Universitat Autònoma de Barcelona
Enginyeria Informàtica
Curs 2020-2021
Sistemas Distribuidos
Análisis de datos con Apache Spark - II

Datos

Profesores de Prácticas: Sergio Villar, Carlos Montemuiño

Material de las prácticas

Para las prácticas, en las máquinas Linux, la documentación será los *HOWTOs* y *manpages* del sistema. La presente guía es orientativa (no funcionará un copy paste sin saber lo que se hace).

En las prácticas se usarán máquinas virtuales (*Virtual Machines VM*) con OpenNebula. A OpenNebula se puede acceder desde todas las máquinas del laboratorio. Existen varios templates con sistema operativo Linux ya instalado.

Para acceder al entorno de prácticas se ha habilitado un acceso remoto a los servicios OpenNebula en esta dirección:

http://nebulacaos.uab.cat:8443

La documentación básica sobre el entorno Open Nebula y cómo utilizarlo puede consultarse en el manual de Operación: http://docs.opennebula.org/5.10/operation/index.html

1. Procesamiento de datos con Spark SQL y DataFrame API

En esta práctica vamos a alejarnos del procesamiento de datos con RDD y utilizaremos otras APIs de alto nivel: Spark SQL y DataFrame. De esta forma, implementaremos lógica de negocio de manera más intuitiva, sin tener que lidiar con los RDDs, que son el API de más bajo nivel (low-level API) que podemos encontrar en Spark.

Documentación: https://spark.apache.org/sql/

Si bien no vamos a trabajar realmente con “big data”, intentaremos acercarnos lo máximo posible según las características del nodo que hemos creado en la práctica de Spark I (2 GB de RAM).

Descargamos los datos desde el campus virtual o desde ese link: https://we.tl/t-iQ8U9cjgDY

Si el nodo no puede procesar los datos de entrada, entonces, o bien ejecutar todo en local (a efectos de la práctica no importa), o utilizar un juego de datos más pequeño: https://we.tl/t-g65SRAuxMf

Luego descomprimimos los ficheros y los ponemos en un directorio de trabajo.

```
.
└── data
    ├── customers.csv
    ├── products.csv
    └── transactions_sample_0.5_pq
        ├── _SUCCESS
        └── part-00000-77190630-54dd-4c1a-ace0-591328645437-c000.snappy.parquet

2 directories, 4 files
```

El resultado serán dos ficheros CSV y un directorio con ficheros en formato Apache Parquet:

- customers.csv: fichero con 5000 clientes random
- products.csv: fichero con 10000 productos random
- transactions_sample_0.50_pq: directorio con transacciones de compra generadas para el 50% de los clientes

---

Para el juego de datos más pequeño, tenemos 3000 clientes y 5000 productos

---

La lectura de ficheros CSV es muy frágil. Además, se pretende mostrar como trabajar con Apache Parquet, un formato binario con una estructura "columnar". Documentación: https://spark.apache.org/docs/3.0.0/sql-data-sources-parquet.html

- El procesamiento de datos con Apache Parquet es un orden de magnitud más eficiente que con CSV descomprimido (más información: https://blog.cloudera.com/benchmarking-apache-parquet-the-allstate-experience/)
- Qué sea "columnar" significa que los datos son seleccionados por columas (vs. filas/"tabular" en caso de CSV). Con esto, los filtros tipo SQL que implementemos serán super eficientes. Alejarse de los formatos tabular descomprimidos (no incluimos Avro!) hacia Parquet, es una de las decisiones más fundamentales que podemos realizar para mejorar el rendimiento en Spark.

Carga de Datos

Ahora vamos a abrir un Spark Shell y cargar los datos de entrada.

```
scala> import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.DataFrame

scala> def info(df: DataFrame)= {
     |     // Helper method to avoid repeating commands
     |     df.printSchema()
     |     df.show(5)
     | }
info: (df: org.apache.spark.sql.DataFrame)Unit

scala> val customers = spark.read.option("header", true).option("inferSchema", true).csv("customers.csv")
customers: org.apache.spark.sql.DataFrame = [cust_id: int, is_male: boolean ... 1 more field]

scala> val products = spark.read.option("header", true).option("inferSchema", true).csv("products.csv")
products: org.apache.spark.sql.DataFrame = [product_id: int, food: boolean ... 3 more fields]

scala> val transactions = spark.read.parquet("transactions_sample_0.5_pq")
transactions: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 1 more field]
```

