Introduction to Google Colab and PySpark

Table Of Contents:

Objective

The objective of this notebook is to:

> - Give a proper understanding about the different PySpark functions available.
> - A short introduction to Google Colab, as that is the platform on which this notebook is written on.

Once you complete this notebook, you should be able to write pyspark programs in an efficent way. The ideal way to use this is by going through the examples given and then trying them on Colab. At the end there are a few hands on questions which you can use to evaluate yourself.

Prerequisite

> - Although some theory about pyspark and big data will be given in this notebook, I recommend everyone to read more about it and have a deeper understanding on how the functions get executed and the relevance of big data in the current scenario.
> - A good understanding on python will be an added bonus.

Notes from the author

This tutorial was made using Google Colab so the code you see here is meant to run on a colab notebook.
It goes through basic PySpark Functions (https://spark.apache.org/docs/latest/api/python/index.html) and a short introduction on how to use Colab (https://colab.research.google.com/notebooks/basic_features_overview.ipynb).
If you want to view my colab notebook for this particular tutorial, you can view it here (https://colab.research.google.com/drive/1G894WS7ItIUTusWWmsCnF_zQhQqZCDOc). The viewing experience and readability is much better there.
If you want to try out things with this notebook as a base, feel free to download it from my repo here (https://github.com/jacobceles/knowledge-repo/blob/master/pyspark/Colab%20and%20PySpark.ipynb) and then use it with jupyter notebook.

Big data, PySpark and Colaboratory

Big data

Big data usually means data of such huge volume that normal data storage solutions cannot efficently store and process it. In this era, data is being generated at an absurd rate. Data is collected for each movement a person makes. The bulk of big data comes from three primary sources:

1. Social data
2. Machine data
3. Transactional data

Some common examples for the sources of such data include internet searches, facebook posts, doorbell cams, smartwatches, online shopping history etc. Every action creates data, it is just a matter of of there is a way to collect them or not. But what's interesting is that out of all this data collected, not even 5% of it is being used fully. There is a huge demand for big data professionals in the industry. Even though the number of graduates with a specialization in big data are rising, the problem is that they don't have the practical knowledge about big data scenarios, which leads to bad architecutres and inefficent methods of processing data.

> If you are interested to know more about the landscape and technologies involved, here is an article (https://hostingtribunal.com/blog/big-data-stats/) which I found really interesting!

PySpark

If you are working in the field of big data, you must have definelty heard of spark. If you look at the Apache Spark (https://spark.apache.org/) website, you will see that it is said to be a `Lightning-fast unified analytics engine` . PySPark is a flavour of Spark used for processing and analysing massive volumes of data. If you are familiar with python and have tried it for huge datasets, you should know that the execution time can get ridiculous. Enter PySpark!

Imagine your data resides in a distributed manner at different places. If you try brining your data to one point and executing your code there, not only would that be inefficent, but also cause memory issues. Now let's say your code goes to the data rather than the data coming to where your code. This will help avoid unneccesary data movement which will thereby decrease the running time.

PySpark is the Python API of Spark; which means it can do almost all the things python can. Machine learning(ML) pipelines, exploratory data analysis (at scale), ETLs for data platform, and much more! And all of them in a distributed manner. One of the best parts of pyspark is that if you are already familiar with python, it's really easy to learn.

Apart from PySpark, there is another language called Scala used for big data processing. Scala is frequently over 10 times faster than Python is native for Hadoop as its based on JVM. But PySpark is getting adopted at a fast rate because of the ease of use, easier learning curve and ML capabilities.

I will briefly explain how a PySpark job works, but I strongly recommend you read more about the architecture (https://data-flair.training/blogs/how-apache-spark-works/) and how everything works. Now, before I get into it, let me talk about some basic jargons first:

Cluster is a set of loosely or tightly connected computers that work together so that they can be viewed as a single system.

Hadoop is an open source, scalable, and fault tolerant framework written in Java. It efficiently processes large volumes of data on a cluster of commodity hardware. Hadoop is not only a storage system but is a platform for large data storage as well as processing.

HDFS (Hadoop distributed file system). It is one of the world's most reliable storage system. HDFS is a Filesystem of Hadoop designed for storing very large files running on a cluster of commodity hardware.

MapReduce is a data Processing framework, which has 2 phases - Mapper and Reducer. The map procedure performs filtering and sorting, and the reduce method performs a summary operation. It usually runs on a hadoop cluster.

Transformation refers to the operations applied on a dataset to create a new dataset. Filter, groupBy and map are the examples of transformations.

Actions Actions refer to an operation which instructs Spark to perform computation and send the result back to driver. This is an example of action.

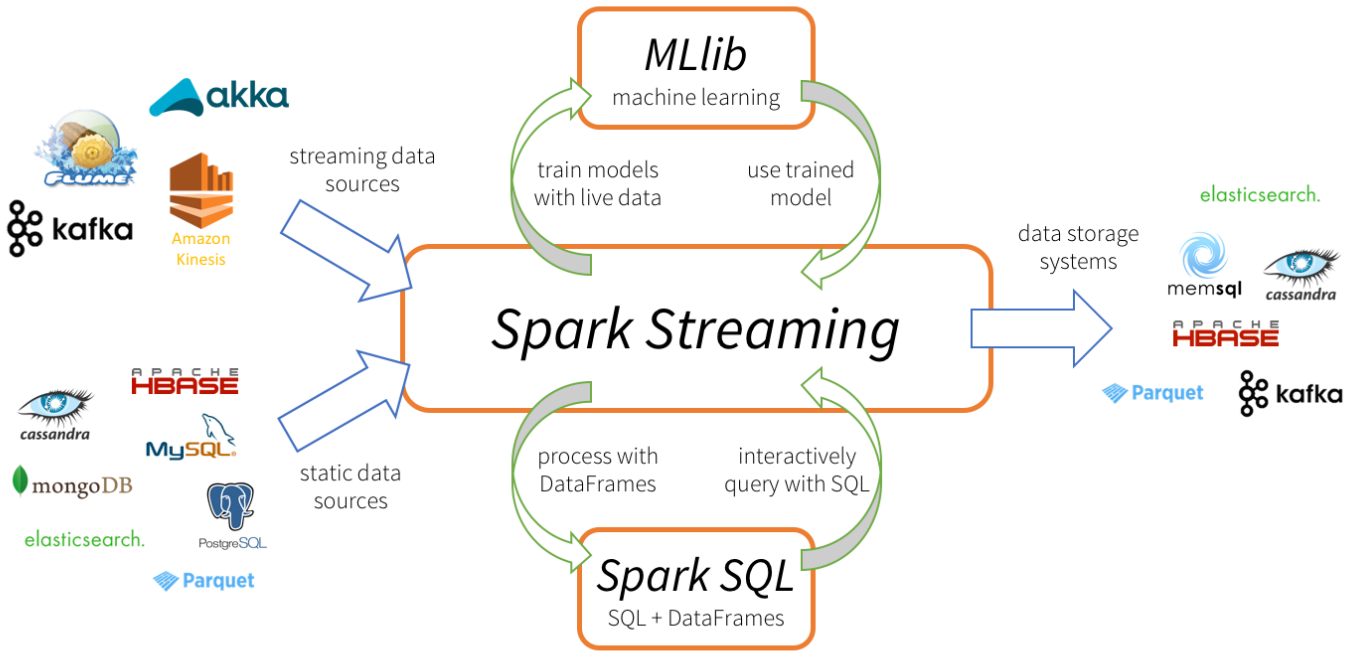
Alright! Now that that's out of the way, let me explain how a spark job runs. In simple terma, each time you submit a pyspark job, the code gets internally converted into a MapReduce program and gets executed in the Java virtual machine. Now one of the thoughts that might be popping in your mind will probably be:
`So the code gets converted into a MapReduce program. Wouldn't that mean MapReduce is faster than pySpark?`
Well, the answer is a big NO. This is what makes spark jobs special. Spark is capable of handling a massive

amount of data at a time, in it's distributed environment. It does this through in-memory processing, which is what makes it almost 100 times faster than Hadoop. Another factor which amkes it fast is Lazy Evaluation. Spark delays its evaluation as much as it can. Each time you submit a job, spark creates an action plan for how to execute the code, and then does nothing. Finally, when you ask for the result(i.e, calls an action), it executes the plan, which is basically all the transofrmations you have mentioned in your code. That's basically the gist of it.

Now lastly, I want to talk about on more thing. Spark mainly consists of 4 modules:

1. Spark SQL - helps to write spark programs using SQL like queries.
2. Spark Streaming - is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. used heavily in processing of social media data.
3. Spark MLLib - is the machine learning component of SPark. It helps train ML models on massive datasets with very high efficeny.
4. Spark GraphX - is the visualization component of Spark. It enables users to view data both as graphs and as collections without data movement or duplication.

Hopefully this image gives a better idea of what I am talking about:



Source: Datanami

Colaboratory

In the words of Google:
`Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs.`

The reason why I used colab is because of its shareability and free GPU and TPU. Yeah you read that right, FREE GPU AND TPU! For using TPU, your program needs to be optimized for the same. Additionally, it helps use different Google services conveniently. It saves to Google Drive and all the services are very closely related. I recommend you go through the offical overview documentation (https://colab.research.google.com/notebooks/basic_features_overview.ipynb) if you want to know more about it. If you have more questions about colab, please refer this link (https://research.google.com/colaboratory/faq.html).

> While using a colab notebook, you will need an active internet connection to keep a session alive. If you lose the connection you will have to download the datasets again.

Jupyter notebook basics

Code cells

```
In [0]: 2*3
Out[0]: 6

In [0]: import pandas as pd

In [0]: print("Hello!")
        Hello!
```

Text cells

Hello world!

