

PySpark Index

- ***Introduction to Spark***
 - ❖ ***Why Spark was developed***
 - ❖ ***What is Spark/Spark Features***
 - ❖ ***Spark Components***
- ***Spark Session***
 - ❖ ***Spark Context***
 - ❖ ***Spark Session***
 - ❖ ***spark-submit***
- ***RDD Fundamentals***
 - ❖ ***What is RDD***
 - ❖ ***RDD Features***
 - ❖ ***When to use RDDs***
 - ❖ ***RDD Properties and Problems***
- ***Create RDD***
- ***RDD Operations***
 - ❖ ***Transformations and Actions***
 - ✓ *Row Level Transformations (map, flatMap, filter)*
 - ✓ *Join (join, cogroup, cartesian)*
 - ✓ *Total Aggregations (Count, reduce)*
 - ✓ *Shuffle and Combiner*
 - ✓ *Key Aggregations (groupByKey ,reduceByKey, aggregateByKey, countByKey)*
 - ✓ *Ranking*

- ✓ *sorting*
- ✓ *set*
- ✓ *sample*
- ✓ *repartition and coalesce*
- ✓ *Reading/writing from rdd to HDFS*

- **Spark Cluster Architecture**

- ❖ *Detailed Spark Cluster Execution Architecture*
- ❖ *YARN Cluster Manager*
- ❖ *JVM Processes*
- ❖ *Commonly Used Terms in Spark Execution Framework*
- ❖ *Narrow and Wide Transformations*
- ❖ *RDD Lineage*
- ❖ *DAG Scheduler*
- ❖ *Task Scheduler*

- **RDD Persistence**

- **Shared Variables**

- ❖ *Broadcast*
- ❖ *Accumulator*

- **Spark SQL**

- ❖ *Architecture*
- ❖ *Catalyst Optimizer*
- ❖ *Volcano Iterator Model*
- ❖ *Tungsten Execution Engine*
- ❖ *Benchmarks*
- ❖ *Understanding Execution Plan*

- **DataFrame Fundamentals**
 - ❖ What is a DataFrame
 - ❖ Sources
 - ❖ Features
 - ❖ Organization
- **SparkSession**
 - ❖ Introduction to SparkSession
 - ❖ Spark Object
 - ❖ Spark Submit
 - ❖ **Commonly used Functions**
 - ✓ *version*
 - ✓ *range*
 - ✓ *createDataFrame*
 - ✓ *sql*
 - ✓ *table*
 - ✓ *sparkContext*
 - ✓ *conf*
 - ✓ *read (csv, text, orc, parquet,json,avro,hive,jdbc)*
 - ✓ *udf*
 - ✓ *newSession*
 - ✓ *stop*
 - ✓ *catalog*

- *DataTypes*
- *DataFrame Rows*
- *DataFrame Columns*
 - ❖ *Column Functions*
- *DataFrame ETL*
 - ❖ *DataFrame APIs*
 - ✓ *selection*
 - ✓ *filter*
 - ✓ *sort*
 - ✓ *set*
 - ✓ *join*
 - ✓ *aggregate*
 - ✓ *groupBy*
 - ✓ *window*
 - ✓ *sample*
 - ✓ *Other Aggregate*
 - ❖ *DataFrame built-in functions*
 - ✓ *new Column*
 - ✓ *encryption*
 - ✓ *String*

- ✓ *Regexp*
- ✓ *Date*
- ✓ *null*
- ✓ *Collection*
- ✓ *na*
- ✓ *math and statistics*
- ✓ *explode and flatten*
- ✓ *Formatting*
- ✓ *json*

❖ **Partition**

- ✓ *What is partition*
- ✓ *Repartition*
- ✓ *Coalesce*
- ✓ *Repartition vs Coalesce*

❖ **Extraction**

- ✓ *csv*
- ✓ *text*
- ✓ *Parquet*
- ✓ *Orc*
- ✓ *Json*
- ✓ *avro*
- ✓ *Hive*
- ✓ *jdbc*

- **Optimization and Management**

- ❖ *Join Strategies*
- ❖ *Driver Configurations*
- ❖ *Executor Configurations*
- ❖ *Parallelism Configurations*
- ❖ *Memory Management*

Introduction to Spark

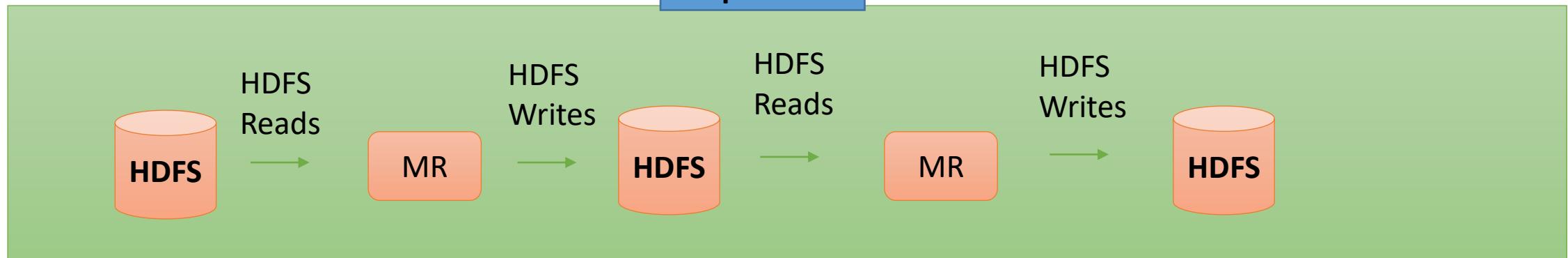
Why Spark was Developed ?

- The biggest problem in any big data project is to achieve the “Scale”. The RDBMS databases like Oracle, sql server etc are the oldest approaches of storing and processing the data. But as data grows, they are unable to scale accordingly.
- The need of new approaches led to creation of different effective systems like GFS, MapReduce.
- Still the problem was not resolved fully:

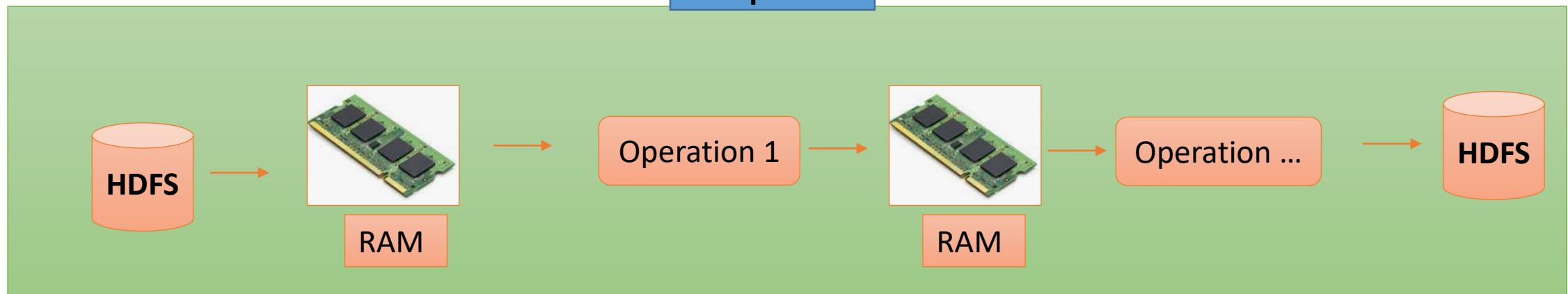
For example – Lets see the shortcomings of MapReduce.

