Pyspark

→ What is spark?

- It is general purpose in-memory computation Engine

1. General purpose:

Data cleaning - PIG
Query - HIVE
ML operation - MAHOUT
⋮

When spark Introducted; each & every operation; we can perform in spark itself

2. Computation: (Spark)

$\boxed{\text{Hadoop vs Spark}}$ ✗ wrong comparison

$\boxed{\text{MapReduce vs. Spark}}$ ✓ right comparison

→ Hadoop Provide 3 components

i. HDFS - storage
ii. Resource manager
iii. Map Reduce - computation

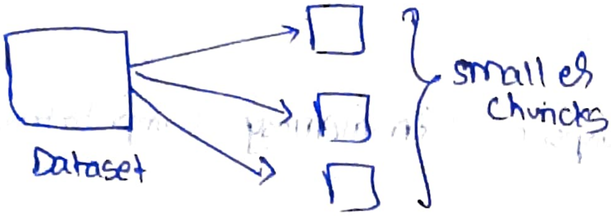
we can even use spark with HDFS

3. In Memory: (RAM)

problem with map reduce

- It stores the Intermediate results like aggregation, joins, filtering, functions; everything gets stores in DISK
- so; for everytime, if the results are needed; need to go to DISK & fetch from DISK
- It's time taking; so spark has Introducted.
- It stores intermediate results in memory(ram)
- So spark is faster than MapReduce

Spark Architecture

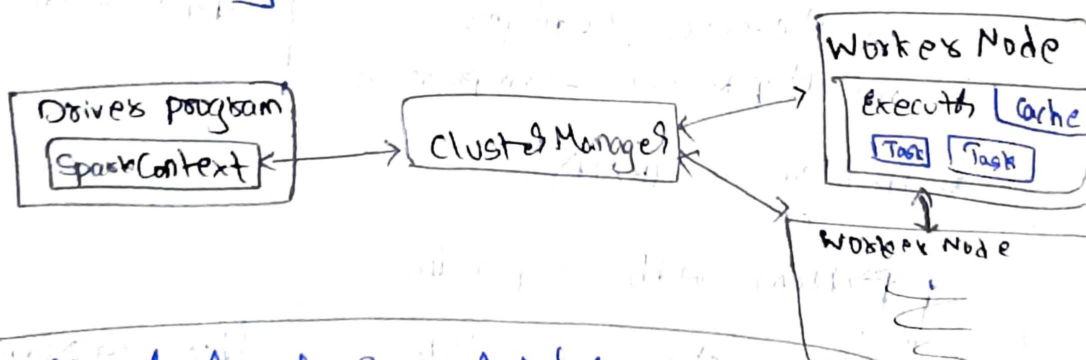


- It follows master-slave Architecture.
- It works on the concept of Clusters.

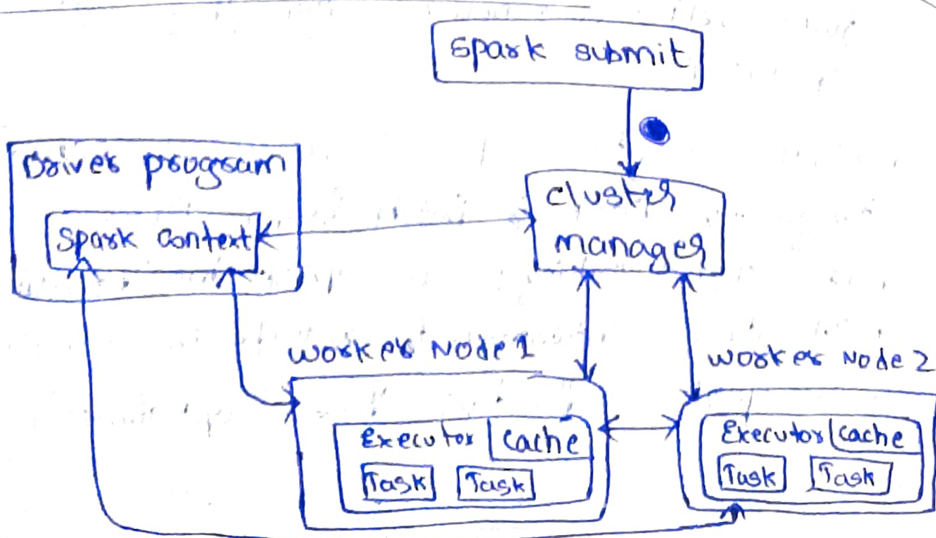
Clusters → groups of machines that can process & Analyze the data

