202 Birch Lane	Melaka	75000	06-4567-8901
E0006	303 Cedar Street	Kuala Lumpur	56100	03-5678-1234
E0007	404 Elm Boulevard	Penang	10300	04-6789-1234
E0008	505 Oak Way	Johor Bahru	80100	07-7890-1234
E0009	606 Pine Crescent	Ipoh	31300	05-8901-2345
E0010	707 Maple Path	Melaka	75100	06-9012-3456

Language Table

language_id	language_name

INSERT INTO Language (Language_ID, Language_Name) VALUES

('L001', 'Mandarin Chinese');

	language_id	language_name
1	L001	Mandarin Chinese
2	L002	Spanish
3	L003	English
4	L004	Hindi
5	L005	Bengali
6	L006	Portuguese
7	L007	Russian
8	L008	Japanese
9	L009	Malay
10	L010	Tamil

Rating table

rating_id	rating

INSERT INTO Rating (Rating_ID, Rating) VALUES

('R001', 'G');

rating_id	rating
R001	G
R002	P13
R003	18
R004	18SG
R005	18PL
R006	Censored

Actor table

actor_id	first_name	last_name

INSERT INTO Actor (Actor_ID, First_Name, Last_Name) VALUES

('A0001', 'Leonardo', 'DiCaprio');

	actor_id	first_name	last_name
1	A0001	Leonardo	DiCaprio
2	A0002	Meryl	Streep
3	A0003	Jackie	Chan
4	A0004	Donnie	Yen
5	A0005	Zhang	Ziyi
6	A0006	Rajinikanth	Shivaji Rao Gaikwad
7	A0007	Kamal	Haasan
8	A0008	Tony	Leung
9	A0009	Chow	Yun-fat
10	A0010	Diljit	Dosanjh

Category table

	category_id	category

INSERT INTO Category (Category_ID, Category) VALUES

('G001', 'Drama');

	category_id	category
1	G001	Drama
2	G002	Action
3	G003	Comedy
4	G004	Thriller
5	G005	Horror
6	G006	Romance
7	G007	Sci-Fi
8	G008	Fantasy
9	G009	Adventure
10	G010	Documentary

Film table

film_id	title	description	release_year	price	rating_id	actor_id	category_id	language_id

INSERT INTO Film (Film_ID, Title, Description, Release_Year, Price, Rating_ID, Actor_ID, Category_ID, Language_ID) VALUES

('F0001', 'Inception', 'A mind-bending thriller', 2010, 4.99, 'R001', 'A0047', 'G001', 'L001');

	film_id	title	description	release_year	price	rating_id	actor_id	category_id	language_id
1	F0001	Inception	A mind-bending thriller	2,010	4.99	R001	A0047	G001	L001
2	F0002	The Devil Wears Prada	A comedy-drama about fashion	2,006	3.99	R001	A0039	G001	L001
3	F0003	The Dark Knight	A superhero action film	2,008	5.99	R002	A0012	G002	L001
4	F0004	The Shawshank Redemption	A prison drama about hope and redemption	1,994	3.49	R001	A0045	G001	L003
5	F0005	Parasite	A dark comedy thriller about class struggle	2,019	4.49	R002	A0034	G001	L001
6	F0006	Avengers: Endgame	Superheroes assemble to stop a global threat	2,019	5.99	R002	A0021	G002	L003
7	F0007	Titanic	A tragic love story set on the doomed ship	1,997	4.99	R001	A0050	G006	L003
8	F0008	The Matrix	A hacker discovers a reality-bending secret	1,999	4.49	R001	A0013	G007	L003
9	F0009	Frozen	Two sisters struggle to control magical powers	2,013	3.99	R001	A0023	G011	L001
10	F0010	The Godfather	A mafia crime drama about family loyalty	1,972	5.49	R002	A0056	G012	L003

Inventory table

inventory_id	stock_left	film_id

INSERT INTO Inventory (Inventory_ID, Stock_Left, Film_ID) VALUES

('I0001', 115, 'F0001');

	inventory_id	stock_left	film_id
1	I0001	15	F0001
2	I0002	12	F0002
3	I0003	17	F0003
4	I0004	14	F0004
5	I0005	13	F0005
6	I0006	16	F0006
7	I0007	18	F0007
8	I0008	15	F0008
9	I0009	14	F0009
10	I0010	16	F0010

Customer table

customer_id	first_name	last_name	email	customer_status	username	password	address_id

INSERT INTO Customer (Customer_ID, First_Name, Last_Name, Email, Customer_Status, Username, Password, Address_ID) VALUES

('C0001', 'John', 'Doe', 'john.doe@example.com', TRUE, 'johndoe', 'password123', 'E0001');

	customer_id	first_name	last_name	email	customer_status	username	password	address_id
1	C0001	John	Doe	john.doe@exa	[v]	johndoe	password123	E0001
2	C0002	Jane	Smith	jane.smith@ex	[v]	janesmith	password456	E0002
3	C0003	Aminah	Abdullah	aminah.abdull	[v]	aminah12	password789	E0003
4	C0004	Mohd	Ali	mohd.ali@exa	[v]	mohdali45	password321	E0004
5	C0005	Siti	Nur	siti.nur@exam	[v]	sitinur	password654	E0005
6	C0006	Zahid	Rahman	zahid.rahman	[v]	zahidrahman	password234	E0006
7	C0007	Fatimah	Ismail	fatimah.ismail	[v]	fatimah85	password567	E0007
8	C0008	Adam	Tan	adam.tan@ex	[v]	adam-tan	password987	E0008
9	C0009	Isha	Mohamad	isha.mohamad	[v]	ishamohamad	password876	E0009
10	C0010	Ravi	Kumar	ravi.kumar@ex	[v]	ravikumar	password543	E0010

Staff table

staff_id	first_name	last_name	email	username	password

INSERT INTO Staff (Staff_ID, First_Name, Last_Name, Email, Username, Password) VALUES

('S0001', 'Ahmad', 'Rahman', 'ahmad.rahman@example.com', 'ahmadr', 'password123');

	staff_id	first_name	last_name	email	username	password
1	S0001	Ahmad	Rahman	ahmad.rahmar	ahmadr	password123
2	S0002	Aisha	Ismail	aisha.ismail@e	aishai	password456
3	S0003	Farid	Omar	farid.omar@ex	farido	password789
4	S0004	Nur	Zahra	nur.zahra@exa	nurzahra	password101
5	S0005	Hasan	Aziz	hasan.aziz@ex	hasana	password202
6	S0006	Maya	Hassan	maya.hassan@	mayah	password303
7	S0007	Zain	Rahman	zain.rahman@	zainr	password404
8	S0008	Siti	Mariam	siti.mariam@e	sitim	password505