1. Hard to manage and administer due to operational complexity.
2. MapReduce forces the data processing into Map and Reduce. Other workflows like join, filter, union etcs are missing.
3. Stateless machine – read and write to disk before or after each map and reduce stages. This repeated performance of disk I/O took its toll: large MR jobs could run for hours, or even days.
4. Java Natively - support for other languages missing.
5. Only support batch processing – Not good for streaming, Machine Learning or interactive sql like queries.

Map Reduce



Spark



- To handle these problems, engineers developed different systems at different times (Hive, Storm, Impala, Giraph, Mahout etc). Each of these systems have their own APIs and cluster configurations. This adds further operational complexity of Hadoop Map Reduce Systems.
- So the idea was to develop a single unified system having capabilities to solve all these problems.
- 2009, The researchers at UC Berkeley who had previously worked on Hadoop MapReduce took on this challenge with a project and they call it Spark.
- Ideas from Hadoop MapReduce are borrowed but enhanced in the Spark Project.
- Spark was developed as highly fault-tolerant, massively parallel, in-memory processing support, easy APIs in multiple language in unified manner.
- By 2013, Spark had gained widespread use and popularity. Some of its creators donated the Spark project to the ASF and formed a company called Databricks.
- Databricks and the community of open source developers worked to release Apache Spark 1.0 in May 2014.

0.5 → June 2012

1.0 → May 2014

1.5 → Sep 2015

2.0 → July 2016

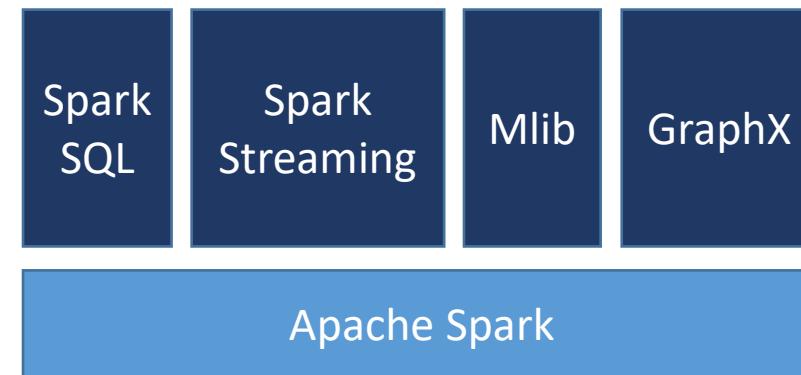
2.4 → Nov 2018

3.0 → June 2020

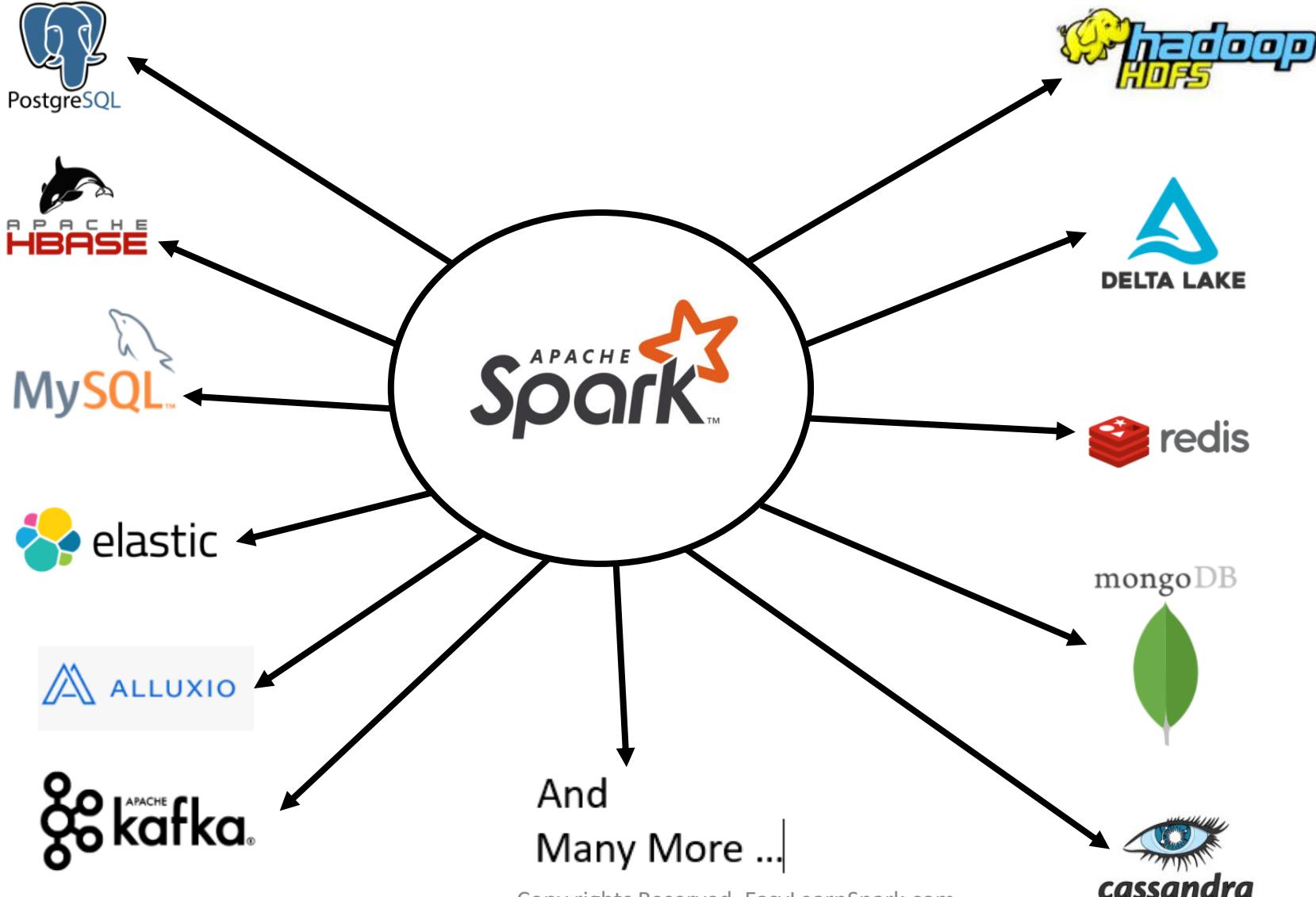
3.1 → Mar 2021

What is Spark ?

- **Unified:** Spark is a unified analytics engine mainly designed for large-scale distributed data processing, on-premises data centers or in the cloud.
- **Speed:** Spark provides in-memory storage for intermediate computations and limited disk I/O, this gives it a huge performance boost. Run workloads 100x faster.
- **Ease of Use (Polyglot):** Write applications quickly in Java, Scala, Python, R, and SQL. Scala is the functional programming language spark is written. Spark achieves simplicity by providing a fundamental abstraction of a simple logical RDD upon which all other higher-level structured data abstractions, such as DataFrames and Datasets are constructed.
- **Generality:** Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.



- **Extensibility:** Decouples Compute and Storage. It focuses on fast and parallel computation engine than on storage. Storage can be extended to read data from other sources.



