databricks HW5_template

(https://databricks.com)

Assignment 5

By Ameya Gaitonde

setup environment (must run first)

3

```
#%python
# Delete or comment this block out when running the notebook on the full dataset
#users_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_sample/Users.json")
#records_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_sample/Records.json
#sessions_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_sample/Sessions.js
#reviews_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_sample/Reviews.json
#upvotes_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_sample/Upvotes.json

#users_df.createOrReplaceTempView("users")
#records_df.createOrReplaceTempView("records")
#sessions_df.createOrReplaceTempView("sessions")
#reviews_df.createOrReplaceTempView("reviews")
#upvotes_df.createOrReplaceTempView("upvotes")
```

4

```
%python
# # Uncomment this block when running the notebook on the full dataset (you must do this in the end before submitting
solution!)
users_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_full/Users.json")
records_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_full/Records.json")
sessions_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_full/Sessions.json"
reviews_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_full/Reviews.json")
upvotes_df = spark.read.format("json").option("multiline", True).load("/FileStore/tables/zotmusic_full/Upvotes.json")

users_df.createOrReplaceTempView("users")
records_df.createOrReplaceTempView("records")
sessions_df.createOrReplaceTempView("sessions")
reviews_df.createOrReplaceTempView("reviews")
upvotes_df.createOrReplaceTempView("upvotes")
```

5

```
%sql
(select "users", count(*) from users)
union all
(select "records", count(*) from records)
union all
(select "sessions", count(*) from sessions)
union all
(select "reviews", count(*) from reviews)
union all
(select "upvotes", count(*) from upvotes)
```

Table

	users	count(1)
1	users	5000
2	records	100000
3	sessions	889979

4	reviews	962143
5	upvotes	9189735

5 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

Solution

1.A: We've printed the users schema below using a Python snippet.

1.B: The Address field is of type: StructType

8

```
%python
#1.A
users_df.printSchema()
#1.B address data type is: StructType
address_type = users_df.schema['address'].dataType

print("Data type of Address field:", address_type)
```

```
 |    |-- city: string (nullable = true)
 |    |-- state: string (nullable = true)
 |    |-- street: string (nullable = true)
 |    |-- zip: string (nullable = true)
 |-- bio: string (nullable = true)
 |-- email: string (nullable = true)
 |-- genres: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- is_artist: boolean (nullable = true)
 |-- is_listener: boolean (nullable = true)
 |-- joined_date: string (nullable = true)
 |-- nickname: string (nullable = true)
 |-- real_name: struct (nullable = true)
 |    |-- first_name: string (nullable = true)
 |    |-- last_name: string (nullable = true)
 |-- stage_name: string (nullable = true)
 |-- subscription: string (nullable = true)
 |-- user_id: string (nullable = true)

Data type of Address field: StructType([StructField('city', StringType(), True), StructField('state', StringType(), Tru
e), StructField('street', StringType(), True), StructField('zip', StringType(), True)])
```

1.C Below we've printed the schema of Records table.

1.D The songs field is of the following datatype: array<structbpm:bigint,length:bigint,mood:string,title:string,track_number:bigint>

10

```
%sql
--1.C
describe records;
--1.D songs data type is: array<struct<bpm:bigint,length:bigint,mood:string,title:string,track_number:bigint>>
```

Table

	col_name	data_type	comment
1	description	string	null

2	genre	string	null
3	is_album	boolean	null
4	is_single	boolean	null
5	record_id	string	null
6	released_by	struct<artist_user_id:string,release_date:string>	null
7	songs	array<struct<bpm:bigint,length:bigint,mood:string,title:string,track_number:bigint>>	null
8	title	string	null
9	video_url	string	null

9 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

2A

The output for the DF query is dense, but we've retrieved all the information for the user with user_id of "user_W2xxLgbyRoqp"

12

```
%python
#2.A DF
specific_id = "user_W2xxLgbyRoqp"
specific_user = users_df.filter(users_df.user_id == specific_id)

# Show the result
specific_user.show()
```

```
+--------------------+--------------------+--------------------+--------------------+---------+-----------+-----------+
----------------+---------+-----------+------------+-----------------+
|             address|                 bio|               email|              genres|is_artist|is_listener|joined_date|
nickname|real_name| stage_name|subscription|          user_id|
+--------------------+--------------------+--------------------+--------------------+---------+-----------+-----------+
----------------+---------+-----------+------------+-----------------+
|{Joelmouth, NULL,...|Full computer tre...|katherinedelgado@...|[Techno, Rock, El...|     true|      false| 2019-05-04|
katherinedelgado|     NULL|stacyobrien|        NULL|user_W2xxLgbyRoqp|
+--------------------+--------------------+--------------------+--------------------+---------+-----------+-----------+
----------------+---------+-----------+------------+-----------------+
```