Date table

	date_id	holiday_date	holiday_name	discount

INSERT INTO Date (Date_ID, holiday_date, holiday_name, discount) VALUES

('H001', '2024-01-01', 'New Years Day', 0.10);

	date_id	holiday_date	holiday_name	discount
1	H001	2024-01-01 00:00:00.000	New Years Day	0.1
2	H002	2024-02-10 00:00:00.000	Chinese New Year	0.3
3	H003	2024-02-11 00:00:00.000	Chinese New Year (Second Day)	0.3
4	H004	2024-05-01 00:00:00.000	Labour Day	0.15
5	H005	2024-05-19 00:00:00.000	Hari Raya Puasa	0.3
6	H006	2024-05-20 00:00:00.000	Hari Raya Puasa (Second Day)	0.3
7	H007	2024-05-23 00:00:00.000	Wesak Day	0.2
8	H008	2024-08-31 00:00:00.000	National Day	0.2
9	H009	2024-11-04 00:00:00.000	Deepavali	0.3
10	H010	2024-12-25 00:00:00.000	Christmas Day	0.15

Payment table

	payment_id	amount	payment_date	customer_id	staff_id	rental_id	late_fee_charge

	payment_id	amount	payment_date	customer_id	staff_id	rental_id	late_fee_charge
1	1	3.99	2020-06-01 10:00:00.000	C0012	S0002	R0001	[NULL]

Rental Table

	rental_id	rental_date	return_date	customer_id	inventory_id	late_return	staff_id	holiday_date

INSERT INTO Rental (Rental_ID, Rental_Date, Return_Date, Customer_ID, Inventory_ID, Staff_ID, Late_Return, Holiday_Date) VALUES

('R0001', '2024-01-01 10:00:00', '2024-01-05 09:00:00', 'C0012', 'I0087', 'S0002', NULL, NULL);

	rental_id	rental_date	return_date	customer_id	inventory_id	late_return	staff_id	holiday_date
1	R0001	2020-06-01 10:00:00.000	2020-06-05 09:00:00.000	C0012	I0087	[NULL]	S0002	[NULL]

4.2 Trigger for automation

4.2.1 Trigger function for Inventory Stock

```
CREATE OR REPLACE FUNCTION decrease_stock()
RETURNS TRIGGER AS $$
DECLARE
    current_stock INT;
    film_name TEXT;
BEGIN
    SELECT i.Stock_Left, f.Film_ID
     INTO current_stock, film_name
     FROM Inventory i
     JOIN Film f ON i.Film_ID = f.Film_ID
    WHERE i.Inventory_ID = NEW.Inventory_ID;

    IF current_stock < 1 THEN
       RAISE EXCEPTION '% is currently out of stock.', film_name;
    END IF;

    UPDATE Inventory
      SET Stock_Left = Stock_Left - 1
    WHERE Inventory_ID = NEW.Inventory_ID;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
---------------------Trigger for decrease_stock----------------------
```

CREATE TRIGGER trig_decrease_stock

BEFORE INSERT ON Rental

FOR EACH ROW

EXECUTE PROCEDURE decrease_stock();

When attempt to insert a row into Rental, the trigger function, trig_decrease_stock trigger fires. The decrease_stock() function will checks the related Inventory item. If Stock_Left is less than 1, it raises an exception and cancels the insert, displaying the custom error message. If Stock_Left is sufficient, it decrements Stock_Left by 1 and allows the insert to continue.

Before:

I0076	15	F0076
I0077	14	F0077
I0078	18	F0078
I0079	16	F0079
I0080	17	F0080
I0081	15	F0081
I0082	19	F0082
I0083	14	F0083
I0084	18	F0084
I0085	16	F0085
I0086	15	F0086
I0087	17	F0087
I0088	19	F0088
I0089	16	F0089
I0090	14	F0090
I0091	17	F0091
I0092	16	F0092
I0093	15	F0093
I0094	19	F0094
I0095	18	F0095
I0096	16	F0096
I0097	17	F0097
I0098	14	F0098
I0099	15	F0099
I0100	19	F0100

(100 rows)

After:

	rental_id	rental_date	return_date	customer_id	inventory_id	late_return	staff_id	holiday_date
1	R0001	2020-06-01 10:00:00.000	2020-06-05 09:00:00.000	C0012	I0087	[NULL]	S0002	[NULL]

	inventory_id	stock_left	film_id
92	I0093	15	F0093
93	I0094	19	F0094
94	I0095	18	F0095
95	I0096	16	F0096
96	I0097	17	F0097
97	I0098	14	F0098
98	I0099	15	F0099
99	I0100	19	F0100
100	I0087	16	F0087

The inventory id I0087 decrease by 1 from 17 to 16 when the rental is made on R0001.

