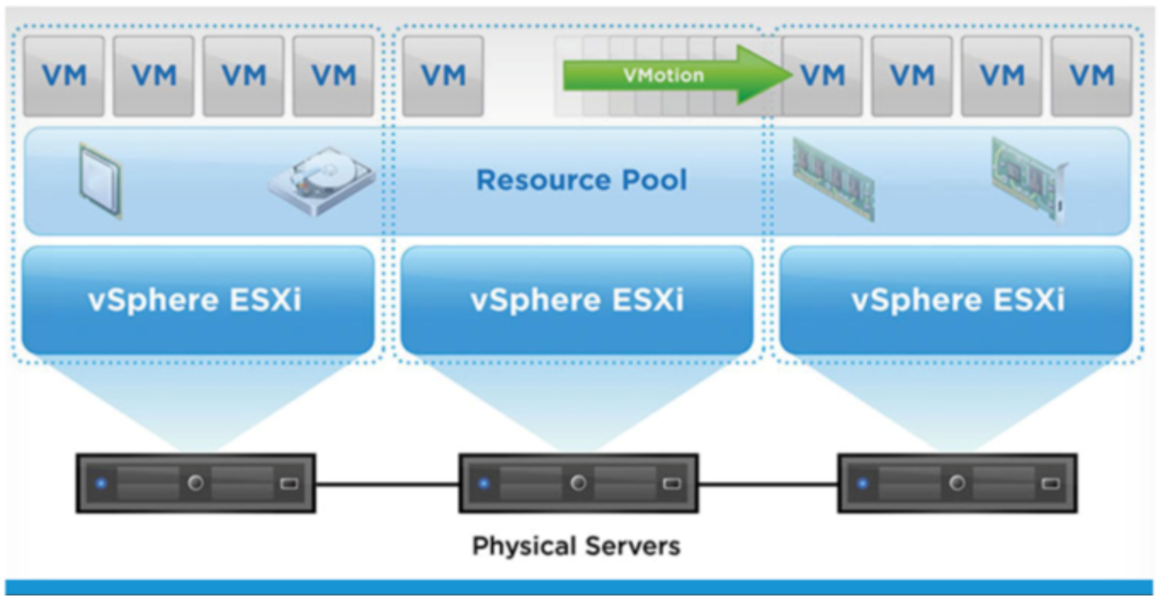
# COMS4121 Assignment 2: Supplement

This is a supplement reading to provide you more context for the assignment. Hopefully, it will be helpful. This doc is by no means an official document. Some of the descriptions are not precise but it might provide you some sense on this project so it will make it easier to get started.

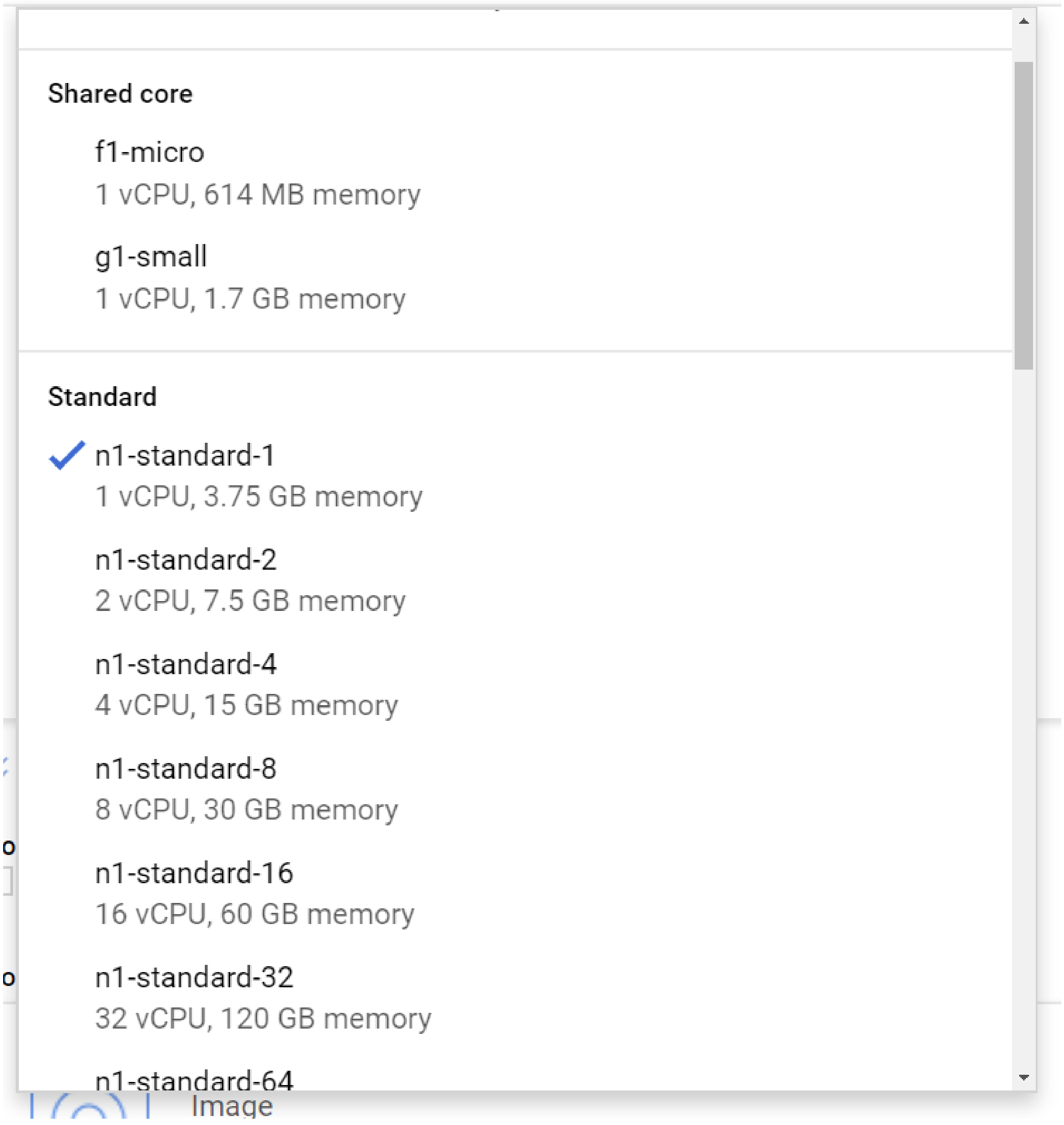
We should first start with the way Google described Dataproc: "*Dataproc makes open source data and analytics processing fast, easy, and more secure in the cloud.*" Which apparently makes it more attractive to customers.

