In [0]:
```
from pyspark.sql.functions import *

# 1. Load the users table
users_df = spark.table("users") # Assuming you named it this

# 2. Load the purchases table
purchases_df = spark.table("purchases")

products_df = spark.table("products")

appointments_df = spark.table("appointments")

# 3. Check that they both loaded
print(f"Total Users: {users_df.count()}")
print(f"Total Purchases: {purchases_df.count()}")

print(f"Total Products: {products_df.count()}")
print(f"Total Appointments: {appointments_df.count()}")
```

```
Total Users: 80000
Total Purchases: 320000
Total Products: 8000
Total Appointments: 160000
```

In [0]:
```
print("--- Top 5 Pet Owners by Number of Purchases ---")

top_shoppers_df = purchases_df.join(
    users_df,
    purchases_df["pet_owner_id"] == users_df["user_id"]
).groupBy(
    users_df["user_id"],
    users_df["name"]
).agg(
    count("purchase_id").alias("total_purchases")
).orderBy(
    desc("total_purchases")
)

top_shoppers_df.show(5)
```

```
--- Top 5 Pet Owners by Number of Purchases ---
+-------+---------------+---------------+
|user_id|           name|total_purchases|
+-------+---------------+---------------+
|   1241| William Garcia|             19|
|  34579|  Jason Walters|             18|
|  64340|      Sue Fritz|             18|
|  11887|James Rodriguez|             17|
|  31602|   David Butler|             16|
+-------+---------------+---------------+
only showing top 5 rows
```

In [0]:
```
from pyspark.sql.functions import col, explode, desc

# Question: What are the most popular pet species?
#
# 1. explode(): To flatten the nested 'pets' array into individual rows.
# 2. Dot notation: To access the 'species' field inside the pet struct.

print("--- Most Popular Pet Species ---")

# 1. Filter for pet owners who actually have pets
pet_owners_df = users_df.filter(
    (col("user_type") == "pet_owner") & (col("pets").isNotNull())
)

# 2. "Explode" the 'pets' array.
#    This creates a new row for each pet in the array.
all_pets_df = pet_owners_df.select(
    explode(col("pets")).alias("pet_details")
)

# 3. Group by the pet's species and count
species_count_df = all_pets_df.groupBy(
    col("pet_details.species")  # Accessing the nested field
).count().orderBy(
    desc("count")
)
```

```
species_count_df.show()
```

```
--- Most Popular Pet Species ---
+--------+-----+
|species|count|
+--------+-----+
|    Cat|22564|
|   Fish|22457|
|    Dog|22423|
|Reptile|22369|
|   Bird|22081|
+--------+-----+
```

In [0]:
```
# 1A

users_df.printSchema()
```

```
root
 |-- address: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- state: string (nullable = true)
 |    |-- street: string (nullable = true)
 |-- email: string (nullable = true)
 |-- groomer_details: struct (nullable = true)
 |    |-- bio: string (nullable = true)
 |    |-- certification: string (nullable = true)
 |    |-- rating: double (nullable = true)
 |    |-- services: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- description: string (nullable = true)
 |    |    |    |-- price: double (nullable = true)
 |    |    |    |-- service_id: long (nullable = true)
 |    |    |    |-- service_type: string (nullable = true)
 |-- join_date: timestamp (nullable = true)
 |-- name: string (nullable = true)
 |-- owner_details: struct (nullable = true)
 |    |-- payment_info: string (nullable = true)
 |-- pets: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- breed: string (nullable = true)
 |    |    |-- dob: timestamp (nullable = true)
 |    |    |-- name: string (nullable = true)
 |    |    |-- notes: string (nullable = true)
 |    |    |-- pet_id: long (nullable = true)
 |    |    |-- photo_urls: array (nullable = true)
 |    |    |    |-- element: string (containsNull = true)
 |    |    |-- species: string (nullable = true)
 |-- user_id: long (nullable = true)
 |-- user_type: string (nullable = true)
```