Access to the shell

In [0]:
```
ls
```

```
sample_data/
```

In [0]:
```
pwd
```

Out[0]:
```
'/content'
```

Install Spark

In [0]:
```
!apt-get update
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://archive.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-hadoop2
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
!pip install -q findspark
```

```
Ign:1 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64
Ign:2 https://developer.download.nvidia.com/compute/machine-learning/repos/ubunt
Hit:3 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64
Hit:4 https://developer.download.nvidia.com/compute/machine-learning/repos/ubunt
Get:5 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Hit:7 http://archive.ubuntu.com/ubuntu bionic InRelease
Get:8 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu bionic InRelease [21.
Get:10 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Get:11 http://ppa.launchpad.net/marutter/c2d4u3.5/ubuntu bionic InRelease [15.4
Get:12 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [90
Get:13 https://cloud.r-project.org/bin/linux/ubuntu bionic-cran35/ InRelease [3,
Get:14 http://archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Get:15 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu bionic/main amd64 Pa
Get:16 https://cloud.r-project.org/bin/linux/ubuntu bionic-cran35/ Packages [91.
Get:17 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [1,20
Get:18 http://security.ubuntu.com/ubuntu bionic-security/restricted amd64 Packag
Get:19 http://security.ubuntu.com/ubuntu bionic-security/multiverse amd64 Packag
Get:20 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages
Get:21 http://ppa.launchpad.net/marutter/c2d4u3.5/ubuntu bionic/main Sources [1,
Get:22 http://archive.ubuntu.com/ubuntu bionic-updates/restricted amd64 Packages
Get:23 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages [
Get:24 http://archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 Packages
Get:25 http://archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [8,
Get:26 http://archive.ubuntu.com/ubuntu bionic-backports/universe amd64 Packages
Get:27 http://ppa.launchpad.net/marutter/c2d4u3.5/ubuntu bionic/main amd64 Packa
Fetched 7,606 kB in 8s (1,014 kB/s)
Reading package lists... Done
```

In [0]:
```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.5-bin-hadoop2.7"
```

In [0]:
```
!ls
```

```
sample_data  spark-2.4.5-bin-hadoop2.7 spark-2.4.5-bin-hadoop2.7.tgz
```

In [0]:
```
import findspark
findspark.init()
from pyspark import SparkContext

sc = SparkContext.getOrCreate()
sc
```

Out[0]:

SparkContext

Spark UI (http://4f2d21480af5:4040)

Version
v2.4.5
Master
local[*]
AppName
pyspark-shell

In [0]:
```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
spark
```

Out[0]:

SparkSession - in-memory
SparkContext

Spark UI (http://4f2d21480af5:4040)

Version
v2.4.5
Master
local[*]
AppName
pyspark-shell

Loading Dataset

In [0]:
```
# Downloading and preprocessing Chicago's Reported Crime Data
!wget https://data.cityofchicago.org/api/views/w98m-zvie/rows.csv?accessType=DOWN
```

```
--2020-05-02 15:12:16--  https://data.cityofchicago.org/api/views/w98m-zvie/rows
Resolving data.cityofchicago.org (data.cityofchicago.org)... 52.206.68.26, 52.20
Connecting to data.cityofchicago.org (data.cityofchicago.org)|52.206.68.26|:443.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/csv]
Saving to: ‘rows.csv?accessType=DOWNLOAD’

rows.csv?accessType     [               <=> ]  58.90M  3.21MB/s    in 19s

2020-05-02 15:12:41 (3.14 MB/s) - ‘rows.csv?accessType=DOWNLOAD’ saved [61759120
```

In [0]:
```
!ls
```

```
'rows.csv?accessType=DOWNLOAD'  spark-2.4.5-bin-hadoop2.7
 sample_data                    spark-2.4.5-bin-hadoop2.7.tgz
```

In [0]:
```
#Renaming the downloaded file
!mv rows.csv?accessType=DOWNLOAD reported-crimes.csv
```

In [0]:
```
df = spark.read.csv('reported-crimes.csv',header=True)
spark.conf.set("spark.sql.repl.eagerEval.enabled", True)
df.show(5)
```

```
+--------+-----------+--------------------+--------------------+----+-----------
|      ID|Case Number|                Date|               Block|IUCR|      Prima
+--------+-----------+--------------------+--------------------+----+-----------
|12040452|   JD220630|11/21/2019 12:00:...|  004XX N WABASH AVE|1153|DECEPTIVE P
|12039199|   JD219081|04/01/2019 08:00:...| 021XX S INDIANA AVE|1153|DECEPTIVE P
|12040466|   JD220650|11/07/2019 12:00:...|026XX N NEW ENGLA...|1152|DECEPTIVE P
|11938499|   JD100565|12/31/2019 11:30:...|070XX S COTTAGE G...|0460|
|11864640|   JC477069|10/18/2019 09:56:...|002XX S LAVERGNE AVE|041A|
+--------+-----------+--------------------+--------------------+----+-----------
only showing top 5 rows
```

Working with the Dataframe API

Viewing Dataframe

In Spark, you have a couple of options to view the DataFrame(DF).

1. `take(3)` will return a list of three row objects.
2. `df.collect()` will get all of the data from the entire DataFrame . Be careful when using it, because if you have a large data set when you run collect, you can easily crash the driver node.
3. If you want Spark to print out your DataFrame in a nice format, then try `df.show()` with the number of rows as paramter. You might notice that the show functions truncates the data as inthe example above. You can use the parameter `truncate=False` to show the entire data. For example, you can use `df.show(5, False)` or `df.show(truncate=False)` to disable truncation of data.

> The limit function **returns a new DataFrame** by taking the first n rows.

Schema of a DataFrame

In [0]:
```
df.dtypes
```

Out[0]:
```
[('ID', 'string'),
 ('Case Number', 'string'),
 ('Date', 'string'),
 ('Block', 'string'),
 ('IUCR', 'string'),
 ('Primary Type', 'string'),
 ('Description', 'string'),
 ('Location Description', 'string'),
 ('Arrest', 'string'),
 ('Domestic', 'string'),
 ('Beat', 'string'),
 ('District', 'string'),
 ('Ward', 'string'),
 ('Community Area', 'string'),
 ('FBI Code', 'string'),
 ('X Coordinate', 'string'),
 ('Y Coordinate', 'string'),
 ('Year', 'string'),
 ('Updated On', 'string'),
 ('Latitude', 'string'),
 ('Longitude', 'string'),
 ('Location', 'string')]
```

In [0]:
```
df.printSchema()
```

```
root
 |-- ID: string (nullable = true)
 |-- Case Number: string (nullable = true)
 |-- Date: string (nullable = true)
 |-- Block: string (nullable = true)
 |-- IUCR: string (nullable = true)
 |-- Primary Type: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Location Description: string (nullable = true)
 |-- Arrest: string (nullable = true)
 |-- Domestic: string (nullable = true)
 |-- Beat: string (nullable = true)
 |-- District: string (nullable = true)
 |-- Ward: string (nullable = true)
 |-- Community Area: string (nullable = true)
 |-- FBI Code: string (nullable = true)
 |-- X Coordinate: string (nullable = true)
 |-- Y Coordinate: string (nullable = true)
 |-- Year: string (nullable = true)
 |-- Updated On: string (nullable = true)
 |-- Latitude: string (nullable = true)
 |-- Longitude: string (nullable = true)
 |-- Location: string (nullable = true)
```

In [0]:
```
# Defining a schema
from pyspark.sql.types import StructType, StructField, StringType, TimestampType
df.columns
```

Out[0]:
```
['ID',
 'Case Number',
 'Date',
 'Block',
 'IUCR',
 'Primary Type',
 'Description',
 'Location Description',
 'Arrest',
 'Domestic',
 'Beat',
 'District',
 'Ward',
 'Community Area',
 'FBI Code',
 'X Coordinate',
 'Y Coordinate',
 'Year',
 'Updated On',
 'Latitude',
 'Longitude',
 'Location']
```

In [0]:
```
labels = [
     ('ID',StringType()),
     ('Case Number',StringType()),
     ('Date',TimestampType()),
     ('Block',StringType()),
     ('IUCR',StringType()),
     ('Primary Type',StringType()),
     ('Description',StringType()),
     ('Location Description',StringType()),
     ('Arrest',StringType()),
     ('Domestic',BooleanType()),
     ('Beat',StringType()),
     ('District',StringType()),
     ('Ward',StringType()),
     ('Community Area',StringType()),
     ('FBI Code',StringType()),
     ('X Coordinate',StringType()),
     ('Y Coordinate',StringType()),
     ('Year',IntegerType()),
     ('Updated On',StringType()),
     ('Latitude',DoubleType()),
     ('Longitude',DoubleType()),
     ('Location',StringType()),
     ('Historical Wards 2003-2015',StringType()),
     ('Zip Codes',StringType()),
     ('Community Areas',StringType()),
     ('Census Tracts',StringType()),
     ('Wards',StringType()),
     ('Boundaries - ZIP Codes',StringType()),
     ('Police Districts',StringType()),
     ('Police Beats',StringType())
]
```

In [0]:
```
schema = StructType([StructField (x[0], x[1], True) for x in labels])
schema
```

Out[0]:
```
StructType(List(StructField(ID,StringType,true),StructField(Case Number,StringTy
```

In [0]:
```
df = spark.read.csv('reported-crimes.csv',schema=schema)
df.printSchema()
# The schema comes as we gave!
```

```
root
 |-- ID: string (nullable = true)
 |-- Case Number: string (nullable = true)
 |-- Date: timestamp (nullable = true)
 |-- Block: string (nullable = true)
 |-- IUCR: string (nullable = true)
 |-- Primary Type: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Location Description: string (nullable = true)
 |-- Arrest: string (nullable = true)
 |-- Domestic: boolean (nullable = true)
 |-- Beat: string (nullable = true)
 |-- District: string (nullable = true)
 |-- Ward: string (nullable = true)
 |-- Community Area: string (nullable = true)
 |-- FBI Code: string (nullable = true)
 |-- X Coordinate: string (nullable = true)
 |-- Y Coordinate: string (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Updated On: string (nullable = true)
 |-- Latitude: double (nullable = true)
 |-- Longitude: double (nullable = true)
 |-- Location: string (nullable = true)
 |-- Historical Wards 2003-2015: string (nullable = true)
 |-- Zip Codes: string (nullable = true)
 |-- Community Areas: string (nullable = true)
 |-- Census Tracts: string (nullable = true)
 |-- Wards: string (nullable = true)
 |-- Boundaries - ZIP Codes: string (nullable = true)
 |-- Police Districts: string (nullable = true)
 |-- Police Beats: string (nullable = true)
```