## VMs / Compute Engine

One of the confusion points for people who don't have experience with computer systems or cloud computing is the idea of VMs. As we learned earlier in the course, the basic unit of a cloud warehouse is the server. Each server is similar to your laptop. They would have CPU(s), memory and disk. VM are virtual computers that run on top of servers. VM software such as ESXi virtually separates those resources and make them look like separate servers. As a result, you could treat each VM as a standalone server.



Google Cloud Computer Engine, in essence, provides you a VM whenever you create an instance on it.


Shared core

f1-micro
1 vCPU, 614 MB memory

g1-small
1 vCPU, 1.7 GB memory

Standard

✓ n1-standard-1
1 vCPU, 3.75 GB memory

n1-standard-2
2 vCPU, 7.5 GB memory

n1-standard-4
4 vCPU, 15 GB memory

n1-standard-8
8 vCPU, 30 GB memory

n1-standard-16
16 vCPU, 60 GB memory

n1-standard-32
32 vCPU, 120 GB memory

n1-standard-64

Image

When you start an instance you could configure your machine and choose how much resources you would want on your VM.

## Google Cloud Storage / S3

Google Cloud Storage / S3 provides data storage on the cloud.

One good comparison is to compare it with Google Drive. The basically functionality are the same. You upload your document to Google Drive and then you document is on Google Server and then you would share the document with anybody who has access to internet.

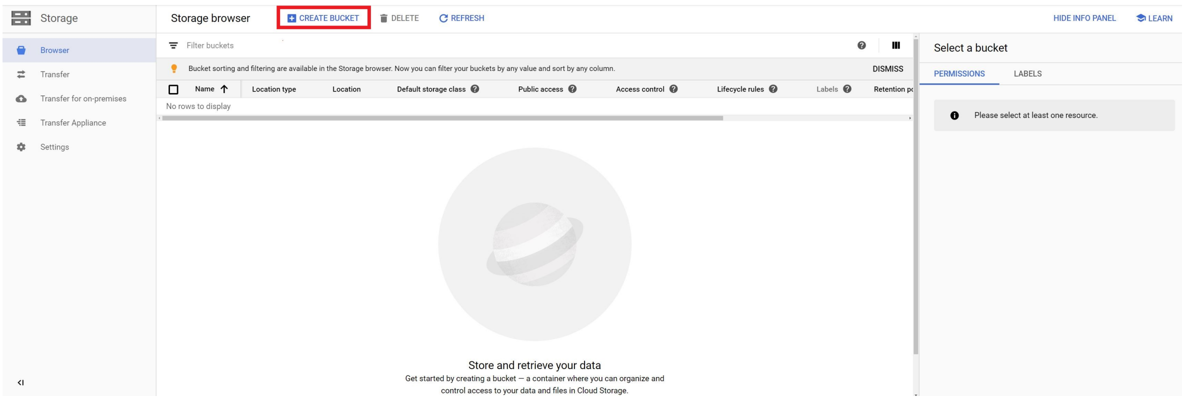
For Google Drive, the main user will be human, so the files are usually documents, slide decks, forms or pictures. Also, it provides better user interfaces.

In contrast, the users of Cloud Storage or S3 are usually programs. As a result, the file could be any type (large or small different formats ). They may not even make sense to human being and the idea of file is simply a chunk of data.

This 7-minute video goes into this topic deeper.

A short tutorial on Google Cloud Storage:

The idea of a folder on Google Drive translates to bucket on Google Cloud Storage.



Bucket sorting and filtering are available in the Storage browser. Now you can filter your buckets by any value and sort by any column.

No rows to display

Please select at least one resource.

Store and retrieve your data

Get started by creating a bucket — a container where you can organize and control access to your data and files in Cloud Storage.

To Create a bucket, you would go to the Storage page on Google Cloud and click on CREATE BUCKET. In most cases you could just name the bucket and use default settings for other properties.

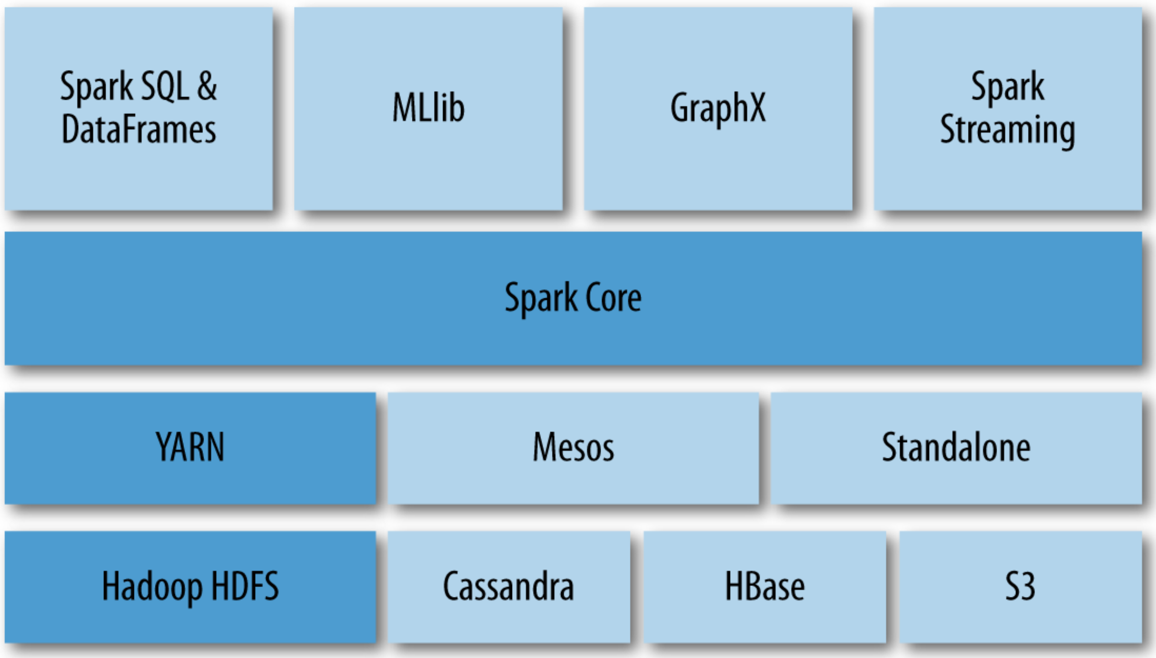
To upload files (e.g. your python file) from your computer you could simply drag and drop the files into the UI and it will be uploaded. To get the URI of the file, click on the name of the file and it would be on the next screen.

