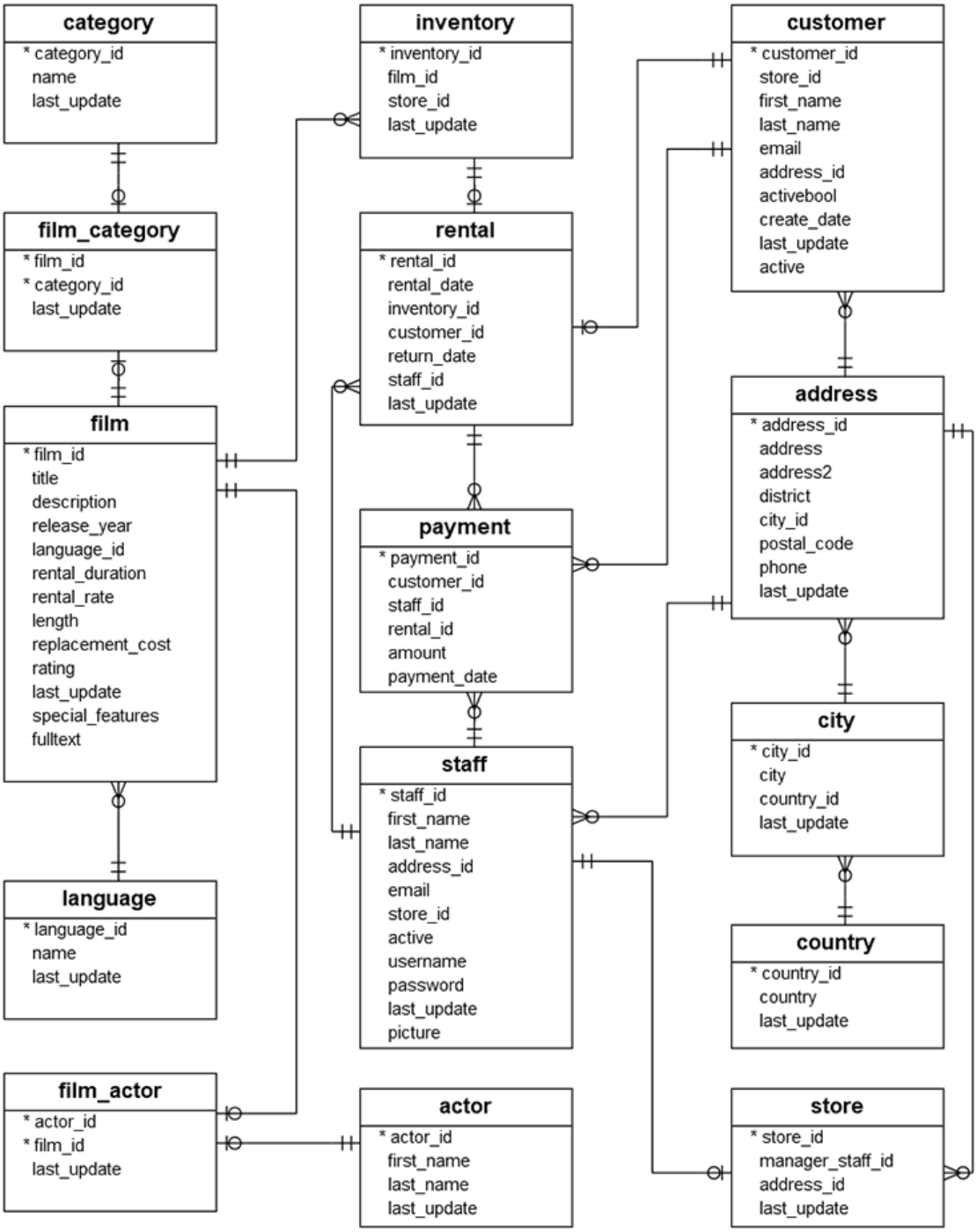
SQL and PySpark Task

Database

The DVD rental database represents the business processes of a DVD rental store. The DVD rental database has many objects.



DVD Rental ER Model

SQL and PySpark Task

There are 15 tables in the DVD Rental database:

actor – stores actor data including first name and last name.

film – stores film data such as title, release year, length, rating, etc.

film_actor – stores the relationships between films and actors.

category – stores film's categories data.

film_category- stores the relationships between films and categories.

store – contains the store data including manager staff and address.

inventory – stores inventory data.

Language – stores language data.

rental – stores rental data.

payment – stores customer's payments.

staff – stores staff data.

customer – stores customer data.

address – stores address data for staff and customers .

city – stores city names.

country – stores country names.

1. SQL Operations:

Write SQL queries to perform the following operations:

1) Select distinct values for key columns in each table.

2) Join relevant tables to create a consolidated view.

3) Calculate summary statistics for important columns.

4) Filter and sort data based on specific conditions.

SQL and PySpark Task

1) Select distinct values for key columns in each table.

-- 1. actor table

```
SELECT DISTINCT actor_id, first_name, last_name FROM actor;
```

-- 2. address table

```
SELECT DISTINCT address_id, address, district, city_id FROM address;
```

-- 3. category table

```
SELECT DISTINCT category_id, name FROM category;
```

-- 4. city table

```
SELECT DISTINCT city_id, city, country_id FROM city;
```

-- 5. country table

```
SELECT DISTINCT country_id, country FROM country;
```

-- 6. customer table

```
SELECT DISTINCT customer_id, first_name, last_name FROM customer;
```

-- 7. film table

```
SELECT DISTINCT film_id, title, release_year FROM film;
```

-- 8. film_actor table

```
SELECT DISTINCT film_id, actor_id FROM film_actor;
```

-- 9. film_category table

```
SELECT DISTINCT film_id, category_id FROM film_category;
```

-- 10. inventory table

```
SELECT DISTINCT inventory_id, film_id, store_id FROM inventory;
```

-- 11. language table

```
SELECT DISTINCT language_id, name FROM language;
```

-- 12. payment table

```
SELECT DISTINCT payment_id, customer_id FROM payment;
```

-- 13. rental table

```
SELECT DISTINCT rental_id, rental_date, inventory_id, customer_id FROM rental;
```

-- 14. staff table

```
SELECT DISTINCT staff_id, first_name, last_name FROM staff;
```

-- 15. store table

```
SELECT DISTINCT store_id, manager_staff_id, address_id FROM store;
```

Explanation:

The provided SQL query selects distinct values for key columns in each table of the DVD rental database. Each SELECT DISTINCT statement retrieves unique entries for the specified columns, ensuring no duplicate values are included in the result sets. This approach allows for a comprehensive understanding of the unique data present in each table, aiding in data exploration and analysis.

2) Join relevant tables to create a consolidated view.

-- 1. Consolidated view of customer table with rental table

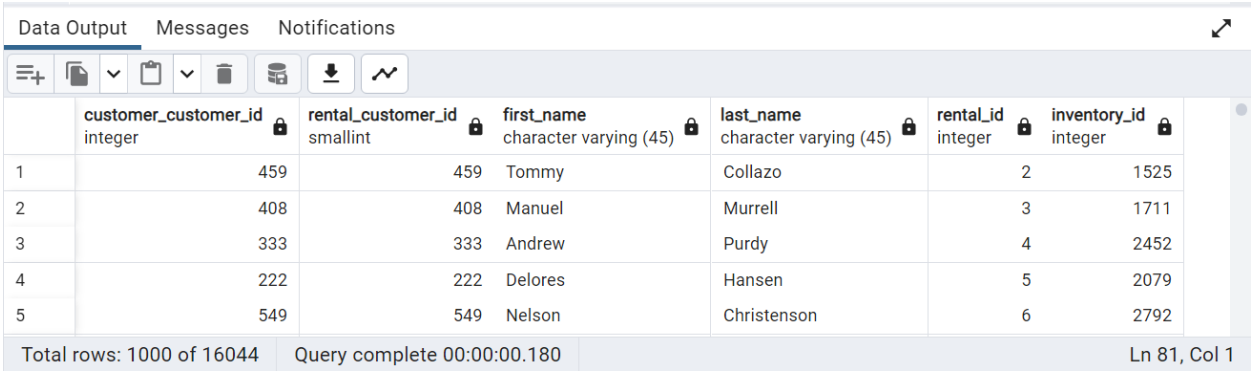
```
CREATE VIEW customer_rental_view AS

SELECT

c.customer_id AS customer_customer_id,

r.customer_id AS rental_customer_id,

c.first_name,

c.last_name,

r.rental_id,

r.inventory_id
```

```
FROM customer c

INNER JOIN rental r

ON c.customer_id = r.customer_id;

SELECT * FROM customer_rental_view;
```

Explanation:

customer_rental_view combines data from the customer and rental tables. It selects relevant columns from both tables and joins them using the customer_id key, providing insights into customer rental transactions including rental IDs and inventory IDs.