4.2.2: Trigger for late return in Rental Table

```
CREATE OR REPLACE FUNCTION calc_late_return()
RETURNS TRIGGER AS $$
DECLARE
  diff_days INT;
BEGIN
  diff_days := EXTRACT(DAY FROM (NEW.Return_Date - NEW.Rental_Date));

  IF diff_days > 7 THEN
    NEW.Late_Return := diff_days;
  ELSE
    NEW.Late_Return := NULL;
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

CREATE TRIGGER trg_calc_late_return

BEFORE INSERT ON Rental

FOR EACH ROW

EXECUTE PROCEDURE calc_late_return();

When attempting to insert a row in rental, a BEFORE INSERT trigger on Rental will calculate the number of late days. If (Return_Date - Rental_Date) > 7 days, set late_return to the number of days late. Otherwise, late_return = NULL.

If no late return:

	rental_id	rental_date	return_date	customer_id	inventory_id	late_return	staff_id	holiday_date
14	R0011	2020-12-10 17:00:00.000	2020-12-14 16:00:00.000	C0029	I0035	[NULL]	S0004	[NULL]
15	R0012	2021-01-02 08:00:00.000	2021-01-06 07:30:00.000	C0042	I0021	[NULL]	S0006	[NULL]
16	R0013	2021-01-15 11:45:00.000	2021-01-19 10:45:00.000	C0037	I0099	[NULL]	S0002	[NULL]
17	R0014	2021-02-02 12:00:00.000	2021-02-06 11:30:00.000	C0018	I0084	[NULL]	S0003	[NULL]
18	R0015	2021-02-20 14:30:00.000	2021-02-24 13:30:00.000	C0070	I0056	[NULL]	S0005	[NULL]
19	R0016	2021-03-01 09:00:00.000	2021-03-05 08:30:00.000	C0040	I0057	[NULL]	S0007	[NULL]

If have late return:

	rental_id	rental_date	return_date	customer_id	inventory_id	late_return	staff_id	holiday_date
1	R0150	2024-05-01 11:00:00.000	2024-05-10 09:00:00.000	C0040	I0009	8	S0007	[NULL]
2	R0160	2024-08-09 11:00:00.000	2024-08-27 09:00:00.000	C0101	I0078	17	S0005	[NULL]
3	R0008	2020-10-15 14:15:00.000	2020-11-06 19:05:00.000	C0015	I0067	22	S0003	[NULL]

4.2.3: Trigger Function to Insert Payment After Rental Insert

CREATE OR REPLACE FUNCTION insert_payment_after_rental()

RETURNS **TRIGGER AS** $$

DECLARE

film_price **DECIMAL**(10,2);

holiday_id **VARCHAR**(50);

holiday_dt **DATE**;

holiday_disc **DECIMAL**(5,2) := 0;

applied_amount **DECIMAL**(10,2);

BEGIN

RAISE NOTICE 'Checking Inventory_ID: %, Rental_Date: %', **NEW**.Inventory_ID, TO_CHAR(**NEW**.Rental_Date, 'YYYY-MM-DD');

```
SELECT f.Price,
    d.Date_ID,
    d.holiday_date,
    COALESCE(d.discount, 0)
 INTO film_price, holiday_id, holiday_dt, holiday_disc
 FROM Inventory i
 JOIN Film f ON i.Film_ID = f.Film_ID
 LEFT JOIN Date d ON d.holiday_date = TO_CHAR(NEW.Rental_Date, 'YYYY-MM-DD')
 WHERE i.Inventory_ID = NEW.Inventory_ID;

IF NOT FOUND THEN
  film_price := NULL;
  holiday_id := NULL;
  holiday_dt := NULL;
  holiday_disc := 0;
END IF;

RAISE NOTICE 'Film Price: %, Holiday ID: %, Holiday Date: %, Discount: %',
     film_price, holiday_id, holiday_dt, holiday_disc;

IF holiday_dt IS NOT NULL THEN
  applied_amount := film_price * (1 - holiday_disc);
  RAISE NOTICE 'Discount applied: %, Applied Amount: %', holiday_disc,
applied_amount;
UPDATE Rental
```

```
SET holiday_date = holiday_id
WHERE rental_id = NEW.rental_id;

  ELSE
    applied_amount := film_price;
    RAISE NOTICE 'No discount applied, Applied Amount: %', applied_amount;
UPDATE Rental
SET holiday_date = holiday_id
WHERE rental_id = NEW.rental_id;
END IF;

  IF NEW.Late_Return IS NOT NULL THEN
    INSERT INTO Payment (Amount, Payment_Date, Customer_ID, Staff_ID, Rental_ID,
Late_Fee_Charge)
    VALUES (applied_amount, NEW.Rental_Date, NEW.Customer_ID, NEW.Staff_ID,
NEW.Rental_ID, NEW.Late_Return * 2);
  ELSE
    INSERT INTO Payment (Amount, Payment_Date, Customer_ID, Staff_ID, Rental_ID)
    VALUES (applied_amount, NEW.Rental_Date, NEW.Customer_ID, NEW.Staff_ID,
NEW.Rental_ID);
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

An AFTER INSERT trigger is triggered in Rental table, it will retrieve film price from film table as film_price, Date_ID, Holiday_Date and discount from Date table as holiday_id, holiday_dt and holiday_disc. The COALESCE(d.discount, 0) will set returns holiday_disc 0 if no discount was applied. For the JOIN, it joins inventory to film, where it fetches the film price using the Film_ID. The LEFT JOIN with date attempts to find a holiday that matches the Rental_Date. The TO_CHAR(NEW.Rental_Date, 'YYYY-MM-DD') converts the timestamp to a string in YYYY-MM-DD format to match the format in d.holiday_date. If the SELECT query returns no rows which means if there's no matching Inventory_ID, the variables are set to default values to prevent errors. When a holiday met, it will set NEW.Holiday_Date to the Date_ID of the holidayand then calculate applied_amount by applying the discount. If no holiday, it remains null. For the late_return_fee in payment table, if late_return is not null in Rental table, then the late fee charge will be calculated. It was set to RM2 per day, means the late return will be multiply by two and all the records will be inserted into the payment table. All the RAISE NOTICE used is for debugging purposes, so that all the data are correctly fetch.

If customers have late return:

Rental Table:

	rental_id	rental_date	return_date	customer_id	inventory_id	late_return	staff_id	holiday_date
1	R0150	2024-05-01 11:00:00.000	2024-05-10 09:00:00.000	C0040	I0009	8	S0007	[NULL]
2	R0160	2024-08-09 11:00:00.000	2024-08-27 09:00:00.000	C0101	I0078	17	S0005	[NULL]
3	R0008	2020-10-15 14:15:00.000	2020-11-06 19:05:00.000	C0015	I0067	22	S0003	[NULL]

Payment Table:

	payment_id	amount	payment_date	customer_id	staff_id	rental_id	late_fee_charge
1	35	3.39	2024-05-01 11:00:00.000	C0040	S0007	R0150	16
2	39	3.29	2024-08-09 11:00:00.000	C0101	S0005	R0160	34
3	40	5.49	2020-10-15 14:15:00.000	C0015	S0003	R0008	44

If Holiday Date match:

Rental Table:

	rental_id	rental_date	return_date	customer_id	inventory_id	late_return	staff_id	holiday_date
31	R0028	2021-09-01 10:00:00.000	2021-09-05 09:30:00.000	C0041	I0072	[NULL]	S0001	[NULL]
32	R0029	2021-09-16 14:30:00.000	2021-09-20 13:30:00.000	C0001	I0096	[NULL]	S0004	[NULL]
33	R0030	2021-10-05 16:30:00.000	2021-10-09 15:00:00.000	C0092	I0064	[NULL]	S0006	[NULL]
34	R0031	2021-10-20 12:00:00.000	2021-10-24 11:30:00.000	C0086	I0033	[NULL]	S0007	[NULL]
35	R0032	2021-11-01 09:00:00.000	2021-11-15 08:30:00.000	C0001	I0052	13	S0002	[NULL]
36	R0033	2024-11-04 14:15:00.000	2024-11-08 13:30:00.000	C0034	I0023	[NULL]	S0005	H009
37	R0034	2024-12-02 11:00:00.000	2024-12-06 10:00:00.000	C0047	I0076	[NULL]	S0004	[NULL]
38	R0035	2024-12-25 15:00:00.000	2024-12-28 14:30:00.000	C0078	I0016	[NULL]	S0003	H010

The Holiday_Date column in rental table will display the Date_ID in date table if the rental day is match with the holiday_date.