In [0]:
```
df.show()
# This comes as null which means the datatypes we gave were wrong.
```

```
+----+-----------+----+-----+----+------------+-----------+--------------------+
|  ID|Case Number|Date|Block|IUCR|Primary Type|Description|Location Description|
+----+-----------+----+-----+----+------------+-----------+--------------------+
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
|null|       null|null| null|null|        null|       null|                null|
+----+-----------+----+-----+----+------------+-----------+--------------------+
only showing top 20 rows
```

In [0]:
```
# So let's just stick with infered schema for now, casting date to date type on i
from pyspark.sql.functions import col, to_timestamp
df = spark.read.csv('reported-crimes.csv',header=True).withColumn('Date',to_time:
df.show(5, False)
```

```
+--------+-----------+-------------------+-------------------------+----+-------
|ID      |Case Number|Date               |Block                    |IUCR|Primary
+--------+-----------+-------------------+-------------------------+----+-------
|12040452|JD220630   |2019-11-21 12:00:00|004XX N WABASH AVE       |1153|DECEPTI
|12039199|JD219081   |2019-04-01 08:00:00|021XX S INDIANA AVE      |1153|DECEPTI
|12040466|JD220650   |2019-11-07 12:00:00|026XX N NEW ENGLAND AVE  |1152|DECEPTI
|11938499|JD100565   |2019-12-31 11:30:00|070XX S COTTAGE GROVE AVE|0460|BATTERY
|11864640|JC477069   |2019-10-18 09:56:00|002XX S LAVERGNE AVE     |041A|BATTERY
+--------+-----------+-------------------+-------------------------+----+-------
only showing top 5 rows
```

Working with Columns

In [0]:
```
df.Block
```

Out[0]:
```
Column<b'Block'>
```

In [0]:
```
df['Block']
```

Out[0]:
```
Column<b'Block'>
```

NOTE:

> **We can't always use the dot notation because this will break when the column names have reserved names or attributes to the data frame class.**

In [0]:
```
df.select(col('Block')).show(truncate=False)
```

```
+-------------------------+
|Block                    |
+-------------------------+
|004XX N WABASH AVE       |
|021XX S INDIANA AVE      |
|026XX N NEW ENGLAND AVE  |
|070XX S COTTAGE GROVE AVE|
|002XX S LAVERGNE AVE     |
|024XX W FOSTER AVE       |
|048XX W AUGUSTA BLVD     |
|062XX S EMERALD DR       |
|062XX S FRANCISCO AVE    |
|120XX S LAFLIN ST        |
|049XX N ALBANY AVE       |
|021XX S HARDING AVE      |
|026XX W BALMORAL AVE     |
|071XX S CYRIL AVE        |
|028XX S CHRISTIANA AVE   |
|052XX S ARCHER AVE       |
|057XX W ROOSEVELT RD     |
|003XX S SPRINGFIELD AVE  |
|002XX N MENARD AVE       |
|003XX N MENARD AVE       |
+-------------------------+
only showing top 20 rows
```

In [0]:
```
df.select(df.Block).show(truncate=False)
```

```
+-------------------------+
|Block                    |
+-------------------------+
|004XX N WABASH AVE       |
|021XX S INDIANA AVE      |
|026XX N NEW ENGLAND AVE  |
|070XX S COTTAGE GROVE AVE|
|002XX S LAVERGNE AVE     |
|024XX W FOSTER AVE       |
|048XX W AUGUSTA BLVD     |
|062XX S EMERALD DR       |
|062XX S FRANCISCO AVE    |
|120XX S LAFLIN ST        |
|049XX N ALBANY AVE       |
|021XX S HARDING AVE      |
|026XX W BALMORAL AVE     |
|071XX S CYRIL AVE        |
|028XX S CHRISTIANA AVE   |
|052XX S ARCHER AVE       |
|057XX W ROOSEVELT RD     |
|003XX S SPRINGFIELD AVE  |
|002XX N MENARD AVE       |
|003XX N MENARD AVE       |
+-------------------------+
only showing top 20 rows
```

In [0]:
```
df.select('Block','Description').show(truncate=False)
```

```
+------------------------+--------------------------------------------------+
|Block                   |Description                                       |
+------------------------+--------------------------------------------------+
|004XX N WABASH AVE      |FINANCIAL IDENTITY THEFT OVER $ 300               |
|021XX S INDIANA AVE     |FINANCIAL IDENTITY THEFT OVER $ 300               |
|026XX N NEW ENGLAND AVE |ILLEGAL USE CASH CARD                             |
|070XX S COTTAGE GROVE AVE|SIMPLE                                           |
|002XX S LAVERGNE AVE    |AGGRAVATED - HANDGUN                              |
|024XX W FOSTER AVE      |FORCIBLE ENTRY                                    |
|048XX W AUGUSTA BLVD    |AGGRAVATED - HANDGUN                              |
|062XX S EMERALD DR      |CHILD ABUSE                                       |
|062XX S FRANCISCO AVE   |AGGRAVATED - HANDGUN                              |
|120XX S LAFLIN ST       |HARASSMENT BY ELECTRONIC MEANS                    |
|049XX N ALBANY AVE      |FROM BUILDING                                     |
|021XX S HARDING AVE     |AGGRAVATED CRIMINAL SEXUAL ABUSE BY FAMILY MEMBER|
|026XX W BALMORAL AVE    |FINANCIAL IDENTITY THEFT OVER $ 300               |
|071XX S CYRIL AVE       |FROM BUILDING                                     |
|028XX S CHRISTIANA AVE  |OTHER VEHICLE OFFENSE                             |
|052XX S ARCHER AVE      |AUTOMOBILE                                        |
|057XX W ROOSEVELT RD    |AGGRAVATED - OTHER                                |
|003XX S SPRINGFIELD AVE |FINANCIAL IDENTITY THEFT $300 AND UNDER           |
|002XX N MENARD AVE      |FINANCIAL IDENTITY THEFT $300 AND UNDER           |
|003XX N MENARD AVE      |FINANCIAL IDENTITY THEFT OVER $ 300               |
+------------------------+--------------------------------------------------+
only showing top 20 rows
```

Adding a New Column

In [0]:
```
#Adding a column in PySpark
# We are adding a column called 'One' at the end
from pyspark.sql.functions import lit
df = df.withColumn('One',lit(1))
df.show(5,truncate=False)
```

```
+--------+-----------+-------------------+------------------------+----+-------
|ID      |Case Number|Date               |Block                   |IUCR|Primary
+--------+-----------+-------------------+------------------------+----+-------
|12040452|JD220630   |2019-11-21 12:00:00|004XX N WABASH AVE      |1153|DECEPTI
|12039199|JD219081   |2019-04-01 08:00:00|021XX S INDIANA AVE     |1153|DECEPTI
|12040466|JD220650   |2019-11-07 12:00:00|026XX N NEW ENGLAND AVE |1152|DECEPTI
|11938499|JD100565   |2019-12-31 11:30:00|070XX S COTTAGE GROVE AVE|0460|BATTERY
|11864640|JC477069   |2019-10-18 09:56:00|002XX S LAVERGNE AVE    |041A|BATTERY
+--------+-----------+-------------------+------------------------+----+-------
only showing top 5 rows
```

Renaming a Column

In [0]:
```
#Renaming a column in PySpark
df = df.withColumnRenamed('One', 'Test')
df.show(truncate=False)
```

```
+--------+-----------+-------------------+-------------------------+----+-------
|ID      |Case Number|Date               |Block                    |IUCR|Primary
+--------+-----------+-------------------+-------------------------+----+-------
|12040452|JD220630   |2019-11-21 12:00:00|004XX N WABASH AVE       |1153|DECEPTI
|12039199|JD219081   |2019-04-01 08:00:00|021XX S INDIANA AVE      |1153|DECEPTI
|12040466|JD220650   |2019-11-07 12:00:00|026XX N NEW ENGLAND AVE  |1152|DECEPTI
|11938499|JD100565   |2019-12-31 11:30:00|070XX S COTTAGE GROVE AVE|0460|BATTERY
|11864640|JC477069   |2019-10-18 09:56:00|002XX S LAVERGNE AVE     |041A|BATTERY
|11766889|JC359918   |2019-07-17 15:07:00|024XX W FOSTER AVE       |0610|BURGLAR
|11740807|JC328262   |2019-06-30 05:04:00|048XX W AUGUSTA BLVD     |041A|BATTERY
|11718084|JC300874   |2019-05-13 13:00:00|062XX S EMERALD DR       |1750|OFFENSE
|11647696|JC215453   |2019-04-07 17:56:00|062XX S FRANCISCO AVE    |041A|BATTERY
|12039931|JD220113   |2019-10-18 08:00:00|120XX S LAFLIN ST        |2826|OTHER O
|12039974|JD220130   |2019-12-24 23:00:00|049XX N ALBANY AVE       |0890|THEFT
|12039885|JD219909   |2019-10-01 02:00:00|021XX S HARDING AVE      |1752|OFFENSE
|12039884|JD220045   |2019-07-01 12:00:00|026XX W BALMORAL AVE     |1153|DECEPTI
|12038367|JD218291   |2019-04-21 12:00:00|071XX S CYRIL AVE        |0890|THEFT
|12038544|JD218327   |2019-12-22 13:00:00|028XX S CHRISTIANA AVE   |5002|OTHER O
|12038308|JD218196   |2019-07-22 08:00:00|052XX S ARCHER AVE       |0910|MOTOR V
|12040067|JD220150   |2019-12-15 18:00:00|057XX W ROOSEVELT RD     |0265|CRIMINA
|12039858|JD219964   |2019-09-03 09:00:00|003XX S SPRINGFIELD AVE  |1154|DECEPTI
|12038342|JD218257   |2019-05-01 00:00:00|002XX N MENARD AVE       |1154|DECEPTI
|12038644|JD218194   |2019-06-01 13:45:00|003XX N MENARD AVE       |1153|DECEPTI
+--------+-----------+-------------------+-------------------------+----+-------
only showing top 20 rows
```

Grouping By Column

In [0]:
```
#Group By a column in PySpark
df.groupBy('Year').count().show(5)
```

```
+----+------+
|Year| count|
+----+------+
|2019|259013|
+----+------+
```