Output:



Data Output | Messages | Notifications

	customer_customer_id integer	rental_customer_id smallint	first_name character varying (45)	last_name character varying (45)	rental_id integer	inventory_id integer
1	459	459	Tommy	Collazo	2	1525
2	408	408	Manuel	Murrell	3	1711
3	333	333	Andrew	Purdy	4	2452
4	222	222	Delores	Hansen	5	2079
5	549	549	Nelson	Christenson	6	2792

Total rows: 1000 of 16044 | Query complete 00:00:00.180 | Ln 81, Col 1

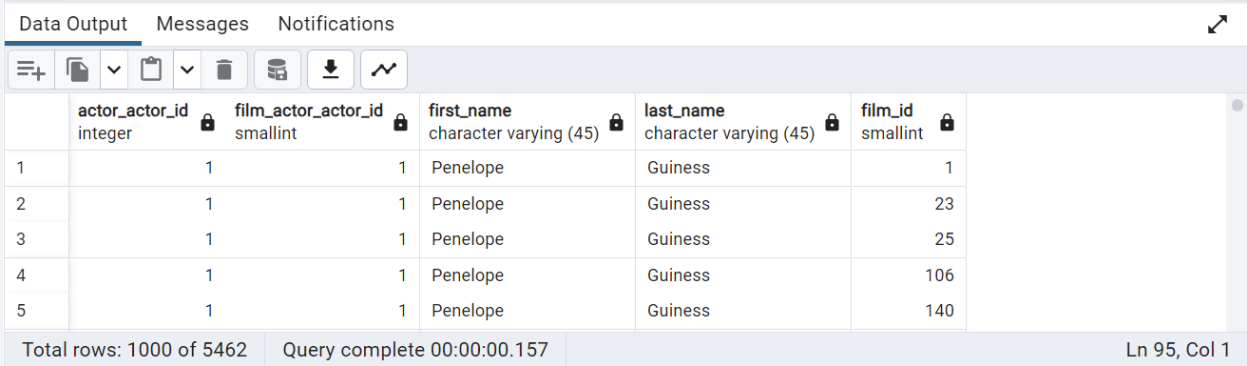
```
-- 2. Consolidated view of actor table with film_actor table

CREATE VIEW actor_flim_actor_view AS

SELECT

a.actor_id AS actor_actor_id,

fa.actor_id AS film_actor_actor_id,

a.first_name,

a.last_name,

fa.film_id

FROM actor a
```

```
LEFT JOIN film_actor fa

ON a.actor_id = fa.actor_id;

SELECT * FROM actor_flim_actor_view;
```

Explanation:

actor_film_actor_view merges data from the actor and film_actor tables. It combines actor and film actor information based on the actor_id key, allowing analysis of actor participation across various films, including their respective IDs and names.

Output:



Data Output | Messages | Notifications

	actor_actor_id integer	film_actor_actor_id smallint	first_name character varying (45)	last_name character varying (45)	film_id smallint
1	1	1	Penelope	Guiness	1
2	1	1	Penelope	Guiness	23
3	1	1	Penelope	Guiness	25
4	1	1	Penelope	Guiness	106
5	1	1	Penelope	Guiness	140

Total rows: 1000 of 5462 | Query complete 00:00:00.157 | Ln 95, Col 1

-- 3. Consolidated view of language table with film table

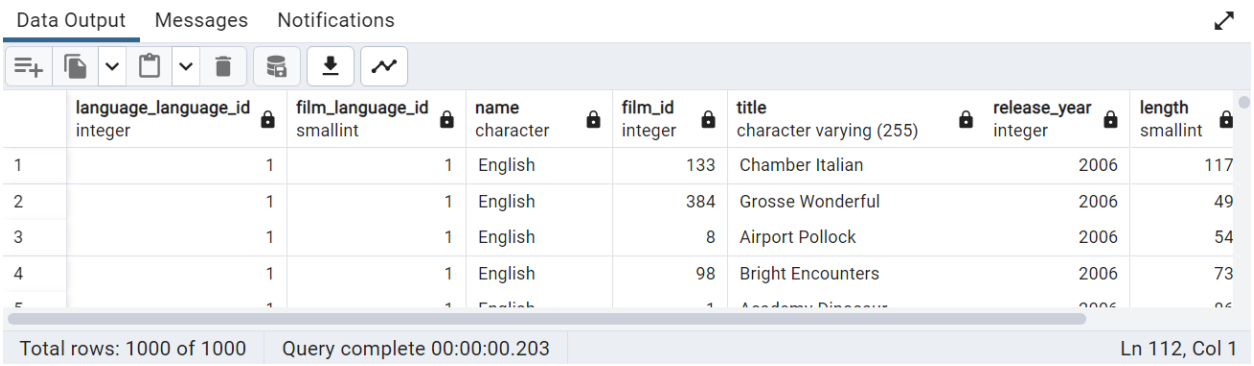
```
CREATE VIEW language_flim_view AS

SELECT

l.language_id AS language_language_id,

f.language_id AS film_language_id,

l.name,

f.film_id,

f.title,

f.release_year,

f.length,
```

```
f.rating

FROM language l

RIGHT JOIN film f

ON l.language_id = f.language_id;

SELECT * FROM language_flim_view;
```

Explanation:

language_film_view integrates data from the language and film tables. It merges language and film details using the language_id key, offering insights into films' language attributes such as title, release year, length, and rating, along with corresponding language details.

Output:



Data Output | Messages | Notifications

	language_language_id integer	film_language_id smallint	name character	film_id integer	title character varying (255)	release_year integer	length smallint
1	1	1	English	133	Chamber Italian	2006	117
2	1	1	English	384	Grosse Wonderful	2006	49
3	1	1	English	8	Airport Pollock	2006	54
4	1	1	English	98	Bright Encounters	2006	73

Total rows: 1000 of 1000 | Query complete 00:00:00.203 | Ln 112, Col 1

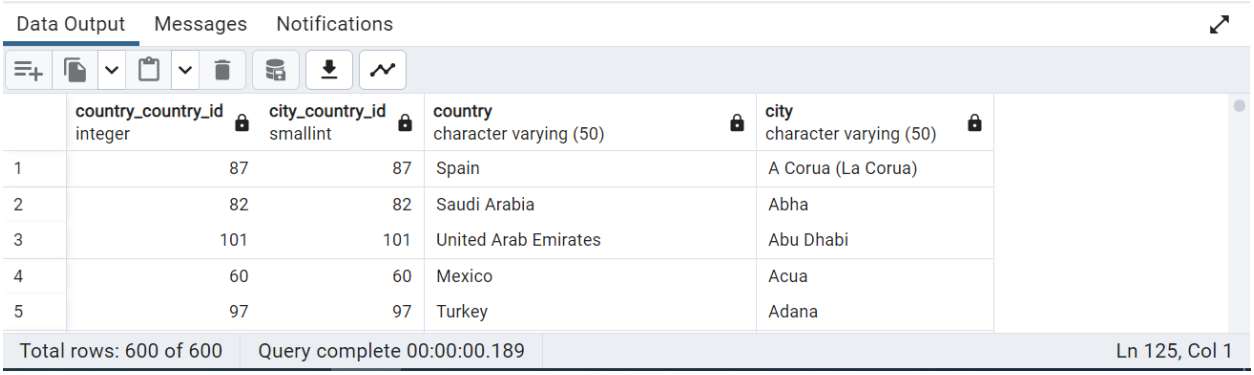
```
-- 4. Consolidated view of country table with city table

CREATE VIEW country_city_view AS

SELECT

co.country_id AS country_country_id,

ci.country_id AS city_country_id,

co.country,

ci.city

FROM country co
```

```
FULL OUTER JOIN city ci
ON co.country_id = ci.country_id;

SELECT * FROM country_city_view;
```

Explanation:

country_city_view consolidates data from the country and city tables. It combines country and city information based on the country_id key, presenting a comprehensive view of countries and their associated cities, including their respective IDs and names.

Output:



Data Output | Messages | Notifications

	country_country_id integer	city_country_id smallint	country character varying (50)	city character varying (50)
1	87	87	Spain	A Corua (La Corua)
2	82	82	Saudi Arabia	Abha
3	101	101	United Arab Emirates	Abu Dhabi
4	60	60	Mexico	Acua
5	97	97	Turkey	Adana

Total rows: 600 of 600 | Query complete 00:00:00.189 | Ln 125, Col 1

3) Calculate summary statistics for important columns.

```
-- 1. Calculate the total number of films in the database

SELECT COUNT(*) AS total_films

FROM film;
```

Explanation:

The query utilizes the COUNT(*) function to count all rows in the film table, which effectively counts the total number of films in the database.

Output:

-- 2. Calculate the average length of films

SELECT AVG(length) AS average_length FROM film;

Explanation:

It calculates the average length of films by using the AVG(length) function, which computes the average value of the length column in the film table.

Output:

-- 3. Determine the number of active and inactive customers

SELECT active, COUNT(customer_id) AS customer_count

FROM customer

GROUP BY active;

Explanation:

The query groups the customer table by the active column and counts the number of customers for each distinct value of 'active'.

Output:

-- 4. Determine the total number of films in each category

```
SELECT c.name AS category, COUNT(fc.film_id) AS film_count
FROM category c
JOIN film_category fc
ON c.category_id = fc.category_id
GROUP BY c.name
ORDER BY c.name;
```

Explanation:

- It joins the category and film_category tables on the category_id column to associate films with their respective categories.
- The query groups the data by category name and counts the occurrences of film IDs within each category, providing a count of films for each category.

Output:


Data Output | Messages | Notifications

	category character varying (25)	film_count bigint
1	Action	64
2	Animation	66
3	Children	60
4	Classics	57
5	Comedy	58

Total rows: 16 of 16 | Query complete 00:00:00.071 | Ln 142, Col 1

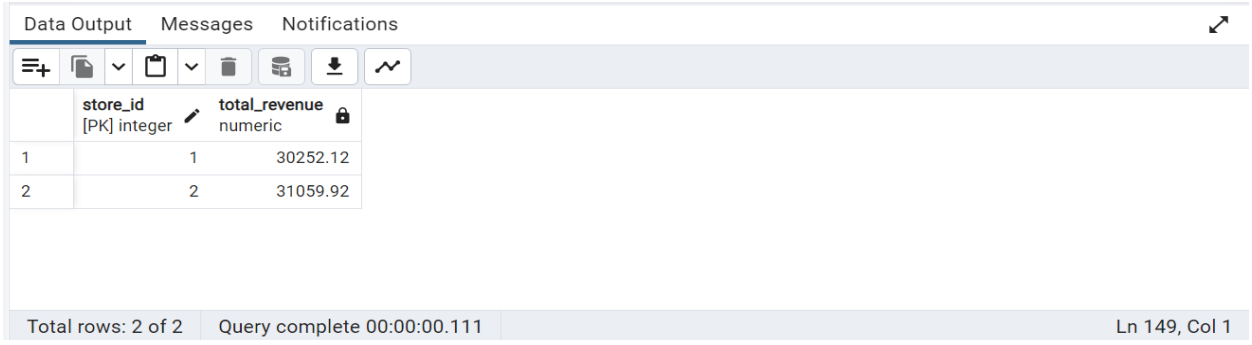
-- 5. Calculate the total revenue generated from each store

```
SELECT s.store_id, SUM(p.amount) AS total_revenue
FROM store s
JOIN staff st ON s.manager_staff_id = st.staff_id
JOIN payment p ON st.staff_id = p.staff_id
GROUP BY s.store_id;
```

Explanation:

- The query joins the store, staff, and payment tables to associate payments with the staff and stores involved.
- It aggregates payment amounts using the SUM() function and groups the data by store ID to compute the total revenue generated by each store.