Customers

Tenemos un DataFrame con 5000 clientes, cada uno con 3 columnas:

```
scala> info(customers)
root
 |-- cust_id: integer (nullable = true)
 |-- is_male: boolean (nullable = true)
 |-- location: integer (nullable = true)

+-------+-------+--------+
|cust_id|is_male|location|
+-------+-------+--------+
| 100000|   true|       1|
| 100001|  false|       2|
| 100002|   true|       3|
| 100003|  false|       4|
| 100004|   true|       1|
+-------+-------+--------+
only showing top 5 rows


scala> customers.count
res1: Long = 5000
```

Products

Tenemos 10000 productos, cada uno con 5 columnas:

```
scala> info(products)
root
 |-- product_id: integer (nullable = true)
 |-- food: boolean (nullable = true)
 |-- price: double (nullable = true)
 |-- brand: string (nullable = true)
 |-- at_location: string (nullable = true)

+----------+-----+------+-------+-----------+
|product_id| food| price|  brand| at_location|
+----------+-----+------+-------+-----------+
|      1000| true| 68.91|premium|         [1]|
|      1001|false| 61.32| luxury|      [1, 2]|
|      1002| true| 85.16|  basic|   [1, 2, 3]|
|      1003|false|125.91|premium|[1, 2, 3, 4]|
|      1004| true| 91.41|premium|         [1]|
+----------+-----+------+-------+-----------+
only showing top 5 rows


scala> products.count
res3: Long = 10000
```

Transactions

Por ultimo, temenos cerca de 25 millones de transacciones, simuladas a lo largo de todo el año 2020:

```
scala> info(transactions)
root
 |-- cust_id: integer (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- date: timestamp (nullable = true)

+-------+----------+--------------------+
|cust_id|product_id|                date|
+-------+----------+--------------------+
| 100118|      1000|2020-04-28 08:37:...|
| 102551|      1000|2020-02-29 23:41:...|
| 101479|      1000|2020-04-07 07:52:...|
| 104229|      1000|2020-04-26 17:27:...|
| 102333|      1000|2020-04-21 09:40:...|
+-------+----------+--------------------+
only showing top 5 rows


scala> transactions.count
res3: Long = 24670000
```

Spark SQL

Vamos a crear una vista en el DataFrame “transactions” para utilizar Spark SQL:

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

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

scala> transactions.createOrReplaceTempView("transactions_view")

scala> spark.sql(
     | """
     | SELECT COUNT(*) FROM transactions_view
     | """).show()
+--------+
|count(1)|
+--------+
|24670000|
+--------+
```

Tener en cuenta que la función “sql” retorna un DataFrame:

```
scala> val df = spark.sql(
     | """
     | SELECT * FROM transactions_view LIMIT 100
     | """)
df: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 1 more field]
```

2. Resolución de Problemas

Nota 1: La resolución de esta práctica requiere de una utilización adecuada de la memoria principal y del espacio en disco para los ficheros de swap. Solamente se precisa utilizar el `***spark-shell***`; por lo tanto, ***no ejecutar el comando `bin/start-master.sh`*** como en la práctica anterior.

Nota 2: La práctica no tiene porqué realizarse en un nodo del NebulaCaos, sino que podéis hacerla en local. Si lo hacéis en un nodo del NebulaCaos, entonces deberéis utilizar el juego de datos **data-small**. Si hacéis todo bien, el consumo de memoria será menor al 70% para los ejercicios 1-7 y menor al 95% para el ejercicio 8. Eso lo podéis comprobar con otro terminal ejecutando el comando `**top**`.

Problema 1 (2 pts)

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.

Podéis utilizar scala.util.Random y luego el método "nextDouble()"

Para agregar la nueva columna, mirar la función "rand" en https://spark.apache.org/docs/3.0.0/api/scala/org/apache/spark/sql/functions$.html

La salida os quedaría similar a:

```
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|0.13684852773713063|
| 100001|      1000|  0.504859233791781|
| 100002|      1000| 0.9126436351522543|
| 100003|      1000| 0.7711029464793209|
| 100004|      1000| 0.7697818215301752|
+-------+----------+-------------------+
only showing top 5 rows
```

Cómo validación, verificar que "scores.count()" da 50 millones (5000 customers x 10000 products !!!).

>>>>>> Para el juego de dato más pequeño, serían 15 millones de registros <<<<<<

Proporcionar el código utilizado para generar el DataFrame requerido y una captura de la salida de "info(scores)".

Problema 2 (0.5 pts)

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/mastering-spark-sql/content/spark-sql-SparkPlan.html

Proveer el plan de ejecución del DataFrame "scores".

A modo de validación, la salida comenzará con:

```
res24: org.apache.spark.sql.execution.SparkPlan =
Project [cust_id#16, product_id#38, 0.28776392391427885 AS score#210]
```

Problema 3 (0.5 pts)

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.

Indicar cuál es el esquema de particionado que aparece en la salida:

- HashPartitioning
- PartitioningCollection
- BroadcastPartitioning
- RoundRobinPartitioning
- RangePartitioning
- SinglePartitioning
- UnknownPartitioning

Responder si es verdadero o falso que "scoresRepartitioned.count()" se observa más rápido que "scores.count()" (no hace falta medir).

Problema 4 (2 pts)

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`**

Se debería obtener una salida parecida a la siguiente cuando se ejecuta **info(customerPurchasingHistory)**:

```
customerPurchasingHistory: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 1 more field]

scala> info(customerPurchasingHistory)
root
 |-- cust_id: integer (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- has_bought: boolean (nullable = false)

+-------+----------+----------+
|cust_id|product_id|has_bought|
+-------+----------+----------+
| 100001|      1420|      true|
| 100001|      1502|      true|
| 100001|      1756|      true|
| 100001|      1942|      true|
| 100001|      1962|      true|
+-------+----------+----------+
only showing top 5 rows
```

Ejemplos de GROUP BY: https://spark.apache.org/docs/3.0.0/sql-ref-syntax-qry-select-groupby.html

Mostrar el SQL generado, la salida de "info(customerPurchasingHistory)" y la cantidad de filas que tiene el DataFrame.

Si por algún motivo no podéis completar el ejercicio 4, en el CV tenéis disponible el fichero `**customer_purchasing_history_pq.tar.gz**` con el resultado del ejercicio 4 en formato Apache Parquet. Lo que tenéis que hacer es descomprimirlo, por ejemplo en `**/home/alumno/Downloads/customer_purchasing_history_pq**`, y luego, poner en el spark-shell: **val customerPurchasingHistory = spark.read.parquet("/home/alumno/Downloads/customer_purchasing_history_pq ")**

Problema 5 (0.5 pts)

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

Proveer el código.

Problema 6 (2 pts)

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")

El resultado se aproximará a:

```
scoresWithPurchasingHistory: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 2 more fields]

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

[4965.967s][warning][gc,alloc] Executor task launch worker for task 320: Retried waiting for GCLocker too often al
20/11/15 23:10:56 WARN TaskMemoryManager: Failed to allocate a page (4194304 bytes), try again.
[4965.983s][warning][gc,alloc] Executor task launch worker for task 320: Retried waiting for GCLocker too often al
[4965.983s][warning][gc,alloc] Executor task launch worker for task 325: Retried waiting for GCLocker too often al
20/11/15 23:10:56 WARN TaskMemoryManager: Failed to allocate a page (4194304 bytes), try again.
20/11/15 23:10:56 WARN TaskMemoryManager: Failed to allocate a page (4194304 bytes), try again.
+-------+----------+--------------------+----------+
|cust_id|product_id|               score|has_bought|
+-------+----------+--------------------+----------+
| 100000|      1132|0.057746031258229635|     false|
| 100000|      1474|0.057746031258229635|     false|
| 100000|      1515|0.057746031258229635|     false|
| 100000|      1543|0.057746031258229635|     false|
| 100000|      1660|0.057746031258229635|     false|
+-------+----------+--------------------+----------+
only showing top 5 rows
```

Ejemplos de JOIN: https://spark.apache.org/docs/3.0.0/sql-ref-syntax-qry-select-join.html

Proveer el código y la captura de “info(scoresWithPurchasingHistory)”.

Problema 7 (0.5 pts)

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)”.

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

Problema 8 (2 pts)

Para este último problema, vamos a recomendar a los customers los 3 top products (de acuerdo a su score) por “brand". Primero enriquecemos las recomendaciones con los detalles de los productos:

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

scala> val scoresRecommendationsWithProductProperties = recommendations.join(products, "product_id")
scoresRecommendationsWithProductProperties: org.apache.spark.sql.DataFrame = [product_id: int, cust_id:

scala> scoresRecommendationsWithProductProperties.createOrReplaceTempView("scores_recommendations_w_pro

scala> info(scoresRecommendationsWithProductProperties)
root
 |-- product_id: integer (nullable = true)
 |-- cust_id: integer (nullable = true)
 |-- score: double (nullable = true)
 |-- has_bought: boolean (nullable = true)
 |-- food: boolean (nullable = true)
 |-- price: double (nullable = true)
 |-- brand: string (nullable = true)
 |-- at_location: string (nullable = true)

+----------+-------+--------------------+----------+-----+------+-------+------------+
|product_id|cust_id|               score|has_bought| food| price|  brand| at_location|
+----------+-------+--------------------+----------+-----+------+-------+------------+
|      1132| 100000|0.057746031258229635|     false| true| 89.64|premium|         [1]|
|      1474| 100000|0.057746031258229635|     false| true|128.27|  basic|   [1, 2, 3]|
|      1515| 100000|0.057746031258229635|     false|false| 63.59|premium|[1, 2, 3, 4]|
|      1543| 100000|0.057746031258229635|     false|false| 89.86| luxury|[1, 2, 3, 4]|
|      1660| 100000|0.057746031258229635|     false| true| 71.75| luxury|         [1]|
+----------+-------+--------------------+----------+-----+------+-------+------------+
only showing top 5 rows
```