- **Runs Everywhere:**
 - ✓ YARN
 - ✓ Kubernetes
 - ✓ Mesos
 - ✓ EC2 (AWS)
 - ✓ Standalone
- **Open Source**
- **Scalable**
- **Distributed**
- **Powerful Caching**
- **Real Time and Batch Processing**

Spark Components



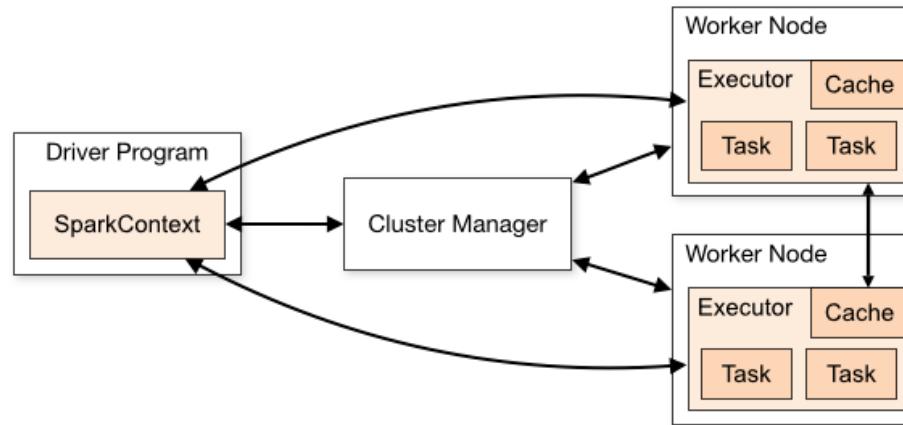
Spark Modules

- Core - Transformations and Actions -APIs such as map, reduce, join, filter etc. They typically work on RDD
- Spark SQL and Data Frames -APIs and Spark SQL interface for batch processing on top of Data Frames or Data Sets(not available for Python)
- Structured Streaming - APIs and Spark SQL interface for stream data processing on top of Data Frames
- Machine Learning Pipelines - Machine Learning data pipelines to apply Machine Learning algorithms on top of Data Frames
- GraphX – For graphs and graph-parallel computation.

Ch 2 (Spark Session)

Initializing Spark

- ✓ The first thing a Spark program is to create a **SparkContext object**, which tells Spark how to access a cluster.
- ✓ When we run any spark application, the driver program starts, which has the main() function and the SparkContext gets initiated here.



- ✓ Prior to 2.0, to create SparkContext we first need to build a SparkConf object that contains information about the application. (This is old way of doing.)

```
conf = SparkConf().setAppName(appName).setMaster(master)  
sc = SparkContext(conf=conf)
```

Initializing Spark

- ✓ In spark 2.0 and onwards, we can create the a spark object using SparkSession class. Then using this object we can access the SparkContext.

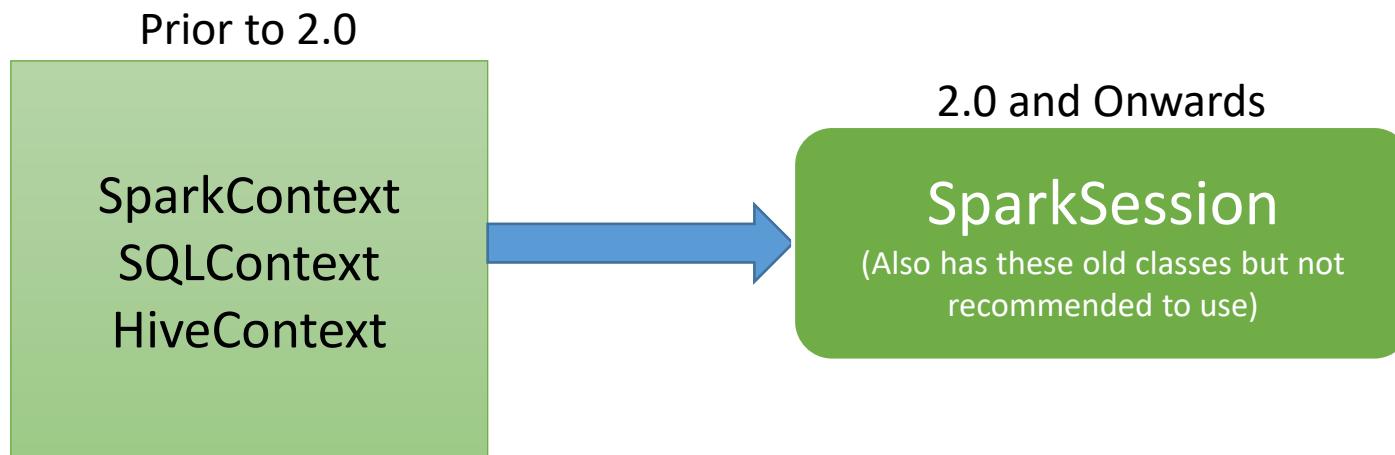
```
from pyspark.sql import SparkSession  
spark = SparkSession \  
    .builder \  
    .master('yarn') \  
    .appName("Python Spark SQL basic example") \  
    .getOrCreate()
```

- ✓ spark.sparkContext

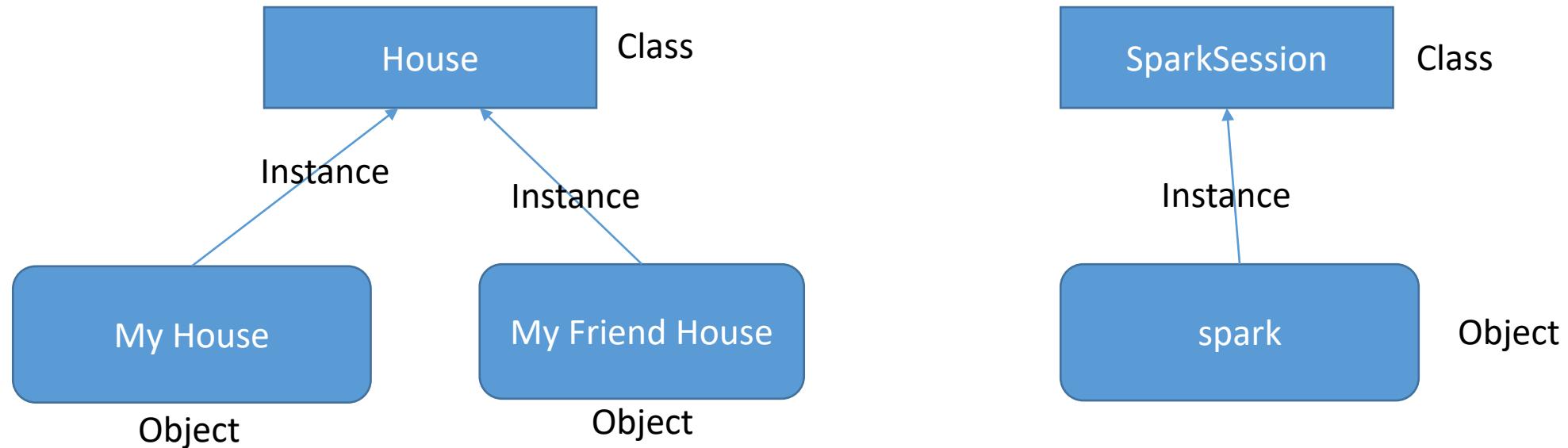
Introduction To Spark Session

SparkSession : The Entry point Spark 2.0 Onwards

- In Spark 2.0, SparkSession is the new entry point to work with RDD, DataFrame and all other functionalities.
- Prior to 2.0, SparkContext used to be an entry point.
- Almost all the APIs available in SparkContext, SQLContext, HiveContext are now available in SparkSession.
SparkContext: Entry point to work with RDD, Accumulators and broadcast variables (< Spark 2.0).
SQLContext: Used for initializing the functionalities of Spark SQL (< spark 2.0).
HiveContext: Super set of SQLContext (< spark 2.0).
- By Default, Spark Shell provides a “spark” object which is an instance of SparkSession class.