In [0]:
```
# 1B

### The data type of pets field in the users_df DataFrame is array.
```

In [0]:
```
%sql

-- 1C

DESCRIBE TABLE products
```

col_name	data_type	comment
name	string	null
price	double	null
product_id	bigint	null

In [0]:
```
# 1D

### The data type of the price field in the zotpets_products table is double.
```

In [0]:
```
# 2A Python

users_df.filter(col("user_id") == 1).show()
```

```
+--------------------+--------------------+--------------+--------------------+-----------+-------------+------------
--------+-------+---------+
|             address|               email|groomer_details|          join_date|       name|owner_details|
pets|user_id|user_type|
+--------------------+--------------------+--------------+--------------------+-----------+-------------+------------
--------+-------+---------+
|{Banksport, MP, 9...|zbailey@example.org|           NULL|2021-04-27 00:00:00|Tyler Brown|     {PayPal}|[{Canary, 20
23-07...|      1|pet_owner|
+--------------------+--------------------+--------------+--------------------+-----------+-------------+------------
--------+-------+---------+
```

In [0]:
```sql
%sql

-- 2A SQL

SELECT * FROM users WHERE user_id = 1
```

address	email	groomer_details	join_date	name	owner_details	pets	user_id	user_ty
List(Banksport, MP, 929 Hart Pine Suite 923)	zbailey@example.org	null	2021-04-27T00:00:00.000Z	Tyler Brown	List(PayPal)	List(List(Canary, 2023-07-20T00:00:00.000Z, Heather, Sense guess yes bed network part center job option subject until reflect., 1, List(), Bird))	1	pet_own

In [0]:
```python
# 2B Python

total_shampoo_product_sales_df = purchases_df.join(
    products_df,
    purchases_df["product_id"] == products_df["product_id"]
).filter(col("name").startswith("Shampoo")
).join(users_df, purchases_df["groomer_id"] == users_df["user_id"]
).groupBy(users_df["user_id"], users_df["name"]
).agg(count("purchase_id").alias("total_shampoo_products_sold")
).orderBy(desc("total_shampoo_products_sold")
).limit(5)

total_shampoo_product_sales_df.show()
```

```
+-------+-------------+--------------------------+
|user_id|         name|total_shampoo_products_sold|
+-------+-------------+--------------------------+
|  29343|  James Moore|                        11|
|  75279|   Philip Lam|                        11|
|  50521|  Brian Smith|                        10|
|  14278|Jennifer Hall|                        10|
|  76015| Adrian Glass|                        10|
+-------+-------------+--------------------------+
```

In [0]:
```sql
%sql

-- 2B SQL

SELECT u.user_id, u.name, COUNT(purch.purchase_id) AS total_shampoo_products_sold
FROM purchases purch
JOIN products prod ON purch.product_id = prod.product_id
JOIN users u ON purch.groomer_id = u.user_id
WHERE prod.name LIKE 'Shampoo%'
GROUP BY u.user_id, u.name
ORDER BY total_shampoo_products_sold DESC
LIMIT 5
```

user_id	name	total_shampoo_products_sold
29343	James Moore	11
75279	Philip Lam	11
50521	Brian Smith	10
14278	Jennifer Hall	10
76015	Adrian Glass	10

In [0]:
```python
# 2C Python

december2024_appointments_df = appointments_df.filter(
    (col("appointment_datetime") >= "2024-12-01") &
    (col("appointment_datetime") < "2025-01-01")
).groupBy(date_format("appointment_datetime", "yyyy-MM-dd").alias("date")
).agg(count("appointment_id").alias("total_appointment_count")
).orderBy(desc("total_appointment_count")
).limit(3)

december2024_appointments_df.show()
```

```
+----------+------------------------+
|      date|total_appointment_count|
+----------+------------------------+
|2024-12-13|                     480|
|2024-12-10|                     468|
|2024-12-26|                     465|
+----------+------------------------+
```