Para calcular el ranking de las recomendaciones, podemos utilizar lo siguiente:

```
scala> val ranking="RANK() OVER (PARTITION BY cust_id , brand ORDER BY score DESC) AS rank"
ranking: String = RANK() OVER (PARTITION BY cust_id , brand ORDER BY score DESC) AS rank
```

Si utilizamos esta fórmula, entonces podemos agregar una columna “rank" para luego seleccionar las filas que tengan “rank” entre 1 y 3:

```
scala> val sqlQuery = s"""
     | SELECT
     |     s.cust_id, s.product_id, s.score, s.brand, $ranking
     | FROM scores_recommendations_w_product_properties s
     | """
sqlQuery: String =
"
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
"

scala> spark.sql(sqlQuery)
res54: org.apache.spark.sql.DataFrame = [cust_id: int, product_id: int ... 3 more fields]

scala> info(res54)
root
 |-- cust_id: integer (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- score: double (nullable = true)
 |-- brand: string (nullable = true)
 |-- rank: integer (nullable = true)

[7147.120s][warning][gc,alloc] Executor task launch worker for task 1028: Retried waiting for GCLocker too often alloc
20/11/15 23:47:17 WARN TaskMemoryManager: Failed to allocate a page (16777216 bytes), try again.
[7156.485s][warning][gc,alloc] Executor task launch worker for task 1079: Retried waiting for GCLocker too often alloc
20/11/15 23:47:26 WARN TaskMemoryManager: Failed to allocate a page (16777216 bytes), try again.
+-------+----------+-------------------+-------+----+
|cust_id|product_id|              score|  brand|rank|
+-------+----------+-------------------+-------+----+
| 100129|      1335|0.05774631258229635|premium|   1|
| 100129|      2162|0.05774631258229635|premium|   1|
| 100129|      3922|0.05774631258229635|premium|   1|
| 100129|      4957|0.05774631258229635|premium|   1|
| 100129|      5025|0.05774631258229635|premium|   1|
+-------+----------+-------------------+-------+----+
only showing top 5 rows
```

Ahora tenemos todos los “building blocks”. Para responder a la pregunta, tenemos que encontrar una condición extra en la siguiente query:

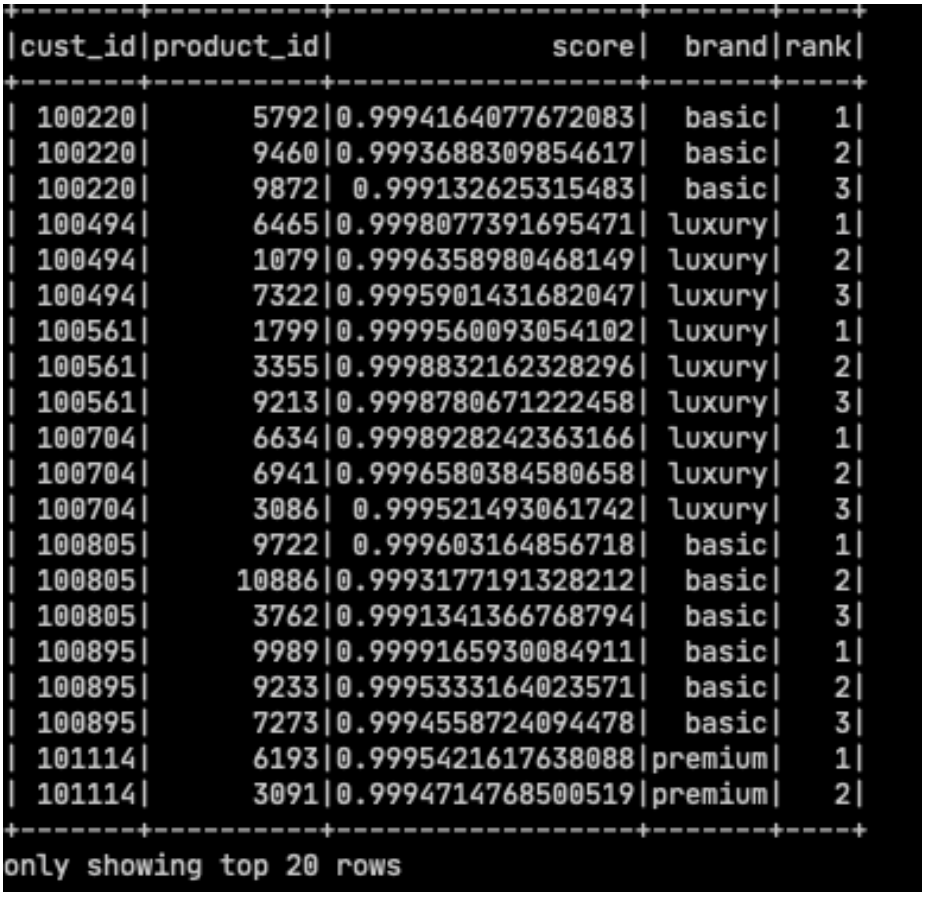
```
scala> val sqlQuery = s"""
     | WITH ranks AS (
     | SELECT
     |     s.cust_id, s.product_id, s.score, s.brand, $ranking
     | FROM scores_recommendations_w_product_properties s
     | )
     | SELECT * FROM ranks WHERE "here_you_put_the_missing_condition!!!!"
     | """
```

IMPORTANTE: la función query anterior no es sintácticamente correcta. Parte del ejercicio es darse cuenta del error.

Documentación relacionada: https://spark.apache.org/docs/3.0.0/sql-ref-syntax-qry-select-cte.html

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

El resultado será similar a:



```
+-------+----------+------------------+-------+----+
|cust_id|product_id|             score|  brand|rank|
+-------+----------+------------------+-------+----+
| 100220|      5792|0.9994164077672083|  basic|   1|
| 100220|      9460|0.9993688309854617|  basic|   2|
| 100220|      9872| 0.999132625315483|  basic|   3|
| 100494|      6465|0.9998077391695471| luxury|   1|
| 100494|      1079|0.9996358980468149| luxury|   2|
| 100494|      7322|0.9995901431682047| luxury|   3|
| 100561|      1799|0.9999560093054102| luxury|   1|
| 100561|      3355|0.9998832162328296| luxury|   2|
| 100561|      9213|0.9998780671222458| luxury|   3|
| 100704|      6634|0.9998928242363166| luxury|   1|
| 100704|      6941|0.9996580384580658| luxury|   2|
| 100704|      3086| 0.999521493061742| luxury|   3|
| 100805|      9722| 0.999603164856718|  basic|   1|
| 100805|     10886|0.9993177191328212|  basic|   2|
| 100805|      3762|0.9991341366768794|  basic|   3|
| 100895|      9989|0.9999165930084911|  basic|   1|
| 100895|      9233|0.9995333164023571|  basic|   2|
| 100895|      7273|0.9994558724094478|  basic|   3|
| 101114|      6193|0.9995421617638088|premium|   1|
| 101114|      3091|0.9994714768500519|premium|   2|
+-------+----------+------------------+-------+----+
only showing top 20 rows
```