13

```
%sql
--2.A SQL
select * from users
where users.user_id = 'user_W2xxLgbyRoqp';
```

Table

	address	bio
1	{"city":"Joelmouth","state":null,"street":"78374 Michael Centers Suite 220","zip":"8041...	Full computer treat mention agreement. Also success or app

1 row

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

2B

15

```
%python
#2.B DF
import pyspark.sql.functions as funcs

PopRecordsdf = (records_df
                  .filter(records_df.genre == 'Pop')  # Filter for genre = 'Pop'
                  .groupBy(funcs.col('released_by.artist_user_id'))  # Group by artist_user_id
                  .agg(funcs.count('*').alias('num_pop_records'))  # Count num records per artist
                  .orderBy(funcs.desc('num_pop_records'))
                  .limit(5))


PopRecordsdf.show()
```

```
+-----------------+---------------+
|   artist_user_id|num_pop_records|
+-----------------+---------------+
|user_W5FmJadbR6OF|              8|
|user_h1_M1rkrRs-F|              7|
|user_giDyfRSuSRKk|              7|
|user_tylP-mPISHS1|              7|
|user_0lVVujY8RC6Y|              7|
+-----------------+---------------+
```

16

```
%sql
--2.B SQL
select released_by.artist_user_id as artist_id,
       COUNT(genre) as num_pop_records
from records
where genre = 'Pop'
group by released_by.artist_user_id
order by num_pop_records desc
limit 5;
```

Table

	artist_id	num_pop_records
1	user_W5FmJadbR6OF	8
2	user_h1_M1rkrRs-F	7
3	user_giDyfRSuSRKk	7
4	user_tylP-mPISHS1	7
5	user_0lVVujY8RC6Y	7

5 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

2C

18

```
#2.C DF

# Convert initiate_at to timestamp
sessions_df = sessions_df.withColumn('initiate_at', funcs.to_timestamp('session_duration.initiate_at'))

# Filter for November 2023
filtered_sessions = sessions_df.filter(
    (funcs.col('initiate_at') >= '2023-11-01 00:00:00') &
    (funcs.col('initiate_at') <= '2023-11-30 23:59:59')
)


# Group by record_id and track_number, count session_id as session_count
# order by session_count and limit to top 3 results.
mostplayed = filtered_sessions.groupBy('song.record_id', 'song.track_number') \
    .agg(funcs.count('session_id').alias('session_count')) \
    .orderBy(funcs.col('session_count').desc()) \
    .limit(3)


mostplayed.show()
```

```
+------------------+------------+-------------+
|          record_id|track_number|session_count|
+------------------+------------+-------------+
|record_llStZtAYQfiA|          11|          887|
|record_rv-yMzjfQSC_|           1|          866|
|record_U-VuvQ46SD6p|           2|            3|
+------------------+------------+-------------+
```

19

```
%sql
--2.C SQL

select song.record_id, song.track_number, COUNT(session_id) as session_count
from sessions
where session_duration.initiate_at BETWEEN '2023-11-01 00:00:00' AND '2023-11-30 23:59:59'
group by song.record_id, song.track_number
order by session_count desc
limit 3;
```

Table

	record_id	track_number	session_count
1	record_llStZtAYQfiA	11	887
2	record_rv-yMzjfQSC_	1	866
3	record_U-VuvQ46SD6p	2	3

3 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

2D

21

```
#2.D DF

# Explode genres array to create a new row for each genre
exploded_users = users_df.withColumn("genre", funcs.explode("genres"))

# Group by 'genre', count distinct 'user_id', and order by the count
populargenres = exploded_users.groupBy("genre") \
    .agg(funcs.countDistinct("user_id").alias("num_users")) \
    .orderBy(funcs.col("num_users").desc()) \
    .limit(1)

populargenres.show()
```

```
+-----+---------+
|genre|num_users|
+-----+---------+
| Funk|     1203|
+-----+---------+
```

22

```
%sql
--2.D SQL

select genre, COUNT(distinct user_id) as num_users
from users
lateral view explode(genres) as genre
group by genre
order by num_users desc
limit 1;
```