## Dataproc

Google listed the following as one of the key features of Dataproc: "*Managed deployment, logging, and monitoring let you focus on your data, not on your cluster. Dataproc clusters are stable, scalable, and speedy.*"  Which definitely makes it more attractive to customers. If you ever need to configure one of the Spark sever you will find it tedious and time consuming. And Google has done that for us.

Once the Dataproc cluster is ready, it would have:

- N Compute Engine VMs (3 if you started a 3 node cluster). One of the VM will be the Name Node and the others will be worker nodes.
- The following software would have been installed and configured for you automatically, all of them are installed on the VMs:
  - HDFS: The Hadoop Distributed File System (HDFS) is a highly fault tolerant file system designed and optimized to be deployed on a distributed infrastructure.
  - YARN:  (yet another resource negotiator) is Hadoop's popular resource scheduling cluster manager. So
  - SPARK: Apache spark is a unified analytics engine for large-scale data processing.



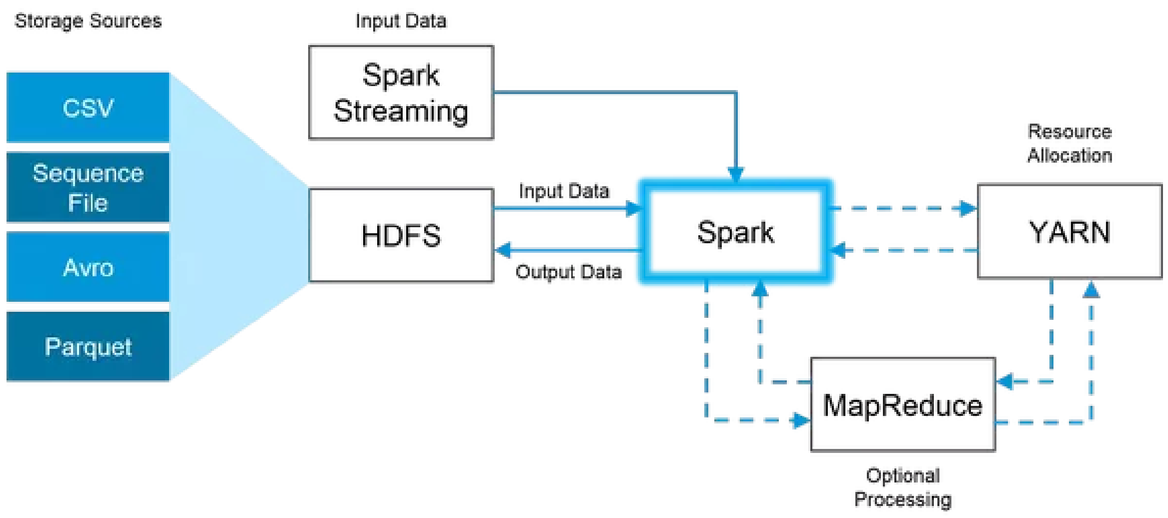
Also, if there are extra components such as Jupyter notebook added while configuring the VMs they will also be installed and configured. Notice that the notebooks that you created are written to the staging bucket on the Cloud Storage so even if you delete your cluster, you will be able to get your notebook files back. However, HDFS files are saved on disks and they will be lost if you delete the cluster.

## HDFS

Like described in the lecture, HDFS is a distributed file system that stores data across different machines.

Here, the more confusing part is probably the relationship between Spark and HDFS.

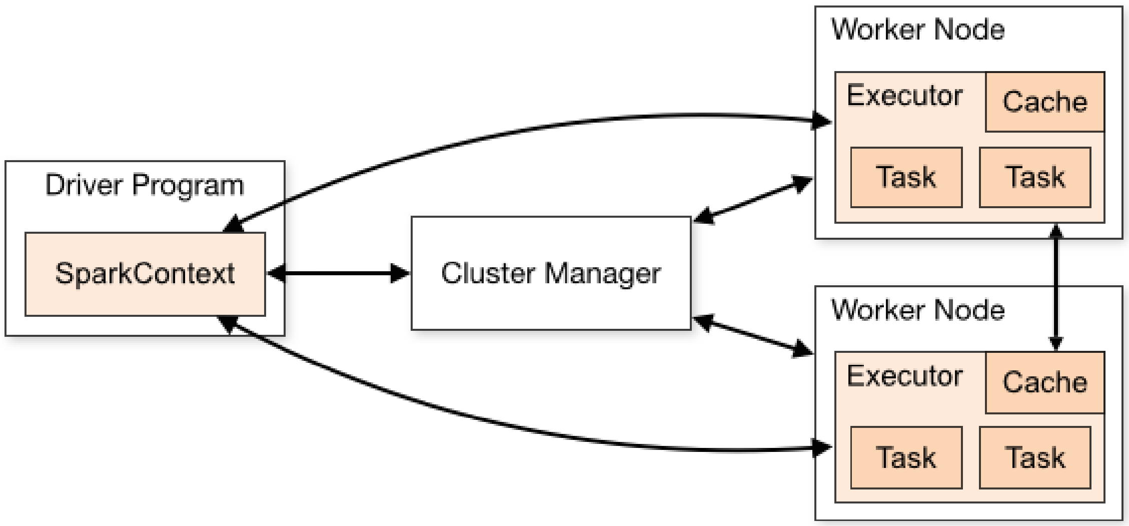
In short, Spark is an analytics engine and it does not handle how the data is stored. HDFS provides the functionality of saving data across servers to Spark and Spark read and write data through HDFS



