Spark SQL y DataFrame

Competencias

- Conocer los distintos componentes de la API de Spark.
- Implementar conexiones generalizadas mediante el objeto `SparkSession`.
- Generar flujos de trabajo con Spark SQL.
- Conocer el objeto `pyspark.sql.dataframe.DataFrame` y sus componentes.
- Generar consumos de distintos datos hacia un objeto `pyspark.sql.dataframe.DataFrame`.
- Conocer los principales modos de trabajo con `pyspark.sql.dataframe.DataFrame`.
- Importar y exportar archivos en formato `parquet`.

Motivación

Hasta el momento tenemos conocimiento sobre el modo de uso de Spark: partimos por generar una conexión dentro de una aplicación, mediante la cual delegaremos acciones a una serie de ejecutores. Sin profundizar mucho en cómo trabaja a nivel de redes, preferimos adentrarnos en el uso de la principal abstracción de Spark: los Resilient Distributed Datasets. Estos permiten encapsular una serie de transformaciones (instrucciones que se compilan en un Grafo Acíclico Dirigido) y posteriormente se ejecutan cuando llamamos una acción.

Parte del motivo que explica la prevalencia de Spark en la industria es el hecho que también presenta una serie de aplicaciones en su API. Mediante estas, se provee de un conjunto de buenas prácticas y procedimientos para que el desarrollador no tenga que implementar casos de uso específico como se realizaba en MapReduce primitivo.

En la siguiente imagen se ejemplifica cómo funcionan las API's de Spark. Siempre tendremos un caso de uso el cual delimitará qué herramienta implementar. Para esta lectura nos centraremos en el consumo de datos estructurados, para lo cual haremos uso de Spark SQL y DataFrame.

Cabe destacar que toda implementación específica de las API tiene que pasar por los componentes Core de Spark. Esta contendrá la funcionalidad básica de Spark incluídos los componentes para la calendarización de tareas, administración de memorias, recuperación de fallas e interacción con los sistemas de almacenamiento. El componente Core también es donde se definen los RDD con los cuales podremos operar.eee

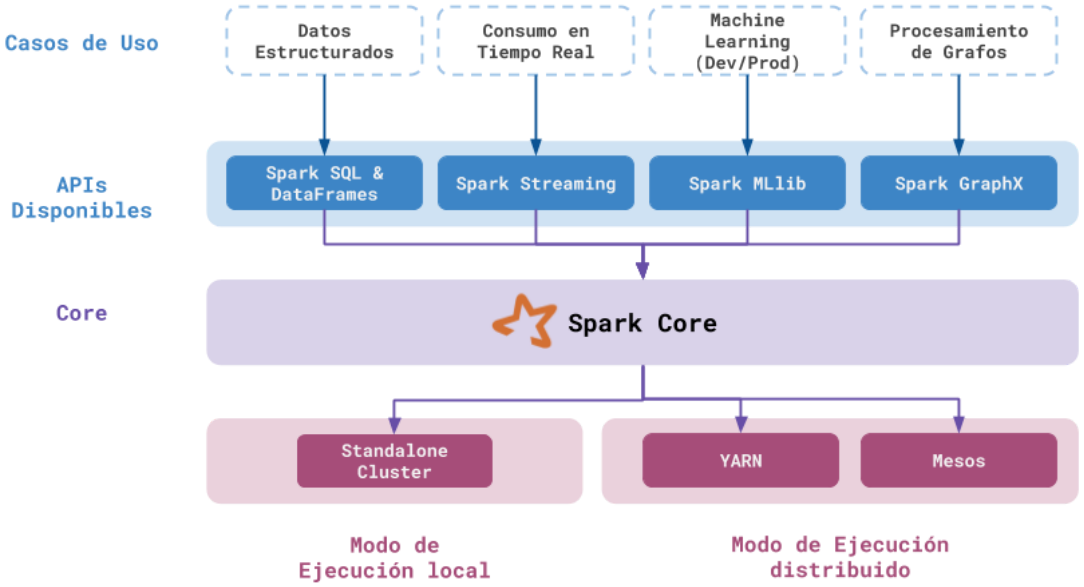

Imagen 1. Spark API working.

Spark SQL

Spark SQL es la interfaz mediante la cual el usuario puede trabajar con datos estructurados o semiestructurados. Por definición, cualquier dato que tenga un **schema** (un conjunto de campos conocidos para cada registro) se puede considerar como estructurado. Spark SQL permite trabajar con este tipo de datos de manera eficiente y fácil mediante tres formas:

1. Permite cargar datos de múltiples formatos estructurados como JSON, Hive y Parquet.

2. Permite realizar consultas utilizando SQL dentro de una aplicación Spark o mediante conectores JDBC/ODBC.

3. Dentro de una aplicación Spark, facilita la integración entre SQL y código en Python/Scala/Java, permitiendo joins entre RDD y tablas SQL, habilitación de funciones personalizadas en SQL, entre otros.

La lectura anterior aprendimos sobre la principal abstracción de Spark, los RDDs. Resulta que estos corresponden a una estructura genérica y responden al Core. Existe un trueque entre la adaptabilidad de un RDD genérico a múltiples escenarios, a expensas de tener una estructura menos específica al caso de uso de trabajo con tablas.

Nuestro objetivo es encontrar una estructura de datos que presente principios de operación similar a los `pd.DataFrames`. Resulta que estos se conocen como `pyspark.sql.dataframe.DataFrame`, y funcionan como un híbrido entre `SQL` y los `pd.DataFrames`. Así, esta estructura tendrá un poco más de restricciones y operaciones, orientadas a la resolución de problemas de analítica.

Algunas de las principales características de los `pyspark.sql.dataframe.DataFrame` de Spark son:

- Los `DataFrame` son un subtipo restringido de RDD.
- Los `DataFrame` almacenan datos bidimensionales, facilitando la abstracción en el procesamiento de datos a como usualmente lo realizamos en hojas de cálculos o tablas en una base de datos relacional.
- Cada columna dentro de un `DataFrame` puede tener sólo un tipo de dato.
- Cada fila dentro de un DataFrame representará una unidad de observación/registro.
- La reorientación de los datos al convertirse a `DataFrame` permiten una mayor optimización para operaciones en el procesamiento de éstos.

Configurando el entry point con `SparkSession`

La manera más fácil de interactuar con Spark SQL es mediante la creación de una conexión con el método `pyspark.sql.SparkSession`. Si bien es posible generar contextos específicos mediante los métodos `SQLContext` y `HiveContext`, privilegiaremos el uso de SparkSession dado que encapsula múltiples contextos.

En sesiones interactivas, `pyspark.sql.SparkSession` se puede encontrar instanciado como `spark`. Este objeto contendrá referencia a un metastore donde se podrán alojar definiciones asociadas a los objetos generados. Si utilizamos el contexto de Hive, implementaremos su base de metadatos. De lo contrario, se implementará una base de metadatos propia.