Payment Table:

	payment_id	amount	payment_date	customer_id	staff_id	rental_id	late_fee_charge
1	35	3.39	2024-05-01 11:00:00.000	C0040	S0007	R0150	16
2	39	3.29	2024-08-09 11:00:00.000	C0101	S0005	R0160	34
3	40	5.49	2020-10-15 14:15:00.000	C0015	S0003	R0008	44

The original film price for the rental id, R0160 is 5.49, but that day is a holiday, so a 40% discount was made, so the amount paid by customer id, C0101 was 3.29.

Raise Notice:

```
Checking Inventory_ID: I0078, Rental_Date: 2024-08-09
Film Price: 5.49, Holiday ID: H012, Holiday Date: 2024-08-09 00:00:00, Discount: .40
Discount applied: .40, Applied Amount: 3.29
```

4.3 User-Defined Function

CREATE OR REPLACE FUNCTION calculate_customer_lifetime_value(cust_id **VARCHAR**(5))

RETURNS DECIMAL(10,2) **AS** $$

DECLARE

clv **DECIMAL**(10,2);

BEGIN

SELECT COALESCE(**SUM**(Amount + **COALESCE**(Late_Fee_Charge,0)), 0)

INTO clv

FROM Payment

WHERE Customer_ID = cust_id;

RETURN clv;

END;

$$ **LANGUAGE** plpgsql;

This function will take Customer_Id as input, then aggregates the total amount the customer has paid from the Payment table by summing Amount and Late_Fee_Charge, the total values will return as a decimal(10,2) value.

After execute:

SELECT calculate_customer_lifetime_value('C0001') **AS** *customer_lifetime_value*;

Payment made by customer C0001:

	payment_id	amount	payment_date	customer_id	staff_id	rental_id	late_fee_charge
1	49	4.49	2021-03-18 16:00:00.000	C0001	S0008	R0017	[NULL]
2	56	5.49	2021-07-02 09:00:00.000	C0001	S0007	R0024	[NULL]
3	61	5.49	2021-09-16 14:30:00.000	C0001	S0004	R0029	[NULL]
4	64	5.49	2021-11-01 09:00:00.000	C0001	S0002	R0032	26

	customer_lifetime_value
1	46.96

4.4 Complex Query

SELECT

r.rental_id,

r.rental_date,

p.payment_date,

f.title **AS** *film_title*,

c.category **AS** *film_category*,

CASE

WHEN *p*.late_fee_charge **IS NOT NULL AND** *p*.late_fee_charge > 0

THEN 'Has Late Fee'

ELSE 'No Late Fee'

END AS *late_fee_status*,

p.amount + **COALESCE**(*p*.late_fee_charge,0) **AS** *total_payment*

FROM Rental *r*

JOIN Payment *p* **ON** *r*.rental_id = *p*.rental_id

JOIN Inventory *i* **ON** *r*.inventory_id = *i*.inventory_id

JOIN Film *f* **ON** *i*.film_id = *f*.film_id

JOIN Category *c* **ON** *f*.category_id = *c*.category_id

WHERE *r*.rental_date >= '2024-01-01'

AND *p*.payment_date <= CURRENT_TIMESTAMP

ORDER BY *p*.payment_date **DESC**;

This query had used Rental, Payment, Inventory, and Film table. The CASE expression is used to check whether it has a late fee or not, if yes, will label as Has Late Fee, else, label as No Late Fee. The filtering in this query is r.rental_date >= '2024-01-01' which means that only rental made after 2024-01-01 will be included and p.payment_date <= CURRENT_TIMESTAMP means payments made up to the current time will only be considered. The line ORDER BY p.payment_date DESC shows the most recent payments first.

	rental_id	rental_date	payment_date	film_title	film_category	late_fee_status	total_payment
1	R0160	2024-08-09 11:00:00.000	2024-08-09 11:00:00.000	Belle de Jour	Sci-Fi	Has Late Fee	37.29
2	R0150	2024-05-01 11:00:00.000	2024-05-01 11:00:00.000	Frozen	Animation	Has Late Fee	19.39
3	R0148	2024-05-01 10:00:00.000	2024-05-01 10:00:00.000	Inception	Drama	No Late Fee	4.99
4	R0036	2024-01-01 09:30:00.000	2024-01-01 09:30:00.000	Persepolis	Animation	No Late Fee	3.59

4.5 Group By and Advanced Grouping

SELECT

COALESCE(*c*.category, 'All Categories') **AS** *category*,

COALESCE(to_char(DATE_TRUNC('month', *p*.payment_date), 'YYYY-MM'), 'All Months') **AS** *month_aggregate*,

SUM(*p*.amount) **AS** *total_amount*

FROM Payment *p*

JOIN <u>Rental</u> *r* **ON** *p*.rental_id = *r*.rental_id

JOIN <u>Inventory</u> *i* **ON** *r*.inventory_id = *i*.inventory_id

JOIN <u>Film</u> *f* **ON** *i*.film_id = *f*.film_id

JOIN <u>Category</u> *c* **ON** *f*.category_id = *c*.category_id

GROUP BY CUBE (*c*.category, DATE_TRUNC('month', *p*.payment_date))

HAVING SUM(p.amount)

ORDER BY *c*.category, *month_aggregate*;

This query will display detailed payment amounts for each (Category, Month) pair, how much total payment was made, subtotal by category, payments were made across all months, subtotal by month, across all categories for a given month and grand total, the total amount across all categories and all months.

The line COALESCE(c.category, 'All Categories') AS category will show the category of the film. COALESCE() is used to display 'All Categories' where the category might be NULL.

Line COALESCE(to_char(DATE_TRUNC('month', p.payment_date), 'YYYY-MM'), 'All Months') AS month_aggregate will extract the month portion from the payment_date. COALESCE() used to show 'All Months' for subtotal rows generated by the CUBE that do not belong to a specific month.