Another potential confusion point is the relationship between HDFS and the local file system. And by local, I mean the file system that already existed on the VMs. In general, a local file system is related to only one server, it manages all the file on that server. While HDFS, creates an abstraction as if all the different servers are one machine and files could be read from it and write to it.

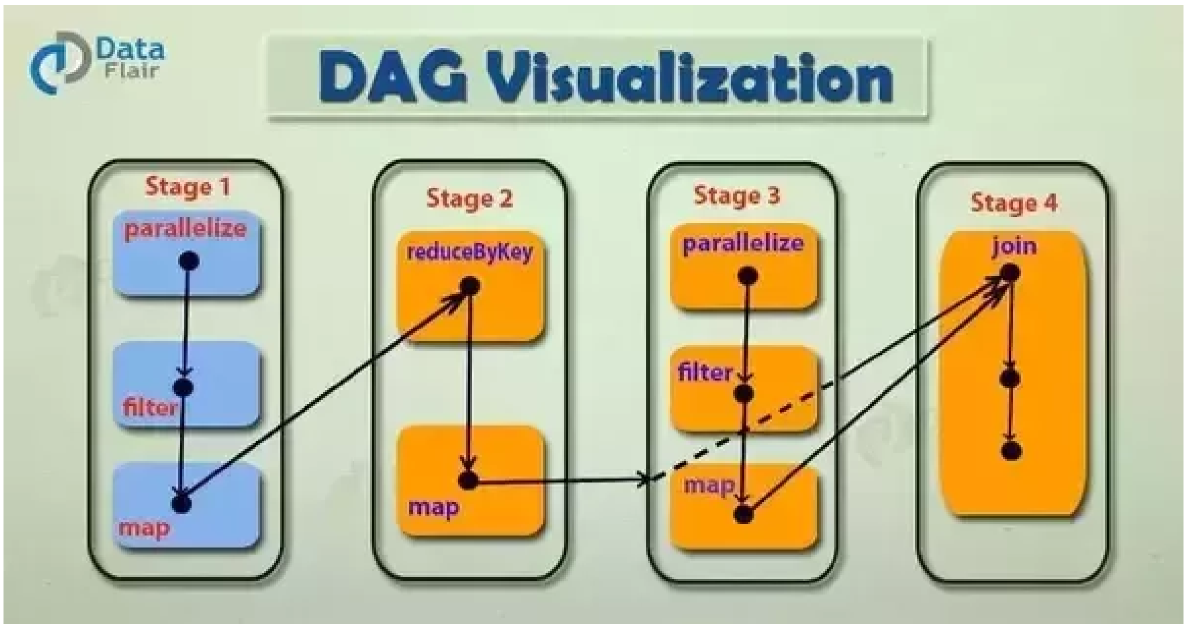
This video gives concrete examples to discuss the difference between HDFS and the local file system.

## Spark Architecture



The major components of Spark are the **Driver**, the **Cluster Manager**, and the **Executors**.

**Spark Driver**: After you submit the job, the driver is the major component that controls the workflow. It would first come up with the DAG (Direct Acyclic Graph)/action plan mentioned in the lecture and then send works to executors which are contained in the worker node(s).