Removing a Column

In [0]:
```
#Remove columns in PySpark
df = df.drop('Test')
df.show(5,truncate=False)
```

```
+--------+-----------+-------------------+-------------------------+----+-------
|ID      |Case Number|Date               |Block                    |IUCR|Primary
+--------+-----------+-------------------+-------------------------+----+-------
|12040452|JD220630   |2019-11-21 12:00:00|004XX N WABASH AVE       |1153|DECEPTI
|12039199|JD219081   |2019-04-01 08:00:00|021XX S INDIANA AVE      |1153|DECEPTI
|12040466|JD220650   |2019-11-07 12:00:00|026XX N NEW ENGLAND AVE  |1152|DECEPTI
|11938499|JD100565   |2019-12-31 11:30:00|070XX S COTTAGE GROVE AVE|0460|BATTERY
|11864640|JC477069   |2019-10-18 09:56:00|002XX S LAVERGNE AVE     |041A|BATTERY
+--------+-----------+-------------------+-------------------------+----+-------
only showing top 5 rows
```

Working with Rows

Filtering Rows

In [0]:
```
# Filtering rows in PySpark
df.filter(col('Date')<'2019-06-01').show(truncate=False)
```

```
+--------+-----------+-------------------+---------------------+----+----------
|ID      |Case Number|Date               |Block                |IUCR|Primary Ty
+--------+-----------+-------------------+---------------------+----+----------
|12039199|JD219081   |2019-04-01 08:00:00|021XX S INDIANA AVE  |1153|DECEPTIVE
|11718084|JC300874   |2019-05-13 13:00:00|062XX S EMERALD DR   |1750|OFFENSE IN
|11647696|JC215453   |2019-04-07 17:56:00|062XX S FRANCISCO AVE|041A|BATTERY
|12038367|JD218291   |2019-04-21 12:00:00|071XX S CYRIL AVE    |0890|THEFT
|12038342|JD218257   |2019-05-01 00:00:00|002XX N MENARD AVE   |1154|DECEPTIVE
|11994131|JD167522   |2019-01-01 09:00:00|013XX W 110TH PL     |1195|DECEPTIVE
|11696379|JC273783   |2019-05-22 14:40:00|029XX S FORT DEARBORN|041A|BATTERY
|11625478|JC188359   |2019-03-16 20:39:00|068XX N OVERHILL AVE |0610|BURGLARY
|12036474|JD216160   |2019-04-05 10:00:00|003XX W HUBBARD ST   |0810|THEFT
|11566441|JC116418   |2019-01-14 03:00:00|036XX N WHIPPLE ST   |0265|CRIMINAL S
|11640117|JC205456   |2019-03-27 09:00:00|082XX S LAFLIN ST    |0281|CRIMINAL S
|11825583|JC430037   |2019-04-02 06:00:00|0000X S CONCOURSE A ST|0810|THEFT
|11699523|JC278066   |2019-05-25 12:49:00|028XX W 51ST ST      |031A|ROBBERY
|11677547|JC250749   |2019-05-05 13:36:00|008XX E 75TH ST      |051A|ASSAULT
|12035312|JD214438   |2019-03-29 08:00:00|049XX W ARTHINGTON ST|1153|DECEPTIVE
|12035047|JD214561   |2019-05-01 00:00:00|002XX W 24TH PL      |1153|DECEPTIVE
|11618719|JC180138   |2019-03-10 10:54:00|010XX S CLARK ST     |0479|BATTERY
|12034436|JD211780   |2019-02-19 09:00:00|055XX W WILSON AVE   |1153|DECEPTIVE
|11700425|JC279076   |2019-05-26 06:09:00|056XX W NORTH AVE    |041A|BATTERY
|11586737|JC140804   |2019-02-04 23:00:00|002XX N WABASH AVE   |0281|CRIMINAL S
+--------+-----------+-------------------+---------------------+----+----------
only showing top 20 rows
```

Get Distinct Rows

In [0]:
```
#Get Unique Rows in PySpark
df.select('Year').distinct().show()
```

```
+----+
|Year|
+----+
|2019|
+----+
```

Sorting Rows

In [0]:
```
# Sort Rows in PySpark
# By default the data will be sorted in ascending order
df.orderBy('Date').show(truncate=False)
```

```
+--------+-----------+-------------------+----------------------+----+----------
|ID      |Case Number|Date               |Block                 |IUCR|Primary Ty
+--------+-----------+-------------------+----------------------+----+----------
|11552674|JC100085   |2019-01-01 00:00:00|092XX S NORMAL AVE    |0910|MOTOR VEHI
|11560011|JC108820   |2019-01-01 00:00:00|068XX S OGLESBY AVE   |0820|THEFT
|11730841|JC315506   |2019-01-01 00:00:00|007XX E 95TH ST       |1563|SEX OFFENS
|11718021|JC300596   |2019-01-01 00:00:00|102XX W ZEMKE RD      |0910|MOTOR VEHI
|11714518|JC294379   |2019-01-01 00:00:00|0000X W 115TH ST      |1210|DECEPTIVE
|11682859|JC256913   |2019-01-01 00:00:00|074XX S HARVARD AVE   |1753|OFFENSE IN
|11552667|JC100123   |2019-01-01 00:00:00|004XX N STATE ST      |0890|THEFT
|11723841|JC300604   |2019-01-01 00:00:00|102XX W ZEMKE RD      |0910|MOTOR VEHI
|11706371|JC286255   |2019-01-01 00:00:00|116XX S ADA ST        |1544|SEX OFFENS
|11552709|JC100020   |2019-01-01 00:00:00|044XX S WASHTENAW AVE |0486|BATTERY
|11552758|JC100058   |2019-01-01 00:00:00|063XX S MARSHFIELD AVE|1310|CRIMINAL D
|11558163|JC106702   |2019-01-01 00:00:00|002XX W ONTARIO ST    |0890|THEFT
|11553168|JC100745   |2019-01-01 00:00:00|008XX N MICHIGAN AVE  |0890|THEFT
|11553381|JC100934   |2019-01-01 00:00:00|022XX N LEAMINGTON AVE|1320|CRIMINAL D
|11553495|JC101115   |2019-01-01 00:00:00|047XX N RACINE AVE    |0281|CRIM SEXUA
|11553914|JC101459   |2019-01-01 00:00:00|062XX W BELMONT AVE   |1310|CRIMINAL D
|11556297|JC104365   |2019-01-01 00:00:00|083XX S YATES BLVD    |2826|OTHER OFFE
|11563351|JC112682   |2019-01-01 00:00:00|045XX S EVANS AVE     |2825|OTHER OFFE
|11566200|JC116315   |2019-01-01 00:00:00|072XX S EAST END AVE  |0320|ROBBERY
|11571975|JC123039   |2019-01-01 00:00:00|0000X N LOCKWOOD AVE  |1153|DECEPTIVE
+--------+-----------+-------------------+----------------------+----+----------
only showing top 20 rows
```

In [0]:
```
# To change the sorting order, you can use the asceding parameter
df.orderBy('Date', ascending=False).show(truncate=False)
```

```
+--------+-----------+-------------------+----------------------+----+----------
|ID      |Case Number|Date               |Block                 |IUCR|Primary Ty
+--------+-----------+-------------------+----------------------+----+----------
|11938228|JD100017   |2019-12-31 23:55:00|0000X W 69TH ST       |143A|WEAPONS VI
|11940078|JD100016   |2019-12-31 23:54:00|063XX S MAY ST        |0420|BATTERY
|11938857|JD100599   |2019-12-31 23:50:00|004XX N Ashland ave   |0820|THEFT
|11938240|JD100002   |2019-12-31 23:48:00|004XX S CICERO AVE    |143A|WEAPONS VI
|11937967|JC567053   |2019-12-31 23:46:00|034XX W JACKSON BLVD  |143A|WEAPONS VI
|11938124|JD100001   |2019-12-31 23:37:00|012XX W 99TH ST       |5112|OTHER OFFE
|11937990|JC567039   |2019-12-31 23:31:00|006XX W 47TH ST       |0460|BATTERY
|11937983|JC567043   |2019-12-31 23:30:00|049XX N CHRISTIANA AVE|051A|ASSAULT
|11937963|JC567049   |2019-12-31 23:21:00|101XX S NORMAL AVE    |5111|OTHER OFFE
|11939218|JD100095   |2019-12-31 23:15:00|001XX N CLARK ST      |0810|THEFT
|11938244|JC567030   |2019-12-31 23:15:00|070XX S CHAPPEL AVE   |0486|BATTERY
|11938338|JC567037   |2019-12-31 23:15:00|001XX W 68TH ST       |0486|BATTERY
|11937952|JC567038   |2019-12-31 23:14:00|007XX N MICHIGAN AVE  |0486|BATTERY
|11938039|JC567055   |2019-12-31 23:14:00|046XX S DAMEN AVE     |143A|WEAPONS VI
|11938007|JC567034   |2019-12-31 23:12:00|030XX W 21ST PL       |0486|BATTERY
|11937955|JC567056   |2019-12-31 23:12:00|080XX S KINGSTON AVE  |0486|BATTERY
|11938003|JC567031   |2019-12-31 23:10:00|100XX S LAFAYETTE AVE |0486|BATTERY
|11938034|JC567024   |2019-12-31 23:06:00|036XX S STATE ST      |143A|WEAPONS VI
|11937958|JC567042   |2019-12-31 23:02:00|014XX E 76TH ST       |0460|BATTERY
|11943133|JD105884   |2019-12-31 23:00:00|007XX N CENTRAL AVE   |1320|CRIMINAL D
+--------+-----------+-------------------+----------------------+----+----------
only showing top 20 rows
```

Union Dataframes

You will see three main methods for performing union of dataframes. It is important to know the difference between them and which one is preferred:

- `union()` – It is used to merge two DataFrames of the same structure/schema. If schemas are not the same, it returns an error
- `unionAll()` – This function is deprecated since Spark 2.0.0, and replaced with union()
- `unionByName()` - This fucntion is used to merge two dataframes based on column name.

> Since `unionAll()` is deprecated, **`union()` is the preferred method for merging dataframes.** The difference between `unionByName()` and `union()` is that `unionByName()` resolves columns by name, not by position.

In other SQLs, Union eliminates the duplicates but UnionAll merges two datasets, thereby including duplicate records. But, in PySpark, both behave the same and includes duplicate records. The recommendation is to use `distinct()` or `dropDuplicates()` to remove duplicate records.