GROUP BY CUBE (c.category, DATE_TRUNC('month', p.payment_date)) will generate the total amount for each category by each month, all months, the total amount for all categories within a single month and a grand total combining all categories and all months.

	category	month_aggregate	total_amount
1	Action	2020-05	5.49
2	Action	2020-10	5.49
3	Action	2020-11	4.99
4	Action	2021-01	5.49
5	Action	2021-03	4.99
6	Action	2021-09	5.49
7	Action	2021-10	5.49
8	Action	2021-11	5.49
9	Action	All Months	42.92
10	Animation	2020-06	3.99
11	Animation	2020-12	3.99
12	Animation	2021-02	4.99
13	Animation	2021-04	4.99
14	Animation	2021-06	4.49
15	Animation	2021-07	4.99
16	Animation	2021-10	3.99
17	Animation	2024-01	3.59
18	Animation	2024-05	3.39
19	Animation	All Months	38.41
58	All Categories	2024-08	3.29
59	All Categories	2024-12	4.67
60	All Categories	All Months	159.15

4.6 View

SQL View Creation for Rental Summary by Film Category:

```
CREATE VIEW Rental_Summary_By_Category AS
SELECT
    cat.Category AS Film_Category,
    COUNT(r.Rental_ID) AS Total_Rentals,
    SUM(p.Amount) AS Total_Rental_Income,
    SUM(COALESCE(p.Late_Fee_Charge, 0)) AS Total_Late_Fees
FROM
    Rental r
JOIN
    Inventory i ON r.Inventory_ID = i.Inventory_ID
JOIN
    Film f ON i.Film_ID = f.Film_ID
JOIN
    Category cat ON f.Category_ID = cat.Category_ID
JOIN
    Payment p ON r.Rental_ID = p.Rental_ID
GROUP BY
    cat.Category;
```

The Rental_Summary_By_Category view is an overall useful database view aimed at improving the understanding of rental data by establishing a metric classification at the level of film category. Hence, this view captures important items such as total rentals for given film category, total rental income, and total late fees for a category, which makes it useful for management reports and decision making processes.

The view pulls out data by making links between multiple tables such as Rental, Inventory, Film, Category, and Payment. These links help the view to associate a rentals to the films, categories and payment all at once. Total rentals for a category, total rental income and total late fees for each category are calculated by relevant aggregations like COUNT, SUM, COALESCE respectively, the COALESCE assuring that any null values in late fees are counted as zero. Film category wise grouping is performed with the aid of GROUP BY clause while query result column headings such as Film_Category, Total_Rentals, Total_Rental_Income and Total_Late_Fees make the output self-explanatory. This perspective has several advantages. First, it integrates critical data scattered in several tables by presenting an information estimate the required data for a given business purpose. It improves reporting because the performance of each of the film categories in terms of popularity and revenue. It guarantees the integrity of financial ratios because it handles nulls appropriately. These capabilities explain the importance of the view in the areas of inventory control, profitability and planning.

SELECT * **FROM** Rental_Summary_By_Category;

As a test for the view, users may run the command SELECT * FROM Rental_Summary_By_Category; which will return a table of results containing information about the film categories, total rentals, income from rentals and late fees.

```
ass=# SELECT * FROM Rental_Summary_By_Category;
 film_category | total_rentals | total_rental_income | total_late_fees
---------------+---------------+---------------------+-----------------
 War           |             2 |                9.98 |           52.00
 Romance       |             7 |               62.20 |          146.00
 Animation     |            63 |              285.37 |         1150.00
 Drama         |            19 |               97.84 |          344.00
 Comedy        |             8 |               65.60 |          178.00
 Fantasy       |            20 |              109.02 |          398.00
 Sci-Fi        |            53 |              297.04 |          994.00
 Action        |            99 |              513.51 |         2018.00
 Crime         |            21 |              112.79 |          382.00
 Documentary   |             4 |               19.96 |           78.00
 Musical       |            12 |               63.42 |          250.00
 Horror        |             2 |               16.98 |           42.00
(12 rows)
```

4.7 One Advanced SQL Features Not Covered in Class

WITH staff_summary **AS** (

SELECT

Staff_ID,

```
        COUNT(*) AS total_transactions,
        SUM(Amount) AS total_amount
    FROM Payment
    GROUP BY Staff_ID )

SELECT
    Staff_ID,
    total_transactions,
    total_amount,
    RANK() OVER (ORDER BY total_amount DESC) AS ranking
FROM staff_summary
ORDER BY ranking;
```

staff_id	total_transactions	total_amount	ranking
S0002	45	253.23	1
S0003	46	240.88	2
S0004	45	239.30	3
S0005	41	216.22	4
S0006	39	210.46	5
S0007	39	200.42	6
S0008	34	181.78	7
S0001	21	111.42	8

(8 rows)

The query begins by defining a Common Table Expression named staff_summary to calculates summary statistics for each staff member. It groups the data by Staff_ID to calculate total number of transactions each staff handled using COUNT(*) then insert into column total_transactions, and total_amount as the total amount collected by each staff were calculated using SUM(Amount) where the amount is referred to Amount column in Payment table. Then, the Staff_ID, total_transactions and total_amount are retrive from staff_summary and assign a rank to each staff based on their total_amount in descending order using RANK() window function with the staff that collected highest amount receiving the higset rank. Finally, the results are displayed in ascending order of rank that show the staff from the highest to the lowest total amount collected by the ranking column.

4.8 Compare row vs. column storage vs Memory-Optimized Tables (MOT) through hands-on queries on the same dataset and pre-loaded datasets.

Test Case	Row Storage	Memory-Optimized Tables (MOT)
Insert data	1196ms	418.30ms
Create user defined function	436.211ms	33.910ms
Creating complex query	504.432ms	3.144ms
Creating Group By and Advanced Grouping	696.789ms	3.772ms
Creating view	39.423ms	6.785ms
Advance query	102.983ms	6.391ms

Overall, Memory-Optimized Tables offer significant performance advantages over row storage, especially in Inserts and aggregations. The reason MOT performance is better than row storage is MOT is roughly 3 times faster for inserts, which aligns with the fact that writing to memory is significantly faster than writing to disk-based tables.