Spark Session : The Entry point Spark 2.0 Onwards



Spark Session : Spark Object & spark-submit

Spark Session : Create

```
from pyspark.sql import SparkSession  
  
spark = SparkSession \  
    .builder \  
    .master('yarn') \  
    .appName("Python Spark SQL basic example") \  
    .getOrCreate()
```

Master can be yarn, mesos, Kubernetes or local(x) , x > 0

How to Run :

1. Organize the folders and create a python file under bin folder.
2. Write above codes in the .py file.
3. Execute the file using spark-submit command.

```
spark2-submit \  
/devl/example1/src/main/python/bin/basic.py
```

Spark Session : spark-submit

Spark-submit is a utility to run a pyspark application job by specifying options and configurations.

```
spark-submit \
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key<=<value> \
--driver-memory <value>g \
--executor-memory <value>g \
--executor-cores <number of cores> \
--jars <comma separated dependencies> \
--packages <package name> \
--py-files \
<application> <application args>
```

Spark Session : spark-submit

--master : Cluster Manager (yarn, mesos, Kubernetes, local, local(k))

local – Use local to run locally with one worker node.

local(k) – Specify k with the number of cores you have locally, this runs application with k worker threads.

--deploy-mode: Either cluster or client

Cluster: Driver runs on one of the worker nodes and you can see the code as a driver on the spark UI of your application. We cant see the logs on the terminal. Logs available only in the UI or the yarn CLI.

yarn logs -applicationId application_1622930712080_16253

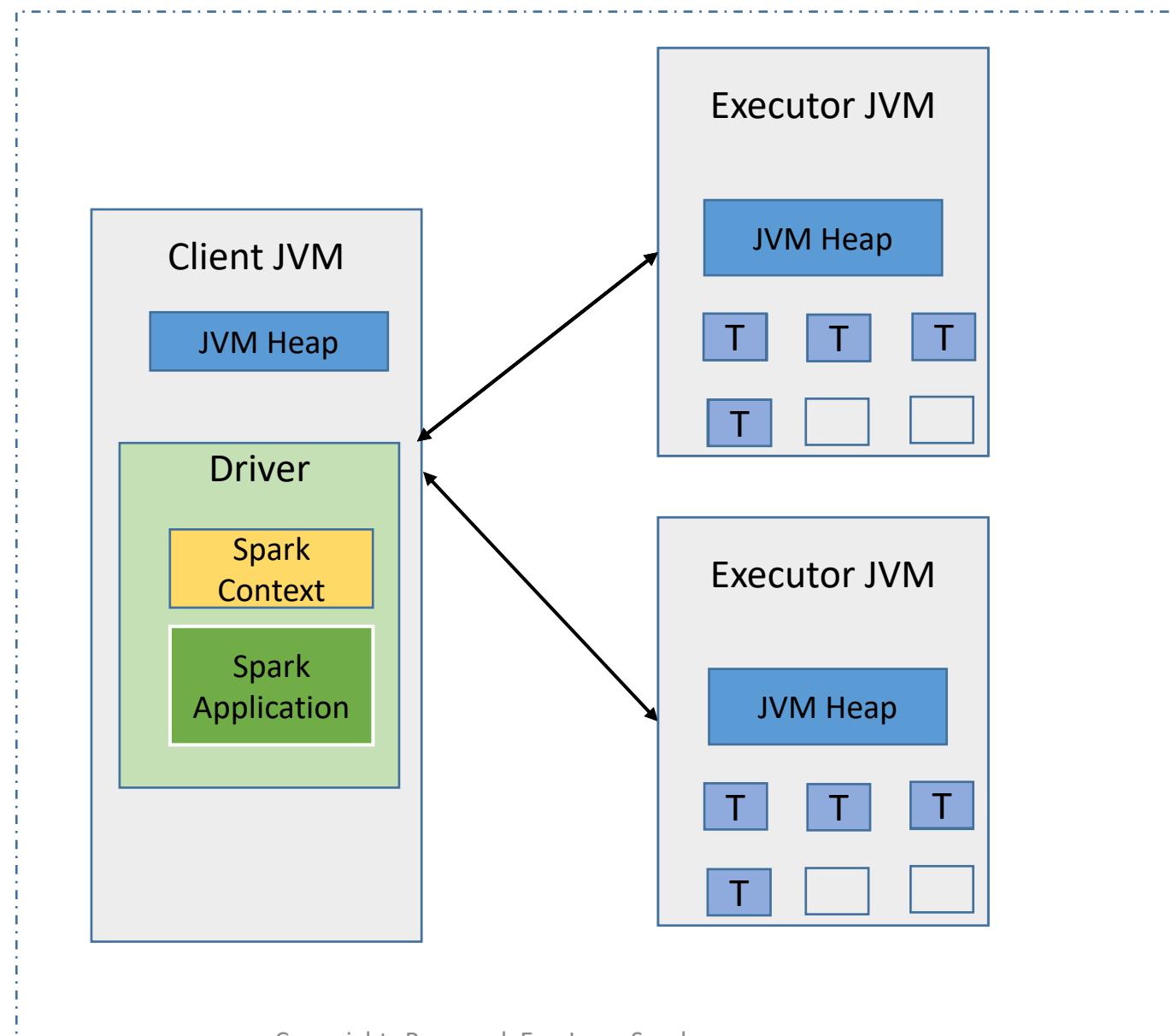
Mainly used for production jobs.

Client: Driver runs locally where we submit the application.

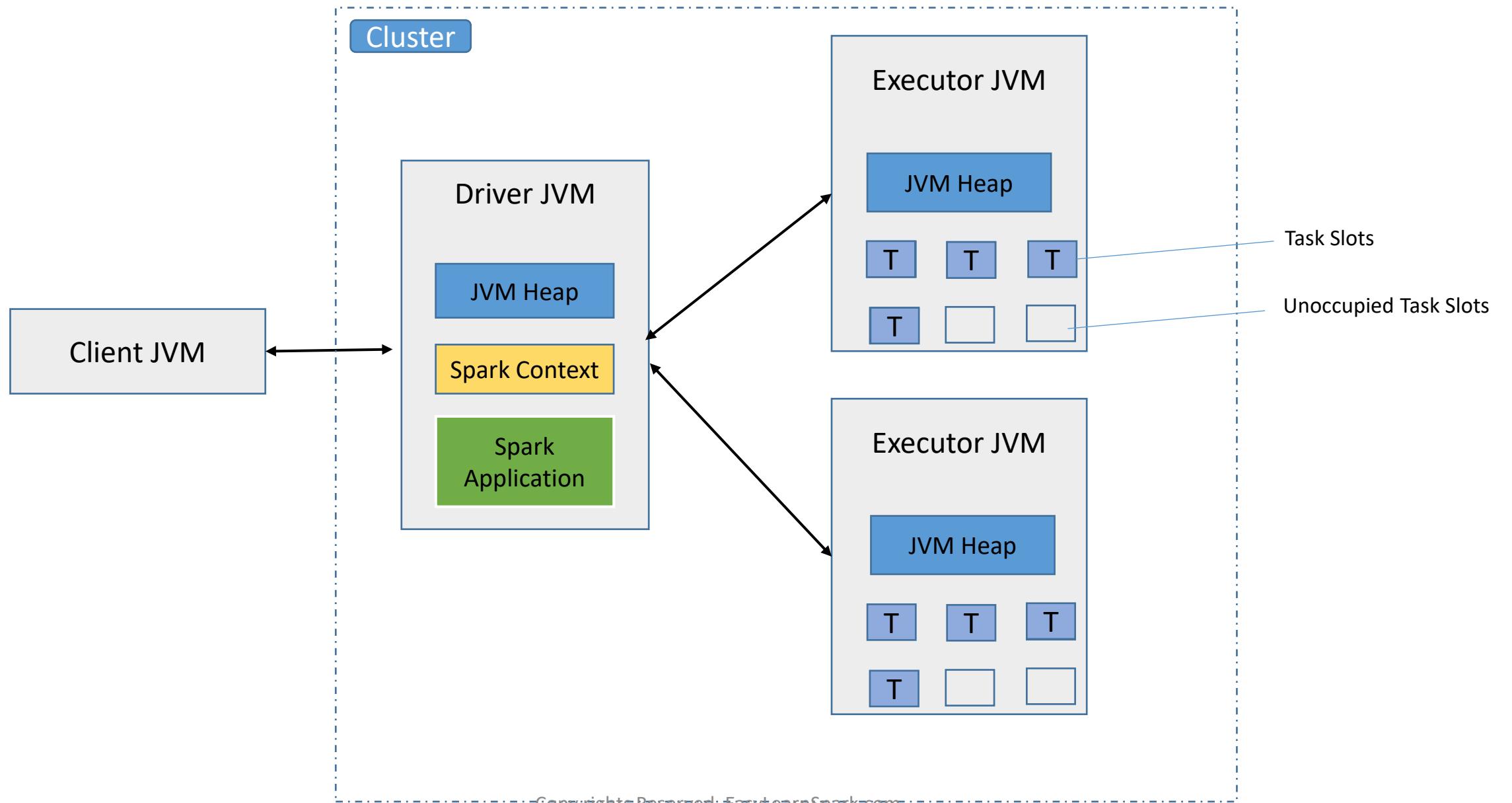
See the logs on the terminal.