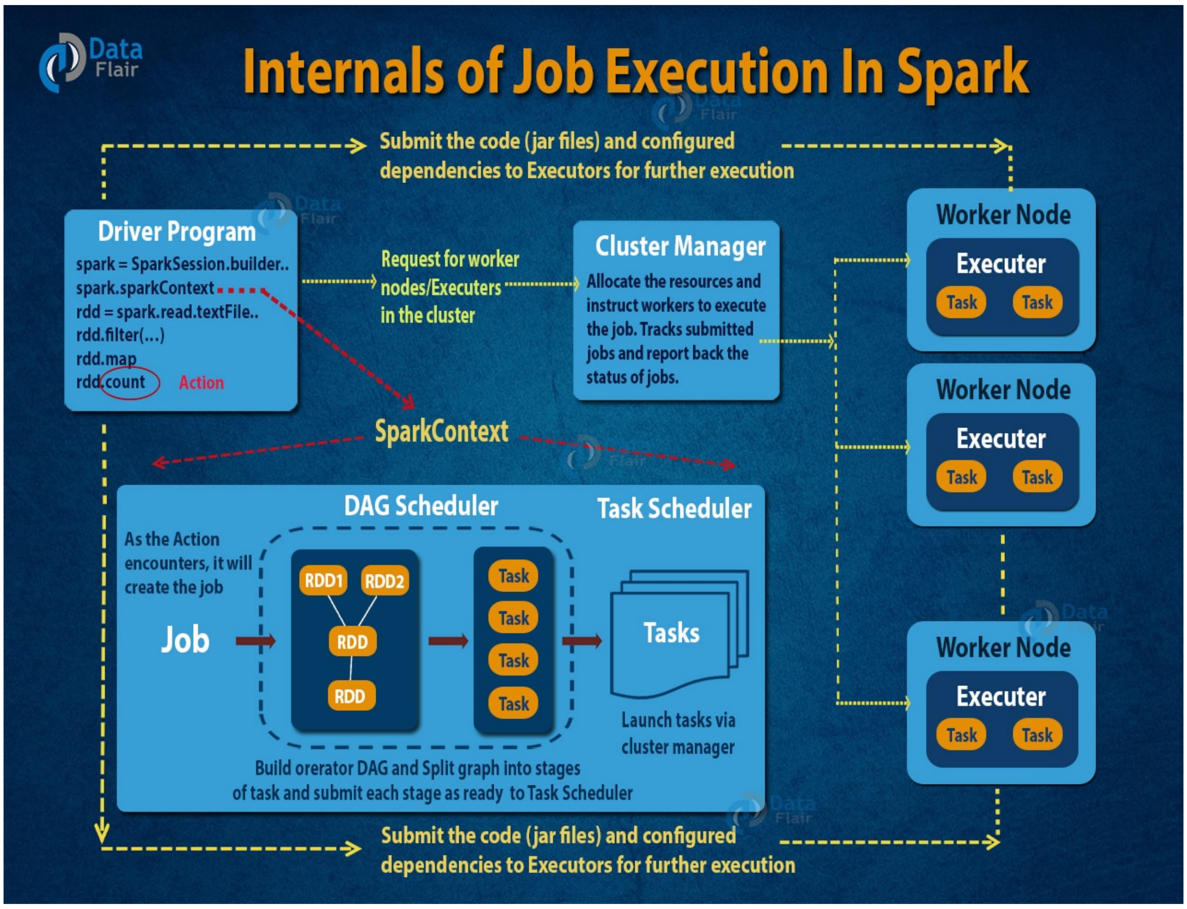


As you can see, a job is split into tasks and stages. Many times some optimization would be done on the DAG for better performance.

**Spark Context**: On the lecture slides you have seen it many times as variable name sc. Each time you submit a job, the Spark Driver would create an instance of an application and that instance of the application is called the context of that application. The context will be destroyed when the job is finished.

**Spark Executors:** After the action plan is made up by a Spark Driver, executors are the actual workers that finish those tasks stage by stage.

**Cluster Manager:** Cluster manager monitors and controls the computational resources on the Worker Nodes and it is responsible for allocating resources to jobs (especially when multiple jobs are submitted at the same time). **Spark Master** plays the role of cluster manager when there is no other manager. Dataproc uses YARN as the Cluster Manager and more discussion about it will come in the next section.



Data Flair
Internals of Job Execution In Spark
Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to Executors for further execution

## YARN

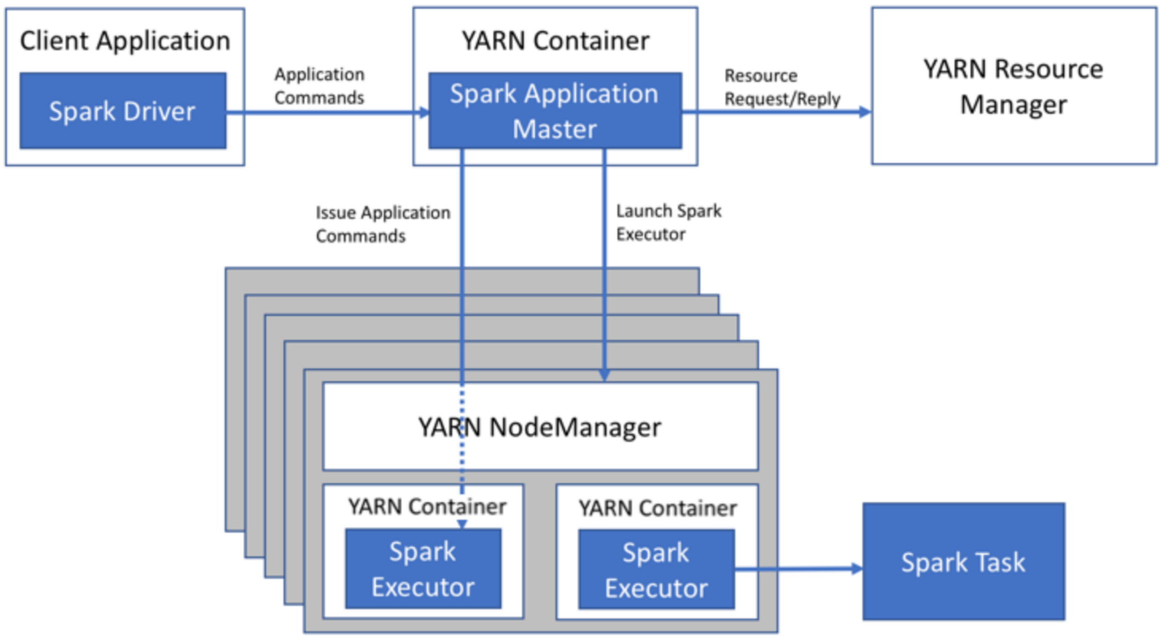
Once the user submits the spark job to the YARN Resource Manager which is the cluster manager, it creates a YARN application master which would serve as the application master. The Spark Driver which is responsible for spark program is also created at the same instance.

YARN virtually separates computing resources into small units called containers. The application master will reserve containers for their tasks and YARN would allocate containers to them.