In [0]:
```
# Append rows in PySpark.
one_day = spark.read.csv('reported-crimes.csv',header=True).withColumn('Date',to_
df.filter(col('Date')==lit('2019-07-30')).count()
```

Out[0]: 10

In [0]:
```
df.union(one_day).filter(col('Date')==lit('2019-07-30')).count()
```

Out[0]: 20

Result:

> As you can see here, there were 6 crimes commited on 2019-07-30, and after union, there's 12 records.

In [0]:
```
# Creating two dataframes with jumbled columns
df1 = spark.createDataFrame([[1, 2, 3]], ["col0", "col1", "col2"])
df2 = spark.createDataFrame([[4, 5, 6]], ["col1", "col2", "col0"])
df1.unionByName(df2).show()
```

```
+----+----+----+
|col0|col1|col2|
+----+----+----+
|   1|   2|   3|
|   6|   4|   5|
+----+----+----+
```

Result:

> As you can see here, the two dataframes have been successfully merged based on their column names.

In [0]:
```
# Top 10 number of reported crimes by Primary Type, in descending order of Occure
df.groupBy("Primary Type").count().orderBy('count', ascending=False).show(10)
```

```
+------------------+-----+
|       Primary Type|count|
+------------------+-----+
|              THEFT|62362|
|            BATTERY|49483|
|    CRIMINAL DAMAGE|26666|
|            ASSAULT|20605|
| DECEPTIVE PRACTICE|18233|
|      OTHER OFFENSE|16682|
|          NARCOTICS|14210|
|           BURGLARY| 9624|
|MOTOR VEHICLE THEFT| 8979|
|            ROBBERY| 7987|
+------------------+-----+
only showing top 10 rows
```

Hands-on Questions 👆 !

What percentage of reported crimes resulted in an arrest?

In [0]:
```
# Answer
```

What are the top 3 locations for reported crimes?

In [0]:
```
# Answer
```

Functions

In [0]:
```
# Functions available in PySpark
from pyspark.sql import functions
print(dir(functions))
```

```
['Column', 'DataFrame', 'DataType', 'PandasUDFType', 'PythonEvalType', 'SparkCon
```

String Functions

In [0]:
```
# Loading the data
from pyspark.sql.functions import col
df = spark.read.csv('reported-crimes.csv',header=True).withColumn('Date',to_timest
```

Display the Primary Type column in lower and upper characters, and the first 4 characters of the column

In [0]:
```
from pyspark.sql.functions import col,lower, upper, substring
help(substring)
df.select(lower(col('Primary Type')),upper(col('Primary Type')),substring(col('Pr
```

```
Help on function substring in module pyspark.sql.functions:

substring(str, pos, len)
    Substring starts at `pos` and is of length `len` when str is String type or
    returns the slice of byte array that starts at `pos` in byte and is of lengt
    when str is Binary type.

    .. note:: The position is not zero based, but 1 based index.

    >>> df = spark.createDataFrame([('abcd',)], ['s',])
    >>> df.select(substring(df.s, 1, 2).alias('s')).collect()
    [Row(s='ab')]

    .. versionadded:: 1.5
```

```
+------------------+------------------+----------------------------+
|lower(Primary Type)|upper(Primary Type)|substring(Primary Type, 1, 4)|
+------------------+------------------+----------------------------+
|deceptive practice |DECEPTIVE PRACTICE |DECE                        |
|deceptive practice |DECEPTIVE PRACTICE |DECE                        |
|deceptive practice |DECEPTIVE PRACTICE |DECE                        |
|battery            |BATTERY            |BATT                        |
|battery            |BATTERY            |BATT                        |
+------------------+------------------+----------------------------+
only showing top 5 rows
```

Numeric functions

Show the oldest date and the most recent date

In [0]:
```
from pyspark.sql.functions import min, max
df.select(min(col('Date')), max(col('Date'))).show()
```

```
+-------------------+-------------------+
|          min(Date)|          max(Date)|
+-------------------+-------------------+
|2019-01-01 00:00:00|2019-12-31 23:55:00|
+-------------------+-------------------+
```

Date

What is 3 days earlier that the oldest date and 3 days later than the most recent date?

In [0]:
```
from pyspark.sql.functions import date_add, date_sub
df.select(date_add(max(col('Date')),3), date_sub(min(col('Date')),3)).show()
```

```
+----------------------+----------------------+
|date_add(max(Date), 3)|date_sub(min(Date), 3)|
+----------------------+----------------------+
|            2020-01-03|            2018-12-29|
+----------------------+----------------------+
```

Working with Dates

> PySpark follows SimpleDateFormat table of Java. Click here to view the docs. (https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html)

In [0]:
```
from pyspark.sql.functions import to_date, to_timestamp, lit
df = spark.createDataFrame([('2019-12-25 13:30:00',)], ['Christmas'])
df.show()
```

```
+-------------------+
|          Christmas|
+-------------------+
|2019-12-25 13:30:00|
+-------------------+
```

In [0]:
```
df.select(to_date(col('Christmas'),'yyyy-MM-dd HH:mm:ss'), to_timestamp(col('Chr
```

```
+-----------------------------------------+----------------------------------
|to_date(`Christmas`, 'yyyy-MM-dd HH:mm:ss')|to_timestamp(`Christmas`, 'yyyy-MM-
+-----------------------------------------+----------------------------------
|                                2019-12-25|                             2019-1
+-----------------------------------------+----------------------------------
```

In [0]:
```
df = spark.createDataFrame([('25/Dec/2019 13:30:00',)], ['Christmas'])
df.select(to_date(col('Christmas'),'dd/MMM/yyyy HH:mm:ss'), to_timestamp(col('Chr
```

```
+-----------------------------------------+----------------------------------
|to_date(`Christmas`, 'dd/MMM/yyyy HH:mm:ss')|to_timestamp(`Christmas`, 'dd/MMM/
+-----------------------------------------+----------------------------------
|                                2019-12-25|                             2019
+-----------------------------------------+----------------------------------
```

In [0]:
```
df = spark.createDataFrame([('12/25/2019 01:30:00 PM',)], ['Christmas'])
df.show(1)
df.show(1, truncate = False)
df.select(to_date(col('Christmas'),'MM/dd/yyyy hh:mm:ss aa'), to_timestamp(col('
```

```
+--------------------+
|           Christmas|
+--------------------+
|12/25/2019 01:30:...|
+--------------------+

+----------------------+
|Christmas             |
+----------------------+
|12/25/2019 01:30:00 PM|
+----------------------+

+-------------------------------------------+----------------------------------
|to_date(`Christmas`, 'MM/dd/yyyy hh:mm:ss aa')|to_timestamp(`Christmas`, 'MM/dd
+-------------------------------------------+----------------------------------
|2019-12-25                                 |2019-12-25 13:30:00
+-------------------------------------------+----------------------------------
```

Working with Joins

In [0]:
```
# Loading the data
from pyspark.sql.functions import col
df = spark.read.csv('reported-crimes.csv',header=True).withColumn('Date',to_times
```

In [0]:
```
# Dowloading police station data
!wget -O police-station.csv https://data.cityofchicago.org/api/views/z8bn-74gv/r
```

```
--2020-05-02 15:16:24--  https://data.cityofchicago.org/api/views/z8bn-74gv/rows
Resolving data.cityofchicago.org (data.cityofchicago.org)... 52.206.140.205, 52.
Connecting to data.cityofchicago.org (data.cityofchicago.org)|52.206.140.205|:44
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/csv]
Saving to: ‘police-station.csv’

police-station.csv      [ <=>                ]   5.57K  --.-KB/s    in 0s

2020-05-02 15:16:28 (587 MB/s) - ‘police-station.csv’ saved [5699]
```

In [0]:
```
ps = spark.read.csv("police-station.csv", header=True)
ps.show()
```

```
+--------------------+----------------+--------------------+-------+-----+-----
|            DISTRICT|    DISTRICT NAME|             ADDRESS|   CITY|STATE|  ZIP
+--------------------+----------------+--------------------+-------+-----+-----
|                   1|         Central|     1718 S State St|Chicago|   IL|60616
|                   6|         Gresham|   7808 S Halsted St|Chicago|   IL|60620
|                  11|        Harrison|  3151 W Harrison St|Chicago|   IL|60612
|                  16|  Jefferson Park|5151 N Milwaukee Ave|Chicago|   IL|60630
|        Headquarters|    Headquarters| 3510 S Michigan Ave|Chicago|   IL|60653
|                  24|     Rogers Park|     6464 N Clark St|Chicago|   IL|60626
|                   2|       Wentworth|5101 S Wentworth Ave|Chicago|   IL|60609
|                   7|       Englewood|      1438 W 63rd St|Chicago|   IL|60636
|                  25|   Grand Central|    5555 W Grand Ave|Chicago|   IL|60639
|                  10|          Ogden|    3315 W Ogden Ave|Chicago|   IL|60623
|                  15|          Austin|   5701 W Madison St|Chicago|   IL|60644
|                   3|  Grand Crossing|7040 S Cottage Gr...|Chicago|   IL|60637
|                  14|     Shakespeare|2150 N California...|Chicago|   IL|60647
|                   8|    Chicago Lawn|      3420 W 63rd St|Chicago|   IL|60629
|                   4|   South Chicago|     2255 E 103rd St|Chicago|   IL|60617
|                  20|         Lincoln|  5400 N Lincoln Ave|Chicago|   IL|60625
|                  18|      Near North|  1160 N Larrabee St|Chicago|   IL|60610
|                  12|       Near West|1412 S Blue Islan...|   null| null| null
|",Chicago,IL,6060...| -87.6569725149)"|                null|   null| null| null
|                   9|         Deering|   3120 S Halsted St|Chicago|   IL|60608
+--------------------+----------------+--------------------+-------+-----+-----
only showing top 20 rows
```

The reported crimes dataset has only the district number. Add the district name by joining with the police station dataset.