Partamos por importar la clase `SparkSession` con la instrucción `from pyspark.sql import SparkSession`. Mediante la expresión `builder` habilitaremos la opción de poder construir nuestra propia sesión. Posterior al `builder`, podremos declarar nuestro motor de asignación de recursos, así como el nombre de la aplicación. Dado que en nuestro cluster estaremos trabajando con `Hive` para realizar consultas `SQL` que se compilen en código MapReduce, una buena opción a incluir en el método es `enableHiveSupport`, que permitirá operar con una base de metadatos persistentes, así como implementar métodos `SerDe` y funciones definidas por el usuario. Finalmente con `getOrCreate`, nos aseguramos de la existencia de un `SparkSession` preexistente.

```
from pyspark.sql import SparkSession

spark = SparkSession\
      .builder\
      .appName("u4lec2")\
      .enableHiveSupport()\
      .getOrCreate()
```

```
spark
```

```
    <div>
        <p><b>SparkSession - hive</b></p>
<div>
    <p><b>SparkContext</b></p>
    <p><a href="http://192.168.2.37:4040">Spark UI</a></p>
    <dl>
      <dt>Version</dt>
        <dd><code>v2.4.3</code></dd>
      <dt>Master</dt>
        <dd><code>local[*]</code></dd>
      <dt>AppName</dt>
        <dd><code>u4lec2</code></dd>
    </dl>
</div>
    </div>
```

Ya con nuestro objeto creado en el ambiente de trabajo, podemos comenzar a interactuar con `Spark`. Como siempre, nuestros ejemplos serán útiles en la medida que tengamos datos, para lo que vamos a ingestar un conjunto. Las opciones para generar lecturas desde un objeto `pyspark.sql.SparkSession` son amplias, para lo cual se entrega la siguiente imagen que busca resumir los principales comportamientos en consumo de datos de SparkSession. Un aspecto importante a destacar es que todo objeto que retorne estos métodos será un `pyspark.sql.dataframe.DataFrame`.

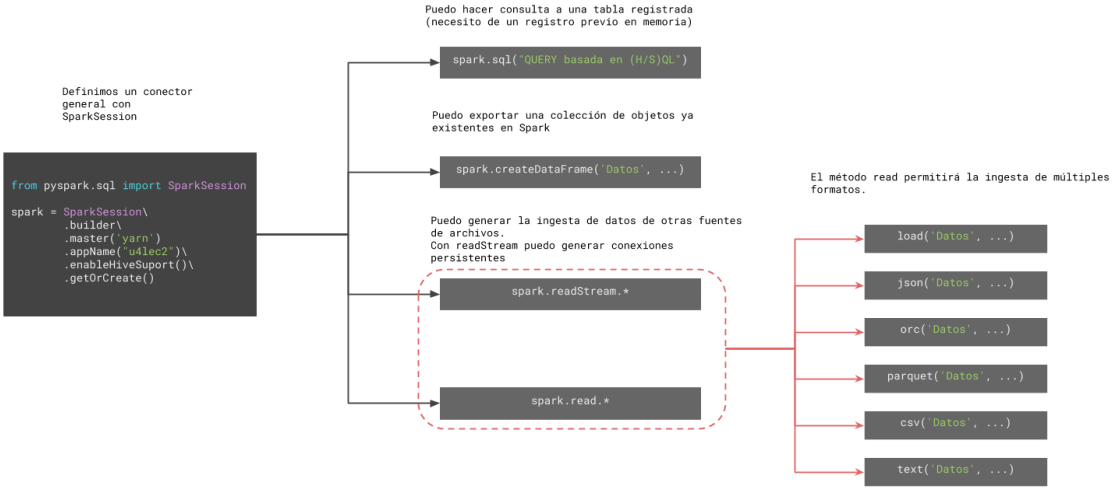

Imagen 2. Spark `DataFrame`.

Elementos conformantes de `pyspark.sql.dataframe.DataFrame`

Para ejemplificar el uso de los `DataFrame` dentro de `pyspark`, realizaremos una ingesta de un archivo llamado `complete_groceries.json` que contiene información sobre una aplicación de pedidos a un supermercado. El archivo se encuentra en el bucket del curso `s3://bigdata-desafio/lecture-replication-data/groceries/complete_groceries.json`. El registro se compone de una serie de números enteros que representan el identificador de la compra, si es que fue o no prioridad, así como el total de la compra. La particularidad de este registro es que tenemos los campos purchases, `storage_location` y `delivery_address` que son arreglos y diccionarios.

```
{"order_id": 283275,
"priority": 0,
"purchases": ["Spaghetti", "Dishwasher Soap", "Dishwasher Soap", "Avocado"],
"ammount": 76,
"storage_location": {
    "street": "1019 Kyle Stream Apt. 825",
    "city": "Jasonview",
    "state": "Illinois"},
"delivery_address": {"street": "0728 Miller Stravenue Suite 277",
                     "city": "Joefort",
                     "state": "Illinois"}
}
```

Comencemos por leer este archivo utilizando `spark.read.json` y asignándole a un objeto llamado `groceries`. Si nos damos cuenta, veremos que a diferencia de cómo lo realizamos en `Hive`, no es estrictamente necesario definir el Schema de los datos. Si preguntamos por el tipo de dato de este objeto, nos dirá que es un `pyspark.sql.dataframe.DataFrame`.

Caveat: *De aquí en adelante, los ejercicios se encuentran implementados en un notebook de JupyterHub.*

```
groceries =
spark.read.json('s3://bigdata-desafio/lecture-replication-data/groceries
/complete_groceries.json')
```

```
type(groceries)
```

```
pyspark.sql.dataframe.DataFrame
```

Schema, Row y Column

Dentro del objeto `pyspark.sql.dataframe.DataFrame` encontraremos un par de definiciones sobre qué constituyen las entidades, los atributos y cómo estos se definen. Para ello debemos interiorizarnos en los objetos `Row`, `Column` y `Schema`.

Resulta que `pyspark` permite inferir el esquema de un conjunto de datos. Si solicitamos la información con `printSchema`, observamos que logró inferir de una manera adecuada qué representa los datos, incluyendo los tipos de datos complejos como `array` y `struct`.

```
groceries.printSchema()
```

```
root
 |-- ammount: long (nullable = true)
 |-- delivery_address: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- state: string (nullable = true)
 |    |-- street: string (nullable = true)
 |-- month: string (nullable = true)
 |-- order_id: long (nullable = true)
 |-- priority: long (nullable = true)
 |-- purchases: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- storage_location: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- state: string (nullable = true)
 |    |-- street: string (nullable = true)
 |-- year: long (nullable = true)
```

La representación que logramos del `json` con el método `printSchema` es simplemente la versión humana de lo que pasa tras bambalinas con el `DataFrame`. Mediante el método `schema` podemos obtener una representación fidedigna de la estructura inferida por `Spark`. Cabe destacar la existencia de los tipos de datos `StructType` y `List`.