Mainly used for interactive or debugging purpose.

Spark Runtime Components in Client deploy mode:



Spark Runtime Components in Cluster deploy mode:



Spark Session : spark-submit

--conf: We can provide runtime configurations, shuffle parameters, application configurations using --conf.

Ex: --conf spark.sql.shuffle.partitions = 300

This configures the number of partitions that are used when shuffling data for joins or aggregations.

<https://spark.apache.org/docs/latest/sql-performance-tuning.html>

--conf spark.yarn.appMasterEnv.HDFS_PATH="practice/retail_db/orders"

We can set environment variables like this when spark is running on yarn.

<https://spark.apache.org/docs/latest/running-on-yarn.html#configuration>

Default Spark Properties File (spark-defaults.conf):

- There is a default spark properties file at \$SPARK_HOME/conf/spark-defaults.conf.
- This could be overridden using spark-submit's "--properties-file" command-line option.

For Ex - spark-submit --properties-file [FILE]

- Individual properties of this file can be overridden by spark-sumit's conf options.

For Ex- *spark.submit.deployMode=client* is in the default file.

We can override this using

spark-submit --deploy-mode "cluster"

Spark Session : spark-submit

--driver-memory : Amount of memory to allocate for a driver (Default: 1024M).

--executor-memory : Amount of memory to use for the executor process.

--executor cores : Number of CPU cores to use for the executor process.

Spark Session : spark-submit

--jars: Dependency .jar files.

Ex : --jars /devl/src/main/python/lib/ojdbc7.jar, fil2.jar, file3.jar

--packages: Pass the dependency packages.

Ex : --packages org.apache.spark:spark-avro_2.11:2.4.4

--py-files: Use --py-files to add .py and .zip files. File specified with --py-files are uploaded to the cluster before it run the application.

Ex - --py-files file1.py, file2.py,file3.zip

Spark Session : spark-submit

```
spark-submit \
--master "yarn" \
--deploy-mode "client"\
--conf spark.sql.shuffle.partitions = 300 \
--conf spark.yarn.appMasterEnv.HDFS_PATH="practice/retail_db/orders"
--driver-memory 1024M \
--executor-memory 1024M \
--num-executors 2 \
--jars --jars /devl/src/main/python/lib/ojdbc7.jar, fil2.jar, file3.jar \
--packages org.apache.spark:spark-avro_2.11:2.4.4 \
--py-files file1.py, file2.py,file3.zip \
/dev/example1/src/main/python/bin/basic.py arg1 arg2 arg3
```

What is RDD ?

Spark Data Structures

- 2 types of data structures in PySpark - RDD and Data Frames. We will see both in detail as we proceed further.
- RDD is the low-level data structure which spark uses to distribute the data between tasks while data is being processed.
- DataFrame is created on top of RDDs We can seamlessly move between RDD and DataFrame.

RDD – Resilient Distributed Datasets

RDD is one of the fundamental abstractions of Spark on which it was created initially. Almost everything in Spark is built on top of RDDs.

Resilient – “able to withstand or recover quickly from difficult conditions”.

- RDDs are immutable and can not be modified once created.
- Fault Tolerance – RDDs track data lineage information to recover lost data quickly and automatically **on failure** at any point of execution cycle.

Distributed – Divided into smaller chunks called Partitions and distributed across multiple nodes across the cluster.

Datasets – Holds data.