In [0]:
```
# Caching the crimes dataset to speed things up, and then, since spark does lazy
df.cache()
df.count()
```

Out[0]: 259013

In [0]:
```
ps.select(col('DISTRICT')).distinct().show()
df.select(col('District')).distinct().show()
```

```
+------------+
|    DISTRICT|
+------------+
|           7|
|          15|
|          11|
|           3|
|           8|
|          22|
|          16|
|           5|
|          18|
|          17|
|           6|
|          19|
|          25|
|Headquarters|
|          24|
|           9|
|           1|
|          20|
|          10|
|           4|
+------------+
only showing top 20 rows

+--------+
|District|
+--------+
|     009|
|     012|
|     024|
|     031|
|     015|
|     006|
|     019|
|     020|
|     011|
|     025|
|     003|
|     005|
|     016|
|     018|
|     008|
|     022|
|     001|
|     014|
|     010|
|     004|
+--------+
only showing top 20 rows
```

In [0]:
```
# Transfomring crime data to remove 0 from beginning, inroder to match the data
from pyspark.sql.functions import lpad
ps = ps.withColumn('Format_district',lpad(col('DISTRICT'),3,'0'))
ps.select(col('Format_district')).show()
```

```
+---------------+
|Format_district|
+---------------+
|            001|
|            006|
|            011|
|            016|
|            Hea|
|            024|
|            002|
|            007|
|            025|
|            010|
|            015|
|            003|
|            014|
|            008|
|            004|
|            020|
|            018|
|            012|
|            ",C|
|            009|
+---------------+
only showing top 20 rows
```

In [0]:
```
# Executing the join and deleting some column so that we don't get too much data
df.join(ps, df.District == ps.Format_district, 'left_outer').drop(
 'ADDRESS',
 'CITY',
 'STATE',
 'ZIP',
 'WEBSITE',
 'PHONE',
 'FAX',
 'TTY',
 'X COORDINATE',
 'Y COORDINATE',
 'LATITUDE',
 'LONGITUDE',
 'LOCATION').show(truncate=False)
```

```
+--------+-----------+-------------------+-------------------------+----+-------
|ID      |Case Number|Date               |Block                    |IUCR|Primary
+--------+-----------+-------------------+-------------------------+----+-------
|12040452|JD220630   |2019-11-21 12:00:00|004XX N WABASH AVE       |1153|DECEPTI
|12039199|JD219081   |2019-04-01 08:00:00|021XX S INDIANA AVE      |1153|DECEPTI
|12040466|JD220650   |2019-11-07 12:00:00|026XX N NEW ENGLAND AVE  |1152|DECEPTI
|11938499|JD100565   |2019-12-31 11:30:00|070XX S COTTAGE GROVE AVE|0460|BATTERY
|11864640|JC477069   |2019-10-18 09:56:00|002XX S LAVERGNE AVE     |041A|BATTERY
|11766889|JC359918   |2019-07-17 15:07:00|024XX W FOSTER AVE       |0610|BURGLAR
|11740807|JC328262   |2019-06-30 05:04:00|048XX W AUGUSTA BLVD     |041A|BATTERY
|11718084|JC300874   |2019-05-13 13:00:00|062XX S EMERALD DR       |1750|OFFENSE
|11647696|JC215453   |2019-04-07 17:56:00|062XX S FRANCISCO AVE    |041A|BATTERY
|12039931|JD220113   |2019-10-18 08:00:00|120XX S LAFLIN ST        |2826|OTHER O
|12039974|JD220130   |2019-12-24 23:00:00|049XX N ALBANY AVE       |0890|THEFT
|12039885|JD219909   |2019-10-01 02:00:00|021XX S HARDING AVE      |1752|OFFENSE
|12039884|JD220045   |2019-07-01 12:00:00|026XX W BALMORAL AVE     |1153|DECEPTI
|12038367|JD218291   |2019-04-21 12:00:00|071XX S CYRIL AVE        |0890|THEFT
|12038544|JD218327   |2019-12-22 13:00:00|028XX S CHRISTIANA AVE   |5002|OTHER O
|12038308|JD218196   |2019-07-22 08:00:00|052XX S ARCHER AVE       |0910|MOTOR V
|12040067|JD220150   |2019-12-15 18:00:00|057XX W ROOSEVELT RD     |0265|CRIMINA
|12039858|JD219964   |2019-09-03 09:00:00|003XX S SPRINGFIELD AVE  |1154|DECEPTI
|12038342|JD218257   |2019-05-01 00:00:00|002XX N MENARD AVE       |1154|DECEPTI
|12038644|JD218194   |2019-06-01 13:45:00|003XX N MENARD AVE       |1153|DECEPTI
+--------+-----------+-------------------+-------------------------+----+-------
only showing top 20 rows
```

As you can see, we have done a left outer join between two dataframes. The following joins are supported by PySpark:

1. inner (default)
2. cross
3. outer
4. full
5. full_outer
6. left
7. left_outer
8. right
9. right_outer
10. left_semi
11. left_anti

Hands-on again!

What is the most frequently reported non-criminal activity?

In [0]:

Find the day of the week with the most reported crime?

In [0]:

Using a bar chart, plot which day of the week has the most number of reported crime.

In [0]:
```
from pyspark.sql.functions import date_format, col
dow = [x[0] for x in df.groupBy(date_format(col('Date'),'E')).count().collect()]
print(dow)
cnt = [x[1] for x in df.groupBy(date_format(col('Date'),'E')).count().collect()]
print(cnt)
```

```
['Sun', 'Mon', 'Thu', 'Sat', 'Wed', 'Tue', 'Fri']
[36026, 36930, 36235, 37978, 36095, 36855, 38894]
```

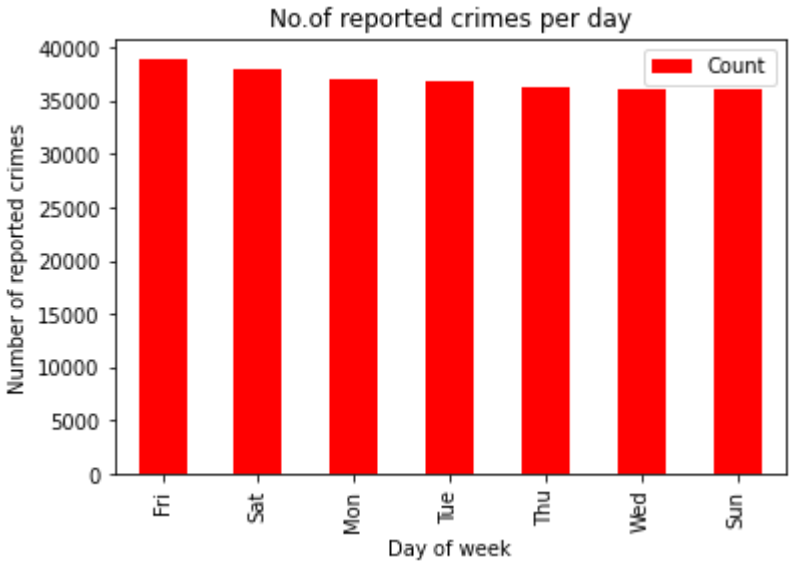
In [0]:
```
import pandas as pd
import matplotlib.pyplot as plt

cp = pd.DataFrame({'Day_of_week':dow, 'Count':cnt})
cp.head()
```

Out[0]:

	Day_of_week	Count
0	Sun	36026
1	Mon	36930
2	Thu	36235
3	Sat	37978
4	Wed	36095

In [0]:
```
cp.sort_values('Count', ascending=False).plot(kind='bar', color= 'red', x='Day_of
plt.xlabel("Day of week")
plt.ylabel("Number of reported crimes")
plt.title("No.of reported crimes per day")
```

Out[0]: Text(0.5, 1.0, 'No.of reported crimes per day')



Spark SQL

SQL has been around since the 1970s, and so one can imagine the number of people who made it their bread and butter. As big data came into popularity, the number of professionals with the technical knowledge to deal with it was in shortage. This led to the creation of Spark SQL. To quote the docs:

> Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations.

Basically, what you need to know is that Spark SQL is used to execute SQL queries on big data. Spark SQL can also be used to read data from Hive tables and views. Let me explain Spark SQL with an example.

In [0]:
```
# Load data
df = spark.read.csv('reported-crimes.csv',header=True).withColumn('Date',to_timest
# Register Temporary Table
df.createOrReplaceTempView("temp")
# Select all data from temp table
spark.sql("select * from temp").show()
# Select count of data in table
spark.sql("select count(*) from temp").show()
```

```
+--------+-----------+-------------------+--------------------+----+------------
|      ID|Case Number|               Date|               Block|IUCR|       Prima
+--------+-----------+-------------------+--------------------+----+------------
|11895315|   JC515309|2019-07-30 00:00:00|097XX S WENTWORTH...|1156| DECEPTIVE P
|11805495|   JC406040|2019-07-30 00:00:00|0000X S MICHIGAN AVE|1156| DECEPTIVE P
|11805303|   JC405788|2019-07-30 00:00:00|067XX S BLACKSTON...|0890|
|11798686|   JC397805|2019-07-30 00:00:00|  010XX W MADISON ST|1153| DECEPTIVE P
|11787134|   JC383936|2019-07-30 00:00:00|110XX S MICHIGAN AVE|1130| DECEPTIVE P
|11781214|   JC376972|2019-07-30 00:00:00|  015XX N LEAVITT ST|0820|
|11775928|   JC370624|2019-07-30 00:00:00|079XX S EBERHART AVE|0820|
|11776857|   JC371434|2019-07-30 00:00:00|070XX S EBERHART AVE|5002|      OTHER
|11776516|   JC371429|2019-07-30 00:00:00|    057XX N CLARK ST|0281|CRIM SEXUAL
|11776159|   JC370564|2019-07-30 00:00:00| 066XX N ROCKWELL ST|0620|           B
+--------+-----------+-------------------+--------------------+----+------------

+--------+
|count(1)|
+--------+
|      10|
+--------+
```

As you can see, we registered the dataframe as temporary table and then ran basic SQL queries on it. How amazing is that?!
If you are a person who is more comfortable with SQL, then this feature is truly a blessing for you! But this raises a question:

> *Should I just keep using Spark SQL all the time?*

And the answer is, ***it depends***.
So basically, the different functions acts in differnet ways, and depending upon the type of action you are trying to do, the speed at which it completes execution also differs. But as time progress, this feature is getting better and better, so hopefully the difference should be a small margin. There are plenty of analysis done on this, but nothing has a definite answer yet. You can read this comparative study done by horton works (https://community.cloudera.com/t5/Community-Articles/Spark-RDDs-vs-DataFrames-vs-SparkSQL/ta-p/246547) or the answer to this stackoverflow question (https://stackoverflow.com/questions/45430816/writing-sql-vs-using-dataframe-apis-in-spark-sql) if you are still curious about it.