Table

	genre	num_users
1	Funk	1203

1 row

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

2E

24

```
%python
#2.E DF


# We can first join reviews with users on the user_id. Then we convert the posted_time to a timestamp
# and filter based on relevant timestamps. Finally, select relevant columns and order/limit results.
result_2e = (reviews_df
             .join(users_df, reviews_df["posted_by.user_id"] == users_df["user_id"])
             .filter(
                 (funcs.to_timestamp(reviews_df["posted_by.posted_time"]) >= "2024-08-03 00:00:00") &
                 (funcs.to_timestamp(reviews_df["posted_by.posted_time"]) < "2024-09-08 00:00:00")
             )
             .select(reviews_df["review_id"], reviews_df["posted_by.posted_time"], users_df["email"])
             .orderBy(funcs.to_timestamp(reviews_df["posted_by.posted_time"]).asc())
             .limit(5)
)

result_2e.show()
```

```
+------------------+------------------+--------------------+
|         review_id|       posted_time|               email|
+------------------+------------------+--------------------+
|review_aArS-aRvRyG5|2024-08-03 00:00:31|robertsaunders@gm...|
|review_O-V9LubWTBKR|2024-08-03 00:01:00|biancadavis@iclou...|
|review_rNdhyxyxQxud|2024-08-03 00:02:01|andrewwatkins@fox...|
|review_xAEdPYgfSGiS|2024-08-03 00:02:07|walleraudrey@gmai...|
|review_2e4fbC-zSWO8|2024-08-03 00:02:39|riverakelsey@yaho...|
+------------------+------------------+--------------------+
```

25

```
%sql
-- 2.E SQL

select reviews.review_id, reviews.posted_by.posted_time, users.email as email
from reviews join users on reviews.posted_by.user_id = users.user_id
where cast(reviews.posted_by.posted_time as TIMESTAMP) >= '2024-08-03 00:00:00'
  and cast(reviews.posted_by.posted_time as TIMESTAMP) < '2024-09-08 00:00:00'
order by cast(reviews.posted_by.posted_time as TIMESTAMP) asc
limit 5;
```

Table

	review_id	posted_time	email
1	review_aArS-aRvRyG5	2024-08-03 00:00:31	robertsaunders@gmail.com
2	review_O-V9LubWTBKR	2024-08-03 00:01:00	biancadavis@icloud.com
3	review_rNdhyxyxQxud	2024-08-03 00:02:01	andrewwatkins@foxmail.com
4	review_xAEdPYgfSGiS	2024-08-03 00:02:07	walleraudrey@gmail.com
5	review_2e4fbC-zSWO8	2024-08-03 00:02:39	riverakelsey@yahoo.com

5 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

2F

27

```
%python
#2.F DF

emails_upvotes = (upvotes_df
          .join(reviews_df, upvotes_df.review_id == reviews_df.review_id)
          .join(users_df, reviews_df.posted_by.user_id == users_df.user_id)
          .groupBy(reviews_df.review_id, reviews_df.posted_by.user_id, users_df.email)
          .agg(funcs.count(upvotes_df.user_id).alias('num_upvotes'))
          .orderBy(funcs.desc('num_upvotes'))
          .limit(3)
          .select(users_df.email.alias('user_email'), 'num_upvotes'))

emails_upvotes.show()
```

```
+--------------------+-----------+
|          user_email|num_upvotes|
+--------------------+-----------+
|    lance08@mail.com|         20|
|   nancy80@gmail.com|         20|
|williamsonraymond...|         20|
+--------------------+-----------+
```

28

```
%sql
--2.F SQL

SELECT users.email AS user_email,
       COUNT(upvotes.user_id) AS num_upvotes
FROM upvotes
JOIN reviews ON upvotes.review_id = reviews.review_id
JOIN users ON reviews.posted_by.user_id = users.user_id
GROUP BY upvotes.review_id, reviews.posted_by.user_id, users.email
ORDER BY num_upvotes DESC
LIMIT 3;
```

Table

	user_email	num_upvotes
1	kimberlyhiggins@icloud.com	20
2	nancy80@gmail.com	20
3	williamsonraymond@mail.com	20