Output:


Data Output | Messages | Notifications

	store_id [PK] integer	total_revenue numeric
1	1	30252.12
2	2	31059.92

Total rows: 2 of 2 | Query complete 00:00:00.111 | Ln 149, Col 1

4) Filter and sort data based on specific conditions.

-- 1. How many actors have 8 letters only in their first_names

```
SELECT COUNT(first_name) AS no_of_actors FROM actor

WHERE LENGTH(first_name) = 8;
```

Explanation:

- This query counts the number of actors whose first names contain exactly 8 letters.
- It utilizes the LENGTH() function to determine the length of the first_name column and filters rows where the length equals 8.

Output:

-- 2. Count the number of actors who's first_names don't start with an 'A'

```
SELECT COUNT(*) AS no_of_actors FROM actor

WHERE first_name NOT LIKE 'A%';
```

Explanation:

- The query counts the number of actors whose first names do not begin with the letter 'A'.
- It uses the NOT LIKE operator to filter out names that start with 'A'.

Output:

-- 3. Find actor names that start with 'P' followed by any letter from a to e then any other letter

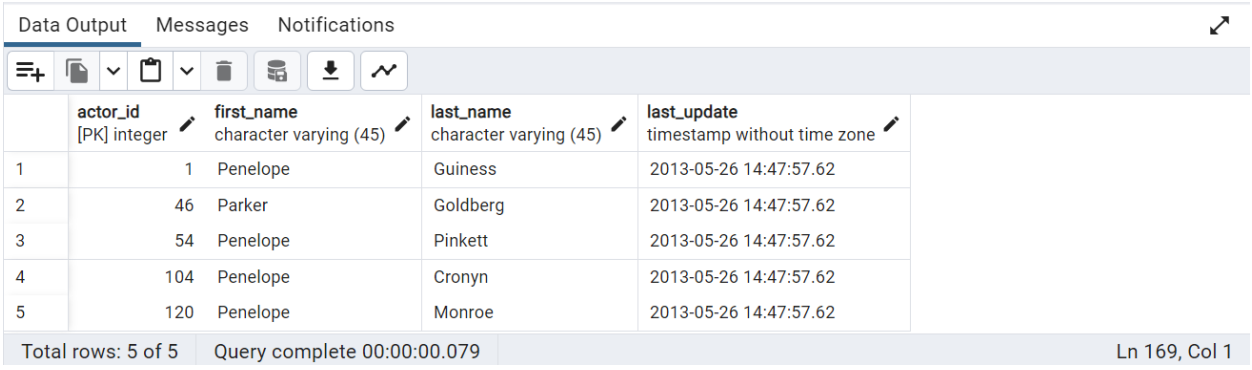
SELECT * FROM actor

WHERE first_name SIMILAR TO 'P[a-e]%';

Explanation:

- It retrieves actor names that start with the letter 'P' followed by any letter from 'a' to 'e' and then any other letter.
- The query employs the SIMILAR TO operator along with a pattern to match the specified criteria.

Output:



-- 4. Which movies have been rented so far

SELECT title FROM film

```
WHERE film_id IN (
    SELECT DISTINCT film_id FROM rental
    JOIN inventory
    on rental.inventory_id = inventory.inventory_id
);
```

Explanation:

- This query lists the titles of movies that have been rented by selecting film titles from the film table.
- It filters films based on whether their IDs appear in the rental records retrieved through a subquery.

Output:

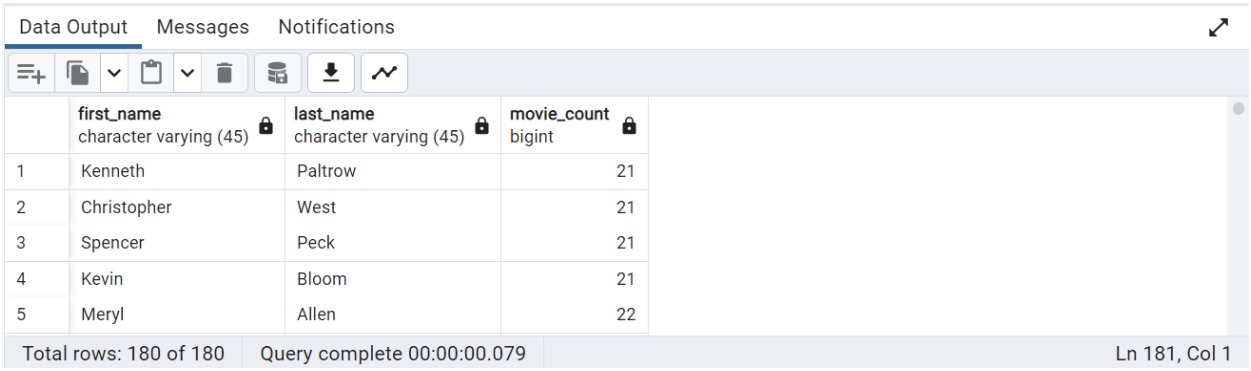

```
-- 5. Display the names of the actors that acted in more than 20 movies
SELECT first_name, last_name, COUNT(fa.film_id) AS movie_count
FROM film_actor fa
JOIN actor a
ON fa.actor_id = a.actor_id
GROUP BY first_name, last_name
HAVING COUNT(fa.film_id) > 20
```

ORDER BY movie_count;

Explanation:

- It fetches the first names, last names, and counts of films for actors who appeared in more than 20 movies.
- The query performs a join between the film_actor and actor tables, groups the data by actor names, and applies a HAVING clause to filter actors based on movie count.

Output:


Data Output | Messages | Notifications

	first_name character varying (45)	last_name character varying (45)	movie_count bigint
1	Kenneth	Paltrow	21
2	Christopher	West	21
3	Spencer	Peck	21
4	Kevin	Bloom	21
5	Meryl	Allen	22

Total rows: 180 of 180 | Query complete 00:00:00.079 | Ln 181, Col 1

2. PySpark Operations:

Write PySpark Code to perform the following operations:

1) Select distinct values for key columns in each table.

2) Join relevant tables to create a consolidated view.

3) Calculate summary statistics for important columns.

4) Filter and sort data based on specific conditions.

1) Select distinct values for key columns in each table.

 # 1. Actor table

actor_distinct = actor_df.select('actor_id', 'first_name', 'last_name').distinct()

actor_distinct.show()

```
# 2. Address table
address_distinct = address_df.select('address_id', 'address', 'district', 'city_id').distinct()
address_distinct.show()
# 3. Category table
category_distinct = category_df.select('category_id', 'name').distinct()
category_distinct.show()
# 4. City table
city_distinct = city_df.select('city_id', 'city', 'country_id').distinct()
city_distinct.show()
# 5. Country table
country_distinct = country_df.select('country_id', 'country').distinct()
country_distinct.show()
# 6. Customer table
customer_distinct = customer_df.select('customer_id', 'first_name', 'last_name').distinct()
customer_distinct.show()
# 7. Film table
film_distinct = film_df.select('film_id', 'title', 'release_year').distinct()
film_distinct.show()
# 8. Film Actor table
film_actor_distinct = film_actor_df.select('film_id', 'actor_id').distinct()
film_actor_distinct.show()
```

 # 9. Film Category table

```
film_category_distinct = film_category_df.select('film_id', 'category_id').distinct()

film_category_distinct.show()
```

 # 10. Inventory table

```
inventory_distinct = inventory_df.select('inventory_id', 'film_id', 'store_id').distinct()

inventory_distinct.show()
```

 # 11. Language table

```
language_distinct = language_df.select('language_id', 'name').distinct()

language_distinct.show()
```

 # 12. Payment table

```
payment_distinct = payment_df.select('payment_id', 'customer_id').distinct()

payment_distinct.show()
```

 # 13. Rental table

```
rental_distinct = rental_df.select('rental_id', 'rental_date', 'inventory_id', 'customer_id').distinct()

rental_distinct.show()
```

 # 14. Staff table

```
staff_distinct = staff_df.select('staff_id', 'first_name', 'last_name').distinct()

staff_distinct.show()
```

 # 15. Store table

```
store_distinct = store_df.select('store_id', 'manager_staff_id', 'address_id').distinct()

store_distinct.show()
```

Explanation:

The provided PySpark code iterates through various DataFrames representing tables within the DVD rental database, selecting distinct values for key columns in each table. By employing the distinct() method on each DataFrame, unique entries for specified columns are retrieved, ensuring no duplicates are included in the result sets. This systematic approach facilitates comprehensive data exploration and analysis, enabling users to gain insights into the unique data stored across different tables within the database.

2) Join relevant tables to create a consolidated view.

 # 1. Join the customer DataFrame with the rental DataFrame

```
customer_rental_join = customer_df.join(rental_df, customer_df.customer_id == rental_df.customer_id, 'inner')

# Select the columns for the new DataFrame

customer_rental_view = customer_rental_join.select(

    customer_df.customer_id.alias("customer_customer_id"),

    rental_df.customer_id.alias("rental_customer_id"),

    customer_df.first_name,

    customer_df.last_name,

    rental_df.rental_id,

    rental_df.inventory_id

)

customer_rental_view.show()
```

Explanation:

The code performs an inner join between the customer and rental DataFrames using the customer_id column as the key. It selects relevant columns from both DataFrames to create a

consolidated view (customer_rental_view) containing customer and rental information such as customer IDs, names, rental IDs, and inventory IDs.