RDD

> With map, you define a function and then apply it record by record. Flatmap returns a new RDD by first applying a function to all of the elements in RDDs and then flattening the result. Filter, returns a new RDD. Meaning only the elements that satisfy a condition. With reduce, we are taking neighboring elements and producing a single combined result. For example, let's say you have a set of numbers. You can reduce this to its sum by providing a function that takes as input two values and reduces them to one.

Some of the reasons you would use a dataframe over RDD are:

1. It's ability to represnt data as rows and columns. But this also means it can only hold structred and semi-structured data.
2. It allows processing data in different formats (AVRO, CSV, JSON, and storage system HDFS, HIVE tables, MySQL).
3. It's superior job Optimization capability.
4. DataFrame API is very easy to use.

In [0]:
```
psrdd = sc.textFile('police-station.csv')
print(psrdd.first())
ps_header = psrdd.first()
ps_rest = psrdd.filter(lambda line: line!=ps_header)
print(ps_rest.first())
```

```
DISTRICT,DISTRICT NAME,ADDRESS,CITY,STATE,ZIP,WEBSITE,PHONE,FAX,TTY,X COORDINATE
1,Central,1718 S State St,Chicago,IL,60616,http://home.chicagopolice.org/communi
```

How many police stations are there?

In [0]:
```
ps_rest.map(lambda line: line.split(",")).count()
```

Out[0]:
```
24
```

Display the District ID, District name, Address and Zip for the police station with District ID 7

In [0]:
```
# District is column  0
(ps_rest.filter(lambda line: line.split(",")[0]=='7').
 map(lambda line: (line.split(",")[0],
    line.split(",")[1],
    line.split(",")[2],
    line.split(",")[5])).collect())
```

Out[0]:
```
[('7', 'Englewood', '1438 W 63rd St', '60636')]
```

Police stations 10 and 11 are geographically close to each other. Display the District ID, District name, address and zip code

In [0]:
```
# District is column  0
(ps_rest.filter(lambda line: line.split(",")[0] in ['10', '11']).
 map(lambda line: (line.split(",")[0],
    line.split(",")[1],
    line.split(",")[2],
    line.split(",")[5])).collect())
```

Out[0]:
```
[('11', 'Harrison', '3151 W Harrison St', '60612'),
 ('10', 'Ogden', '3315 W Ogden Ave', '60623')]
```

User-Defined Functions (UDF)

PySpark User-Defined Functions (UDFs) help you convert your python code into a scalable version of itself. It comes in handy more than you can imagine, but beware, as the performance is less when you compare it with pyspark functions. You can view examples of how UDF works here (https://docs.databricks.com/spark/latest/spark-sql/udf-python.html). What I will give in this section is some theory on how it works, and why it is slower.

When you try to run a UDF in PySpark, each executor creates a python process. Data will be serialised and deserialised between each executor and python. This leads to lots of performance impact and overhead on spark jobs, making it less efficent than using spark dataframes. Apart from this, sometimes you might have memory issues while using UDFs. The Python worker consumes huge off-heap memory and so it often leads to memoryOverhead, thereby failing your job. Keeping these in mind, I wouldn't recommend using them, but at the end of the day, your choice.

Common Questions

Recommended IDE

I personally prefer PyCharm (https://www.jetbrains.com/pycharm/) while coding in Python/PySpark. It's based on IntelliJ IDEA so it has a lot of features! And the main advantage I have felt is the ease of installing PySpark and other packages. You can customize it with themes and plugins, and it lets you enhance productivity while coding by providing some features like suggestions, local VCS etc.

Submitting a Spark Job

The python syntax for running jobs is: `python <file_name>.py <arg1> <arg2> ...`
But when you submit a spark job you have to use spark-submit to run the application.

Here is a simple example of a spark-submit command: `spark-submit filename.py --named_argument 'arguemnt value'`
Here, named_argument is an argument that you are reading from inside your script.

There are other options you can pass in the command, like:
`--py-files` which helps you pass a python file to read in your file,
`--files` which helps pass other files like txt or config,
`--deploy-mode` which tells wether to deploy your worker node on cluster or locally
`--conf` which helps pass different configurations, like memoryOverhead, dynamicAllocation etc.

There is an entire page (https://spark.apache.org/docs/latest/submitting-applications.html) in spark documentation dedicated to this. I highly recommend you go through it once.

Creating Dataframes

When getting started with dataframes, the most common question is: *'How do I create a dataframe?'*
Below, you can see how to create three kinds of dataframes:

Create a totally empty dataframe

In [0]:
```
from pyspark.sql.types import StructType
#Create empty df
schema = StructType([])
empty = spark.createDataFrame(sc.emptyRDD(), schema)
empty.show()
```

```
++
||
++
++
```

Create an empty dataframe with header

In [0]:
```
 from pyspark.sql.types import StructType, StructField
#Create empty df with header
schema_header = StructType([StructField("name", StringType(), True)])
empty_with_header = spark.createDataFrame(sc.emptyRDD(), schema_header)
empty_with_header.show()
```

```
+----+
|name|
+----+
+----+
```

Create a dataframe with header and data

In [0]:
```
from pyspark.sql import Row
mylist = [
  {"name":'Alice',"age":13},
  {"name":'Jacob',"age":24},
  {"name":'Betty',"age":135},
]
spark.createDataFrame(Row(**x) for x in mylist).show()
```

```
+---+-----+
|age| name|
+---+-----+
| 13|Alice|
| 24|Jacob|
|135|Betty|
+---+-----+
```

In [0]:
```
# You can achieve the same using this - note that we are using spark context here
from pyspark.sql import Row
df = sc.parallelize([
        Row(name='Alice', age=13),
        Row(name='Jacob', age=24),
        Row(name='Betty', age=135)]).toDF()
df.show()
```

```
+---+-----+
|age| name|
+---+-----+
| 13|Alice|
| 24|Jacob|
|135|Betty|
+---+-----+
```

Drop Duplicates

As mentioned earlier, there are two wasy to remove duplicates from a dataframe. We have already seen the usage of distinct under Get Distinct Rows section. I will expalin how to use the `dropDuplicates()` function to achieve the same.

> `drop_duplicates()` is an alias for `dropDuplicates()`

In [0]:
```
from pyspark.sql import Row
from pyspark.sql import Row
mylist = [
  {"name":'Alice',"age":5,"height":80},
  {"name":'Jacob',"age":24,"height":80},
  {"name":'Alice',"age":5,"height":80}
]
df = spark.createDataFrame(Row(**x) for x in mylist)
df.dropDuplicates().show()
```

```
+---+------+-----+
|age|height| name|
+---+------+-----+
|  5|    80|Alice|
| 24|    80|Jacob|
+---+------+-----+
```

`dropDuplicates()` can also take in an optional parameter called *subset* which helps specify the columns on which the duplicate check needs to be done on.

In [0]:
```
df.dropDuplicates(subset=['height']).show()
```

```
+---+------+-----+
|age|height| name|
+---+------+-----+
|  5|    80|Alice|
+---+------+-----+
```

Fine Tuning a Spark Job

Before we begin, please note that this entire section is written purely based on experience. It might differ with use cases, but it will help you get a better understanding of what you should be looking for, or act as a guidance to achieve your aim.

> Spark Performance Tuning refers to the process of adjusting settings to record for memory, cores, and instances used by the system. This process guarantees that the Spark has a flawless performance and also prevents bottlenecking of resources in Spark.

Considering you are using Amazon EMR to execute your spark jobs, there are three aspects you need to take care of:

1. EMR Sizing
2. Spark Configurations
3. Job Tuning

EMR Sizing

Sizing your EMR is extremely important, as this affects the efficency of your spark jobs. Apart from the cost factor, the maximum number of nodes and memory your job can use will be decided by this. If you spin up a EMR with high specifications, that obviously means you are paying more for it, so we should ideally utilize it to the max. These are the guidelines that I follow to make sure the EMR is rightly sized:

1. Size of the input data (include all the input data) on the disk.
2. Whether the jobs have transformations or just a straight pass through.
   Assess the joins and the complex joins involved.
3. Size of the output data on the disk.

Look at the above criteria against the memory you need to process, and the disk space you would need. Start with a small configuration, and keep adding nodes to arrive at an optimal configuration. In case you are wondering about the *Execution time vs EMR configuration* factor, please understand that it is okay for a job to run longer, rather than adding more resources to the cluster. For example, it is okay to run a job for 40 mins job on a 5 node cluster, rather than running a job in 10 mins on a 15 node cluster.

Another thing you need to know about EMRs, are the different kinds of EC2 instance types provided by Amazon. I will briefly talk about them, but I strongly recommend you to read more about it from the official documentation (https://aws.amazon.com/ec2/instance-types/). There are 5 types of instance classes. Based on the job you want to run, you can decide which one to use:

Instance Class	Description
General purpose	Balance of compute, memory and networking resources
Compute optimized	Ideal for compute bound applications that benefit from high performance processors
Memory optimized	Designed to deliver fast performance for workloads that process large data sets in memory
Storage optimized	For workloads that require high, sequential read and write access to very large data sets on local storage
GPU instances	Use hardware accelerators, or co-processors, to perform high demanding functions, more efficiently than is possible in software running on CPUs

The configuration (memory, storage, cpu, network performance) will differ based on the instance class you choose. To help make life easier, here is what I do when I get into a predicament about which one to go with:

1. Visit ec2instances (https://www.ec2instances.info/)
2. Choose the EC2 instances in question
3. Click on compare selected

This will easily help you undesrstand what you are getting into, and thereby help you make the best choice! The site was built by Garret Heaton (https://github.com/powdahound)(founder of Swoot), and has helped me countless number of times to make an informed decision.

Spark Configurations

There are a ton of configurations (https://spark.apache.org/docs/latest/configuration.html) that you can tweak when it comes to Spark. Here, I will be noting down some of the configurations which I use, which have worked well for me. Alright! let's get into it!