- Spark distributes the data & computations across multiple nodes in the cluster, allowing for parallel processing & faster data processing

→ One of the node acts as master
remaining act as worker nodes



Official Apache Spark Architecture



1) When a user submit a spark Application;

2) The cluster manager takes the spark-context & assigned a node & share it.

3) That node will act as Driver (or) Master node.

4) Cluster manager (or) resource manager will basically installed on top of all the nodes

5) Driver Node communicates with cluster manager (YARN, Mesos) whose work is to manage the resources.

- Driver program calls the main application & SparkContext is created which is the entry point of spark functionally.
- It would communicate with cluster manager whose job is to allocate executors inside worker nodes.
- Driver has components like DAG scheduler, Task scheduler etc..
- Cluster manager would send status & details of executors to Driver.
- Then Driver would schedule job.
- It would take RDD (Resilient Distributed Dataset) and translate into execution graph called DAG (Directed Acyclic Graph).
- These jobs would further split into different stages.

- In each stage, it would take split the work for each executors called task & send to executors for processing.
- Executor would perform that task execution given by driver & send the result back to driver.

RDD:

- Resilient Distributed Dataset

Dataset is nothing but the data that we provided.

- The distribution means the input data is stored across all worker nodes.
- Resilient means fault tolerance

- RDD is distributed collection of dataset in Memory

So, file1 is in HDFS (lazy loading)

rdd1 = load file from hdfs
rdd2 = rdd1.filter(-)

This will won't be happen untill

rdd3.collect() → action got it.

→ There are 2 type of operations happens in rdd
1. transformation
2. action

DAG:

- (Directed Acyclic Graph)

Spark practical setup

1. use databricks (✓)
2. use jupyter notebook for spark

- Anaconda
- Java jdk
- Apache spark
- winutils

→ **create dataframe:**

Dataframe:

→ Dataframe is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows & columns)

df =

```
df = spark.read.csv('——')
df.show()
display(df)
```

⇒ **dataframe with header**

```
df1 = spark.read.csv('——'
                     , header = True)
```

```
df1 = spark.read.csv('—', header = True)
```

→ How to print/get schema of a dataframe!

```
df.printSchema()
```

Dataframe with header and proper schema :

```
df = spark.read.csv('—',
                    header = True,
                    inferSchema = True
)
```

if we didnot specify it;
By Default values in the schema (columns) will be consider as 'string'

```
df.printSchema()
```

(or)

```
df = spark.read.format("csv") \
     .option('header', True) \
     .option('inferSchema', True) \
     .load("—")
```

(or)

(or)

```
df = spark.read.format('csv') \
        .options(header='True', inferSchema='True') \
        .load('—')
```

↳ How to read json file?

```
df = spark.read.json("—")
```

Then this will be okay.

if we have json files as single-line

{'a' : 'A'}, {'b' : 'B'}, {...}, {...}

But;

how to read multiple-line json file?

[{① ②}, {① ②}, {① ②}, {—}, ...]

```
df = spark.read.json("—", multiLine = True)
```

```
df = spark.read.format("json") \
        .option("multiLine", True) \
        .load("—")
```

```
df = spark.read.option("multiLine", True) \
        .json("—")
```

select:

→ select() function is used to select single, multiple, ~~complex~~ column by index, all columns from the list & the nested columns from a Dataframe

→ df.select("col1", "col2").display()

(or)

* imp

df.select(df.col1, df.col2, df.col3)

select() using column by index:

emp	name	salary
index → 0	1	2

⇒ df.select(df.columns[index])

df.select(df.columns[2])

df.select(df.columns[1:4])

with Column:

- Pyspark withColumn() is a transformation function of Dataframe which is used to change the value, convert the datatype of an existing column, create a new column() & more.

