databricks Big Data Processing with Spark

In this big data era, Spark, which is a fast and general engine for large-scale data processing, is the hottest big data tool. Spark is a cluster computing framework which is used for scalable and efficient analysis of big data. Other data analysis tools such as R and Pandas run on a single machine but with Spark, we can use many machines which divide the tasks among themselves and perform fault tolerant computations by distributing the data over a cluster.

Spark programs run up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. Among the many capabilities of Spark, which made it favous, is its ability to be used with various programing languages through APIs. We can write Spark operations in Java, Scala, Python or R. Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

As shown in the figure below, Spark components consist of Core Spark, Spark SQL, MLlib and ML for machine learning and GraphX for graph analytics. In this blog post, we will focus on Spark SQL.

Here, I am using the flights data from the nycflights13 (https://cran.r-project.org/web/packages/nycflights13/index.html) R package. I downloaded the data from R and uploaded it to Databricks (https://databricks.com). Databricks have a free community edition (https://databricks.com/try-databricks). So, you can use it to learn Spark.

Read data

In order to use Spark and its DataFrame API, we will need to use a SQLContext. When running Spark, we start Spark application by creating a SparkContext. We can then create a SQLContext from the SparkContext.

```
> spark_DF = sqlContext.read.format("csv").options(header='true',
  inferSchema='true').load("FileStore/tables/jvo1773d1466128441021/flights.csv")
```

Let's display the first 1000 rows.

```
> display(spark_DF)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time
1	2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227
2	2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227
3	2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160
4	2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183
5	2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116

Showing the first 1000 rows.

But how many rows do we have in the data frame?

```
> print spark_DF.count()

336776
```

Before doing any analysis, let's see the schema. We can also see the type of the DataFrame.

```
> spark_DF.printSchema()
```

```
root
 |-- : integer (nullable = true)
 |-- year: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- day: integer (nullable = true)
 |-- dep_time: string (nullable = true)
 |-- sched_dep_time: integer (nullable = true)
 |-- dep_delay: string (nullable = true)
 |-- arr_time: string (nullable = true)
 |-- sched_arr_time: integer (nullable = true)
 |-- arr_delay: string (nullable = true)
 |-- carrier: string (nullable = true)
 |-- flight: integer (nullable = true)
 |-- tailnum: string (nullable = true)
 |-- origin: string (nullable = true)
 |-- dest: string (nullable = true)
 |-- air_time: string (nullable = true)
 |-- distance: integer (nullable = true)
 |-- hour: integer (nullable = true)
 |-- minute: integer (nullable = true)
 |-- time_hour: timestamp (nullable = true)
```

```
> print type(spark_DF)
```

```
<class 'pyspark.sql.dataframe.DataFrame'>
```

Spark Operations

Now, let's perform various Spark DataFrame operations.

select helps us to select one or more columns. * means all columns

```
> m=spark_DF.select("*")

  display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time
1	2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227
2	2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227
3	2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160
4	2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183
5	2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116

Showing the first 1000 rows.

```
> spark_DF.select('year','month','day').show(10)
```

```
+----+-----+---+
|year|month|day|
+----+-----+---+
|2013|    1|  1|
|2013|    1|  1|
|2013|    1|  1|
|2013|    1|  1|
|2013|    1|  1|
|2013|    1|  1|
|2013|    1|  1|
```

```
|2013|    1|  1|
|2013|    1|  1|
|2013|    1|  1|
+----+-----+---+
only showing top 10 rows
```

drop helps us to drop column(s).

```
> display(m.drop("month"))
```

	year	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distan
1	2013	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400
2	2013	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416
3	2013	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089
4	2013	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183	1576
5	2013	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116	762

Showing the first 1000 rows.

We can also change the column names.

```
> m=spark_DF.select(spark_DF.year.alias('Year'), spark_DF.month.alias('Month'), spark_DF.day.alias('Day'))
  display(m)
```

Year	Month	Day
2013	1	1
2013	1	1
2013	1	1
2013	1	1
2013	1	1
2013	1	1
2013	1	1
2013	1	1
2013	1	1

Showing the first 1000 rows.

```
> from pyspark.sql.functions import length

  m=spark_DF.select(spark_DF.origin,length(spark_DF.origin).alias('Length'))
  display(m)
```

origin	Length
EWR	3
LGA	3
JFK	3
JFK	3
LGA	3
EWR	3
EWR	3

LGA	3

Showing the first 1000 rows.

We can use **filter** or **where** to filter certain rows.

```
> m=spark_DF.filter("month =1") # selecting January only
  display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time
1	2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227
2	2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227
3	2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160
4	2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183
5	2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116

Showing the first 1000 rows.