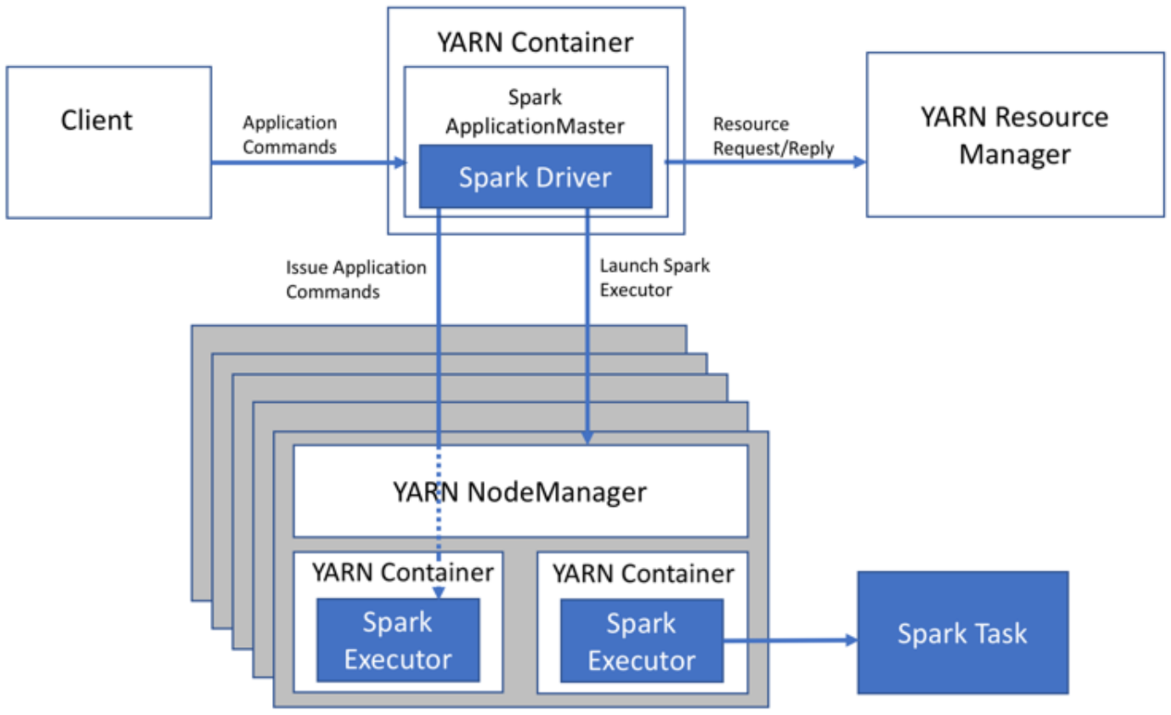
There are two modes for YARN. Client mode and Cluster mode. The default running mode on Dataproc is the Client mode.

The major difference is that whether Spark Driver runs inside the YARN cluster or not. The detailed difference is outside of the scope of this course and this blog post might be helpful if you want to learn more about YARN.

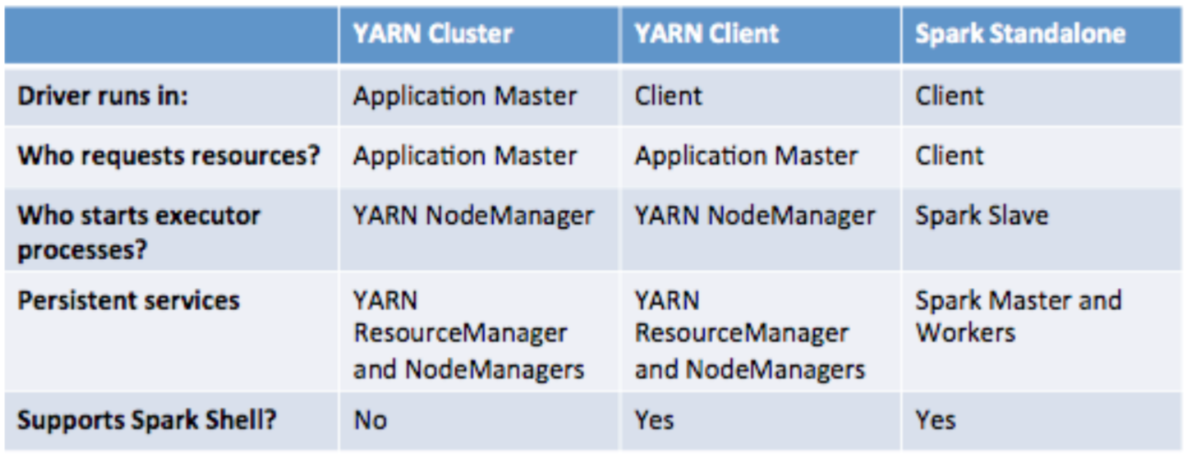
**Client mode:**



**Cluster mode:**



**Comparision:**



| | YARN Cluster | YARN Client | Spark Standalone |
|---|---|---|---|
| **Driver runs in:** | Application Master | Client | Client |
| **Who requests resources?** | Application Master | Application Master | Client |
| **Who starts executor processes?** | YARN NodeManager | YARN NodeManager | Spark Slave |
| **Persistent services** | YARN ResourceManager and NodeManagers | YARN ResourceManager and NodeManagers | Spark Master and Workers |
| **Supports Spark Shell?** | No | Yes | Yes |

## Regular Expression

In part 1 task 2 of the homework, you are asked to extract internal links from the text field. One of the way to do it is to use regular expression.

Regular expressions, commonly known as  regex or regexp, uses a sequence of characters to create a searchable pattern. The pattern then would be applied over text or data to find strings that matches the pattern.

Here is an simple example using python:

```
import re
text = "The park in NewYork"
x = re.search("The.*NewYork", txt)
```

Here the '.' would match any character and * means it would match '.' zero or more repetitions.

So any string that contains "The" at the beginning and ending with "NewYork"  would match the pattern.

Here is a short tutorial that you could get some practice or brush up your regex skill.

One good practice is to use online regular expression testers such as this one to debug your regular expression before you apply them in your program.

## UDF

In traditional SQL environments, SQL syntax doesn't allow you to define functions to take in input argument(s) and output some result, and many databases allow you to define User Defined Functions to achieve that.

In the context of Spark Dataframe, UDFs transform values from a single row within a table to produce a single corresponding output value per row.