El método `schema` está representando cómo se declaran los campos y estructura de un registro específico. El tipo de dato `StructType` hace referencia a una fila dentro de nuestro `DataFrame`. Por su parte, el tipo de dato `List` es el que define el nombre y el tipo de dato en cada campo a completar.

```
groceries.schema
```

```
StructType(List(StructField(ammount,LongType,true),StructField(delivery_
address,StructType(List(StructField(city,StringType,true),StructField(st
ate,StringType,true),StructField(street,StringType,true))),true),StructF
ield(month,StringType,true),StructField(order_id,LongType,true),StructFi
eld(priority,LongType,true),StructField(purchases,ArrayType(StringType,t
rue),true),StructField(storage_location,StructType(List(StructField(city
,StringType,true),StructField(state,StringType,true),StructField(street,
StringType,true))),true),StructField(year,LongType,true)))
```

Para solicitar un retorno de datos en nuestro objeto `DataFrame` creado, vamos a tener dos posibles opciones. La primera es implementar el método `show`, que retornará en un print las `n` observaciones inferidas. El resultado será algo entendible y similar a una consulta en `SQL`. La segunda opción es implementar el método `take` que ya conocemos de nuestro trabajo con RDD. La principal diferencia es que este método nos entregará una representación cruda de los datos. En específico, nos informa que la primera observación está envuelta en un método `Row`.

```
groceries.show(1)
```

```
+-------+--------------------+-----+--------+--------+-----------------
--+--------------------+----+
|ammount|    delivery_address|month|order_id|priority|
purchases|    storage_location|year|
+-------+--------------------+-----+--------+--------+-----------------
--+--------------------+----+
|    253|[Johntown, South ...| June|  281053|       0|[Rake, Eggs,
Tea,...|[South Pamelaton,...|2015|
+-------+--------------------+-----+--------+--------+-----------------
--+--------------------+----+
only showing top 1 row
```

```
groceries.take(1)
```

```
[Row(ammount=253, delivery_address=Row(city='Johntown', state='South
Dakota', street='84430 Caroline Point Apt. 462'), month='June',
order_id=281053, priority=0, purchases=['Rake', 'Eggs', 'Tea', 'Trout',
'Rice', 'Yoghurt', 'Tea', 'Toothpaste', 'Eggs', 'Steak', 'Trout'],
storage_location=Row(city='South Pamelaton', state='South Dakota',
street='559 Wagner Oval Suite 872'), year=2015)]
```

Un objeto `Row` simplemente representa un registro específico dentro del `DataFrame`. Su comportamiento difiere entre Scala/Java y Python. En Python permiten acceder a una serie de valores mediante el nombre de columna específica. Si deseamos extraer el valor `ammount` dentro de este objeto, podemos utilizar la notación `row.column_name`.

Digresión: *Con el método `_` podemos acceder a la última ejecución válida en Python.*

```
_[0].ammount
```

```
253
```

Podemos acceder a una columna específica mediante la implementación del método `DataFrame.column_name`. Este retornará un objeto del tipo `Column`. Dentro de este objeto podemos implementar funciones que retornarán una reexpresión. Si estamos interesados en seleccionar todos los registros que están entre 100 y 200 dólares, podremos utilizar la sintaxis `DataFrame.Columna.between()`.

```
groceries.ammount
```

```
Column<b'ammount'>
```

```
type(_)
```

```
pyspark.sql.column.Column
```

```
groceries\
    .ammount\
    .between(100, 200)
```

```
Column<b'((ammount >= 100) AND (ammount <= 200))'>
```

Esta expresión se considera lazy, dado que todavía no se aplica a un conjunto de datos. Recordemos que el objetivo es implementar un indicador de selección para los valores entre 100 y 200. Para generar un retorno válido, deberemos incluir esta expresión dentro del `DataFrame` con el método `select`, de la siguiente forma:

```
groceries\
    .select(groceries.delivery_address.state,
            groceries.ammount,
            groceries.ammount.between(100, 200))\
    .sample(.01)\
    .show(5)
```

```
+----------------------+-------+---------------------------------------+
|delivery_address.state|ammount|((ammount >= 100) AND (ammount <= 200))|
+----------------------+-------+---------------------------------------+
|                  Iowa|    306|                                  false|
|                 Idaho|    138|                                   true|
|           Connecticut|    114|                                   true|
|             Minnesota|    108|                                   true|
|                 Maine|    144|                                   true|
+----------------------+-------+---------------------------------------+
only showing top 5 rows
```

Tipos de operaciones en `pyspark.sql.dataframe.DataFrame`

El ejemplo anterior demuestra una de las principales características del objeto `DataFrame`: De manera similar a como lo realizamos con `pandas`, privilegiamos trabajar siempre sobre columnas. De hecho, los `DataFrame` de `spark` operan en el principio de ser abstracciones RDD columna-orientadas. A grandes rasgos las operaciones en un objeto `DataFrame` de `pyspark` las podemos agrupar en tres grandes categorías:

1. **Operaciones sobre los metadatos:** Son aquellas que están orientadas a entregar información sobre la estructura de los datos, por sobre los datos en sí mismos. Ejemplos como `printSchema` y los retornos con `Row` y `Column` son casos donde implementamos operaciones que nos entregan información de la composición de los datos.

2. **Operaciones basadas en RDD:** Toda operación en un `DataFrame` se puede remontar a una operación basada en Resilient Distributed Datasets. Así, podremos implementar nuestros procesos clásicos como `map`, `reduceByKey` entre otros.

3. **Operaciones orientadas a la estadística:** Existen una serie de funciones que vienen incluidas por defecto en el módulo `pyspark.sql.functions`, las cuales están orientadas al procesamiento ágil de columnas para entregar resultados descriptivos.

Dado la vasta cantidad de operaciones posibles dentro de los `DataFrame`, se entrega la siguiente imagen a modo resumen sobre sus componentes.