5.Performance Optimization

Task: Optimize the performance of your data warehouse by creating indexes and using partitioning where necessary

5.1 Index critical columns (e.g., foreign keys, frequently queried fields).

Identification of Critical Columns

We analyzed the database schema and identified critical columns based on:

1. Foreign Key Relationships (columns in fact and dimension tables):
   - Film table: Rating_ID, Actor_ID, Category_ID, Language_ID
   - Inventory table: Film_ID
   - Customer table: Address_ID
   - Rental table: Customer_ID, Inventory_ID, Staff_ID
   - Payment table: Customer_ID, Staff_ID, Rental_ID
2. Frequently Queried Fields (columns in WHERE clauses or JOIN condition):
   - Film table: Title, Release_Year (for search and filtering)
   - Customer table: First_Name, Last_Name (for customer lookups)
   - Rental table: Rental_Date, Return_Date (for date-based queries)
   - Payment table: Payment_Date, Amount (for financial reports)

Implementation Strategy

Based on the identified critical columns, we implemented 2 categories of indexes:

1. Foreign Key Indexes

CREATE INDEX idx_film_rating ON Film(Rating_ID);

CREATE INDEX idx_film_actor ON Film(Actor_ID);

CREATE INDEX idx_film_category ON Film(Category_ID);

CREATE INDEX idx_film_language ON Film(Language_ID);

CREATE INDEX idx_inventory_film ON Inventory(Film_ID);

CREATE INDEX idx_customer_address ON Customer(Address_ID);

CREATE INDEX idx_rental_customer ON Rental(Customer_ID);

CREATE INDEX idx_rental_inventory ON Rental(Inventory_ID);

CREATE INDEX idx_rental_staff ON Rental(Staff_ID);

CREATE INDEX idx_payment_customer ON Payment(Customer_ID);

CREATE INDEX idx_payment_staff ON Payment(Staff_ID);

CREATE INDEX idx_payment_rental ON Payment(Rental_ID);

2. Frequently Queried Fields Indexes

CREATE INDEX idx_film_title ON Film(Title);

CREATE INDEX idx_film_release_year ON Film(Release_Year);

CREATE INDEX idx_customer_name ON Customer(First_Name, Last_Name);

CREATE INDEX idx_rental_date ON Rental(Rental_Date);

CREATE INDEX idx_return_date ON Rental(Return_Date);

CREATE INDEX idx_payment_date ON Payment(Payment_Date);

CREATE INDEX idx_payment_amount ON Payment(Amount);

Performance Testing

We conducted performance testing using a complex query that joins multiple tables and include date filtering:

EXPLAIN ANALYZE

SELECT r.Rental_Date, f.Title, p.Amount

FROM Rental r

JOIN Inventory i ON r.Inventory_ID = i.Inventory_ID

JOIN Film f ON i.Film_ID = f.Film_ID

JOIN Payment p ON r.Rental_ID = p.Rental_ID

WHERE r.Customer_ID = 'C0001'

AND r.Rental_Date BETWEEN '2024-01-01' AND '2024-03-31';

Performance Results & Analysis

Before Optimization:

```
ass=# EXPLAIN ANALYZE
ass-# SELECT r.Rental_Date, f.Title, p.Amount
FROM Rental r
JOIN Inventory i ON r.Inventory_ID = i.Invass-# ass-# entory_ID
JOIN Film f ON i.Film_ID = f.Film_ID
JOIN Payment p ON r.Rental_ID = p.Rental_ID
WHEREass-# ass-# ass-#  r.Customer_ID = 'C0001'
AND r.Rental_Date BETWEEN '2024-01-01' AND '2024-03-31';ass-#
                                                                                                          QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------
 Hash Join  (cost=27.02..44.04 rows=3 width=540) (actual time=5.889..5.889 rows=0 loops=1)
   Hash Cond: ((p.rental_id)::text = (r.rental_id)::text)
   ->  Seq Scan on payment p  (cost=0.00..15.59 rows=559 width=40) (actual time=0.686..0.686 rows=1 loops=1)
   ->  Hash  (cost=27.01..27.01 rows=1 width=548) (actual time=0.864..0.864 rows=0 loops=1)
         Buckets: 32768  Batches: 1  Memory Usage: 256kB
         ->  Nested Loop  (cost=0.00..27.01 rows=1 width=548) (actual time=0.861..0.861 rows=0 loops=1)
               ->  Nested Loop  (cost=0.00..26.68 rows=1 width=56) (actual time=0.858..0.858 rows=0 loops=1)
                     ->  Seq Scan on rental r  (cost=0.00..18.40 rows=1 width=56) (actual time=0.853..0.853 rows=0 loops=1)
                           Filter: ((rental_date >= '2024-01-01 00:00:00'::timestamp without time zone) AND (rental_date <= '2024-03-31 00:00:00'::timestamp without t
ime zone) AND ((customer_id)::text = 'C0001'::text))
                           Rows Removed by Filter: 146
                     ->  Index Scan using inventory_pk on inventory i  (cost=0.00..8.27 rows=1 width=48) (Actual time: never executed)
                           Index Cond: ((inventory_id)::text = (r.inventory_id)::text)
               ->  Index Scan using film_pk on film f  (cost=0.00..0.32 rows=1 width=540) (Actual time: never executed)
                     Index Cond: ((film_id)::text = (i.film_id)::text)
 Total runtime: 11.479 ms
(15 rows)

ass=#
```

Execution Time: 11.479 ms

Query Plan Characteristics:

- Sequential scans on tables
- Less efficient join operations
- No index usage for filtering

