Sistemas Distribuidos

Spark II

Carlos Gómez Gómez -1460840
Julio Velásquez Cárdenas - 1397896

Problema 1

Definir un Cross Join (producto cartesiano) entre los DataFrame customers y products, con las columnas "cust_id" y "product_id", agregando una columna "score" que contenga un valor random de tipo Double.

```
scala> val customers = spark.read.option("header", true).option("inferSchema", true).csv("/home/alumno/Escritori
o/Spark2/data-small/customers.csv")
customers: org.apache.spark.sql.DataFrame = [cust_id: int, is_male: boolean ... 1 more field]
```

```
scala> val scores = (customers.crossJoin(products).select("cust_id","product_id").withColumn("score",rand()))
scores: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 1 more field]
```

```
scala> info(scores)
root
 |-- cust_id: integer (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- score: double (nullable = false)

+-------+----------+--------------------+
|cust_id|product_id|               score|
+-------+----------+--------------------+
| 100000|      1000| 0.18531872528540094|
| 100001|      1000|0.0083727259555538174|
| 100002|      1000|  0.7347864748745944|
| 100003|      1000| 0.42174083646950244|
| 100004|      1000|  0.5208993531123938|
+-------+----------+--------------------+
only showing top 5 rows
```

```
scala> scores.count
res5: Long = 15000000
```

Problema 2

El proceso de optimización de queries en Apache Spark requiere una comprensión cabal de los planes de ejecución. En este punto pedimos se provea el plan de ejecución físico (o "SparkPlan") del DataFrame "scores". Toda la información necesaria se puede encontrar en https://jaceklaskowski.gitbooks.io/masteringspark-sql/content/spark-sql-SparkPlan.html
Proveer el plan de ejecución del DataFrame "scores".

El plan de ejecución del DataFrame ''scores'', se puede visualizar mediante los dos siguientes comandos:

Scores.explain

```
scala> scores.explain
== Physical Plan ==
*(1) Project [cust_id#16, product_id#38, rand(4475100773560392423) AS score#64]
+- BroadcastNestedLoopJoin BuildLeft, Cross
   :- BroadcastExchange IdentityBroadcastMode, [id=#85]
   :  +- FileScan csv [cust_id#16] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[fil
e:/home/alumno/spark-3.0.1-bin-hadoop2.7/data-small/customers.csv], PartitionFilters: [], PushedFilters: [], Rea
dSchema: struct<cust_id:int>
   +- FileScan csv [product_id#38] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[fil
e:/home/alumno/spark-3.0.1-bin-hadoop2.7/data-small/products.csv], PartitionFilters: [], PushedFilters: [], Read
Schema: struct<product_id:int>
```

scores.queryExecution.sparkPlan

```
scala> scores.queryExecution.sparkPlan
res5: org.apache.spark.sql.execution.SparkPlan =
Project [cust_id#16, product_id#38, rand(4475100773560392423) AS score#64]
+- BroadcastNestedLoopJoin BuildLeft, Cross
   :- FileScan csv [cust_id#16] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[file:/
home/alumno/spark-3.0.1-bin-hadoop2.7/data-small/customers.csv], PartitionFilters: [], PushedFilters: [], ReadSc
hema: struct<cust_id:int>
   +- FileScan csv [product_id#38] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[fil
e:/home/alumno/spark-3.0.1-bin-hadoop2.7/data-small/products.csv], PartitionFilters: [], PushedFilters: [], Read
Schema: struct<product_id:int>
```

Problema 3

Una de las formas a mejorar el rendimiento de las queries es incrementar el nivel de paralelismo de los RDDs en los que se traduce un DataFrame (recordar que el DataFrame es tan solo una API de alto nivel). Para esto definimos un número de particiones. No hay una fórmula mágica para este número, por lo que para resolver este problema pedimos crear un nuevo DataFrame, con nombre "scoresRepartitioned", que se construya de igual manera que "scores", pero indicando 10 particiones para el DataFrame customers (hint: función "repartition").

Proveer el código para este nuevo DataFrame y el plan de ejecución.