- withColumnRenamed() to rename a Dataframe column, we often need to rename one column (or) multiple (or all) columns on pyspark Dataframe.

withColumn()

1. add new column based on existing column
2. add new column
3. change datatype
4. update the value of an existing column

→ df.withColumn("total_cost", df.price * df.quantity)
df.withColumn("new_column", lit("val"))   lit → { pyspark function used to add a constant (literal) value }
df.withColumn("salary", col("salary").cast("Integer"))
df.withColumnRenamed("old_col", "new_col_name")
df.withColumn("salary", col("salary") - 5000)

Filter():

- pyspark filter() function is used to create a new Dataframe by filtering the elements from an existing Dataframe based on the given condition.

df.filter(df.col1 == "val 1")
df.filter(df.col1 != "val1")
df.filter((df.col1 == "val1") & (df.col2 == "val2"))
df.filter((df.col1 == "val1") | (df.col2 == "val2"))
df.filter(col("col1").startswith("r")) df.filter(col("col1").endswith("l"))
df.filter(col("col1").like("%oul%"))

Distint() and dropDuplicates()

distint() : transformation used to drop/remove the duplicates rows (all columns) from dataframe.

dropDuplicates() : used to drop rows, based on selected (one or multiple) columns

⇒ df = df.distint()

df = df.dropDuplicates(['col1', 'col2'])

↓

if there are duplicates in the attributes then those will be removed with their respecitive record.

dropDuplicates()	: removes rows that are identical across ALL columns
dropDuplicates(['col1','col2'])	: removes duplicates based only on selected columns.
dropDuplicates(subset=['col1'])	: same as above

→ How to deduplicate using latest ~~stam~~ timestamp
↓
window + row_number()

RDD:

- RDD (Resilient Distributed Dataset) is a spark's low-level Distributed data structure that stores data across multiple machines & processes it in parallel.

Why RDD exists?

- Before Dataframes existed; spark needed:
  - → Distributed data
  - → fault tolerance
  - → Parallel Computation

- RDD was the first core abstraction of spark

Note:
Dataframes internally use RDDs but add schema + optimizations

Feature	RDD	Dataframe
API Level	Low-level	high-level
schema	✗	✓
optimization	✗	✓
speed	slower	faster
SQL support	✗	✓
usage Today	Rare	90% jobs

```
rdd = spark.sparkContext.parallelize([1,2,3,4,5])
rdd = spark.sparkContext.parallelize([ —— ])
```
python collection

from a file

```
rdd = spark.sparkContext.textFile("——")
```

RDD operations

```
rdd2 = rdd.map(lambda x : x*2)
rdd3 = rdd.filter( lambda x : x>3)
```

Actions

```
rdd.collect()
rdd.count()
rdd.take(3)
```

when should we use RDD? (rare)

use rdd only if:

- → we need low-level control
- → we have unstructured data
- → Complex logic not supported in Dataframe

```
rdd.mapPartitions(——)
```

Dataframe ——→ RDD

```
rdd = df.rdd
```

RDD ——→ Dataframe

initially need to add schema

①

```
rdd = spark.sparkContext.parallelize([
        (1, 'A'), (2, 'B')])
df = rdd.toDF(['id', 'name'])
```

②

```
df = spark.createDataframe(rdd, ['id', 'name'])
```

to_date() (vs) date_format()

① to_date()

$\boxed{\text{to\_date()}}$ converts string → Date

- It doesn't control how the date is displayed

→ spark has two different things:

① Data type (what it is)

② Display format (how it looks)

Note:

to_date()

only changes the data type, not the display format

Input		Output
"12/31/2003"	→ to_date() →	12/31/2003

Converted correctly

But format will not be parenfeet

② date_format()

↳ This converts Date to String

date_format ("____", "MM/dd/yyyy")

"dd/MM/yyyy"