3 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

2G

30

```
%python
#2.G DF

# Similar to what I've done below with SQL, I will first create the artist_ratings view using
# SQL, I tried with Python but kept getting attribute errors, so for simplicity, I will take this
# approach.
spark.sql("""
CREATE OR REPLACE TEMP VIEW artist_ratings AS
SELECT rec.released_by.artist_user_id, rev.rating,
       u.real_name.first_name, u.real_name.last_name
FROM reviews rev
JOIN upvotes upv ON rev.review_id = upv.review_id
JOIN sessions sess ON rev.posted_by.user_id = sess.user_id
JOIN records rec ON rev.record_id = rec.record_id
JOIN users u ON rec.released_by.artist_user_id = u.user_id
GROUP BY rev.review_id, rev.record_id, rev.rating, rev.posted_by.user_id,
         rec.released_by.artist_user_id,
         u.real_name.first_name, u.real_name.last_name
HAVING COUNT(upv.user_id) >= 5  -- Review needs >= 5 upvotes
   AND COUNT(sess.session_id) >= 275  -- User needs >= 275 sessions
""")


# Now, I will ;oad the artist_ratings view into a dataframe, and use python query on that
artist_ratings_df = spark.table("artist_ratings")

final_result = (artist_ratings_df
    .groupBy('artist_user_id', 'first_name', 'last_name')
    .agg(
        funcs.avg('rating').alias('ACRR'),  # Average rating per artist
        funcs.count('*').alias('num_certified_ratings')  # Count of reviews per artist
    )
    .orderBy(funcs.col('ACRR').desc())  # Order by average rating in descending order
    .limit(5)  # Get top 5 artists
)

final_result.show()
```

```
+----------------+----------+---------+------------------+--------------------+
|   artist_user_id|first_name|last_name|              ACRR|num_certified_ratings|
+----------------+----------+---------+------------------+--------------------+
|user_CbFuaCAaRXuo|      NULL|     NULL|3.2253886010362693|                 386|
|user_A3OUtbXAQhqg|      Ryan|    Young|3.2118380062305296|                 321|
|user_bSebtS-RQveD|      NULL|     NULL|          3.196875|                 320|
|user_ivT6E0p0Tgqe|      NULL|     NULL| 3.181286549707602|                 342|
|user_OIMtKmJgQe2X|    Joshua|    Jones|3.1666666666666665|                 312|
+----------------+----------+---------+------------------+--------------------+
```

```
%sql
--2.G SQL

-- After acquiring the relevant artist_user_id, rating, first_name, last_name values, we
-- can create a view to access the relevant data since the query is getting complex.
CREATE OR REPLACE TEMP VIEW artist_ratings AS
SELECT rec.released_by.artist_user_id, rev.rating,
       u.real_name.first_name, u.real_name.last_name
FROM reviews rev
JOIN upvotes upv ON rev.review_id = upv.review_id
JOIN sessions sess ON rev.posted_by.user_id = sess.user_id
JOIN records rec ON rev.record_id = rec.record_id
JOIN users u ON rec.released_by.artist_user_id = u.user_id
GROUP BY rev.review_id, rev.record_id, rev.rating, rev.posted_by.user_id,
         rec.released_by.artist_user_id,
         u.real_name.first_name, u.real_name.last_name
HAVING COUNT(upv.user_id) >= 5  -- Review needs >= 5 upvotes
   AND COUNT(sess.session_id) >= 275; -- User needs >= 275 sessions


SELECT
    artist_user_id,
    AVG(rating) AS ACRR,
    COUNT(*) AS num_certified_ratings,
    first_name,
    last_name
FROM artist_ratings
GROUP BY
    artist_user_id,
    first_name,
    last_name
ORDER BY ACRR DESC
LIMIT 5;
```

Table

	artist_user_id	ACRR	num_certified_ratings	first_name	last_name
1	user_CbFuaCAaRXuo	3.2253886010362693	386	null	null
2	user_A3OUtbXAQhqg	3.2118380062305296	321	Ryan	Young
3	user_bSebtS-RQveD	3.196875	320	null	null
4	user_ivT6E0p0Tgqe	3.181286549707602	342	null	null
5	user_OIMtKmJgQe2X	3.1666666666666665	312	Joshua	Jones

5 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.


```
# 3
# Go BACK to the beginning and run the whole notebook on the full dataset, both in Databricks Community Version and
Databricks Premium Version.
# Write down your observations about the differences of using single-node community version vs. using multi-node premi
version of Databricks.

# Using the community version, it took about 17 minutes to run the whole notebook on the full dataset. With the premiu
# version, it took about 4 minutes to run the whole notebook on the full dataset.
```

3: Observations

Using the full dataset in the community version with single node, the notebook ran in about 17 minutes.

Using the full dataset in the PREMIUM version with 3 nodes, the notebook ran in about 4 minutes.

The execution times were faster across most cells, particularly for the cells for 2G. In the community version, each cell for problem 2G took about 7 minutes to run, while each of them took less than 1 minute in the premium version.

So for the full dataset, we were able to run it in 4 minutes using the premium version with 3 nodes, significantly less than the 17 minutes using the single node version.

34