Here is an example UDF from Stack Overflow:

```
+---------------------------------------+----------+
|                  Text                 | Key_word |
+---------------------------------------+----------+
| First random text tree cheese cat      | tree     |
| Second random text apple pie three     | text     |
| Third random text burger food brain    | brain    |
| Fourth random text nothing thing chips | random   |
+---------------------------------------+----------+
```

The task was to use the word that appeared before the Key_word to construct the word_bef_key_word column. The result would be like the following table. For example, the word **text** appeared before the word **tree** in the first row and the value on the third column became **text** after the operation.

```
+---------------------------------------+----------+-------------------+---+
|                  Text                 | Key_word | word_bef_key_word |   |
+---------------------------------------+----------+-------------------+---+
| First random text tree cheese cat      | tree     | text              |   |
| Second random text apple pie three     | text     | random            |   |
| Third random text burger food brain    | brain    | food              |   |
| Fourth random text nothing thing chips | random   | Fourth            |   |
+---------------------------------------+----------+-------------------+---+
```

Here is one of the answers that used user defined functions:

```
import re
from pyspark.sql.functions import udf

def get_previous_word(text, key_word):
    matches = re.findall(r'\w+(?= {kw})'.format(kw=key_word), text)
    return matches[0] if matches else None

get_previous_word_udf = udf(
    lambda text, key_word: get_previous_word(text, key_word),
    StringType()
)

df = df.withColumn('word_bef_key_word', get_previous_word_udf('Text',
'Key_word'))
```

## Spark Dataframe Operations

In the lecture, many RDD operations have been taught and the slide deck is the best place to find some of the useful RDD operations. Here we have listed some of the useful Dataframe operations which are from thisdocumentation.

`cache`()[source]

Persists the `DataFrame` with the default storage level (`MEMORY_AND_DISK`).

`dropna`(*how='any'*, *thresh=None*, *subset=None*)[source]

`collect`()[source]

Returns all the records as a list of `Row`.

```
>>> df.collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
```

*property* `columns`

Returns all column names as a list.

```
>>> df.columns
['age', 'name']
```

`count`()[source]

Returns the number of rows in this `DataFrame`.

```
>>> df.count()
2
```

`distinct`()[source]

Returns a new `DataFrame` containing the distinct rows in this `DataFrame`.

```
>>> df.distinct().count()
2
```

`dropna`(*how='any'*, *thresh=None*, *subset=None*)[source]

Returns a new `DataFrame` omitting rows with null values. `DataFrame.dropna()` and `DataFrameNaFunctions.drop()` are aliases of each other.

- Parameters

  **how** – 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null.

  **thresh** – int, default None If specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.

  **subset** – optional list of column names to consider.

```
>>> df4.na.drop().show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
+---+------+-----+
```

`filter`(*condition*)[source]

Filters rows using the given condition.

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

- Parameters

  **condition** – a `Column` of `types.BooleanType` or a string of SQL expression.

```
>>> df.filter(df.age > 3).collect()
[Row(age=5, name='Bob')]
>>> df.where(df.age == 2).collect()
[Row(age=2, name='Alice')]
>>> df.filter("age > 3").collect()
[Row(age=5, name='Bob')]
>>> df.where("age = 2").collect()
[Row(age=2, name='Alice')]
```

`first`()[source]

Returns the first row as a `Row`.

```
>>> df.first()
Row(age=2, name='Alice')
```

`foreach`(*f*)[source]

Applies the `f` function to all `Row` of this `DataFrame`.

This is a shorthand for `df.rdd.foreach()`.

```
>>> def f(person):
...     print(person.name)
>>> df.foreach(f)
```

`groupBy`(**cols*)[source]

Groups the `DataFrame` using the specified columns, so we can run aggregation on them. See `GroupedData` for all the available aggregate functions.

`groupby()` is an alias for `groupBy()`.

- Parameters

  **cols** – list of columns to group by. Each element should be a column name (string) or an expression (`Column`).

```
>>> df.groupBy().avg().collect()
[Row(avg(age)=3.5)]
>>> sorted(df.groupBy('name').agg({'age': 'mean'}).collect())
[Row(name='Alice', avg(age)=2.0), Row(name='Bob', avg(age)=5.0)]
>>> sorted(df.groupBy(df.name).avg().collect())
[Row(name='Alice', avg(age)=2.0), Row(name='Bob', avg(age)=5.0)]
>>> sorted(df.groupBy(['name', df.age]).count().collect())
[Row(name='Alice', age=2, count=1), Row(name='Bob', age=5, count=1)]
```

`head`(*n=None*)[source]

Returns the first `n` rows.

Note

This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

- Parameters

  **n** – int, default 1. Number of rows to return.

- Returns

  If n is greater than 1, return a list of `Row`. If n is 1, return a single Row.

```
>>> df.head()
Row(age=2, name='Alice')
>>> df.head(1)
[Row(age=2, name='Alice')]
```

`join`(*other*, *on=None*, *how=None*)[source]

Joins with another `DataFrame`, using the given join expression.

- Parameters

  **other** – Right side of the join**on** – a string for the join column name, a list of column names, a join expression (Column), or a list of Columns. If on is a string or a list of strings indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join.**how** – str, default `inner`. Must be one of: `inner`, `cross`, `outer`, `full`, `full_outer`, `left`, `left_outer`, `right`, `right_outer`, `left_semi`, and `left_anti`.

The following performs a full outer join between `df1` and `df2`.

```
>>> df.join(df2, df.name == df2.name, 'outer').select(df.name,
df2.height).collect()
[Row(name=None, height=80), Row(name='Bob', height=85), Row(name='Alice',
height=None)]
>>> df.join(df2, 'name', 'outer').select('name', 'height').collect()
[Row(name='Tom', height=80), Row(name='Bob', height=85), Row(name='Alice',
height=None)]
>>> cond = [df.name == df3.name, df.age == df3.age]
>>> df.join(df3, cond, 'outer').select(df.name, df3.age).collect()
[Row(name='Alice', age=2), Row(name='Bob', age=5)]
>>> df.join(df2, 'name').select(df.name, df2.height).collect()
[Row(name='Bob', height=85)]
>>> df.join(df4, ['name', 'age']).select(df.name, df.age).collect()
[Row(name='Bob', age=5)]
```

`limit`(*num*)[source]

Limits the result count to the number specified.

```
>>> df.limit(1).collect()
[Row(age=2, name='Alice')]
>>> df.limit(0).collect()
[]
```

`orderBy`(**cols*, **kwargs*)

Returns a new `DataFrame` sorted by the specified column(s).