Output:

```
+--------------------+------------------+----------+-----------+---------+------------+
|customer_customer_id|rental_customer_id|first_name|  last_name|rental_id|inventory_id|
+--------------------+------------------+----------+-----------+---------+------------+
|                 459|               459|     Tommy|    Collazo|        2|        1525|
|                 408|               408|    Manuel|    Murrell|        3|        1711|
|                 333|               333|    Andrew|      Purdy|        4|        2452|
|                 222|               222|   Delores|     Hansen|        5|        2079|
|                 549|               549|    Nelson|Christenson|        6|        2792|
```

 # 2. Join the actor DataFrame with the film_actor DataFrame

actor_film_actor_join = actor_df.join(film_actor_df, actor_df.actor_id == film_actor_df.actor_id, 'left')

Select the columns for the new DataFrame

actor_film_actor_view = actor_film_actor_join.select(

    actor_df.actor_id.alias("actor_actor_id"),

    film_actor_df.actor_id.alias("film_actor_actor_id"),

    actor_df.first_name,

    actor_df.last_name,

    film_actor_df.film_id

)

actor_film_actor_view.show()

Explanation:

This section executes a left join between the actor and film_actor DataFrames, linking entries based on the actor_id column. The resulting DataFrame (actor_film_actor_view) includes actor details along with corresponding film IDs, facilitating analysis of actor-film relationships.

Output:

```
+--------------+------------------+----------+---------+-------+
|actor_actor_id|film_actor_actor_id|first_name|last_name|film_id|
+--------------+------------------+----------+---------+-------+
|             1|                 1|  Penelope|  Guiness|    980|
|             1|                 1|  Penelope|  Guiness|    970|
|             1|                 1|  Penelope|  Guiness|    939|
|             1|                 1|  Penelope|  Guiness|    832|
|             1|                 1|  Penelope|  Guiness|    749|
```

 # 3. Join the language DataFrame with the film DataFrame

language_film_join = language_df.join(film_df, language_df.language_id == film_df.language_id, 'right')

Select the columns for the new DataFrame

language_film_view = language_film_join.select(

language_df.language_id.alias("language_language_id"),

film_df.language_id.alias("film_language_id"),

language_df.name,

film_df.film_id,

film_df.title,

film_df.release_year,

film_df.length,

film_df.rating

)

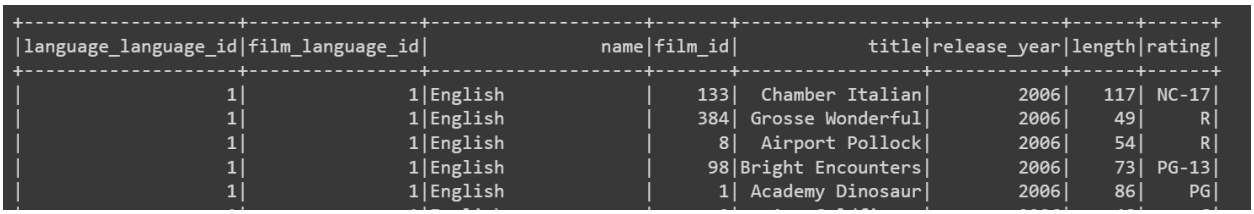
language_film_view.show()

Explanation:

The code performs a right join between the language and film DataFrames using the language_id column as the key. It selects pertinent columns to generate a consolidated view

(language_film_view) comprising language details and corresponding film attributes such as title, release year, length, and rating.

Output:



language_language_id	film_language_id	name	film_id	title	release_year	length	rating
1	1	English	133	Chamber Italian	2006	117	NC-17
1	1	English	384	Grosse Wonderful	2006	49	R
1	1	English	8	Airport Pollock	2006	54	R
1	1	English	98	Bright Encounters	2006	73	PG-13
1	1	English	1	Academy Dinosaur	2006	86	PG

 # 4. Join the country DataFrame with the city DataFrame

```
country_city_join = country_df.join(city_df, country_df.country_id == city_df.country_id, 'full_outer')

# Select the columns for the new DataFrame

country_city_view = country_city_join.select(

    country_df.country_id.alias("country_country_id"),

    city_df.country_id.alias("city_country_id"),

    country_df.country,

    city_df.city

)

country_city_view.show()
```

Explanation:

A full outer join is executed between the country and city DataFrames, utilizing the country_id column as the linking key. The resulting DataFrame (country_city_view) combines country and city information, offering insights into the geographical distribution of cities across different countries.

Output:

```
+------------------+---------------+-------------+--------------------+
|country_country_id|city_country_id|      country|                city|
+------------------+---------------+-------------+--------------------+
|                 1|              1|  Afghanistan|               Kabul|
|                 2|              2|      Algeria|               Batna|
|                 2|              2|      Algeria|               Bchar|
|                 2|              2|      Algeria|              Skikda|
|                 3|              3|American Samoa|              Tafuna|
```

3) Calculate summary statistics for important columns.

 # 1. Calculate the total number of films in the database

```
total_films = film_df.count()

print("Total number of films in the database:", total_films)
```

Explanation:

The code retrieves the total count of films in the database by invoking the count() function on the film DataFrame. This straightforward operation provides a fundamental insight into the scale of the film collection.

Output:

```
Total number of films in the database: 1000
```

 # 2. Calculate the average length of films

```
average_length = film_df.agg({'length': 'avg'}).collect()[0][0]

print("Average length of films:", average_length)
```

Explanation:

Utilizing the agg() function with the 'avg' aggregation, the code calculates the average length of films in the database. This metric offers a measure of the typical duration of movies available for rental.

Output:

```
Average length of films: 115.272
```

 # 3. Determine the number of active and inactive customers

```
customer_count = customer_df.groupBy('active').count()

customer_count.show()
```

Explanation:

Grouping the customer DataFrame by the 'active' column, the code determines the count of active and inactive customers. This breakdown provides visibility into the distribution of customer engagement with the rental service.

Output:

```
+------+-----+
|active|count|
+------+-----+
|     1|  584|
|     0|   15|
+------+-----+
```

 # 4. Determine the total number of films in each category

```
film_count_per_category = film_category_df.join(category_df, film_category_df.category_id == category_df.category_id) \
                          .groupBy(category_df.name.alias('category')) \
                          .agg(F.count('film_id').alias('film_count')) \
                          .orderBy('category')

film_count_per_category.show()
```

Explanation:

By joining the film_category DataFrame with the category DataFrame and aggregating the count of film IDs per category, the code generates a summary of the film count within each category. This breakdown aids in understanding the distribution of films across different genres.

Output:

```
+-----------+----------+
|   category|film_count|
+-----------+----------+
|     Action|        64|
|  Animation|        66|
|   Children|        60|
|   Classics|        57|
|     Comedy|        58|
```

 # 5. Calculate the total revenue generated from each store

```
total_revenue_per_store = store_df.join(staff_df, store_df.manager_staff_id == staff_df.staff_id) \
                          .join(payment_df, staff_df.staff_id == payment_df.staff_id) \
                          .groupBy(store_df["store_id"]) \
                          .agg({'amount': 'sum'}) \
                          .withColumnRenamed('sum(amount)', 'total_revenue')
total_revenue_per_store.show()
```

Explanation:

Through a series of joins between the store, staff, and payment DataFrames, followed by aggregation on the 'amount' column, the code computes the total revenue generated from each store. This analysis offers insights into the financial performance of individual rental outlets.

Output:

```
+--------+------------------+
|store_id|     total_revenue|
+--------+------------------+
|       1|30252.120000004612|
|       2|31059.920000004782|
+--------+------------------+
```

4) Filter and sort data based on specific conditions.

 # 1. How many actors have 8 letters only in their first names.

```
actors_with_8_letters = actor_df.filter(F.length("first_name") == 8).count()
print("Number of actors with 8 letters in their first names:", actors_with_8_letters)
```

Explanation:

Utilizing the filter() method with the length() function from the pyspark.sql.functions module, the code identifies actors whose first names consist of precisely 8 letters. The count of such actors provides a succinct overview of this particular subset.

Output:

```
Number of actors with 8 letters in their first names: 16
```

 # 2. Count the number of actors whose first_names don't start with an 'A'.

actors_without_A = actor_df.filter(~F.col("first_name").startswith("A")).count()

print("Number of actors whose first names don't start with 'A':", actors_without_A)

Explanation:

Leveraging the filter() method with the negation operator ~ and startswith() function, the code counts actors whose first names do not commence with the letter 'A'. This criterion aids in understanding the distribution of actor names across different alphabets.

Output:

```
Number of actors whose first names don't start with 'A': 187
```

 # 3. Find actor names that start with 'P' followed by any letter from 'a' to 'e' then any other letter.

pattern = "^P[a-e].*"

actors_matching_pattern = actor_df.filter(F.col("first_name").rlike(pattern)).show()

Explanation:

Employing the rlike() method, the code extracts actor names starting with 'P', followed by any letter from 'a' to 'e', and then any other letter. Regular expressions facilitate flexible pattern matching, offering insights into specific name patterns within the dataset.

Output:

```
+--------+----------+---------+-----------+
|actor_id|first_name|last_name|last_update|
+--------+----------+---------+-----------+
|       1|  Penelope|  Guiness|    47:57.6|
|      46|    Parker| Goldberg|    47:57.6|
|      54|  Penelope|  Pinkett|    47:57.6|
|     104|  Penelope|   Cronyn|    47:57.6|
|     120|  Penelope|   Monroe|    47:57.6|
+--------+----------+---------+-----------+
```

 # 4. Which movies have been rented so far.

joined_df = rental_df.join(inventory_df, rental_df.inventory_id == inventory_df.inventory_id)

distinct_film_ids = joined_df.select("film_id").distinct()

movies_rented_so_far = film_df.join(distinct_film_ids, film_df.film_id == distinct_film_ids.film_id).select("title")

movies_rented_so_far.show()

Explanation:

By joining the rental and inventory DataFrames and selecting distinct film IDs, the code identifies the movies that have been rented. This analysis provides visibility into the popularity and utilization of different films within the rental service.