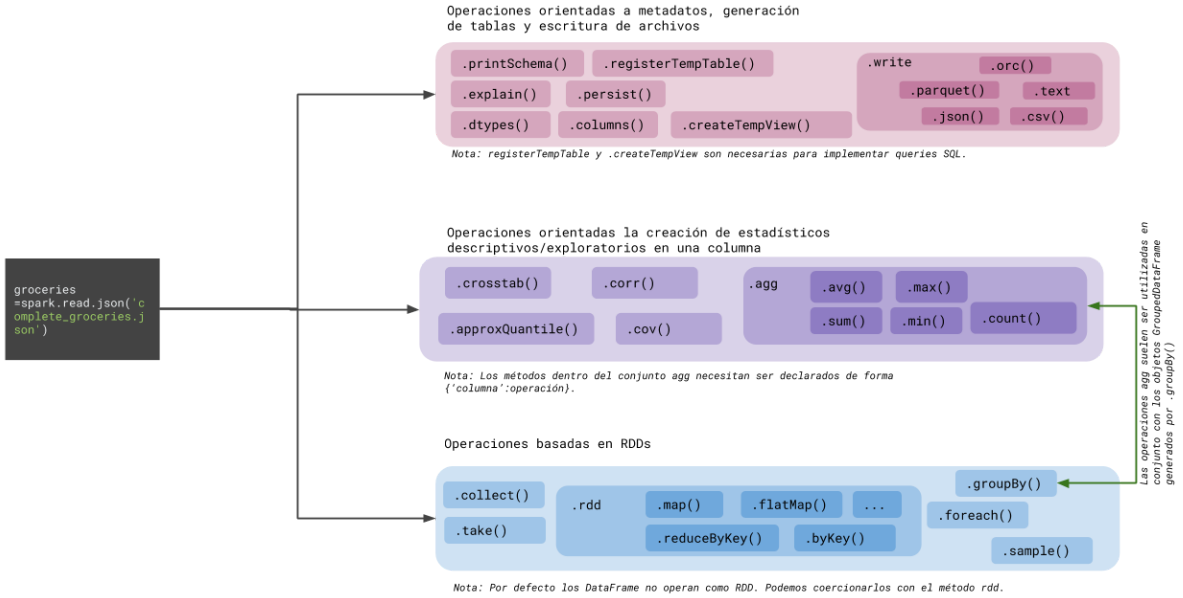

Imagen 3. Operaciones `DataFrame`.

Supongamos que estamos interesados en extraer la cantidad de filas y columnas existentes en este objeto `DataFrame`. Para identificar la cantidad de filas podemos utilizar el método `count`, y para extraer la cantidad de columnas, vamos a contar el largo del arreglo retornado por `groceries.columns`. En este pequeño ejemplo comenzamos a implementar funciones asociadas a operaciones de estadística y metadatos.

```
print("Cantidad de registros: ", groceries.count())
print("Cantidad de columnas asociadas:", len(groceries.columns) )
```

```
Cantidad de registros:  101000
Cantidad de columnas asociadas: 8
```

Nuestra primera tarea va a ser evaluar cómo se comporta el volumen de compras en este conjunto de datos. Recordemos que con el método `groceries.columns` podremos tener un listado del nombre de todas las columnas. Lo que vamos a realizar es generar un subconjunto con las columnas `year`, `month` y `ammount` mediante el método `select`. Mediante el método `show` podremos visualizar cómo está organizada la base de datos. Cada registro corresponderá a una compra específica.

```
ammount_by_year = groceries.select( 'year', 'month', 'ammount')
ammount_by_year.show(5)
```

```
+----+-------+-------+
|year|  month|ammount|
+----+-------+-------+
|2015|   June|    253|
|2018|   July|    396|
|2016|   June|    561|
|2015|January|    108|
|2014|    May|      4|
+----+-------+-------+
only showing top 5 rows
```

```
print("Cantidad de registros: ", ammount_by_year.count())
print("Cantidad de columnas asociadas:", len(ammount_by_year.columns) )
```

```
Cantidad de registros:  101000
Cantidad de columnas asociadas: 3
```

Ya con este nuevo objeto, vamos a evaluar una serie de estadísticas descriptivas con el comando `describe` en el `DataFrame`. Este se comporta de una manera similar a su análogo en `pandas`. Algunas de las diferencias es que éste se aplica a todas las columnas, irrestricto si la columna es un dato string o no. La otra diferencia es que el resultado debe ser evaluado con un `show`.

```
df_describe = ammount_by_year.describe()
df_describe.show()
```

```
+-------+------------------+---------+------------------+
|summary|              year|    month|           ammount|
+-------+------------------+---------+------------------+
|  count|            101000|   101000|            101000|
|   mean|2015.9906138613862|     null|179.15418811881187|
| stddev|1.4142524248846893|     null|157.00952691722873|
|    min|              2014|    April|                 1|
|    max|              2018|September|               792|
+-------+------------------+---------+------------------+
```

Supongamos que estamos interesados en identificar todas aquellas observaciones donde el total gastado (`ammount`) sea menor a la media global de 179.154 como 1, 0 para todas las demás observaciones que no satisfagan esta condición. De manera adicional, deseamos agregar esta información en una nueva columna en nuestro `DataFrame`.

Sabemos que en nuestro flujo clásico de `pandas` el procedimiento sería algo similar a: `df['flag_underflow'] = np.where(df['ammount'] < df['ammount'].mean(), 1, 0)`. Para implementar el operador ternario en `pyspark`, podemos importar la función `pyspark.sql.functions.when`. Por defecto en `when` vamos a definir la condición lógica, seguido del valor a imputar. Para todos los demás casos, utilizamos la operación `otherwise`.

El otro punto a considerar es que la asignación de una nueva columna en un `DataFrame` existente difiere un poco a cómo lo haríamos en `pandas`. Acá el objetivo es sobreescribir el objeto con una nueva expresión. Esta nueva expresión se implementará mediante la función `DataFrame.withColumn`.

```
from pyspark.sql.functions import when

ammount_by_year = ammount_by_year\
                .withColumn('flag_underflow',
                    when(ammount_by_year.ammount < 179.154, 1)\
                    .otherwise(0))
```

```
# visualicemos los resultados
ammount_by_year.show(5)
```

```
+----+-------+-------+--------------+
|year|  month|ammount|flag_underflow|
+----+-------+-------+--------------+
|2015|   June|    253|             0|
|2018|   July|    396|             0|
|2016|   June|    561|             0|
|2015|January|    108|             1|
|2014|    May|      4|             1|
+----+-------+-------+--------------+
only showing top 5 rows
```

En base a esta nueva columna, vamos a extraer el porcentaje de consumos bajo la media. Para ello implementaremos la función `agg`. Resulta que ésta opera mediante un diccionario donde le indicamos la columna a operar y qué tipo de operación. Para este caso, necesitamos extraer la media de la columna `flag_underflow` y la cantidad de casos. La sintáxis se verá de la siguiente manera. Nos percatamos que tenemos 60\% de encontrar un registro aleatorio que presente un consumo inferior a la media.

```
ammount_by_year.agg({'ammount': 'count', 'flag_underflow':
'avg'}).show()
```

```
+------------------+--------------+
|avg(flag_underflow)|count(ammount)|
+------------------+--------------+
| 0.5998712871287128|        101000|
+------------------+--------------+
```

```
ammount_by_year.sort
```