Para crear el nuevo DataFrame particionado, se utiliza el siguiente comando:
val scoresRepartitioned = scores.repartition(10)

```
scala> val scoresRepartitioned = scores.repartition(10)
scoresRepartitioned: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [cust_id: int, product_id: int ...
 1 more field]

scala> info(scoresRepartitioned)
root
 |-- cust_id: integer (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- score: double (nullable = false)

+-------+----------+-------------------+
|cust_id|product_id|              score|
+-------+----------+-------------------+
| 100168|      5296|0.28849569505601147|
| 101010|      2438| 0.4660082981098189|
| 100448|      5747| 0.7683091623323203|
| 102952|      1019|0.30526587580860975|
| 102181|      1842|0.13794323030851308|
+-------+----------+-------------------+
only showing top 5 rows
```

Si observamos el esquema de particionado de la salida se puede ver que el esquema que se utiliza es RoundRobinPartitioning

```
scala> scoresRepartitioned.queryExecution.sparkPlan
res1: org.apache.spark.sql.execution.SparkPlan =
Exchange RoundRobinPartitioning(10), false, [id=#51]
+- Project [cust_id#16, product_id#38, rand(3835008529393167129) AS score#64]
   +- BroadcastNestedLoopJoin BuildLeft, Cross
      :- FileScan csv [cust_id#16] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[fil
e:/home/alumno/spark-3.0.1-bin-hadoop2.7/data-small/customers.csv], PartitionFilters: [], PushedFilters: [], Rea
dSchema: struct<cust_id:int>
      +- FileScan csv [product_id#38] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[
file:/home/alumno/spark-3.0.1-bin-hadoop2.7/data-small/products.csv], PartitionFilters: [], PushedFilters: [], R
eadSchema: struct<product_id:int>
```

Se han ejecutado los comandos scores.count() y scoresRepartitiones.count(), y se observa que scores.count(), se ejecuta más rápido que la versión con repartition.

```
scala> scoresRepartitioned.count()
res2: Long = 15000000

scala> scores.count()
res3: Long = 15000000
```

Problema 4

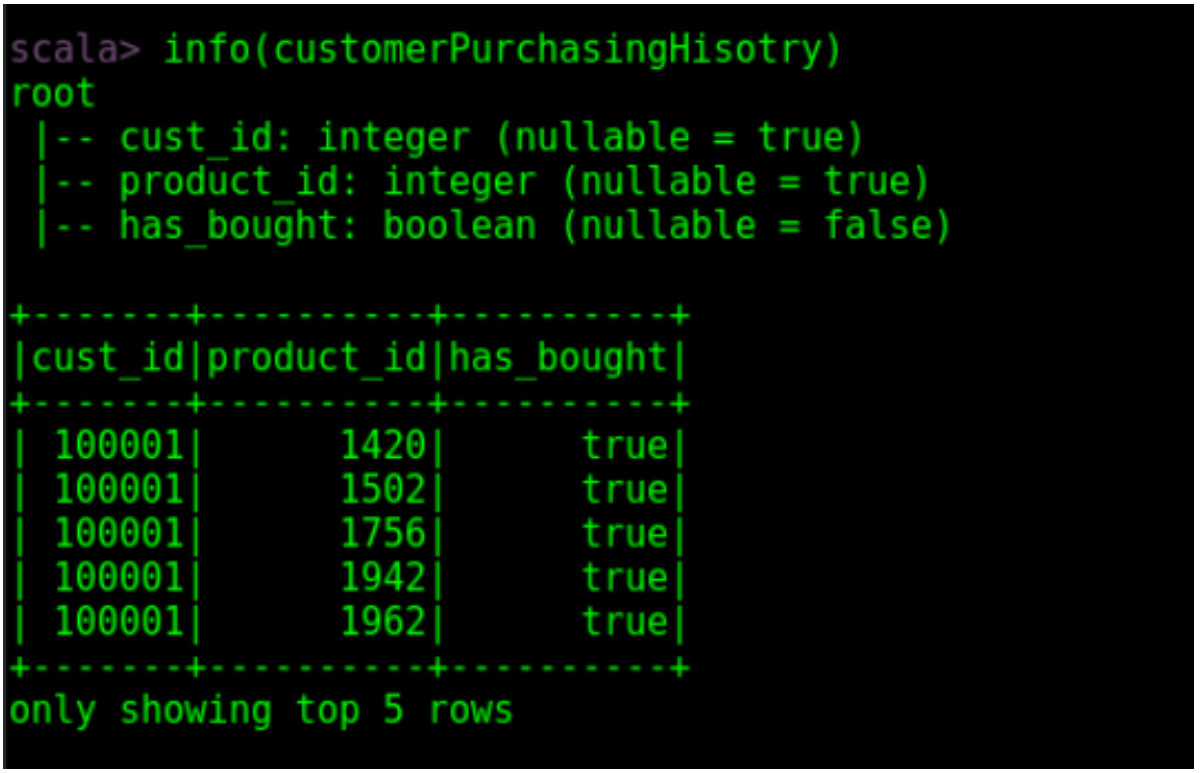
Crear un DataFrame, llamado “customerPurchasingHistory” que contenga la historia de compra de los clientes.