In [0]:
```sql
%sql

-- 2C SQL

SELECT DATE(appointment_datetime) AS date, COUNT(appointment_id) AS total_appointment_count
FROM appointments
WHERE appointment_datetime >= '2024-12-01'
  AND appointment_datetime < '2025-01-01'
GROUP BY DATE(appointment_datetime)
ORDER BY total_appointment_count DESC
LIMIT 3
```

date	total_appointment_count
2024-12-13	480
2024-12-10	468
2024-12-26	465

In [0]:
```python
# 2D Python

popular_pet_species_df = users_df.filter(
    col("pets").isNotNull()
).select(explode("pets").alias("pet")
).groupBy(col("pet.species")
).agg(count("*").alias("count")
).orderBy(desc("count")
).limit(5)

popular_pet_species_df.show()
```

```
+-------+-----+
|species|count|
+-------+-----+
|    Cat|22564|
|   Fish|22457|
|    Dog|22423|
|Reptile|22369|
|   Bird|22081|
+-------+-----+
```

In [0]:
```sql
%sql

-- 2D SQL

SELECT pet.species, COUNT(*) as count
FROM users
LATERAL VIEW EXPLODE(pets) exploded_pets AS pet
GROUP BY pet.species
ORDER BY count DESC
LIMIT 5
```

species	count
Cat	22564
Fish	22457
Dog	22423
Reptile	22369
Bird	22081

In [0]:
```python
# 2E Python

appointments_scheduled_df = appointments_df.filter(
    col("appointment_datetime") >= "2025-11-01"
).join(
    users_df,
    appointments_df["pet_owner_id"] == users_df["user_id"]
).select("appointment_id", "appointment_datetime", "email"
).orderBy("appointment_datetime"
).limit(5)

appointments_scheduled_df.show()
```

```
+--------------+--------------------+--------------------+
|appointment_id|appointment_datetime|               email|
+--------------+--------------------+--------------------+
|          4459|2025-11-01 00:02:...| tammy91@example.com|
|         33603|2025-11-01 00:04:...|williambaker@exam...|
|         49920|2025-11-01 00:07:...|robertsonwilliam@...|
|        118439|2025-11-01 00:09:...|wongjeremy@exampl...|
|         22987|2025-11-01 00:10:...|michael97@example...|
+--------------+--------------------+--------------------+
```

In [0]:
```sql
%sql

-- 2E SQL

SELECT app.appointment_id, app.appointment_datetime, u.email
FROM appointments app
JOIN users u ON app.pet_owner_id = u.user_id
WHERE app.appointment_datetime >= '2025-11-01'
ORDER BY app.appointment_datetime ASC
LIMIT 5
```

appointment_id	appointment_datetime	email
4459	2025-11-01T00:02:51.301Z	tammy91@example.com
33603	2025-11-01T00:04:11.912Z	williambaker@example.com
49920	2025-11-01T00:07:58.033Z	robertsonwilliam@example.net
118439	2025-11-01T00:09:10.077Z	wongjeremy@example.com
22987	2025-11-01T00:10:22.819Z	michael97@example.com

In [0]:
```python
# 2F Python

top_groomer_revenues_df = purchases_df.join(
    products_df, purchases_df["product_id"] == products_df["product_id"]
).join(users_df, purchases_df["groomer_id"] == users_df["user_id"]
).groupBy(users_df["name"], users_df["email"]
).agg(sum("price").alias("total_revenue")
).orderBy(desc("total_revenue")
).limit(3)

top_groomer_revenues_df.show()
```

```
+-----------------+--------------------+-----------------+
|             name|               email|    total_revenue|
+-----------------+--------------------+-----------------+
|   Joshua Mcmahon|monica83@example.net|           882.72|
|Joseph Richardson|sharon92@example.org|856.5300000000001|
|      Kristin Key|gregory66@example...|798.8299999999999|
+-----------------+--------------------+-----------------+
```