Entonces, ¿cómo replicamos este procedimiento si queremos sacar el porcentaje de registros bajo la media, pero por cada año? Para ello vamos a utilizar la función `groupBy` que permite generar objetos `GroupedDataFrame`, mediante los cuales podemos implementar funciones como `agg`. Posterior a la implementación de `agg`, vamos a ordenar los datos de forma descendente con el método `sort`. Observamos que si bien la varianza entre años es bastante baja, el 2018 fue el año que tuvo un menor porcentaje de pedidos bajo el promedio, mientras que 2015 fue el año con una mayor probabilidad.

```
ammount_by_year\
    .groupBy('year')\
    .agg({'ammount': 'count', 'flag_underflow': 'avg'})\
    .sort('avg(flag_underflow)')\
    .show()
```

```
+----+-------------------+--------------+
|year|avg(flag_underflow)|count(ammount)|
+----+-------------------+--------------+
|2018| 0.5974565663645922|         19973|
|2014| 0.5995304705076787|         20446|
|2017| 0.6001983143282102|         20170|
|2016| 0.6008696081822225|         20239|
|2015|  0.601279000594884|         20172|
+----+-------------------+--------------+
```

Sigamos en un espíritu similar y reportemos tanto la media y desviación estándar, así como el valor mínimo y máximo para cada año. Para implementar múltiples funciones sobre una misma columna, vamos a tener que hacer un procedimiento distinto. Partamos por incluir todas las funciones mencionadas que se encuentran en el módulo `pyspark.sql.functions`. También deberemos incluir el operador `col` que nos permitirá asignar una columna específica asociada a una función.

De esta forma, vamos a generar una lista donde cada elemento va a estar compuesto de la siguiente expresión: `funcion(col('nombre_columna'))`. De esta forma, vamos a pasar de manera explícita el comportamiento de cada función. Esta lista generada la pasaremos dentro de `agg`.

```
from pyspark.sql.functions import min, max, col, mean, stddev

evaluate_functions = [min(col('ammount')), max(col('ammount')),
count(col("ammount")), mean(col("ammount")), stddev(col("ammount"))]

ammount_by_year\
    .groupBy('year')\
    .agg(*evaluate_functions)\
    .show()
```

```
+----+------------+------------+--------------+------------------+------
--------------+
|year|min(ammount)|max(ammount)|count(ammount)|
avg(ammount)|stddev_samp(ammount)|
+----+------------+------------+--------------+------------------+------
--------------+
|2014|           1|         792|         20446|178.98420228895628|
156.93844997272186|
|2016|           1|         792|         20239|178.87207866001285|
156.72054515041364|
|2018|           1|         792|         19973|179.58649176388124|
158.0530720194283|
|2017|           1|         792|         20170|179.26688150718888|
156.96411333817608|
|2015|           1|         792|         20172| 179.0688082490581|
156.39330491402123|
+----+------------+------------+--------------+------------------+------
--------------+
```

Por último, debemos tener en consideración el hecho que todo objeto `DataFrame` resultante de `pyspark` puede ser convertido a un objeto `pandas.DataFrames`, lo cual facilitará el posterior procesamiento de los datos.

Supongamos que estamos interesados en ver la cantidad de compras realizadas por mes y año. Para ello vamos a generar un objeto `DataFrame` que estará agrupado por dos atributos: `'year'` y `'month'`. Si implementamos el código, obtendremos una salida similar a:

```
ammount_by_year_month = ammount_by_year\
    .groupBy(['year', 'month'])\
    .agg({'ammount': 'count'})
```

```
ammount_by_year_month.show(5)
```

```
+----+---------+--------------+
|year|    month|count(ammount)|
+----+---------+--------------+
|2014|    April|          1671|
|2015| November|          1619|
|2014|September|          1621|
|2018|    March|          1634|
|2014|   August|          1723|
+----+---------+--------------+
only showing top 5 rows
```

Para pasar este `DataFrame` a `pandas.DataFrames`, tan solo necesitamos utilizar el método `toPandas`, el cual se encargará de pasar esta información a un objeto con el cual podremos seguir en nuestro procesamiento.

```
ammount_by_year_month = ammount_by_year_month.toPandas()
print("Tipo de dato: ", type(ammount_by_year_month),"\n")
print(ammount_by_year_month.head(5))
```

```
Tipo de dato:  <class 'pandas.core.frame.DataFrame'>

   year      month  count(ammount)
0  2014      April            1671
1  2015   November            1619
2  2014  September            1621
3  2018      March            1634
4  2014     August            1723
```

Para generar un reporte gráfico de la cantidad de registros por mes y año, vamos a partir por generar una nueva columna que concatene ambos valores y que podamos implementar en el gráfico. Esto lo podemos lograr con la siguiente línea:

```
ammount_by_year_month['concat_ym'] =
ammount_by_year_month['year'].astype(str) + " "+
ammount_by_year_month['month']
```

Finalmente, vamos a ordenar los atributos por el tamaño de `count(ammount)` y graficamos utilizando `matplotlib`. Se observa que los primeros tres meses que tuvieron más registros asociados fueron Mayo del 2017, Julio del 2016 y Diciembre del 2014.