Output:

```
+------------------+
|             title|
+------------------+
|    Island Exorcist|
|      Kick Savannah|
|   Instinct Airport|
|    Splendor Patton|
|      Submarine Bed|
```

 # 5. Display the names of the actors that acted in more than 20 movies.

actors_more_than_20_movies = (film_actor_df

.join(actor_df, film_actor_df.actor_id == actor_df.actor_id)

.groupBy("first_name", "last_name")

```
    .agg(F.count("film_id").alias("movie_count"))
    .filter("movie_count > 20")
    .orderBy(F.asc("movie_count"))
)
actors_more_than_20_movies.show()
```

Explanation:

Through a series of joins, groupings, and aggregations, the code determines actors who have appeared in more than 20 movies. Sorting the results by movie count in ascending order offers clarity on the most prolific actors in terms of movie appearances, aiding in talent assessment and analysis within the film industry.

Output:

```
+-----------+-----------+-----------+
| first_name|  last_name|movie_count|
+-----------+-----------+-----------+
|Christopher|       West|         21|
|    Kenneth|    Paltrow|         21|
|      Kevin|      Bloom|         21|
|    Spencer|       Peck|         21|
|        Dan|       Torn|         22|
```

3. SQL Operations:

Formulate 15 questions based on the database, ranging from easy to difficult.

Questions:

1) Retrieve all the distinct country names from the country table.

2) List the titles of films along with their categories.

3) Calculate the maximum length of films in the database.

4) Retrieve all the rental records where the return date is null.

5) What are the addresses of each store?

6) Count the number of films in each category, but only for categories with more than 10 films.

7) What is the name of the customer who lives in the city 'Apeldoorn'?

8) Update the email of the staff member with ID 101 to 'newemail@example.com'.

9) Write a query to create a count of movies in each of the 4 filmlen_groups:

1 hour or less

Between 1-2 hours

Between 2-3 hours

More than 3 hours

10) Select the titles of the movies that have the highest replacement cost.

11) Insert a new category named 'Documentary' into the category table.

12) Combine first_name and last_name from the customer table to become full_name.

13) Show how many inventory items are available at each store.

14) What is the total amount paid by each customer for all their rentals? For each customer, print their name and the total amount paid.

15) What payments have amounts between 3 USD and 5 USD?

-- 1. Retrieve all the distinct country names from the country table.

SELECT DISTINCT country FROM country;

Explanation:

The query utilizes the DISTINCT keyword to fetch unique country names from the country table. By selecting only the 'country' column, the query ensures no duplicate entries are included in the result set.

Output:

Data Output | Messages | Notifications

	country character varying (50)
1	Thailand
2	Virgin Islands, U.S.
3	Indonesia
4	Faroe Islands
5	Bangladesh

Total rows: 109 of 109 | Query complete 00:00:00.067 | Ln 221, Col 1

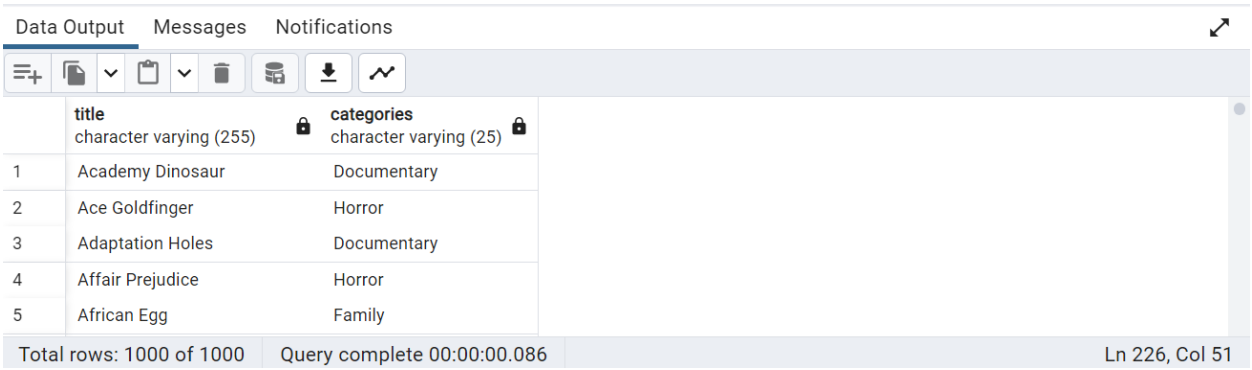
-- 2. List the titles of films along with their categories.

```
SELECT f.title, c.name AS categories FROM film f

JOIN film_category fc

ON f.film_id = fc.film_id

JOIN category c ON fc.category_id = c.category_id;
```

Explanation:

This query employs multiple JOIN operations between the film, film_category, and category tables to associate each film title with its corresponding category name. The SELECT statement fetches the film titles from the film table and the category names from the category table based on the common film IDs stored in the film_category table.

Output:



Data Output | Messages | Notifications

	title character varying (255)	categories character varying (25)
1	Academy Dinosaur	Documentary
2	Ace Goldfinger	Horror
3	Adaptation Holes	Documentary
4	Affair Prejudice	Horror
5	African Egg	Family

Total rows: 1000 of 1000 | Query complete 00:00:00.086 | Ln 226, Col 51

-- 3. Calculate the maximum length of films in the database.

SELECT MAX(length) AS maximum_length_of_films FROM film;

Explanation:

Using the MAX() aggregate function, this query calculates the maximum length of films present in the film table. By selecting the 'length' column and applying the MAX() function, it retrieves the highest value from the 'length' column, representing the longest film duration in the database.

Output:

-- 4. Retrieve all the rental records where the return date is null.

SELECT * FROM rental WHERE return_date IS NULL;

Explanation:

The query filters records from the rental table where the 'return_date' column is null, indicating ongoing rentals. By using the IS NULL condition, it selects rental records that have not been returned yet, providing insights into current rental activities.

Output:

-- 5. What are the addresses of each store?

```
SELECT s.store_id, a.address, a.address2

FROM store s

JOIN address a ON s.address_id = a.address_id;
```

Explanation:

This query utilizes an inner JOIN operation between the store and address tables to associate each store ID with its corresponding address details. By selecting relevant columns from both tables and joining them based on the common 'address_id' column, it retrieves the addresses of all stores in the database.

Output:

Data Output | Messages | Notifications

	store_id integer	address character varying (50)	address2 character varying (50)
1	1	47 MySakila Drive	[null]
2	2	28 MySQL Boulevard	[null]

Total rows: 2 of 2 | Query complete 00:00:00.071 | Ln 235, Col 1

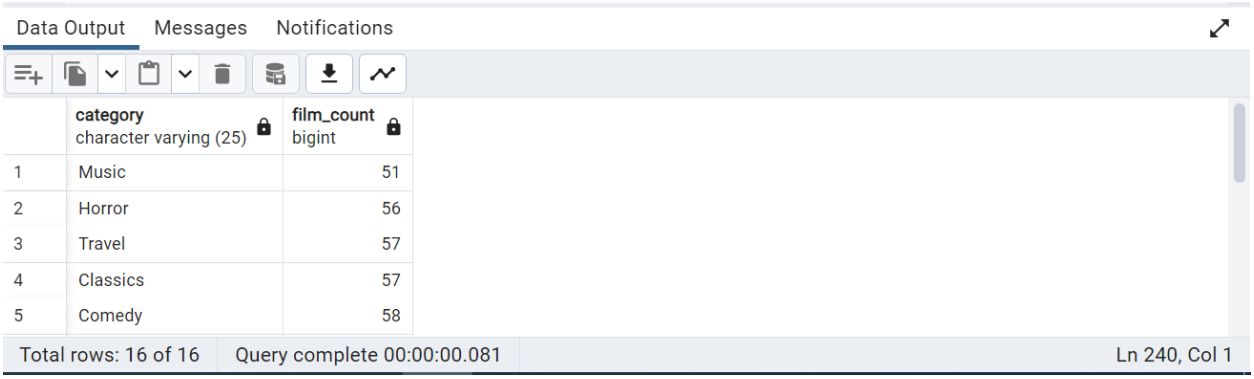
-- 6. Count the number of films in each category, but only for categories with more than 10 films.

```
SELECT c.name AS category, COUNT(fc.film_id) AS film_count FROM category c

JOIN film_category fc

ON c.category_id = fc.category_id

GROUP BY c.name

HAVING COUNT(fc.film_id) > 10

ORDER BY film_count;
```

Explanation:

The query performs a JOIN operation between the category and film_category tables to associate each film with its corresponding category. It then uses GROUP BY and HAVING clauses to count the number of films in each category and filter out categories with fewer than 10 films. Finally, the result set is sorted based on the film count in ascending order.

Output:


Data Output | Messages | Notifications

	category character varying (25)	film_count bigint
1	Music	51
2	Horror	56
3	Travel	57
4	Classics	57
5	Comedy	58

Total rows: 16 of 16 | Query complete 00:00:00.081 | Ln 240, Col 1

-- 7. What is the name of the customer who lives in the city 'Apeldoorn'?