- Parameters

  **cols** – list of `Column` or column names to sort by.**ascending** – boolean or list of boolean (default `True`). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the cols.

```
>>> df.sort(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.sort("age", ascending=False).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> from pyspark.sql.functions import *
>>> df.sort(asc("age")).collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
>>> df.orderBy(desc("age"), "name").collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(["age", "name"], ascending=[0, 1]).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
```

`printSchema`()[source]

Prints out the schema in the tree format.

```
>>> df.printSchema()
root
 |-- age: integer (nullable = true)
 |-- name: string (nullable = true)
```

`repartition`(*numPartitions*, **cols*)[source]

Returns a new `DataFrame` partitioned by the given partitioning expressions. The resulting `DataFrame` is hash partitioned.

- Parameters

  **numPartitions** – can be an int to specify the target number of partitions or a Column. If it is a Column, it will be used as the first partitioning column. If not specified, the default number of partitions is used.

*Changed in version 1.6:* Added optional arguments to specify the partitioning columns. Also made numPartitions optional if partitioning columns are specified.

```
>>> df.repartition(10).rdd.getNumPartitions()
10
>>> data = df.union(df).repartition("age")
>>> data.show()
+---+-----+
|age| name|
+---+-----+
|  5|  Bob|
|  5|  Bob|
|  2|Alice|
|  2|Alice|
+---+-----+
>>> data = data.repartition(7, "age")
>>> data.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
|  2|Alice|
|  5|  Bob|
+---+-----+
>>> data.rdd.getNumPartitions()
7
>>> data = data.repartition("name", "age")
>>> data.show()
+---+-----+
|age| name|
+---+-----+
|  5|  Bob|
|  5|  Bob|
|  2|Alice|
|  2|Alice|
+---+-----+
```

`select`(**cols*)[source]

Projects a set of expressions and returns a new `DataFrame`.

- Parameters

  **cols** – list of column names (string) or expressions (`Column`). If one of the column names is '*', that column is expanded to include all columns in the current `DataFrame`.

```
>>> df.select('*').collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
>>> df.select('name', 'age').collect()
[Row(name='Alice', age=2), Row(name='Bob', age=5)]
>>> df.select(df.name, (df.age + 10).alias('age')).collect()
[Row(name='Alice', age=12), Row(name='Bob', age=15)]
```

`selectExpr`(**expr*)[source]

Projects a set of SQL expressions and returns a new `DataFrame`.

This is a variant of `select()` that accepts SQL expressions.

```
>>> df.selectExpr("age * 2", "abs(age)").collect()
[Row((age * 2)=4, abs(age)=2), Row((age * 2)=10, abs(age)=5)]
```

`show`(*n=20*, *truncate=True*, *vertical=False*)[source]

Prints the first `n` rows to the console.

- Parameters

  **n** – Number of rows to show.**truncate** – If set to `True`, truncate strings longer than 20 chars by default. If set to a number greater than one, truncates long strings to length `truncate` and align cells right.**vertical** – If set to `True`, print output rows vertically (one line per column value).

```
>>> df
DataFrame[age: int, name: string]
>>> df.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
+---+-----+
>>> df.show(truncate=3)
+---+----+
|age|name|
+---+----+
|  2| Ali|
|  5| Bob|
+---+----+
>>> df.show(vertical=True)
-RECORD 0-----
```

```
 age  | 2
 name | Alice
-RECORD 1-----
 age  | 5
 name | Bob
```

`sort`(**cols***, ***kwargs*)[source]

Returns a new `DataFrame` sorted by the specified column(s).

- Parameters

  **cols** – list of `Column` or column names to sort by.**ascending** – boolean or list of boolean (default `True`). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the cols.

```
>>> df.sort(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.sort("age", ascending=False).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> from pyspark.sql.functions import *
>>> df.sort(asc("age")).collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
>>> df.orderBy(desc("age"), "name").collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(["age", "name"], ascending=[0, 1]).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
```

`toDF`(**cols*)[source]

Returns a new class:DataFrame that with new specified column names

- Parameters

  **cols** – list of new column names (string)

```
>>> df.toDF('f1', 'f2').collect()
[Row(f1=2, f2='Alice'), Row(f1=5, f2='Bob')]
```

`withColumn`(*colName*, *col*)[source]

Returns a new `DataFrame` by adding a column or replacing the existing column that has the same name.

The column expression must be an expression over this `DataFrame`; attempting to add a column from some other `DataFrame` will raise an error.

- Parameters

  **colName** – string, name of the new column.**col** – a `Column` expression for the new column.

```
>>> df.withColumn('age2', df.age + 2).collect()
[Row(age=2, name='Alice', age2=4), Row(age=5, name='Bob', age2=7)]
```

`avg`(**cols*)[source]

Computes average values for each numeric columns for each group.

`mean()` is an alias for `avg()`.

- Parameters

  **cols** – list of column names (string). Non-numeric columns are ignored.

```
>>> df.groupBy().avg('age').collect()
[Row(avg(age)=3.5)]
>>> df3.groupBy().avg('age', 'height').collect()
[Row(avg(age)=3.5, avg(height)=82.5)]
```

*class* `pyspark.sql.``Column`(*jc*)[source]

A column in a DataFrame.

`Column` instances can be created by:

```
# 1. Select a column out of a DataFrame

df.colName
df["colName"]

# 2. Create from an expression
df.colName + 1
1 / df.colName
```

`desc`()

Returns a sort expression based on the descending order of the column.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([('Tom', 80), ('Alice', None)], ["name",
"height"])
>>> df.select(df.name).orderBy(df.name.desc()).collect()
[Row(name='Tom'), Row(name='Alice')]
```

`isNotNull`()

True if the current expression is NOT null.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(name='Tom', height=80), Row(name='Alice',
height=None)])
>>> df.filter(df.height.isNotNull()).collect()
[Row(height=80, name='Tom')]
```

`isNull`()

True if the current expression is null.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(name='Tom', height=80), Row(name='Alice',
height=None)])
>>> df.filter(df.height.isNull()).collect()
[Row(height=None, name='Alice')]
```