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('ggplot')
plt.figure(figsize=(10, 15))
ammount_by_year_month =
ammount_by_year_month.sort_values(by='count(ammount)')
plt.plot(ammount_by_year_month['count(ammount)'],
         ammount_by_year_month['concat_ym'],'o')
plt.xlabel('Uso promedio a nivel mensual');
plt.ylabel('Combinación Año - Mes');
```

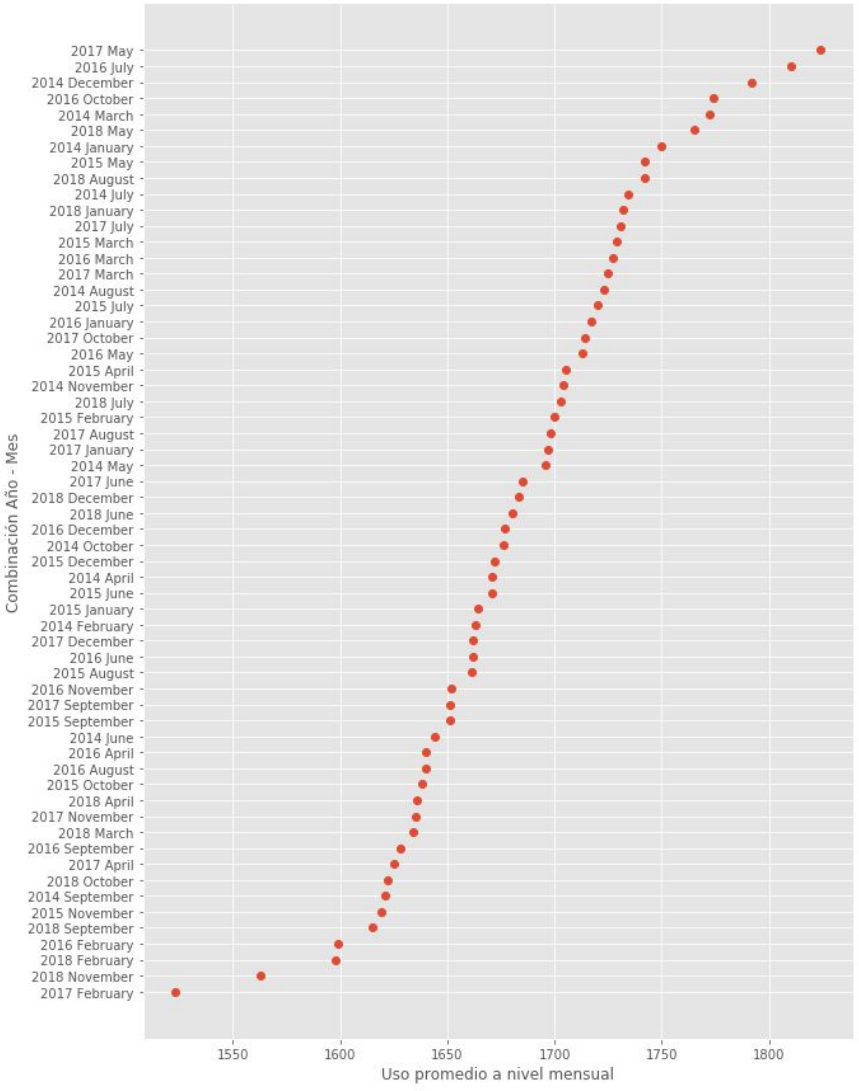

Imagen 4. Uso promedio a nivel mensual.

Consultas basadas en SQL en pyspark

De manera alternativa a las consultas implementadas **dentro** del mismo objeto `DataFrame`, podemos implementar consultas desde declaraciones SQL en `pyspark`. Para poder implementar operaciones SQL, debemos generar el registro en una tabla temporal de los datos con los cuales estamos trabajando. En este caso utilizaremos el mismo `DataFrame`.

Para implementar una tabla temporal, debemos utilizar el método `.registerTempTable` que encontraremos en el `DataFrame`. El argumento que va entre paréntesis del método corresponde al nombre de la tabla que haremos referencia dentro de las consultas SQL. Vamos crear esta tabla temporal con el nombre `groceries`.

```
groceries.registerTempTable('groceries')
```

Ya con nuestra tabla temporal registrada, podremos comenzar a implementar consultas. Para ello, vamos a utilizar el método `sql` que se encuentra dentro de nuestro objeto donde instanciamos la clase `SparkSession`. Dentro de los paréntesis del método `sql`, escribiremos nuestra consulta. Dado que nuestro objeto `SparkSession` tiene la opción de utilizar Hive, las queries que escribiremos podrán contener expresiones de HQL.

Nuestra primera consulta será evaluar la relación existente entre la cantidad de artículos comprados y el gasto asociado. Partamos por definir una consulta que nos devuelva la cantidad de compras creadas y el gasto asociado en cada registro. La expresión quedaría constituida de la siguiente forma.

```
tamano_precio = spark.sql( """
                        SELECT size(purchases), ammount FROM groceries
                        """).toPandas()
```

Dado que sabemos que todo objeto `DataFrame` puede ser exportado a `pandas.DataFrames`, concatenaremos el método `toPandas` para poder visualizar los datos. Visualizaremos los datos en un gráfico de dispersión con el método `sns.regplot`.

```
import seaborn as sns
plt.figure(figsize=(12, 8))
sns.regplot('size(purchases)', 'ammount', tamano_precio, marker='.');
```

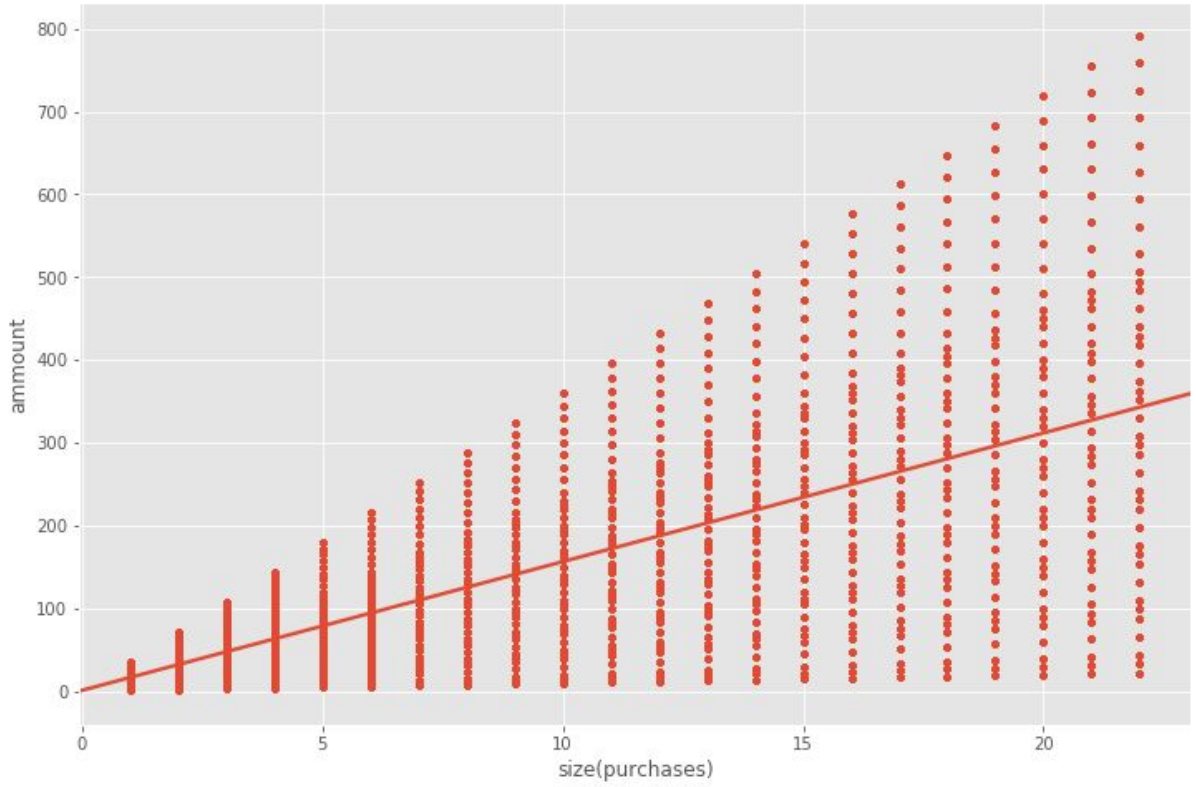

Imagen 5. Gráfico de dispersión.

Observamos que a nivel global, la tendencia tiende a ser positiva. En la medida que se agregan más productos al carrito de compras, esperaremos un gasto total más alto.

Ahora, ¿Cómo se comporta esto a nivel de estado? Vamos a levantar información similar mediante la query definida a continuación. Cabe destacar el hecho que estamos implementando UDF nativas de Hive como los métodos `corr` y `size` para este ejemplo.