*****Correction - I said 100MB. It is 200MB File would be divided into 2 partitions – 128MB and 72MB.**

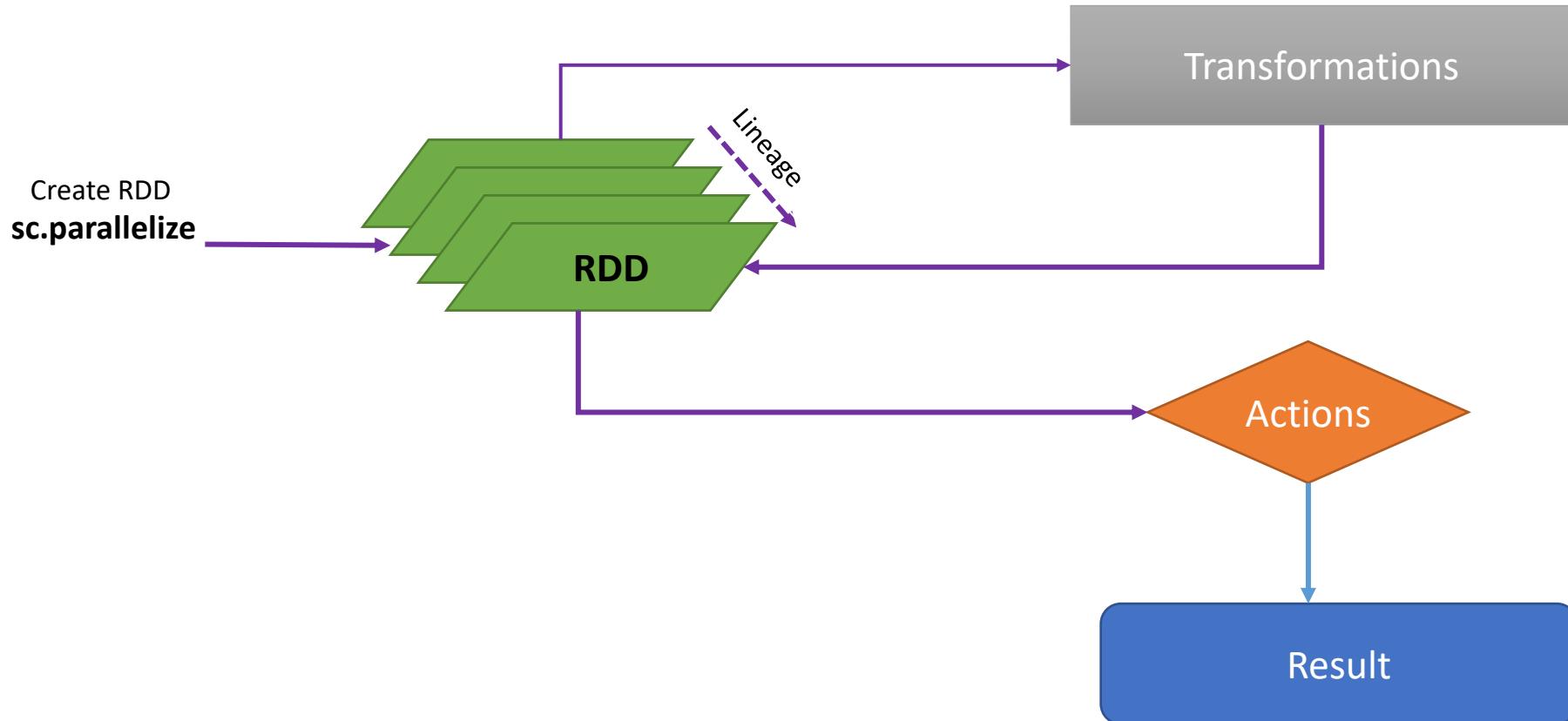
RDD Main Features

1. Resilience
2. Distributed
3. Lazy Evaluation
4. Immutability
5. In-memory Computation
6. Structured or semi-structured Data

RDD Main Features

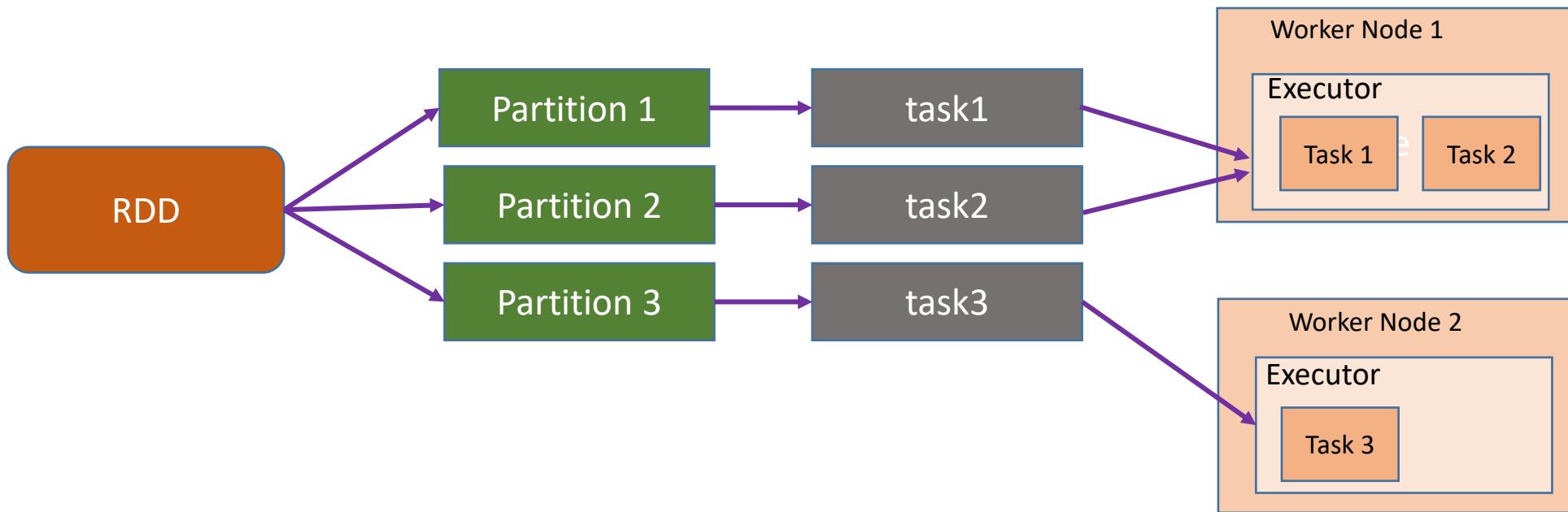
1. Resilience

- Fault Tolerance – RDDs track data lineage information to recover lost data quickly and automatically on failure at any point of execution cycle.
- Spark keeps a record of lineage while tracking the transformations that have been performed. If any part of RDD is lost, then spark will utilize this lineage record to quickly and efficiently re-compute the RDD using the identical operations.



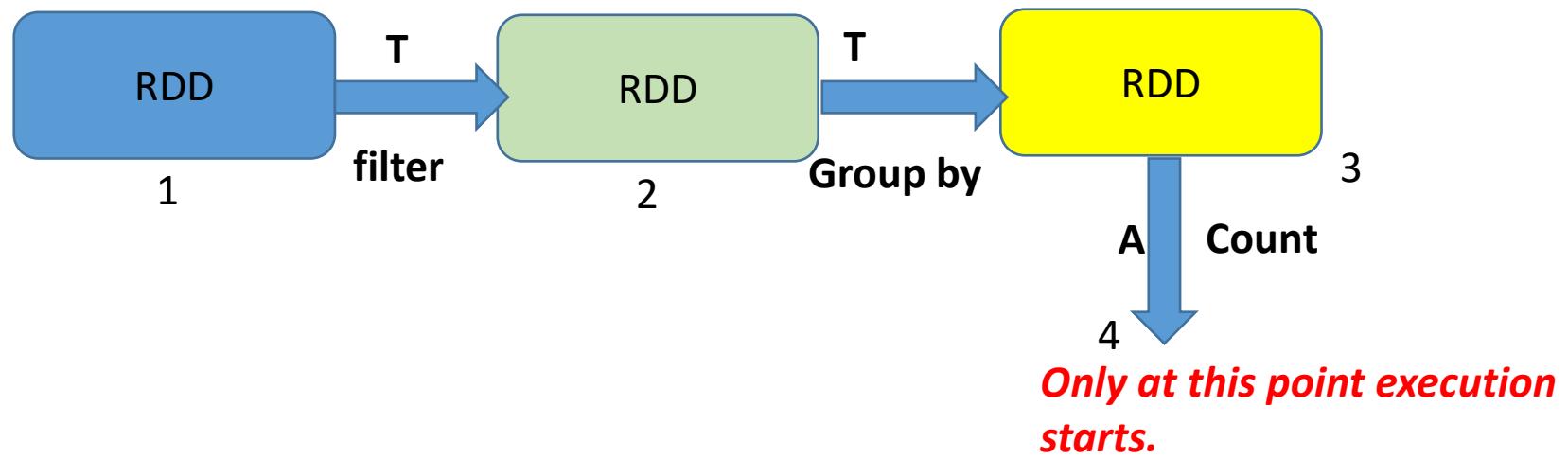
2. Distributed

- RDD will be divided into partitions while data being processed. Each partition will be processed by one task.
- If spark is running in YARN cluster, the number of RDD partitions is typically based on HDFS block size which is 128MB by default. We can control the number of minimum partitions by using additional arguments while invoking APIs such as textFile.



3. Lazy Evaluation:

- Each Transformation is a Lazy Operation. Evaluation is not started until an action is triggered.



Example – Find out 10 sample records having a string “Robert” from a file(1TB).