Para resolver el problema hay que construir una sentencia SQL que lea la vista “transactions_view”, haciendo un GROUP BY de las columnas del customer y producto. La primera parte del SQL es: SELECT cust_id, product_id, TRUE AS has_bought

Para generar el nuevo DataFrame se utiliza el siguiente comando:
val customerPurchasingHisotry = spark.sql(“SELECT cust_id, product_id, TRUE AS has_bought FROM transactions_view GROUP BY cust_id, product_it ”)

```
scala> val customerPurchasingHisotry = spark.sql("SELECT cust_id, product_id, TRUE AS has_bought FROM transactio
ns_view GROUP BY cust_id, product_id")
customerPurchasingHisotry: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 1 more field]
```

El resultado del DataFrame generado es el siguiente:



Problema 5

Crear una vista para los DataFrame "customerPurchasingHistory" y "score" (o "scoreRepartitioned"), llamados "customer_purchasing_history" y "scores", respectivamente.

Proveer el código.

```
scala> scores.createOrReplaceTempView("scores")

scala> spark.sqk(
     | """
     | SELECT COUNT(*) FROM
     |
     |
You typed two blank lines.  Starting a new command.

scala> scores.createOrReplaceTempView("scores_view")

scala> spark.sql(
     | """
     | SELECT COUNT(*) FROM scores_view
     | """).show()
+--------+
|count(1)|
+--------+
|15000000|
+--------+
```

```
scala> val customerPurchasingHisotry = spark.sql("SELECT cust_id, product_id, TRUE AS has_bought FROM transactio
ns_view GROUP BY cust_id, product_id")
customerPurchasingHisotry: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 1 more field]

scala> customer
customerPurchasingHisotry   customers

scala> customerPurchasingHisotry.createOrReplaceTempView("customerPuchasingHistory")

scala> spark.sql(
     | """
     | SELECT COUNT(*) FROM
     |
     |
You typed two blank lines.  Starting a new command.

scala> customerPurchasingHisotry.createOrReplaceTempView("customer_purchasing_history")

scala> spark.sql(
     | """
     | SELECT COUNT(*) FROM customer_purchasing_history
     | """).show()
+--------+
|count(1)|
+--------+
| 7445000|
+--------+
```

Los siguientes ejercicios han sido realizados de forma local ya que no era suficiente con la memoria de la máquina virtual.

Problema 6

Ahora vamos a hacer un FULL JOIN entre la vista "scores" y "customer_purchasing_history". Nos interesa que el DataFrame resultante, llamado "scoresWithPurchasingHistory", tenga las columnas "cust_id", "product_id", "score", y una columna llamada "has_bought", que sea FALSE si tiene un valor NULL en la vista "customer_purchasing_history" o el valor que hay en la vista "customer_purchasing_history" (hint: ver funciones "if" y "isnul")

Para generar el nuevo DataFrame se utiliza el siguiente comando:
val scoreswithPurchasingHistory = spark.sql("SELECT s.cust_id, s.product_id, s.score, if(isnull(c.has_bought), FALSE, c.has_bought) AS has_bought FROM scores s FULL JOIN customer_purchasing_history c ON s.cust_id = c.cust_id AND s.product_id = c.product_id")

```
scala> val scoreswithPurchasingHistory = spark.sql("SELECT s.cust_id, s.product_id, s.score, if(isnull(c.has_bought), FALSE, c.has_bought) AS has_bought FROM scores s FULL JOIN customer_purchasing_history c ON s.cust_id = c.cust_id AND s
.product_id = c.product_id")
scoreswithPurchasingHistory: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 2 more fields]
```

El resultado del DataFrame generado es el siguiente:

```
scala> info(scoreswithPurchasingHistory)
root
 |-- cust_id: integer (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- score: double (nullable = true)
 |-- has_bought: boolean (nullable = true)

+-------+----------+-------------------+----------+
|cust_id|product_id|              score|has_bought|
+-------+----------+-------------------+----------+
| 100000|      1132| 0.7683074317043797|     false|
| 100000|      1474|0.25841466337951924|     false|
| 100000|      1515| 0.7306247703107563|     false|
| 100000|      1543|0.03580177220170233|     false|
| 100000|      1660| 0.5626099636131205|     false|
+-------+----------+-------------------+----------+
only showing top 5 rows
```

Problema 7

Data la información de los pasos anteriores, vamos a recomendar productos a aquellos customers que no han comprado el producto.Para esto, utilizaremos el filtro "has_bought = FALSE"sobre el DataFrame "scoresWithPurchasingHistory". Al DataFrame resultando lo llamaremos "recommendations". Hay varias opciones para implementar este filtro.
Proveer el código y la salida de "info(recommendations)".