```
correlación_tamano_precio_estatal = spark.sql(
    """
    SELECT storage_location.state,
    corr(size(purchases), ammount)
    FROM groceries
    GROUP BY storage_location.state
    """).toPandas()

correlación_tamano_precio_estatal.columns = ['state', 'corr']
correlación_tamano_precio_estatal =
correlación_tamano_precio_estatal.sort_values(by='corr')

plt.figure(figsize=(10, 15))
plt.plot(correlación_tamano_precio_estatal['corr'],
         correlación_tamano_precio_estatal['state'], 'o-')
plt.xlabel('Correlación de Pearson entre Cantidad de compras y Gasto a
nivel estatal');
```

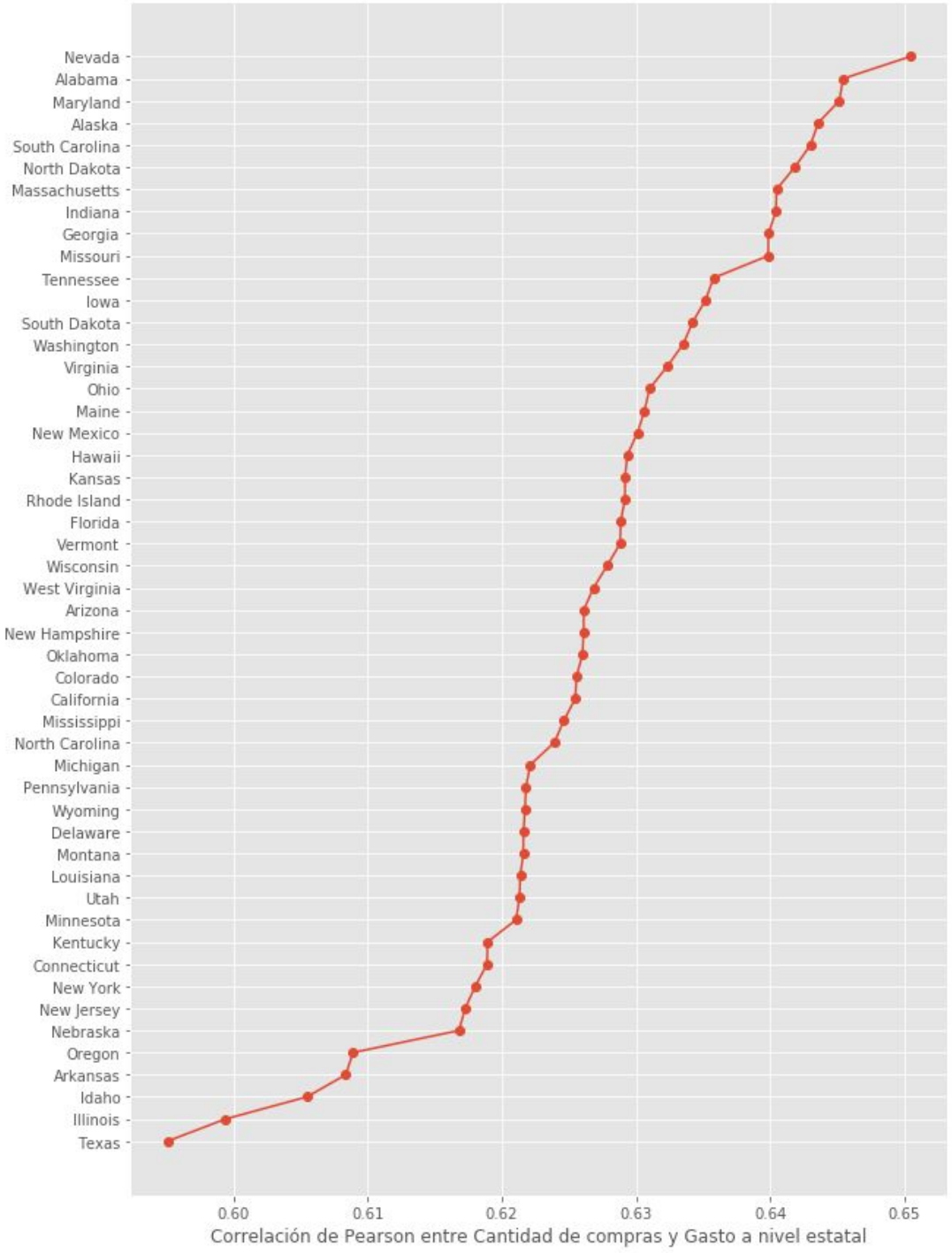

Imagen 6. Correlación de Pearson entre Cantidad de compras y Gasto a nivel estatal.

El gráfico reporta que la relación entre compras y gasto tiende a ser mucho más débil en estados como Texas, Illinois y Idaho; mientras que en estados como Nevada, Alabama y Maryland la correlación tiende a ser un poco más fuerte. La última query de ejemplo nos permitirá visualizar en qué estados hay una mayor probabilidad de hacer entregas erróneas fuera de este.