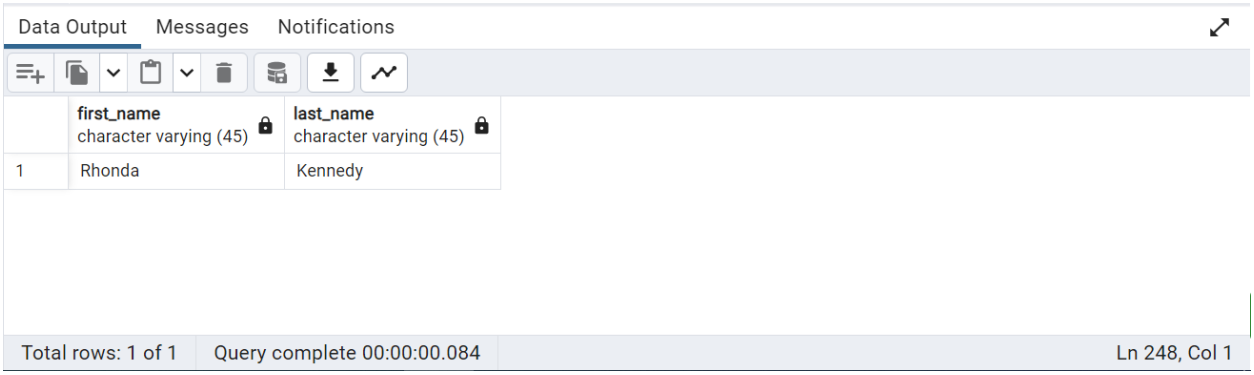
```
SELECT first_name, last_name FROM customer
WHERE address_id IN (
   SELECT address_id FROM address
   WHERE city_id = (
      SELECT city_id FROM city
      WHERE city = 'Apeldoorn'
   )
);
```

Explanation:

This query involves a series of subqueries to retrieve the first and last names of customers residing in the city 'Apeldoorn'. It starts by selecting the city ID corresponding to 'Apeldoorn', then finds

the address IDs associated with the city. Finally, it fetches the customer names based on the address IDs, providing the name of the customer living in 'Apeldoorn'.

Output:



-- 8. Update the email of the staff member with ID 1 to 'newemail@example.com'.

UPDATE staff SET email = 'newemail@example.com' WHERE staff_id = 1;

Explanation:

This SQL statement updates the email address of the staff member with ID 1 to 'newemail@example.com'. It uses the UPDATE command to modify the 'email' column of the staff table for the staff member with the specified ID.

Output:

/* 9. Write a query you to create a count of movies in each of the 4 filmlen_groups: 1 hour or less, Between 1-2 hours, Between 2-3 hours, More than 3 hours.

filmlen_groups ***filmcount_bylencat***

1 hour or less ***104***

Between 1-2 hours ***439***

Between 2-3 hours ***418***

More than 3 hours ***39***

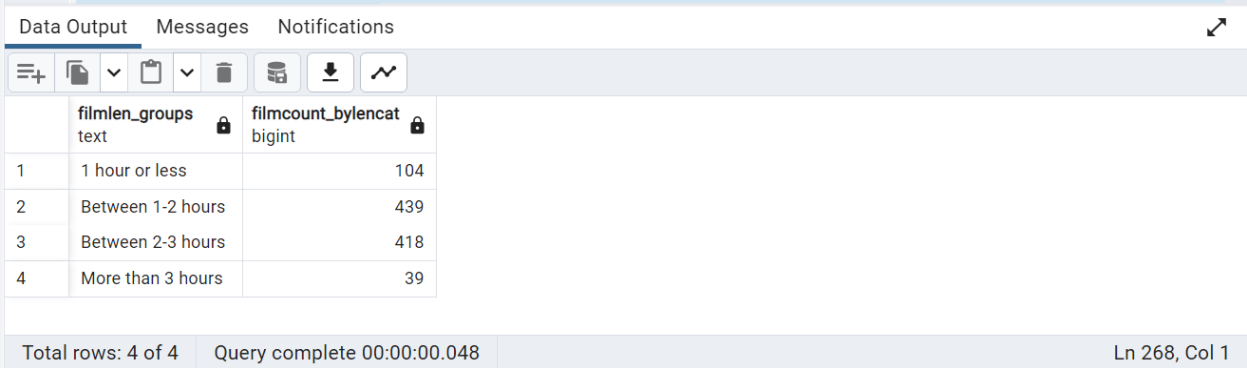
/

```
SELECT DISTINCT(filmlen_groups),
    COUNT(title) OVER (PARTITION BY filmlen_groups) AS filmcount_bylencat
FROM
   (SELECT title,length,
    CASE WHEN length <= 60 THEN '1 hour or less'
    WHEN length > 60 AND length <= 120 THEN 'Between 1-2 hours'
    WHEN length > 120 AND length <= 180 THEN 'Between 2-3 hours'
    ELSE 'More than 3 hours' END AS filmlen_groups
    FROM film ) t1
ORDER BY  filmlen_groups;
```

Explanation:

The query categorizes films into four groups based on their length: 1 hour or less, Between 1-2 hours, Between 2-3 hours, and More than 3 hours. It calculates the count of movies in each group using a CASE statement within a subquery, then presents the results with the respective film count for each group.

Output:



Data Output | Messages | Notifications

	filmlen_groups text	filmcount_bylencat bigint
1	1 hour or less	104
2	Between 1-2 hours	439
3	Between 2-3 hours	418
4	More than 3 hours	39

Total rows: 4 of 4 | Query complete 00:00:00.048 | Ln 268, Col 1

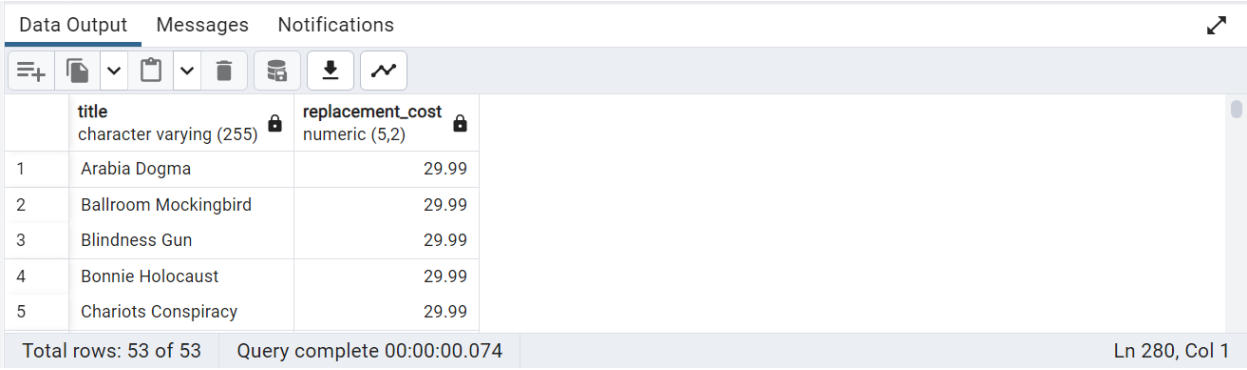
-- 10. Select the titles of the movies that have the highest replacement cost.

```
SELECT title, replacement_cost FROM film

WHERE replacement_cost = (

    SELECT MAX(replacement_cost) FROM film

);
```

Explanation:

This query retrieves the titles of movies with the highest replacement cost by comparing each movie's replacement cost with the maximum replacement cost obtained using a subquery. It filters the films based on the maximum replacement cost.

Output:



Data Output | Messages | Notifications

	title character varying (255)	replacement_cost numeric (5,2)
1	Arabia Dogma	29.99
2	Ballroom Mockingbird	29.99
3	Blindness Gun	29.99
4	Bonnie Holocaust	29.99
5	Chariots Conspiracy	29.99

Total rows: 53 of 53 | Query complete 00:00:00.074 | Ln 280, Col 1

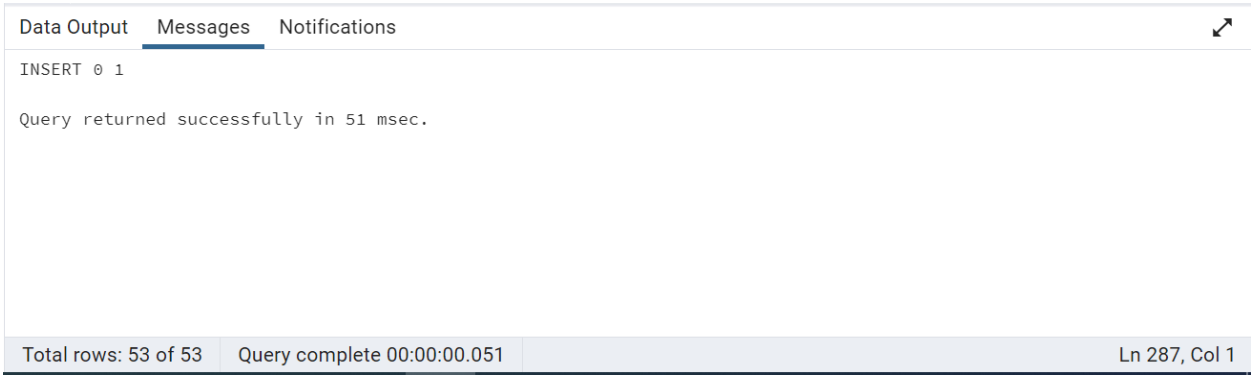
-- 11. Insert a new category named 'Documentary' into the category table.

```
INSERT INTO category (name, last_update) VALUES ('Documentary', NOW());
```

Explanation:

This SQL statement inserts a new category named 'Documentary' into the category table, specifying the category name and the current timestamp using the NOW() function.

Output:



-- 12. Combine first_name and last_name from the customer table to become full_name.

SELECT first_name, last_name, CONCAT(first_name, ' ', last_name) AS full_name

FROM customer;

Explanation:

The SQL query combines the **first_name** and **last_name** columns from the **customer** table to create a new column named **full_name**. It utilizes the **CONCAT()** function, which concatenates strings together, to combine the first and last names with a space in between.