With Out Lazy Evaluation:

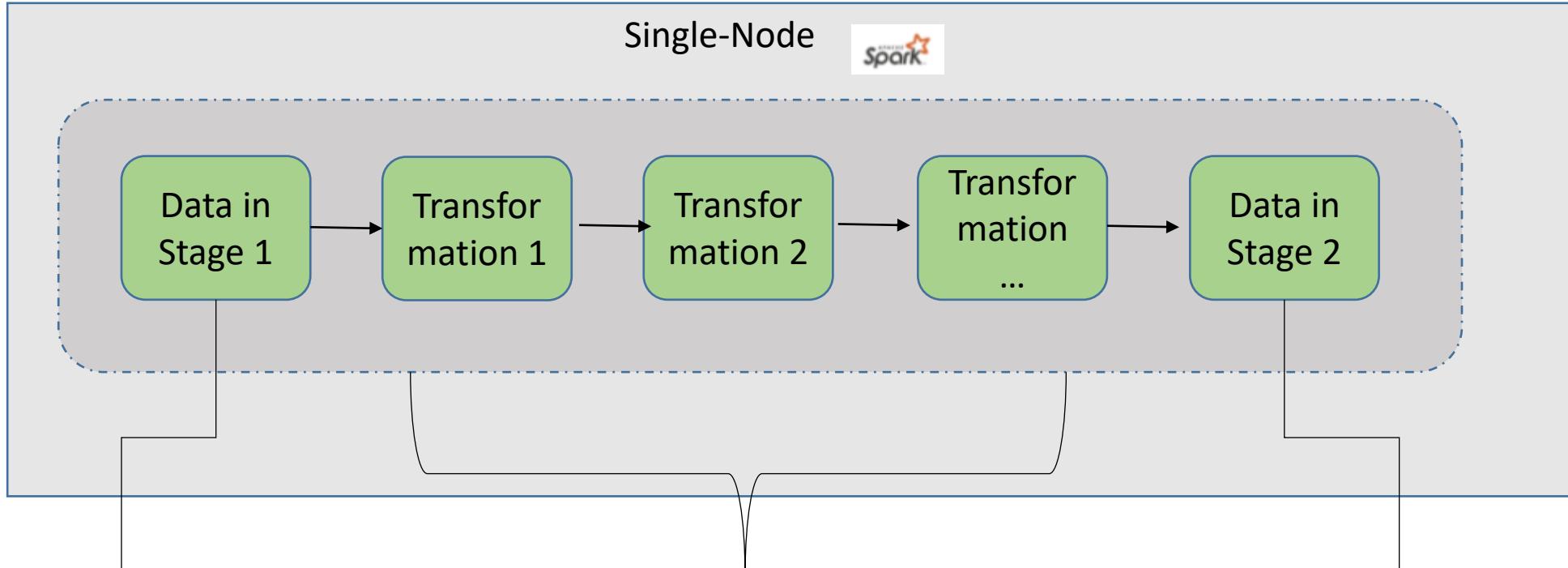
- Step 1 Load the 1TB File
- Step 2 Perform full scan to find out all Records having “Robert”
- Step 3 Retrieve 10 Records

With Lazy Evaluation:

- Step 1 Wait for an Action and then Load the 1TB File
- Step 2 Filter out first 10 records having “Robert” .

4. Immutability:

RDDs are considered to be “Immutable Storage”.



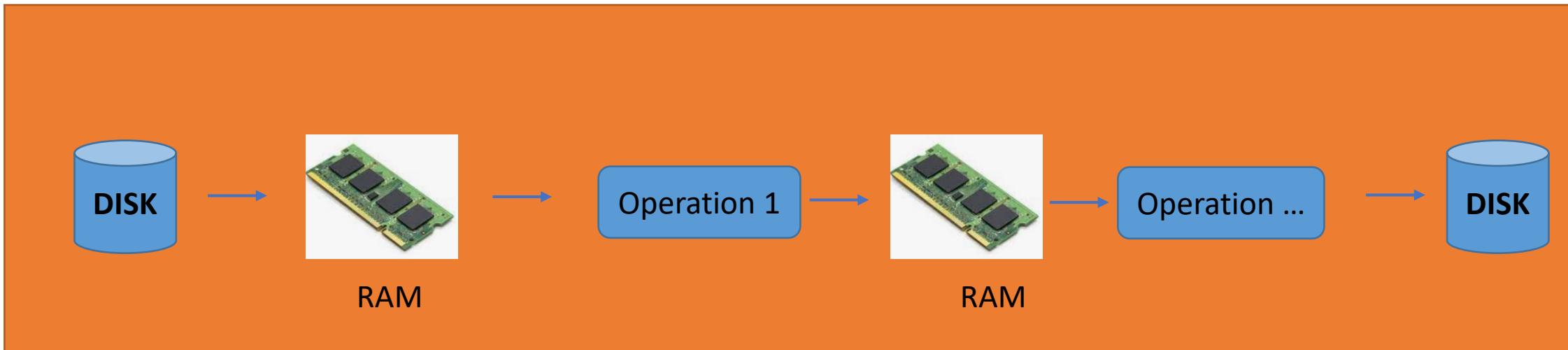
Data Stored
Immutably

The Transformation Recipe is Stored
*Ex – Logic to print all the even
numbers is col%2 == 0*

Data is not
Stored

5. In-memory Computation:

- The data is kept in *RAM* instead of disk drives and is processed in parallel.
- This has become popular because it reduces the cost of memory.
- The two main columns of in-memory computation are-RAM storage and Parallel distributed processing.



6. Structured or Semi-structured Data:

Along with structured data, RDD can also be used for semi structured data like text, media streaming data etc.

When to Use RDDs?

- Low Level API & control of dataset
- Dealing with unstructured data(text or media streams)
- Don't care schema or structure of data
- Don't care about optimization, performance

Problems in RDD ?

RDD Properties

RDD is defined by five main properties:

1. List of Parent RDDs (Dependencies)
2. An array of partitions that a dataset is divided into.
3. A Compute function to do a computation on partitions.
4. Optional practitioner that defines how keys are hashed and the pairs partitioned (key value RDDs).
5. Optional Preferred locations – Information about the locations of the split block for an HDFS file (if on YARN).

Main Problems in RDD ?

Problem#1 Can not Optimize !!!

Compute function:

Partition => Iterator(T)

Opaque Computation
&
Opaque data

Main Problems in RDD ?

Problem#2 (Pretty verbose to work with)

```
emp.map(lambda x: (x.deptid, (x.age, 1))) \  
    .reduceByKey(lambda x,y: (x(0) + y(0), x(1) + y(1))) \  
    .map(lambda x: (x(0), x(1)(0) / x(1)(1))) \  
    .collect()
```

```
SELECT deptid, AVG(age) FROM emp GROUP BY deptid
```

Create RDD

Ways to create RDD :

- ✓ External Data (HDFS, local etc)
- ✓ Local Data
- ✓ Python List/Parallelized Collections
- ✓ Other RDDs
- ✓ Existing DataFrame

Create RDD using `textFile`:

Using Spark Context Object `sc`.

```
ordRDD = sc.textFile('practice/retail_db/orders',8)
```

- By default `minPartitions` is the file block numbers in HDFS. We can give a different number.
- But if you use a number less than number of blocks, that will not be used and number of blocks tasks will be executed.