```
import numpy as np
count_mismatched = spark.sql("""
                    SELECT delivery_address.state,
                    count(delivery_address.state)
                    FROM groceries WHERE storage_location.state !=
delivery_address.state
                    GROUP BY delivery_address.state
                """).toPandas()
count_mismatched.columns = ['state', 'count']
count_mismatched = count_mismatched.sort_values(by='count')
plt.figure(figsize=(15, 7))
plt.bar(count_mismatched['state'],
        count_mismatched['count'],
        color=np.where(count_mismatched['count'] <
count_mismatched['count'].mean(),
                    'dodgerblue', 'tomato'))
plt.xticks(rotation=90);
```

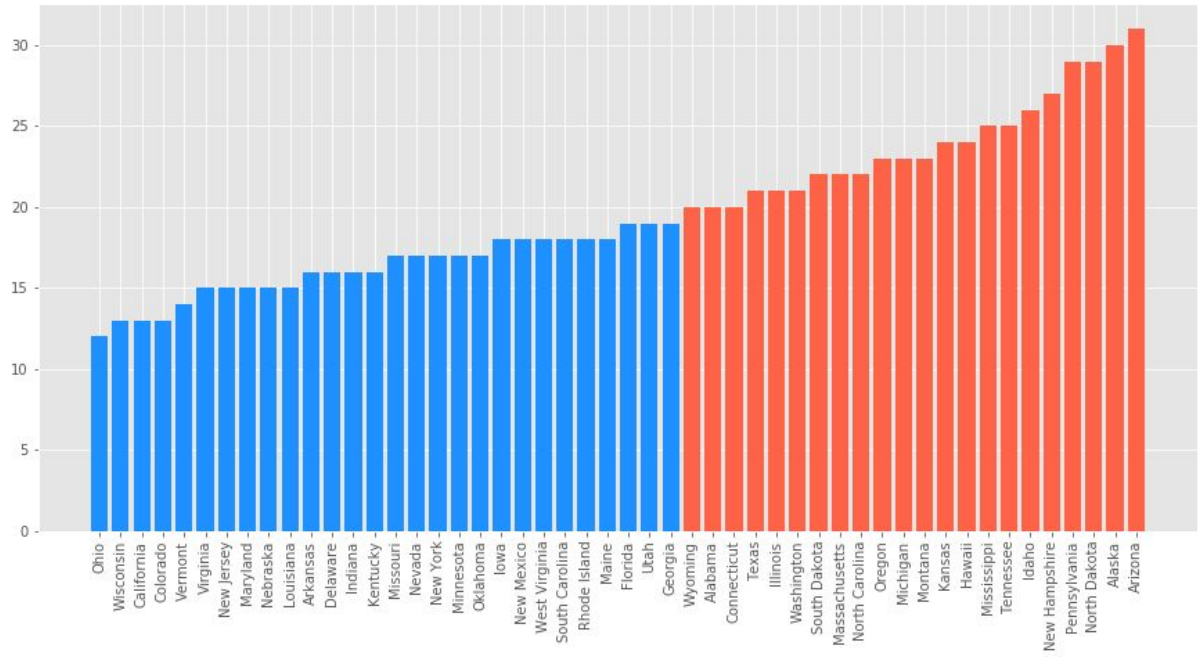

Imagen 7. Estados con mayor probabilidad de realizar entregas erróneas.

En rojo se encuentran aquellos estados donde la cantidad de pedidos derivados desde otro estado es superior al comportamiento global de la muestra, mientras que en azul se encuentran aquellos estados donde el comportamiento es menor al comportamiento global de la muestra.

Cierre: Comentarios sobre los archivos columnares

Spark SQL hace uso de almacenamiento columnar para guardar los objetos en memoria. Un archivo columnar busca organizar los datos mediante columnas, por sobre el formato por defecto que es fila. La implementación de sistemas de almacenamiento columnar permite reducir de manera substancial el tamaño de los archivos y el desempeño de consultas dentro de los datos.

Dentro de la suite de Hadoop existen varias alternativas para crear archivos columnares, dentro de las cuales se destaca Parquet. Este es un formato de archivos para generar representaciones columnares de los datos almacenados, el cual se considera un ciudadano de primera clase en Spark y es el formato preferido para leer y escribir archivos y resultados.

Comencemos por generar un archivo columnar a partir de nuestro `DataFrame` `groceries`. Para ello vamos a utilizar la opción `write.parquet`, donde especificaremos un nombre.

```
groceries.write.parquet('groceries.parquet')
```

Los resultados del parquet por defecto se van a escribir en una carpeta dentro del directorio en el que estemos trabajando. Así, observaremos que tenemos nuestra carpeta con el nombre `groceries.parquet`.

```
!ls
```

```
Untitled.ipynb    derby.log         lec1.ipynb
[1m[36mmetastore_db[m[m
Untitled1.ipynb   [1m[36mgroceries.parquet[m[m lec1.md
```

Si evaluamos cuáles son los elementos dentro de esta carpeta, observaremos que corresponden a un trabajo de MapReduce, donde nos notifican el éxito de la tarea con `_SUCCESS`.

```
!ls groceries.parquet/
```

```
_SUCCESS
part-00000-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00001-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00002-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00003-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00004-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00005-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00006-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00007-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00008-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00009-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
part-00010-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.snappy.parquet
```

Dentro de cada archivo `part-000*.parquet`, el contenido estará en un formato binario para facilitar su consumo por Spark y Hive.

```
!head -n 3
groceries.parquet/part-00000-256a381b-8474-45ec-ad75-914bb2ae3e40-c000.s
nappy.parquet
```

```
PAR1'#'L''' '1l   T*
'v''''4a''
<@''''',Û'T'y*Û  @'''(X''@ D'0'B
0d a 'C'>!b'&N''!'',Z'x1'2B''''6n'''c';0''''A'dB'''G'$Qrd
''8yF'R'4'b''W
```

Otra de las virtudes es implementar formatos parquet, es la capacidad de particionar los datos por algún criterio. En este ejemplo, vamos a particionar los datos de groceries por los años de la medición. Esto lo podemos lograr con la opción `partitionBy` dentro del método `write.parquet`. Como observamos, dentro de la carpeta `groceries-partitioned.parquet` vamos a encontrar subdirectorios que representan a cada uno de los años.

```
groceries.write.parquet('groceries-partitioned.parquet',
partitionBy='year')
```

```
!ls groceries-partitioned.parquet/
```

```
_SUCCESS  [1m[36myear=2014[m[m [1m[36myear=2015[m[m [1m[36myear=2016[m[m
[1m[36myear=2017[m[m [1m[36myear=2018[m[m
```

```
!ls groceries-partitioned.parquet/year=2014
```

```
part-00000-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00001-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00002-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00003-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00004-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00005-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00006-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00007-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00008-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00009-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
part-00010-4013b333-5808-442e-9fc7-86fe6191dd6b.c000.snappy.parquet
```

Para leer un conjunto de archivos parquet, tan solo necesitamos implementar el método `read.parquet` desde nuestro `SparkSession`. El método se encargará de juntar las piezas dentro del directorio, irrestricto de sus niveles de partición.

```
groceries_parquet = spark.read.parquet('groceries-partitioned.parquet')
```

Para concluir, podemos observar que la misma query entrega los mismos resultados en ambos tipos de objetos.

Archivo parquetizado

```
groceries_parquet.groupBy('year').count().sort('year').show()
```

```
+----+-----+
|year|count|
+----+-----+
|2014|20446|
|2015|20172|
|2016|20239|
|2017|20170|
|2018|19973|
+----+-----+
```

Archivo original (sin parquetizar)

```
groceries.groupBy('year').count().sort('year').show()
```

```
+----+-----+
|year|count|
+----+-----+
|2014|20446|
|2015|20172|
|2016|20239|
|2017|20170|
|2018|19973|
+----+-----+
```

Referencias

- Karau, H; Konwinski, A; Wendell, P; Zaharia, M. 2015. Learning Spark: Lighting-fast Data Analysis. Sebastopol, CA: O'Reilly Media Inc.

- Ryza, S; Laserson, U; Owen, S; Wills, J. 2015. Advanced Analytics with Spark: Patterns for Learning from Data at Scale. Sebastopol, CA: O'Reilly Media Inc.

- White, T. 2014. Hadoop The Definitive Guide: Storage and Analysis at Internet Scale. Sebastopol, CA: O'Reilly Media Inc.