Output:

-- 13. Show how many inventory items are available at each store.

```
SELECT store.store_id, COUNT(inventory_id) FROM store, inventory

WHERE inventory.store_id = store.store_id

GROUP BY store.store_id;
```

Explanation:

This query counts the number of inventory items available at each store by joining the store and inventory tables on the store_id column. It then uses the COUNT() function to count the number of inventory items for each store.

Output:

-- 14. What is the total amount paid by each customer for all their rentals? For each customer print their name and the total amount paid.

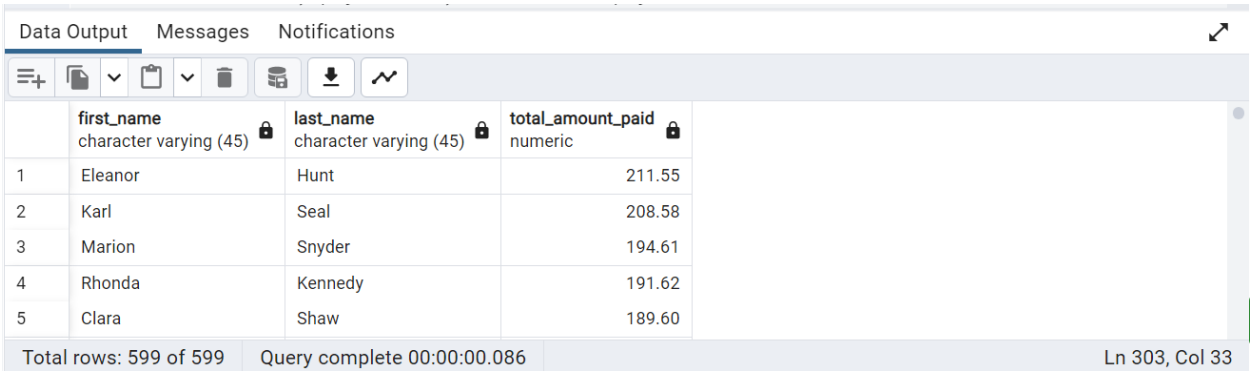
```
SELECT c.first_name, c.last_name, SUM(p.amount) AS total_amount_paid FROM customer c

JOIN payment p ON c.customer_id = p.customer_id

GROUP BY c.customer_id, c.first_name, c.last_name

ORDER BY total_amount_paid DESC;
```

Explanation:

The query calculates the total amount paid by each customer for all their rentals by joining the customer and payment tables on the customer_id column. It then utilizes the SUM() function to

calculate the total amount paid for each customer, grouping the results by customer ID and retrieving their first and last names.

Output:



Data Output | Messages | Notifications

	first_name character varying (45)	last_name character varying (45)	total_amount_paid numeric
1	Eleanor	Hunt	211.55
2	Karl	Seal	208.58
3	Marion	Snyder	194.61
4	Rhonda	Kennedy	191.62
5	Clara	Shaw	189.60

Total rows: 599 of 599 | Query complete 00:00:00.086 | Ln 303, Col 33

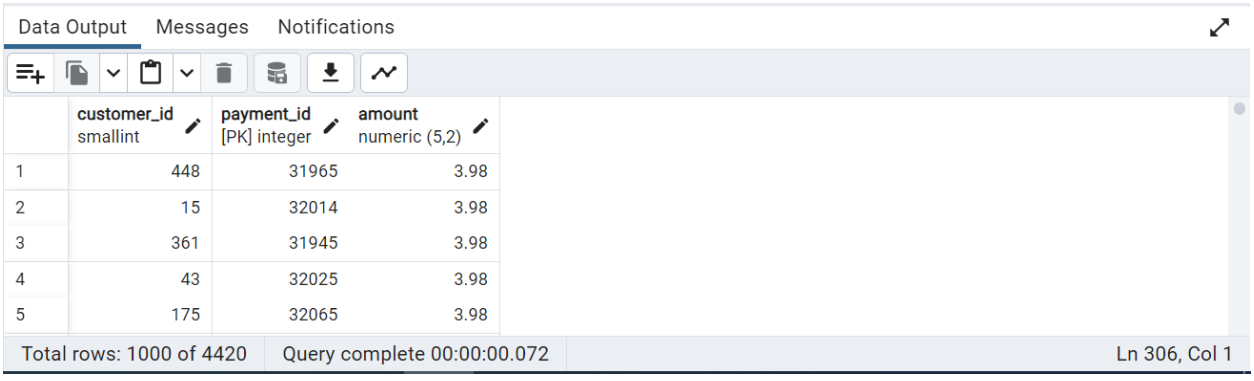
-- 15. What payments have amounts between 3 USD and 5 USD?

```
SELECT customer_id, payment_id, amount FROM payment

WHERE amount BETWEEN 3 AND 5

ORDER BY amount;
```

Explanation:

This query retrieves payments with amounts between 3 and 5 USD using the BETWEEN operator within a WHERE clause to specify the range of payment amounts. It selects the customer ID, payment ID, and amount from the payment table and orders the results by the payment amount.

Output:



Data Output | Messages | Notifications

	customer_id smallint	payment_id [PK] integer	amount numeric (5,2)
1	448	31965	3.98
2	15	32014	3.98
3	361	31945	3.98
4	43	32025	3.98
5	175	32065	3.98

Total rows: 1000 of 4420 | Query complete 00:00:00.072 | Ln 306, Col 1

SQL and PySpark Task

4. PySpark Operations:

Formulate 15 questions based on the database, ranging from easy to difficult.

Questions:

1) Retrieve all the distinct country names from the country table.

2) List the titles of films along with their categories.

3) Calculate the maximum length of films in the database.

4) Retrieve all the rental records where the return date is null.

5) What are the addresses of each store?

6) Count the number of films in each category, but only for categories with more than 10 films.

7) What is the name of the customer who lives in the city 'Apeldoorn'?

8) Update the email of the staff member with ID 101 to 'newemail@example.com'.

9) Write a query to create a count of movies in each of the 4 filmlen_groups:

1 hour or less

Between 1-2 hours

Between 2-3 hours

More than 3 hours

10) Select the titles of the movies that have the highest replacement cost.

11) Insert a new category named 'Documentary' into the category table.

12) Combine first_name and last_name from the customer table to become full_name.

13) Show how many inventory items are available at each store.

14) What is the total amount paid by each customer for all their rentals? For each customer, print their name and the total amount paid.

15) What payments have amounts between 3 USD and 5 USD?

SQL and PySpark Task

 # 1. Retrieve all the distinct country names from the country table.

```
distinct_countries = country_df.select('country').distinct()

distinct_countries.show()
```

Explanation:

The PySpark code selects the 'country' column from the country DataFrame and applies the distinct() function to retrieve only unique country names. It then displays the distinct country names using the show() function, providing a list of all unique countries present in the dataset.

Output:

```
+--------------------+
|             country|
+--------------------+
|                Chad|
|            Anguilla|
|            Paraguay|
|               Yemen|
|             Senegal|
```

 # 2. List the titles of films along with their categories.

```
film_category_join = film_df.join(film_category_df, film_df.film_id ==
film_category_df.film_id)\
                .join(category_df, film_category_df.category_id == category_df.category_id)

film_category_join.select('title', F.col('name').alias('categories')).orderBy('title').show()
```

Explanation:

The code performs a series of joins between the film, film_category, and category DataFrames based on their respective IDs. It selects the 'title' column from the film DataFrame and the 'name' column from the category DataFrame, aliasing the latter as 'categories'. The result is ordered by film title and displays the titles of films along with their corresponding categories.

Output:

```
+--------------------+-----------+
|               title| categories|
+--------------------+-----------+
|   Academy Dinosaur|Documentary|
|     Ace Goldfinger|     Horror|
|   Adaptation Holes|Documentary|
|   Affair Prejudice|     Horror|
|        African Egg|     Family|
```

 # 3. Calculate the maximum length of films in the database.

```
max_length = film_df.select(F.max('length').alias('maximum_length_of_films'))

max_length.show()
```

Explanation:

Using PySpark's agg() function, the code calculates the maximum length of films in the database by selecting the 'length' column from the film DataFrame and applying the max() aggregation function to it. The result is displayed with the alias 'maximum_length_of_films'.

Output:

```
+----------------------+
|maximum_length_of_films|
+----------------------+
|                   185|
+----------------------+
```

 # 4. Retrieve all the rental records where the return date is null.

```
rental_df.filter(rental_df.return_date.isNull()).show()
```

Explanation:

The PySpark code filters the rental DataFrame using the isNull() function to select records where the 'return_date' column has a null value. This effectively retrieves all rental records where the return date is yet to be recorded or completed.

Output:

```
+---------+---------------+------------+-----------+-----------+--------+--------------+
|rental_id|    rental_date|inventory_id|customer_id|return_date|staff_id|   last_update|
+---------+---------------+------------+-----------+-----------+--------+--------------+
|    11496|2/14/2006 15:16|        2047|        155|       NULL|       1|2/16/2006 2:30|
|    11541|2/14/2006 15:16|        2026|        335|       NULL|       1|2/16/2006 2:30|
|    12101|2/14/2006 15:16|        1556|        479|       NULL|       1|2/16/2006 2:30|
|    11563|2/14/2006 15:16|        1545|         83|       NULL|       1|2/16/2006 2:30|
|    11577|2/14/2006 15:16|        4106|        319|       NULL|       2|2/16/2006 2:30|
```

 # 5. What are the addresses of each store?

```
store_address_join = store_df.join(address_df, store_df.address_id == address_df.address_id)

store_address_join.select('store_id', 'address', 'address2').show()
```

Explanation:

By performing an inner join between the store and address DataFrames based on their respective address IDs, the code combines information from both tables. It selects the 'store_id', 'address', and 'address2' columns and displays the addresses of each store.