Job Scheduling

When you submit your job in a cluster, it will be given to Spark Schedulers, which is responsible for materializing a logical plan for your job. There are two types of job scheduling (https://spark.apache.org/docs/latest/job-scheduling.html):

1. FIFO
   By default, Spark's scheduler runs jobs in FIFO fashion. Each job is divided into stages (e.g. map and reduce phases), and the first job gets priority on all available resources while its stages have tasks to launch, then the second job gets priority, etc. If the jobs at the head of the queue don't need to use the whole cluster, later jobs can start to run right away, but if the jobs at the head of the queue are large, then later jobs may be delayed significantly.
2. FAIR
   The fair scheduler supports grouping jobs into pools and setting different scheduling options (e.g. weight) for each pool. This can be useful to create a high-priority pool for more important jobs, for example, or to group the jobs of each user together and give users equal shares regardless of how many concurrent jobs they have instead of giving jobs equal shares. This approach is modeled after the Hadoop Fair Scheduler.

> I personally prefer using the FAIR mode, and this can be set by adding `.config("spark.scheduler.mode", "FAIR")` when you create your SparkSession.

Serializer

We have two types of serializers (https://spark.apache.org/docs/latest/tuning.html#data-serialization) available:

1. Java serialization
2. Kryo serialization

Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all Serializable types and requires you to register the classes you'll use in the program in advance for best performance.

Java serialization is used by default because if you have custom class that extends Serializable it can be easily used. You can also control the performance of your serialization more closely by extending java.io.Externalizable

> The general recommendation is to use Kyro as the serializer whenver possible, as it leads to much smaller sizes than Java serialization. It can be added by using `.config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")` when you create your SparkSession.

Shuffle Behaviour

It is generally a good idea to compress the output file after the map phase. The `spark.shuffle.compress` property decides whether to do the compression or not. The compression used is `spark.io.compression.codec` .

> The property can be added by using `.config("spark.shuffle.compress", "true")` when you create your SparkSession.

Compression and Serialization

There are 4 defaiult codecs spark provides to compress internal data such as RDD partitions, event log, broadcast variables and shuffle outputs. They are:

1. lz4
2. lzf
3. snappy
4. zstd

> The decision on which to use rests upon the use case. I generally use the `snappy` compression. Google created Snappy because they needed something that offered very fast compression at the expense of final size. Snappy is fast, stable and free, but it increases the size more than the other codecs. At the same time, since compute costs will be less, it seems like balanced trade off. The property can be added by using `.config("spark.io.compression.codec", "snappy")` when you create your SparkSession.

This session (https://databricks.com/session/best-practice-of-compression-decompression-codes-in-apache-spark) explains the best practice of compression/decompression codes in Apache Spark. I recommend you to take a look at it before taking a decision.

Scheduling

The property `spark.speculation` performs speculative execution of tasks. This means if one or more tasks are running slowly in a stage, they will be re-launched. Speculative execution will not stop the slow running task but it launches the new task in parallel.

> I usually disable this option by adding `.config("spark.speculation", "false")` when I create the SparkSession.

Application Properties

There are mainly two application properties that you should know about:

1. spark.driver.memoryOverhead - The amount of off-heap memory to be allocated per driver in cluster mode, in MiB unless otherwise specified. This is memory that accounts for things like VM overheads, interned strings, other native overheads, etc. This tends to grow with the container size (typically 6-10%). This option is currently supported on YARN and Kubernetes.
2. spark.executor.memoryOverhead - The amount of off-heap memory to be allocated per executor, in MiB unless otherwise specified. This is memory that accounts for things like VM overheads, interned strings, other native overheads, etc. This tends to grow with the executor size (typically 6-10%). This option is currently supported on YARN and Kubernetes.

> If you ever face an issue like `Container killed by YARN for exceeding memory limits` , know that it is because you have not specified enough memory Overhead for your job to successfully execute. The default value for Overhead is 10% of avaialbe memory (driver/executor sepearte), with minimum of 384.

Dynamic Allocation

Lastly, I want to talk about Dynamic Allocation. This is a feature I constantly use while executing my jobs. This property is by defualt set to False. As the name suggests, it sets whether to use dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload. Truly a wonderful feature, and the greatest benefit of using it is that it will help make the best use of all the resources you have! The disadvantage of this feature is that it does not shine well when you have to execute tasks in parallel. Since most of the resources will be used by the first task, the second one will have to wait till some resource gets released. At the same time, if both get submitted at the exact same time, the resources will be shared between them, although not equally. Also, it is not guaranteed to *always* use the most optimal configurations. But in all my tests, the results have been great!

> If you are planning on using this feature, you can pass the configurations as required through the spark-submit command. The four configurations which you will have to keep in mind are:
>
> ```
> --conf spark.dynamicAllocation.enabled=true
> --conf spark.dynamicAllocation.initialExecutors
> --conf spark.dynamicAllocation.minExecutors
> --conf spark.dynamicAllocation.maxExecutors
> ```

You can read more about this feature here (https://spark.apache.org/docs/latest/configuration.html#dynamic-allocation) and here (https://stackoverflow.com/questions/40200389/how-to-execute-spark-programs-with-dynamic-resource-allocation).

Job Tuning

Apart from EMR and Spark tuning, there is another way to approach opttimizations, and that is by tuning your job itself to produce results efficently. I will be going over some such techniques which will help you achieve this. The Spark Programming Guide (https://spark.apache.org/docs/2.1.1/programming-guide.html) talks more about these concepts in detail. If you guys prefer watching a video over reading, I highly recommend A Deep Dive into Proper Optimization for Spark Jobs (https://youtu.be/daXEp4HmS-E) by Daniel Tomes from Databricks, which I found really useful and informative!

Broadcast Joins (Broadcast Hash Join)

For some jobs, the efficenecy can be increased by caching them in memory. Broadcast Hash Join(BHJ) is such a technique which will help you optimize join queries when the size of one side of the data is low.

> BroadCast joins are the fastest but the drawaback is that it will consume more memory on both the executor and driver.

This following steps give a sneak peek into how it works, which will help you understand the use cases where it can be used:

1. Input file(smaller of the two tables) to be broadcasted is read by the executors in parallel into its working memory.
2. All the data from the executors is collected into driver (Hence, the need for higher memory at driver).
3. The driver then broadcasts the combined dataset (full copy) into each executor.
4. The size of the broadcasted dataset could be several (10-20+) times bigger the input in memory due to factors like deserialization.
5. Executors will end up storing the parts it read first, and also the full copy, thereby leading to a high memory requirement.

Some things to keep in mind about BHJ:

1. It is advisable to use broadcast joins on small datasets only (dimesnion table, for example).
2. Spark does not guarantee BHJ is always chosen, since not all cases (e.g. full outer join) support BHJ.
3. You could notice skews in tasks due to uneven partition sizes; especially during aggregations, joins etc. This can be evened out by introducing Salt value (random value).
   Suggested formula for salt value: random(0 – (shuffle partition count – 1))

Spark Partitions

A partition in spark is an atomic chunk of data (logical division of data) stored on a node in the cluster. Partitions are the basic units of parallelism in Spark. Having too large a number of partitions or too few is not an ideal solution. The number of partitions in spark should be decided based on the cluster configuration and requirements of the application. Increasing the number of partitions will make each partition have less data or no data at all. Generally, spark partitioning can be broken down in three ways:

1. Input
2. Shuffle
3. Output

Input

Spark usually does a good job of figuring the ideal configuration for this one, except in very particular cases. It is advisable to use the spark default unless:

1. Increase parallelism
2. Heavily nested data
3. Generating data (explode)
4. Source is not optimal
5. You are using UDFs

`spark.sql.files.maxpartitionBytes` : This property indicates the maximum number of bytes to pack into a single partition when reading files (Default 128 MB) . Use this to increase the parallelism in reading input data. For example, if you have more cores, then you can increase the number of parallel tasks which will ensure usage of the all the cores of the cluster, and increase the speed of the task.

Shuffle

One of the major reason why most jobs lags in performance is, for the majority of the time, because they get the shuffle partitions count wrong. By default, the value is set to 200. In almost all situations, this is not ideal. If you are dealing with shuffle satge of less than 20 GB, 200 is fine, but otherwise this needs to be changed. For most cases, you can use the following equation to find the right value:

> `Partition Count = Stage Input Data / Target Size` where
> `Largest Shuffle Stage (Target Size) < 200MB/partition` in most cases.
> `spark.sql.shuffle.partitions` property is used to set the ideal partition count value.

If you ever notice that target size at the range of TBs, there is something terribly wrong, and you might want to change it back to 200, or recalculate it. Shuffle partitions can be configured for every action (not transformation) in the spark script.

Let us use an example to explain this scenario:
Assume shuffle stage input = 210 GB.
Partition Count = Stage Input Data / Target Size = 210000 MB/200 MB = 1050.
As you can see, my shuffle partitions should be 1050, not 200.

But, if your cluster has 2000 cores, then set your shuffle partitions to 2000.

> In a large cluster dealing with a large data job, never set your shuffle partitions less than your total core count.

Shuffle stages almost always precede the write stages and having high shuffle partition count creates small files in the output. To address this, use localCheckPoint just before write & do a coalesce call. This localCheckPoint writes the Shuffle Partition to executor local disk and then coalesces into lower partition count and hence improves the overall performance of both shuffle stage and write stage.

Output

There are different methods to write the data. You can control the size, composition, number of files in the output and even the number of records in each file while writing the data. While writing the data, you can increase the parallelism, thereby ensuring you use all the resources that you have. But this approach would lead to a larger number of smaller files. Usually, this isn't a problem, but if you want bigger files, you will have to use one of the compaction techniques, preferably in a cluster with lesser configuration. There are multiple ways to change the composition of the output. Keep these two in mind about composition:

1. Coalesce: Use this to reduce the number of partitions.
2. Repartition: Use this very rarely, and never to reduce the number of partitions
   a. Range Paritioner - It partitions the data either based on some sorted order OR set of sorted ranges of keys.
   b. Hash Partioner - It spreads around the data in the partitioning based upon the key value. Hash partitioning can make distributed data skewed.

Best Practices

Try to incorporate these to your coding habits for better performance:

1. Do not use NOT IN use NOT EXISTS.
2. Remove Counts, Distinct Counts (use approxCountDIstinct).
3. Drop Duplicates early.
4. Always prefer SQL functions over PandasUDF.
5. Use Hive partitions effectively.
6. Leverage Spark UI effectively. Avoid Shuffle Spills.
7. Drop Duplicates early
8. Aim for target cluster utilization of atleast 70%