In [0]:
```
%sql

-- 2F SQL

SELECT u.name, u.email, SUM(prod.price) AS total_revenue
FROM purchases purch
JOIN products prod ON purch.product_id = prod.product_id
JOIN users u ON purch.groomer_id = u.user_id
GROUP BY u.user_id, u.name, u.email
ORDER BY total_revenue DESC
LIMIT 3
```

name	email	total_revenue
Joshua Mcmahon	monica83@example.net	882.72
Joseph Richardson	sharon92@example.org	856.5300000000001
Kristin Key	gregory66@example.org	798.8299999999999

In [0]:
```
# 2G Python

owners_who_had_appointment_in_2024 = appointments_df.filter(
    (col("appointment_datetime") >= "2024-01-01") &
    (col("appointment_datetime") < "2025-01-01")
).select("pet_owner_id").distinct()

result_df = owners_who_had_appointment_in_2024.join(
    users_df.alias("u"), col("pet_owner_id") == col("u.user_id")
).filter(col("u.pets").isNotNull()
).select(col("u.user_id"), explode("u.pets").alias("pet")
).join(purchases_df.alias("purch"), col("u.user_id") == col("purch.pet_owner_id")
).join(products_df.alias("prod"), col("purch.product_id") == col("prod.product_id")
).groupBy(col("pet.species")
).agg(sum("prod.price").alias("total_revenue"), countDistinct("u.user_id").alias("unique_owner_count")
).orderBy(desc("total_revenue"))

result_df.show()
```

```
+-------+------------------+------------------+
|species|     total_revenue|unique_owner_count|
+-------+------------------+------------------+
|    Cat|1118836.3999999976|              6973|
|   Bird|1095087.8500000006|              6802|
|    Dog|1082523.7000000032|              6746|
|Reptile| 1079131.019999998|              6769|
|   Fish|1075447.8999999962|              6755|
+-------+------------------+------------------+
```

In [0]:
```
%sql

-- 2G SQL

WITH owners_who_had_appointment_in_2024 AS (
    SELECT DISTINCT pet_owner_id
    FROM appointments
    WHERE appointment_datetime >= '2024-01-01'
      AND appointment_datetime < '2025-01-01'
),
owner_of_pets AS (
    SELECT u.user_id, pet.species
    FROM owners_who_had_appointment_in_2024 owha
    JOIN users u ON owha.pet_owner_id = u.user_id
    LATERAL VIEW EXPLODE(u.pets) exploded AS pet
)
SELECT own.species, SUM(prod.price) AS total_revenue, COUNT(DISTINCT own.user_id) AS unique_owner_count
FROM owner_of_pets own
JOIN purchases purch ON own.user_id = purch.pet_owner_id
JOIN products prod ON purch.product_id = prod.product_id
GROUP BY own.species
ORDER BY total_revenue DESC
```

species	total_revenue	unique_owner_count
Cat	1118836.3999999976	6973
Bird	1095087.8500000006	6802
Dog	1082523.7000000032	6746
Reptile	1079131.019999998	6769
Fish	1075447.8999999962	6755

3A PostgreSQL

Spark's native array support allows for simpler and faster queries using built-in functions like EXPLODE. PostgreSQL, however, would require slower JSON processing and lateral joins to handle the same semi-structured pet data, making this query more complex.

3B MongoDB

To get data from four different "collections", joining multiple collections in document databases would be required, but this is inefficient since it needs separate queries and manual data combining. While MongoDB can handle some joins, it becomes slow and complicated with large amounts of data across different collections, which is one of the major weaknesses of document databases that this query exposes.

3C Cassandra

Cassandra's denormalized design works well for simple key-based lookups but struggles with complex analytical queries. It's optimized for fast reads and writes on single data partitions, not for joining multiple tables or analyzing data across the entire dataset. This is why it would be a terrible choice for this kind of ad-hoc analytical query.