```
> m=spark_DF.filter(spark_DF.month > 9)
  display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_ti
27005	2013	10	1	447	500	-13	614	648	-34	US	1877	N538UW	EWR	CLT	69
27006	2013	10	1	522	517	5	735	757	-22	UA	252	N556UA	EWR	IAH	174
27007	2013	10	1	536	545	-9	809	855	-46	AA	2243	N630AA	JFK	MIA	132
27008	2013	10	1	539	545	-6	801	827	-26	UA	1714	N37252	LGA	IAH	172
27009	2013	10	1	539	545	-6	917	933	-16	B6	1403	N789JB	JFK	SJU	186

Showing the first 1000 rows.

```
> m=spark_DF.filter(spark_DF.month < 9).filter('month>=5')
  display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_t
193412	2013	5	1	9	1655	434	308	2020	408	VX	413	N628VA	JFK	LAX	341
193413	2013	5	1	451	500	-9	641	640	1	US	1219	N196UW	EWR	CLT	94
193414	2013	5	1	537	540	-3	836	840	-4	AA	701	N5BYAA	JFK	MIA	144
193415	2013	5	1	544	545	-1	818	827	-9	UA	450	N494UA	LGA	IAH	190
193416	2013	5	1	548	600	-12	831	854	-23	B6	371	N523JB	LGA	FLL	140

Showing the first 1000 rows.

`distinct()` (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.distinct) filters out duplicate rows, and it considers all columns.

```
> m=spark_DF.select('month').filter(spark_DF.month < 9).filter('month>=5').distinct().collect()
  display(m)
```

month
5
6
7
8

`where()` is an alias for `filter()`.

```
> m=spark_DF.where("month <= 4")
  display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time
1	2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227
2	2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227
3	2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160
4	2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183
5	2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116

Showing the first 1000 rows.

```
> m=spark_DF.where(spark_DF.month==12)
  display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_ti
83162	2013	12	1	13	2359	14	446	445	1	B6	745	N715JB	JFK	PSE	195
83163	2013	12	1	17	2359	18	443	437	6	B6	839	N593JB	JFK	BQN	186
83164	2013	12	1	453	500	-7	636	651	-15	US	1895	N197UW	EWR	CLT	86
83165	2013	12	1	520	515	5	749	808	-19	UA	1487	N69804	EWR	IAH	193
83166	2013	12	1	536	540	-4	845	850	-5	AA	2243	N634AA	JFK	MIA	144

Showing the first 1000 rows.

We can also use user defined functions

```
> from pyspark.sql.types import BooleanType
  even_months = udf(lambda s: s%2==0, BooleanType())  # select even months only
  m = spark_DF.filter(even_months(spark_DF.month))
  display(m.select('month').distinct())    # want to see the distinct months
```

month
2
4
6
8
10
12

```
> even_months = udf(lambda s: s in (6,7,8,12), BooleanType())  # select June-August and December
  m = spark_DF.filter(even_months(spark_DF.month))
  display(m.select('month').distinct())
```

month
6
7
8
12

`groupBy()` ((http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.groupBy) is one of the most powerful transformations. It allows you to perform aggregations on a DataFrame.

Unlike other DataFrame transformations, `groupBy()` does *not* return a DataFrame. Instead, it returns a special GroupedData (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData) object that contains various aggregation functions.

The most commonly used aggregation function is count() (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.count), but there are others (like sum() (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.sum), max() (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.max), and avg() (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.avg).

These aggregation functions typically create a new column and return a new DataFrame.

```
> m=spark_DF.groupBy(spark_DF.carrier).avg('distance')
  display(m)
```

carrier	avg(distance)
AA	1340.2359986556264
HA	4983
AS	2402
UA	1529.1148725816074
B6	1068.621524663677
US	553.4562719127387
OO	500.8125
VX	2499.4821774506004
WN	996.269083503055

Averages of groups using DataFrames. `orderBy()` (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.distinct) allows you to sort a DataFrame by one or more columns, producing a new DataFrame.

```
> m=spark_DF.groupBy(spark_DF.carrier).avg('distance').orderBy('carrier')
  display(m)
```

carrier	avg(distance)
9E	530.235752979415
AA	1340.2359986556264
AS	2402
B6	1068.621524663677
DL	1236.9012055705675
EV	562.9917301977
F9	1620
FL	664.8294478527607
HA	4983

in orderBy(), the default is ascending but we can change it to descending by setting "ascending=False".

```
> m=spark_DF.groupBy(spark_DF.carrier).avg('distance').orderBy('carrier',ascending=False)
  display(m)
```

carrier	avg(distance)
YV	375.0332778702163
WN	996.269083503055
VX	2499.4821774506004
US	553.4562719127387
UA	1529.1148725816074
OO	500.8125
MQ	569.5327120506118
HA	4983
FL	664.8294478527607

```
> m=spark_DF.groupBy(spark_DF.carrier).avg('distance')
  display(m.sort('carrier',ascending=False))
```

carrier	avg(distance)
YV	375.0332778702163
WN	996.269083503055
VX	2499.4821774506004
US	553.4562719127387
UA	1529.1148725816074
OO	500.8125
MQ	569.5327120506118
HA	4983
FL	664.8294478527607

Use the `count` function (http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.count) to find the number of times each carrier occurs

