1️⃣SQL and Pyspark Question 1: Identify VIP Users for Netflix

Question: To better cater to its most dedicated users, Netflix would like to identify its “VIP users” - those who are most active in terms of the number of hours of content they watch. Write a SQL query that will retrieve the top 10 users with the most watched hours in the last month.

Tables:
* users table: user_id (integer), sign_up_date (date), subscription_type (text)
* watching_activity table: activity_id (integer), user_id (integer), date_time (timestamp), show_id (integer), hours_watched (float)

```
%sql
select u.user_id, u.subscription_type, sum(wa.hours_watched) as total_hours_watched from users u join
watching_activity wa on u.user_id = wa.user_id where wa.data_time >= add_months(trunc(sysdate,'Month'), -1)
and wa.data_time < Trunc(sysdate,'Month') group by u.user_id, u.subscription_type order by total_hours_watched
desc fetch first 10 rows only;
```

```
%python
from pyspark.sql import functions as F
from pyspark.sql.window import Window

current_date = F.current_date()
first_day_of_month = F.date_trunc("month",current_date)
last_month_start = F.add_months(first_day_of_month, -1)

vip_users = (watching_activity_df.filter(F.col("date_time") >= last_month_start).filter(F.col("date_time") <
first_day_of_month).groupBy("user_id").agg(F.sum("hours_watched").alias("total_hours_watched")).join(users_df,"user
_id").orderBy(F.desc("total_hours_watched")).limit(10))
```

2️⃣SQL and Pyspark Question 2: Analyzing Ratings For Netflix Shows

Question: Given a table of user ratings for Netflix shows, calculate the average rating for each show within a given month. Assume that there is a column for user_id, show_id, rating (out of 5 stars), and date of review. Order the results by month and then by average rating (descending order).

Tables:
* show_reviews table: review_id (integer), user_id (integer), review_date (timestamp), show_id (integer), stars (integer)

```
%sql
```

```
select show_id, extract(month from review_date) as month, extract(year from review_date) as year, avg(stars) as
avg_rating, count(*) as review_count from show_reviews group by show_id, extract(month from review_date),
extract(year from review_date) order by year,month,avg_rating desc;
```

```
%python
from pyspark.sql import functions as F

rating_by_month = (show_reviews_df.withColumn("month",
F.month("review_date")).withColumn("year",F.year("review_date")).groupBy("show_id","month","year").agg(F.avg("st
ars").alias("avg_rating"), F.count("*").alias("review_count")).orderBy("year","month",F.desc("avg_rating")))
rating_by_month.show()
```

3️⃣ SQL Question 3: What does EXCEPT / MINUS SQL commands do?

Question: Explain the purpose and usage of the EXCEPT (or MINUS in some SQL dialects) SQL commands.

The EXCEPT (or MINUS in Oracle) operator is used to return all distinct rows from the first query that are not present in the results of the second query.

Key characteristics:
Both queries must have the same number of columns
Corresponding columns must have compatible data types
Duplicates are removed from the final result
Similar to set difference operation in mathematics

```
%sql
-- Users who watched show A but not show B
SELECT user_id FROM watching_activity WHERE show_id = 101
EXCEPT
SELECT user_id FROM watching_activity WHERE show_id = 102;
```

```
%python
# Users who watched show A but not show B
show_a_watchers = watching_activity_df.filter(F.col("show_id") == 101).select("user_id")
show_b_watchers = watching_activity_df.filter(F.col("show_id") == 102).select("user_id")

a_not_b_watchers = show_a_watchers.exceptAll(show_b_watchers)
a_not_b_watchers.show()
```

4️⃣ SQL Question 4: Filter Netflix Users Based on Viewing History and Subscription Status

Question: You are given a database of Netflix's user viewing history and their current subscription status. Write a SQL query to find all active customers who watched more than 10 episodes of a show called "Stranger Things" in the last 30 days.

Tables:
* users table: user_id (integer), active (boolean)
* viewing_history table: user_id (integer), show_id (integer), episode_id (integer), watch_date (date)
* shows table: show_id (integer), show_name (text)

```
%sql
select u.user_id from users u join viewing_history vh on u.user_id = vh.user_id join shows s on vh.show_id = s.how_id
where u.active = True and s.show_name = 'Stringer things and vh.watch_date >= Sysdate - 30 group by u.user_id
having count(Distinct vh.episode_id) > 10;
```

```
%python
from pyspark.sql import functions as F

active_stranger_things_viewers = (users_df.filter(F.col("active") == True).join(viewing_history_df,
"user_id").join(shows_df.filter(F.col("show_name") == "Stranger Things"),
"show_id").groupBy("user_id").agg(F.countDistinct("episode_id").alias("episodes_watched")).filter(F.col("episode_wat
ched") > 10))
active_stranger_things_viewers.show()
```

5️⃣ SQL Question 5: What does it mean to denormalize a database?

Question: Explain the concept and implications of denormalizing a database.

Denormalization is the process of intentionally introducing redundancy into a database design to improve read performance by reducing the number of joins needed for queries.

Key aspects:
Purpose: Optimize read performance at the cost of write performance
Techniques: Adding redundant columns, combining tables, creating summary tables
When to use: For read-heavy applications, reporting systems, data warehouses

Trade-offs:
Pros: Faster reads, simpler queries
Cons: Data redundancy, potential inconsistency, more complex updates
Increased storage requirements

In PySpark, denormalization typically involves:
Creating wide DataFrames with redundant data
Using join() operations to combine tables
Persisting denormalized DataFrames for better performance

```
# Denormalize by joining user data with viewing history
denormalized_df = (viewing_history_df
    .join(users_df, "user_id")
    .join(shows_df, "show_id")
    .cache()  # Persist in memory for faster access)
```

6 SQL Question 6: Filter and Match Customer’s Viewing Records

Question: As a data analyst at Netflix, you are asked to analyze the customer’s viewing records. You confirmed that Netflix is especially interested in customers who have been continuously watching a particular genre - ‘Documentary’ over the last month. The task is to find the name and email of those customers who have viewed more than five ‘Documentary’ movies within the last month. ‘Documentary’ could be a part of a broader genre category in the genre field (for example, ‘Documentary, History’). Therefore, the matching pattern could occur anywhere within the string.

Tables:
* movies table: movie_id (integer), title (text), genre (text), release_year (integer)
* customer table: user_id (integer), name (text), email (text), last_movie_watched (integer), date_watched (date)

```
%sql
select c.user_id, c.name, c.email, count(*) as documentary_count from customer c join movies m on
c.last_movie_watched = m.movie_id where c.date_watched >= sysdate - 30 and m.genre like '%Documentary%'
group by c.user_id, c.name, c.email having count(*) > 5 order by documentry_count DESC;
```

```
%python
from pyspark.sql import functions as F

documentary_viewers = (customer_df
.join(movies_df, customer_df.last_movie_watched == movies_df.movie_id)
.filter(F.col("date_watched") >= F.date_sub(F.current_date(), 30))
.filter(F.col("genre").contains("Documentry"))
.groupBy("user_id", "name", "email")
.agg(F.count("*").alias("documentry_count"))
.filter(F.col("documentry_count") > 5)
.orderBy(F.desc("documentry_count")))

documentray_viewers.show()
```