Output:

```
+--------+-----------------+--------+
|store_id|          address|address2|
+--------+-----------------+--------+
|       1| 47 MySakila Drive|    NULL|
|       2|28 MySQL Boulevard|    NULL|
+--------+-----------------+--------+
```

 # 6. Count the number of films in each category, but only for categories with more than 10 films.

```
film_category_count = film_category_df.groupBy('category_id').count()

film_category_count_filtered = film_category_count.filter(film_category_count['count'] > 10)

film_category_count_filtered = film_category_count_filtered.join(category_df,
                                                                 'category_id',
                                                                 'inner') \
    .select(F.col('name').alias('category'), 'count') \
    .orderBy('count')

film_category_count_filtered.show()
```

Explanation:

The code groups the film_category DataFrame by 'category_id' and counts the number of films in each category using the count() function. It then filters the result to include only categories with more than 10 films, providing insights into categories with significant film representation.

Output:

```
+-----------+-----+
|   category|count|
+-----------+-----+
|      Music|   51|
|     Horror|   56|
|     Travel|   57|
|   Classics|   57|
|     Comedy|   58|
```

 # 7. What is the name of the customer who lives in the city 'Apeldoorn'?

```
customer_city_join = customer_df.join(address_df, customer_df.address_id == address_df.address_id)\
                    .join(city_df, address_df.city_id == city_df.city_id)

customer_city_join.filter(city_df.city == 'Apeldoorn').select('first_name', 'last_name').show()
```

Explanation:

By joining the customer, address, and city DataFrames based on their corresponding IDs, the code filters customers living in the city 'Apeldoorn'. It selects the 'first_name' and 'last_name' columns to display the name of the customer residing in the specified city.

Output:

```
+----------+---------+
|first_name|last_name|
+----------+---------+
|    Rhonda|  Kennedy|
+----------+---------+
```

 # 8. Update the email of the staff member with ID 101 to 'newemail@example.com'.

```
staff_df_upd = staff_df.withColumn('email', F.when(staff_df.staff_id == 1, 'newemail@example.com').otherwise(staff_df.email))

staff_df_upd.select('staff_id', 'email').show()
```

Explanation:

Using PySpark's withColumn() function, the code conditionally updates the 'email' column of the staff DataFrame where the staff ID is 101. It replaces the existing email with the new email address 'newemail@example.com'.

Output:

```
+--------+--------------------+
|staff_id|               email|
+--------+--------------------+
|       1|newemail@example.com|
|       2|Jon.Stephens@saki...|
+--------+--------------------+
```

"""

9. Write a query you to create a count of movies in each of the 4 filmlen_groups:

1 hour or less, Between 1-2 hours, Between 2-3 hours, More than 3 hours.

filmlen_groups	***filmcount_bylencat***
1 hour or less	***104***
Between 1-2 hours	***439***
Between 2-3 hours	***418***
More than 3 hours	***39***

"""

```
# Define the film length groups using when and otherwise functions
film_df = film_df.withColumn("filmlen_groups",
                    F.when(F.col("length") <= 60, "1 hour or less")
                    .when((F.col("length") > 60) & (F.col("length") <= 120), "Between 1-2
hours")
                    .when((F.col("length") > 120) & (F.col("length") <= 180), "Between 2-3
hours")
                    .otherwise("More than 3 hours"))
# Calculate the count of films in each length category
film_count_by_length = film_df \
   .groupBy("filmlen_groups") \
   .agg(F.count("title").alias("filmcount_bylencat")) \
   .orderBy("filmlen_groups")
film_count_by_length \
```

.select(F.col("filmlen_groups").alias("filmlen_groups"),

F.col("filmcount_bylencat").alias("filmcount_bylencat")).show()

Explanation:

The code categorizes films into four groups based on their length using PySpark's when() and otherwise() functions. It then calculates the count of movies in each length category and orders the result accordingly, providing insights into film distribution based on duration.

Output:

```
+-----------------+-----------------+
|   filmlen_groups|filmcount_bylencat|
+-----------------+-----------------+
|   1 hour or less|              104|
|Between 1-2 hours|              439|
|Between 2-3 hours|              418|
|More than 3 hours|               39|
+-----------------+-----------------+
```

 # 10. Select the titles of the movies that have the highest replacement cost.

max_replacement_cost = film_df.agg({"replacement_cost": "max"}).collect()[0][0]

highest_replacement_cost_films = film_df.filter(film_df.replacement_cost == max_replacement_cost)

titles_highest_replacement_cost = highest_replacement_cost_films.select("title", "replacement_cost")

titles_highest_replacement_cost.show()

Explanation:

The code determines the maximum replacement cost across all films using PySpark's agg() function. It then filters films with replacement costs matching the maximum value, retrieving titles of movies with the highest replacement cost.

Output:

```
+--------------------+----------------+
|               title|replacement_cost|
+--------------------+----------------+
|        Arabia Dogma|           29.99|
|Ballroom Mockingbird|           29.99|
|       Blindness Gun|           29.99|
|    Bonnie Holocaust|           29.99|
| Chariots Conspiracy|           29.99|
```

 # 11. Insert a new category named 'Documentary' into the category table.

```
# Define the schema for the new DataFrame

schema = StructType([

    StructField("category_id", StringType(), nullable=True),

    StructField("name", StringType(), nullable=False),

    StructField("last_update", TimestampType(), nullable=False)

])

# Create a new DataFrame with the 'Documentary' category

new_category_df = spark.createDataFrame([(None, 'Documentary', datetime.now())], schema)

# Union the new DataFrame with the existing category DataFrame

category_df_upd = category_df.union(new_category_df)

category_df_upd.show()
```

Explanation:

To add the new category 'Documentary', the PySpark code creates a new DataFrame with a schema matching the category table. It appends a row containing the details of the new category, such as its name and the current timestamp. Finally, it concatenates the new DataFrame with the existing category DataFrame to reflect the addition of the 'Documentary' category.

Output:

```
+-----------+-----------+-------------------+
|category_id|       name|        last_update|
+-----------+-----------+-------------------+
|          1|     Action|     2/15/2006 9:46|
|          2|  Animation|     2/15/2006 9:46|
|          3|   Children|     2/15/2006 9:46|
|          4|   Classics|     2/15/2006 9:46|
|          5|     Comedy|     2/15/2006 9:46|
```

 # 12. Combine first_name and last_name from the customer table to become full_name.

```
customer_full_name = customer_df.withColumn('full_name', F.concat(F.col('first_name'), F.lit(' '), F.col('last_name')))
```

customer_full_name.select('first_name', 'last_name', 'full_name').show()

Explanation:

This PySpark code creates a new column named 'full_name' by concatenating the 'first_name' and 'last_name' columns from the customer DataFrame using the concat() function provided by PySpark. It then selects the original 'first_name' and 'last_name' columns along with the newly created 'full_name' column and displays the result.

Output:

```
+----------+---------+----------------+
|first_name|last_name|       full_name|
+----------+---------+----------------+
|     Jared|      Ely|       Jared Ely|
|      Mary|    Smith|      Mary Smith|
|  Patricia|  Johnson|Patricia Johnson|
|     Linda| Williams|  Linda Williams|
|   Barbara|    Jones|   Barbara Jones|
```

 # 13. Show how many inventory items are available at each store.

inventory_count_per_store = inventory_df.groupBy('store_id').count()

inventory_count_per_store.show()

Explanation:

The code groups the inventory DataFrame by the 'store_id' column and counts the number of items available in each store using the count() function provided by PySpark. It displays the count of inventory items available at each store, providing insight into the inventory distribution across different store locations.

Output:

```
+--------+-----+
|store_id|count|
+--------+-----+
|       1| 2270|
|       2| 2311|
+--------+-----+
```

"""

14. What is the total amount paid by each customer for all their rentals?

For each customer print their name and the total amount paid.

```
"""

# Group by customer_id and sum the amount paid

total_amount_paid_per_customer = payment_df.groupBy('customer_id') \

    .agg(F.sum('amount').alias('total_amount_paid'))

# Join with customer_df to get customer names

total_amount_paid_per_customer = total_amount_paid_per_customer.join(customer_df,

                                                    'customer_id',

                                                    'inner') \

    .select('first_name', 'last_name', 'total_amount_paid')

# Order the result by total_amount_paid in descending order

total_amount_paid_per_customer =
total_amount_paid_per_customer.orderBy(F.desc('total_amount_paid'))

total_amount_paid_per_customer.show()
```

Explanation:

This query groups payment records by 'customer_id' and calculates the sum of the 'amount' paid by each customer using the sum() aggregation function provided by PySpark. It then joins the result with the customer DataFrame to retrieve the names of the customers. The final DataFrame includes each customer's name along with the total amount they paid for all their rentals, ordered by the total amount paid in descending order.

Output:

```
+----------+---------+------------------+
|first_name|last_name| total_amount_paid|
+----------+---------+------------------+
|   Eleanor|     Hunt| 211.5500000000001|
|      Karl|     Seal|208.58000000000013|
|    Marion|   Snyder|194.61000000000007|
|    Rhonda|  Kennedy|191.62000000000006|
|     Clara|     Shaw|189.60000000000005|
```

 # 15. What payments have amounts between 3 USD and 5 USD?

```
payments_between_3_and_5 = payment_df.filter((payment_df.amount >= 3) &
(payment_df.amount <= 5))

payments_between_3_and_5.select('customer_id', 'payment_id',
'amount').orderBy('amount').show()
```

Explanation:

The code filters payment records using the filter() function provided by PySpark to include only those with amounts between 3 and 5 USD. It then selects specific columns ('customer_id', 'payment_id', 'amount') from the filtered DataFrame and orders the result by the 'amount' column to display payments within the specified range. This query helps identify payments falling within the specified amount range for further analysis or auditing purposes.

Output:

```
+-----------+----------+------+
|customer_id|payment_id|amount|
+-----------+----------+------+
|        269|     31919|  3.98|
|        361|     31945|  3.98|
|        448|     31965|  3.98|
|        457|     31969|  3.98|
|         15|     32014|  3.98|
```