```
> m=spark_DF.groupBy(spark_DF.carrier).count()
  display(m)
```

carrier	count
AA	32729
HA	342
AS	714
UA	58665
B6	54635
US	20536
OO	32
VX	5162
WN	12275

We can also calculate maximum, minimum and sum of groups using DataFrames

```
> m=spark_DF.groupBy(spark_DF.carrier).max('distance','hour')
  display(m)
```

carrier	max(distance)	max(hour)
AA	2586	21
HA	4983	10
AS	2402	18
UA	4963	23
B6	2586	23
US	2153	21
OO	1008	18
VX	2586	20
WN	2133	21

```
> m=spark_DF.groupBy(spark_DF.carrier).min('distance','hour')
  display(m)
```

carrier	min(distance)	min(hour)
AA	187	5
HA	4983	9
AS	2402	7
UA	116	5
B6	173	5
US	17	1
OO	229	11
VX	2248	7
WN	169	6

```
> m=spark_DF.groupBy(spark_DF.carrier).sum('distance')
  display(m)
```

carrier	sum(distance)
AA	43864584
HA	1704186
AS	1715028
UA	89705524
B6	58384137
US	11365778
OO	16026
VX	12902327
WN	12229203

SQL statements in Spark

```
# SQL statements can be run by using the sql methods provided by `spark`

# Register this DataFrame as a table.

spark_DF.registerTempTable("spark_DF")
```

```
m = sqlContext.sql("SELECT * FROM spark_DF WHERE month >= 5 AND month <= 9")
display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_t
193412	2013	5	1	9	1655	434	308	2020	408	VX	413	N628VA	JFK	LAX	341
193413	2013	5	1	451	500	-9	641	640	1	US	1219	N196UW	EWR	CLT	94
193414	2013	5	1	537	540	-3	836	840	-4	AA	701	N5BYAA	JFK	MIA	144
193415	2013	5	1	544	545	-1	818	827	-9	UA	450	N494UA	LGA	IAH	190
193416	2013	5	1	548	600	-12	831	854	-23	B6	371	N523JB	LGA	FLL	140

Showing the first 1000 rows.

```
m = sqlContext.sql("SELECT * FROM spark_DF WHERE month >= 5 AND month <= 9")
display(m)
```

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_t
193412	2013	5	1	9	1655	434	308	2020	408	VX	413	N628VA	JFK	LAX	341
193413	2013	5	1	451	500	-9	641	640	1	US	1219	N196UW	EWR	CLT	94
193414	2013	5	1	537	540	-3	836	840	-4	AA	701	N5BYAA	JFK	MIA	144
193415	2013	5	1	544	545	-1	818	827	-9	UA	450	N494UA	LGA	IAH	190
193416	2013	5	1	548	600	-12	831	854	-23	B6	371	N523JB	LGA	FLL	140

Showing the first 1000 rows.

```
> m = sqlContext.sql("SELECT carrier, max(distance) AS Maximum_Distance FROM spark_DF GROUP BY carrier ORDER BY
  Maximum_Distance DESC")
  display(m)
```

carrier	Maximum_Distance
HA	4983
UA	4963
AA	2586
B6	2586
DL	2586
VX	2586
AS	2402
US	2153
WN	2133

```
> m = sqlContext.sql("SELECT DISTINCT(carrier) FROM spark_DF")
  display(m)
```

carrier
AA
HA
AS
UA
B6
US
OO
VX
WN

```
> m = sqlContext.sql("SELECT DISTINCT(month) FROM spark_DF")
  display(m)
```

month
1
2
3
4
5
6
7
8
9

```
> m = sqlContext.sql("SELECT DISTINCT month FROM spark_DF WHERE month > 3 AND month < 10")
  display(m)
```

month
4
5
6

7
8
9

```
> m = sqlContext.sql("SELECT DISTINCT month FROM spark_DF WHERE month BETWEEN 4 AND 9")
  display(m)
```

month
4
5
6
7
8
9

```
> m = sqlContext.sql("SELECT carrier, max(distance) AS Maximum_Distance FROM spark_DF GROUP BY carrier ORDER BY
  Maximum_Distance DESC LIMIT 5")
  display(m)
```

carrier	Maximum_Distance
HA	4983
UA	4963
B6	2586
VX	2586
AA	2586

```
> m = sqlContext.sql("SELECT carrier, max(distance) AS Maximum_Distance FROM spark_DF GROUP BY carrier HAVING
  Maximum_Distance > 2000 ORDER BY Maximum_Distance DESC")

  display(m)
```

carrier	Maximum_Distance
HA	4983
UA	4963
B6	2586
AA	2586
VX	2586
DL	2586
AS	2402
US	2153
WN	2133

```
> m = sqlContext.sql("SELECT DISTINCT month FROM spark_DF WHERE month !=2")
  display(m)
```

month
1
3

4
5
6
7
8
9

```
> m = sqlContext.sql("SELECT DISTINCT carrier FROM spark_DF WHERE month =1 OR month = 12")
  display(m)
```

carrier
AA
HA
AS
UA
B6
US
OO
VX
WN

carrier
AA
AS