Crearemos el filtro necesario con el siguiente comando. Este nos servirá para escoger a todos los clientes que no hayan comprado ningún producto.

```
scala> val filtro = "has_bought = FALSE"
filtro: String = has_bought = FALSE
```

Usamos el comando *.where()* para realizar este filtrado en el DataFrame *recommendations* y vemos el resultado obtenido con *info()*.

```
scala> val recommendations = scoreswithPurchasingHistory.where(filtro)
recommendations: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [cust_id: int, product_id: int ... 2 more fields]
```

```
scala> info(recommendations)
root
 |-- cust_id: integer (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- score: double (nullable = true)
 |-- has_bought: boolean (nullable = true)

+-------+----------+-------------------+----------+
|cust_id|product_id|              score|has_bought|
+-------+----------+-------------------+----------+
| 100000|      1132| 0.7683074317043797|     false|
| 100000|      1474|0.25841466337951924|     false|
| 100000|      1515| 0.7306247703107563|     false|
| 100000|      1543|0.03580177220170233|     false|
| 100000|      1660| 0.5626099636131205|     false|
+-------+----------+-------------------+----------+
only showing top 5 rows
```

¿Cuál es la ratio de "recommendations" y el total de filas de"scoresWithPurchasingHistory"?

```
scala> scoreswithPurchasingHistory.count
res15: Long = 15000000

scala> recommendations.count
res16: Long = 7555000
```

El ratio es de casi la mitad de los customers.

Problema 8

Para este último problema, vamos a recomendar a los customers los 3 top products (de acuerdo a su score) por "brand".

Proveer la query completa, el código necesario para ejecutarla y las primeras 20 filas del DataFrame resultante.

La query necesaria para realizar el problema es la siguiente. La condición que faltaba era la *WHERE rank BETWEEN 1 AND 3*.

```
scala> val sqlQuery = s"""
     | WITH ranks AS (
     | SELECT
     | s.cust_id, s.product_id, s.score, s.brand, $ranking
     | FROM scores_recommendations_w_product_properties s
     | )
     | SELECT * FROM ranks WHERE rank BETWEEN 1 AND 3
     | """
sqlQuery: String =
"
WITH ranks AS (
SELECT
s.cust_id, s.product_id, s.score, s.brand, RANK() OVER (PARTITION BY cust_id, brand ORDER BY score DESC) AS rank
FROM scores_recommendations_w_product_properties s
)
SELECT * FROM ranks WHERE rank BETWEEN 1 AND 3
"
```

Ejecutamos la query creando el DataFrame *top* y vemos el resultado obtenido convirtiendo este en una vista SQL temporal y usando el comando *show()*.

```
scala> val top = spark.sql(sqlQuery)
top: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 3 more fields]
```

```
scala> top.createOrReplaceTempView("top")

scala> top.show()
+-------+----------+------------------+-------+----+
|cust_id|product_id|             score|  brand|rank|
+-------+----------+------------------+-------+----+
| 100490|      3066|0.9987848355362482|premium|   1|
| 100490|      3508|0.9982934845688217|premium|   2|
| 100490|      4082| 0.997429609751207|premium|   3|
| 100497|      2181|0.9993102828918152|premium|   1|
| 100497|      5873|  0.99889768912067|premium|   2|
| 100497|      3785|0.9987582767773848|premium|   3|
| 100561|      5973|0.9991377172046426| luxury|   1|
| 100561|      4079|0.9987071257689649| luxury|   2|
| 100561|      1612|0.9981603208349445| luxury|   3|
| 100993|      2471| 0.999925114089399| luxury|   1|
| 100993|      2360|0.9996156668172836| luxury|   2|
| 100993|      3648|0.9992689581778793| luxury|   3|
| 101231|      4314|0.9995390905388906|  basic|   1|
| 101231|      2922|0.9993405250808322|  basic|   2|
| 101231|      1338|0.9986770117680874|  basic|   3|
| 101340|      5666|0.9998864487008675|premium|   1|
| 101340|      3154|0.9994856485060524|premium|   2|
| 101340|      5684|0.9992784807075688|premium|   3|
| 101426|      4171|0.9994408403538513|  basic|   1|
| 101426|      2300|0.9978408603839024|  basic|   2|
+-------+----------+------------------+-------+----+
only showing top 20 rows
```