After Optimization:

```
ass=# EXPLAIN ANALYZE
ass-# SELECT r.Rental_Date, f.Title, p.Amount
FROM Rental r
JOIN Inventory i ON r.Inventory_ID = i.Invass-# ass-# entory_ID
JOIN Film f ON i.Film_ID = f.Film_ID
JOIN Payment p ON r.Rental_ID = p.Rental_ID
WHEREass-# ass-# ass-#  r.Customer_ID = 'C0001'
AND r.Rental_Date BETWEEN '2024-01-01' AND '2024-03-31';ass-#
                                                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----
 Nested Loop  (cost=8.54..29.64 rows=3 width=540) (actual time=0.300..0.300 rows=0 loops=1)
   ->  Nested Loop  (cost=4.27..18.23 rows=1 width=548) (actual time=0.298..0.298 rows=0 loops=1)
         ->  Nested Loop  (cost=4.27..17.90 rows=1 width=56) (actual time=0.298..0.298 rows=0 loops=1)
               ->  Bitmap Heap Scan on rental r  (cost=4.27..9.62 rows=1 width=56) (actual time=0.294..0.294 rows=0 loops=1)
                     Recheck Cond: ((customer_id)::text = 'C0001'::text)
                     Filter: ((rental_date >= '2024-01-01 00:00:00'::timestamp without time zone) AND (rental_date <= '2024-03-31 00:00:00'::timestamp without time zo
ne))
                     ->  Bitmap Index Scan on idx_rental_customer  (cost=0.00..4.26 rows=2 width=0) (actual time=0.111..0.111 rows=0 loops=1)
                           Index Cond: ((customer_id)::text = 'C0001'::text)
               ->  Index Scan using inventory_pk on inventory i  (cost=0.00..8.27 rows=1 width=48) (Actual time: never executed)
                     Index Cond: ((inventory_id)::text = (r.inventory_id)::text)
         ->  Index Scan using film_pk on film f  (cost=0.00..0.32 rows=1 width=540) (Actual time: never executed)
               Index Cond: ((film_id)::text = (i.film_id)::text)
   ->  Bitmap Heap Scan on payment p  (cost=4.27..11.38 rows=3 width=40) (Actual time: never executed)
         Recheck Cond: ((rental_id)::text = (r.rental_id)::text)
         ->  Bitmap Index Scan on idx_payment_rental  (cost=0.00..4.27 rows=3 width=0) (Actual time: never executed)
               Index Cond: ((rental_id)::text = (r.rental_id)::text)
 Total runtime: 1.889 ms
(17 rows)

ass=#
```

Execution Time: 1.889 ms

Query Plan Characteristics:

- Efficient bitmap and index scans
- Optimized join operations using indexes
- Better handling of date range filters

Runtime Reduction: 11.479 - 1.889 = 9.59 ms

Percentage Improvement: (9.59/11.479) x 100 = 83.54%

The implementation of indexes on critical columns has yielded significant performance improvements in our database. Our testing revealed a substantial runtime reduction of 9.59 milliseconds, decreasing from 11.479 ms to 1.889 ms. This represents an impressive 83.5% improvement in query performance. Several key improvements were observed such as the query execution plan now utilized efficient index scans instead of sequential table scans, join operations are optimized through the use of indexes, and the filtering of customer and date ranges is handled more effectively. The overall query execution time has been reduced, demonstrating that our strategic placement of indexes on critical columns has successfully enhanced database performance, particularly for complex queries that involve multiple table joins and date-range filtering. These results confirm that our indexing strategy effectively addresses the performance optimization requirements for the data warehouse.

5.2 Implement partitioning (e.g., by date or region) to improve query performance.

Deliverable: Submit the indexing and partitioning strategy with before/after performance comparison.

Identification of Partitioning Strategy

This implementation adopts Range Partitioning to efficiently manage transactional data in the movie rental system. This strategy partitions the RENTAL and PAYMENT tables, which handle high transaction volumes, using their respective date fields (Rental_Date and Payment_Date) as partition keys. This decision was driven by the nature of the queries, which frequently filter data by date ranges. Range partitioning allows the database engine to scan only relevant partitions, reducing query times for such operations. The data is segmented into yearly partitions (2023 and 2024), enabling efficient data retrieval and management based on temporal access patterns.

Implementation Strategy

The implementation process began with analyzing the existing schema to identify RENTAL and PAYMENT tables as prime candidates for partitioning, based on their frequent date-based queries. The strategy involved recreating these tables using the PARTITION BY RANGE clause, with Rental_Date and Payment_Date serving as partition keys. Each table was structured with two distinct partitions: one handling pre-2024 data and another for 2024 data, ensuring efficient temporal data segregation. Primary key constraints were maintained through the CONSTRAINT clause, preserving data integrity while enabling partition-based data management.

1. Rental Table Partitioning

```
CREATE TABLE Rental_Partitioned (
   Rental_ID VARCHAR(5) NOT NULL,
   Rental_Date TIMESTAMP NOT NULL,
   Return_Date TIMESTAMP NOT NULL,
   Customer_ID VARCHAR(5) NOT NULL,
   Inventory_ID VARCHAR(5) NOT NULL,
   Staff_ID VARCHAR(5) NOT NULL,
   Late_Return INT DEFAULT NULL,
   Holiday_Date VARCHAR(5) DEFAULT NULL,
   CONSTRAINT rental_partitioned_pk PRIMARY KEY (Rental_ID), -- Renamed primary key
   CONSTRAINT rental_fk_customer FOREIGN KEY (Customer_ID) REFERENCES Customer (Customer_ID),
   CONSTRAINT rental_fk_inventory FOREIGN KEY (Inventory_ID) REFERENCES Inventory (Inventory_ID),
   CONSTRAINT rental_fk_staff FOREIGN KEY (Staff_ID) REFERENCES Staff (Staff_ID),
   CONSTRAINT rental_fk_date FOREIGN KEY (Holiday_Date) REFERENCES Date (Date_ID)
)
PARTITION BY RANGE (Rental_Date) (
   PARTITION rental_2023 VALUES LESS THAN ('2024-01-01'),
   PARTITION rental_2024 VALUES LESS THAN ('2025-01-01')
```

);

2. Payment Table Partitioning

```
CREATE TABLE Payment_Partitioned (
   Payment_ID SERIAL NOT NULL,
   Amount DECIMAL(10, 2) NOT NULL CHECK (Amount >= 0),
   Payment_Date TIMESTAMP NOT NULL,
   Customer_ID VARCHAR(5) NOT NULL,
   Staff_ID VARCHAR(5) NOT NULL,
   Rental_ID VARCHAR(5) NOT NULL,
   Late_Fee_Charge DECIMAL(10, 2) DEFAULT NULL,
   CONSTRAINT payment_partitioned_pk PRIMARY KEY (Payment_ID), -- Renamed primary key
   CONSTRAINT payment_fk_customer FOREIGN KEY (Customer_ID) REFERENCES Customer (Customer_ID),
   CONSTRAINT payment_fk_staff FOREIGN KEY (Staff_ID) REFERENCES Staff (Staff_ID),
   CONSTRAINT payment_fk_rental FOREIGN KEY (Rental_ID) REFERENCES Rental (Rental_ID)
)
PARTITION BY RANGE (Payment_Date) (
   PARTITION payment_2023 VALUES LESS THAN ('2024-01-01'),
   PARTITION payment_2024 VALUES LESS THAN ('2025-01-01')
);
```

Performance Testing

To evaluate the effectiveness of partitioning, performance tests were conducted using EXPLAIN ANALYZE on both the original and partitioned tables.:

For the before(original) table:

EXPLAIN ANALYZE SELECT * FROM Rental WHERE Rental_Date BETWEEN '2024-01-01' AND '2024-12-31';

EXPLAIN ANALYZE SELECT * FROM Payment WHERE Payment_Date BETWEEN '2024-01-01' AND '2024-12-31';

For the after(partitioned) table:

EXPLAIN ANALYZE SELECT * FROM Rental_Partitioned WHERE Rental_Date BETWEEN '2024-01-01' AND '2024-12-31';

EXPLAIN ANALYZE SELECT * FROM Payment_Partitioned WHERE Payment_Date BETWEEN '2024-01-01' AND '2024-12-31';

Performance Result & Analysis

Before Partitioning:

Rental:

```
ass=# EXPLAIN ANALYZE SELECT * FROM Rental WHERE Rental_Date BETWEEN '2024-01-01' AND '2024-12-31';
                                                                        QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------
-----------------------------
 Seq Scan on rental  (cost=0.00..7.65 rows=310 width=68) (actual time=0.043..0.569 rows=310 loops=1)
   Filter: ((rental_date >= '2024-01-01 00:00:00'::timestamp without time zone) AND (rental_date <= '2024-12-31 00:00:00'::t
imestamp without time zone))
 Total runtime: 0.767 ms
(3 rows)
```

Execution Time: 0.767 ms

Query Plan: Sequential Scan on Rental

Observation: A sequential scan is performed across the entire table to fetch the required rows.

Payment:

```
ass=# EXPLAIN ANALYZE SELECT * FROM Payment WHERE Payment_Date BETWEEN '2024-01-01' AND '2024-12-31';
                                                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------
-------------------------------
 Seq Scan on payment  (cost=0.00..7.65 rows=310 width=41) (actual time=0.045..0.565 rows=310 loops=1)
   Filter: ((payment_date >= '2024-01-01 00:00:00'::timestamp without time zone) AND (payment_date <= '2024-12-31 00:00:00':
:timestamp without time zone))
 Total runtime: 0.695 ms
(3 rows)
```

Execution Time: 0.695 ms

Query Plan: Sequential Scan on Payment

Observations: A sequential scan is performed across the entire table.

After Partitioning:

Rental_Partitionend:

```
ass=# EXPLAIN ANALYZE SELECT * FROM Rental_Partitioned WHERE Rental_Date BETWEEN '2024-01-01' AND '2024-12-31';
                                                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------
-----------------------------------
 Partition Iterator  (cost=0.00..12.20 rows=2 width=140) (actual time=0.015..0.015 rows=0 loops=1)
   Iterations: 1
   ->  Partitioned Seq Scan on rental_partitioned  (cost=0.00..12.20 rows=2 width=140) (actual time=0.002..0.002 rows=0 loop
s=1)
         Filter: ((rental_date >= '2024-01-01 00:00:00'::timestamp without time zone) AND (rental_date <= '2024-12-31 00:00:
00'::timestamp without time zone))
         Selected Partitions:  2
 Total runtime: 1.203 ms
(6 rows)
```

Execution Time: 1.203 ms

Query Plan:

- Partition Iterator with Partitioned Seq Scan across relevant partitions.
- Selected Partitions: 2

Observations: The query scans only the partitions that match the date range condition, but the overhead of partitioning logic slightly increases the execution time.

Payment_Partitioned:

```
ass=# EXPLAIN ANALYZE SELECT * FROM Payment_Partitioned WHERE Payment_Date BETWEEN '2024-01-01' AND '2024-12-31';
                                                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------
-------------------------------------
 Partition Iterator  (cost=0.00..13.38 rows=3 width=116) (actual time=0.018..0.018 rows=0 loops=1)
   Iterations: 1
   ->  Partitioned Seq Scan on payment_partitioned  (cost=0.00..13.38 rows=3 width=116) (actual time=0.001..0.001 rows=0 loo
ps=1)
         Filter: ((payment_date >= '2024-01-01 00:00:00'::timestamp without time zone) AND (payment_date <= '2024-12-31 00:0
0:00'::timestamp without time zone))
         Selected Partitions:  2
 Total runtime: 0.145 ms
(6 rows)
```

Execution Time: 0.145 ms

Query Plan:

- Partition Iterator with Partitioned Seq Scan across relevant partitions.
- Selected Partitions: 2

Observations: The query scans only the relevant partitions, resulting in significant improvement in execution time compared to the original table.

Table	Query Type	Time (ms)	Observations
Rental	Before/Original Table Query	0.767	Sequential scan across the entire table.
Rental_Partitioned	After/Partitioned Table Query	1.203	Partition pruning; slight overhead noted.
Payment	Before/ Original Table Query	0.695	Sequential scan across the entire table.
Payment_Partitioned	After/Partitioned Table Query	0.145	Partition pruning leads to faster results.

The implementation of range partitioning demonstrated mixed but promising results across the database system. The PAYMENT table showed remarkable improvement, with execution time reducing from 0.695 ms to 0.145 ms through efficient partition pruning. While the RENTAL table experienced a slight performance overhead, increasing from 0.767 ms to 1.203 ms due to partition iterator costs, the strategy's benefits are expected to become more pronounced as the dataset grows. The query planner successfully demonstrated partition pruning capabilities, scanning only relevant partitions during execution. Overall, this partitioning implementation establishes a robust foundation for scalable data management, particularly beneficial for handling larger datasets in the future.

6.Challenges Faced and Decision Made During the Project

The first challenge we faced was that the Sakila database was not completely suitable for our project. For example, the Sakila database did not calculate late return fees and discounts, and the date table that stores the dates of Malaysian holidays was not in the Sakila database. This prevented us from extracting data directly from the database, so we decided to create the data ourselves.

In addition to this, inserting data into the database was also a big challenge for us. Since our dataset was more than 1000 rows, it was difficult to enter each row directly in openGauss. We decided to put the code for creating the table and inserting the values into two sql files and then import it into the database. Finding a way to insert the sql files into the database also gave us a hard time. However, we have overcome it and successfully inserted all the data we needed into the database.

The third challenge we encountered was coordinating and managing our SQL files as a team. With over a thousand insert statements for movies, customers, and rentals, plus table creations and trigger definitions, we needed to ensure everyone was working with the latest version of the code. Sometimes changes made by one team member would affect code written by another, causing errors in our database setup. We solved this by carefully organizing our code into two main SQL files and making sure to communicate any changes to the team.