Create RDD using `wholeTextFiles`:

Read entire file as a single record into RDD.

```
ordRDD = sc.wholeTextFiles('practice/retail_db/orders',8)
```

Create Empty RDD:

Create empty RDD with no partition.

```
rdd = spark.sparkContext.emptyRDD()
```

Python List:

```
lst = open('/staging/test/sample.txt').read().splitlines()
rdd = sc.parallelize(lst)
```

Other RDDs

```
rdd1 = rdd
Rdd1.rdd.map(lambda x: x(1))
```

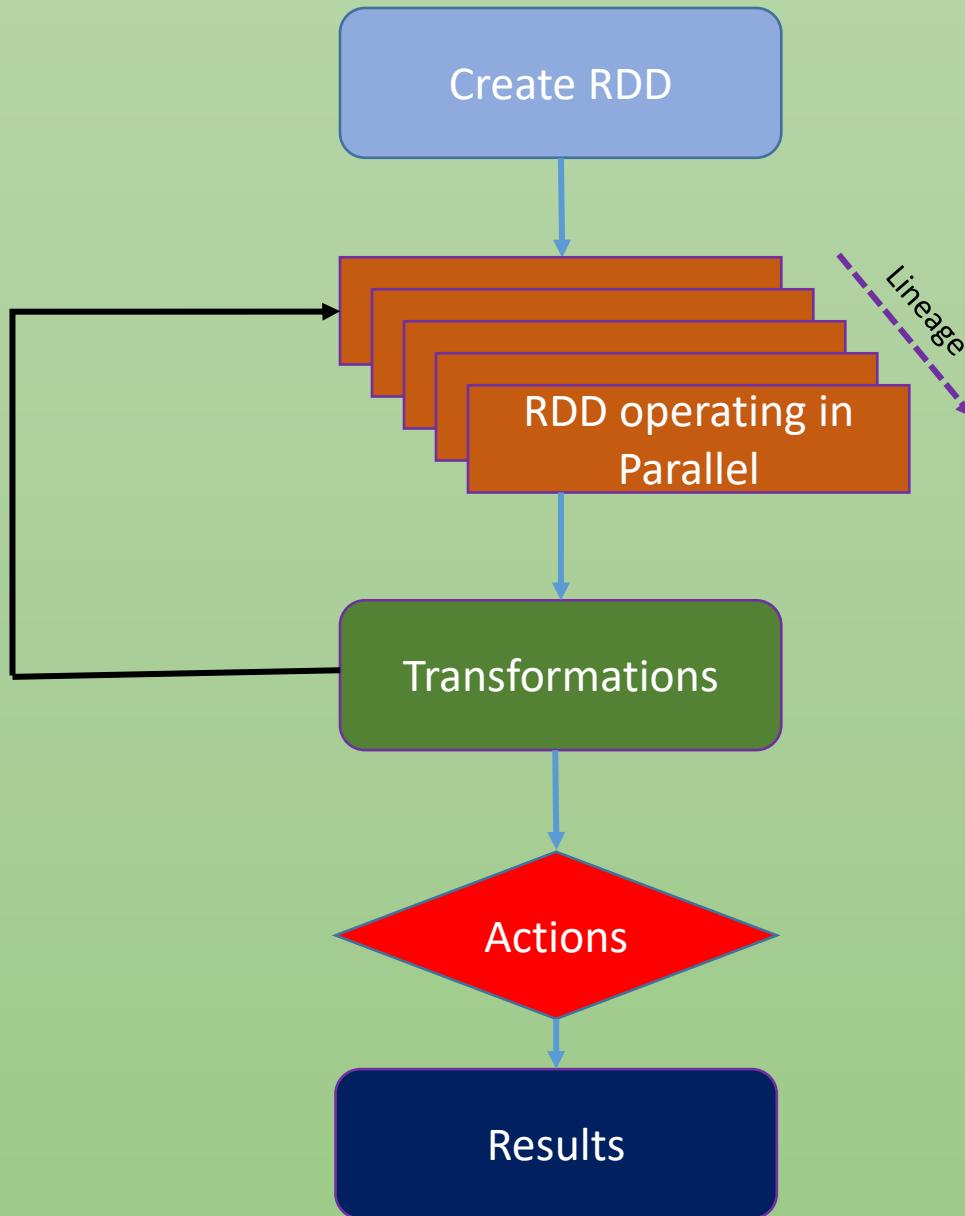
Existing DataFrame

```
df=spark.createDataFrame(data=([('robert',35),('Mike',45)]),schema=('name','age'))
new_rdd= df.rdd
```

Create RDD from Local File:

```
rdd = sc.textFile("file://<local_path>")
Ex - Local Path: /staging/test
```

RDD Operations



<https://spark.apache.org/docs/latest/rdd-programming-guide.html>

| Transformations | | | | | | | |
|------------------|-----------|----------------|-----------|--------------|----------|------|------------------------------------|
| Row Level | Joining | Key Agg | Sorting | Set | Sampling | Pipe | Partitions |
| map | join | reduceByKey | sortByKey | union | sample | pipe | Coalesce |
| flatMap | cogroup | aggregateByKey | | intersection | | | Repartition |
| filter | cartesian | groupByKey | | distinct | | | repartitionAndSortWithinPartitions |
| <i>mapValues</i> | | countByKey | | subtract | | | |

| Actions | | | |
|-------------|-----------|--------------------|---------|
| Display | Total Agg | File Extraction | foreach |
| take | reduce | saveAsTextFile | foreach |
| takeSample | count | saveAsSequenceFile | |
| takeOrdered | | saveAsObjectFile | |
| first | | | |
| collect | | | |

Low Level Transformations (map, flatMap, filter)

map : map(f, preservesPartitioning=False)

- Perform row level transformations where one record transforms into another record.
- Number of records in input is equal to output.
- Return a new RDD by applying a function to each element of this RDD.
- When we apply a map function to an RDD, a pipelineRDD is formed, a subclass of RDD. It has all the APIs defined in the RDD.

```
ord = sc.textFile('practice/retail_db/orders')
ordItems = sc.textFile('practice/retail_db/order_items')
```

PS: Project all the Order_ids.

```
ordMap = ord.map(lambda x : x.split(',') (0))  
for i in ordMap.take(5) : print(i)
```

PS: Project all the Orders and their status.

```
ord.map(lambda x : (x.split(',') (0),x.split(',') (1))).take(5)
```

PS: Combine Order id and status with '#'

```
ord.map(lambda x : x.split(',') (0) + '#' + x.split(',') (3)).take(5)
```

PS: Convert the Order date into YYYY/MM/DD Format.

```
ord.map(lambda x : x.split(',') (1).split(' ') (0).replace('-', '/')).first()
```

PS: Create key-value pairs with key as Order id and values as whole records.

```
ordMap = ord.map(lambda x : (x.split(',') (0),x))
```

PS: Project all the Order_item_ids and their subtotal.

```
ordItemsMap = ordItems.map(lambda x : (x.split(',') (0),x.split(',') (4)))
```

PS: Applied user defined function to convert status into lowercase.

```
def lowerCase(str):  
    return str.lower()
```

```
ord.map(lambda x : lowerCase(x.split(',') (3))).first()
```

flatMap : flatMap(f, preservesPartitioning=False)

- Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
- Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). Number of records in input is less than or equal to output.

PS : Word count in orders file.

```
ord = sc.textFile('practice/retail_db/orders')
wordCount = ord.flatMap(lambda x : x.split(',')).map(lambda w : (w,1)).reduceByKey(lambda x,y : x+y)
```

filter: filter(f)

Return a new dataset formed by selecting those elements of the source on which *func* returns true.

PS: Print all the orders which are closed or Complete and ordered in the year 2013.

```
ord = sc.textFile('practice/retail_db/orders')
filteredOrd = ord.filter(lambda x : (x.split(',')[3] in ("CLOSED","COMPLETE")) and (x.split(',')[1].split('-')[0] == '2014'))
```

mapValues: mapValues(f)

- Only applicable to pair RDDs RDD((A,B)).
- Do not change the key. Apply the function ‘f’ to all values of the same key.
- The difference with map is map operates on the entire record.

Ex - rdd = sc.parallelize(((“a”, (1,2,3)), (“b”, (3,4,5)), (“a”, (1,2,3,4,5))))

```
def f(x): return len(x)  
rdd.mapValues(f).collect()
```

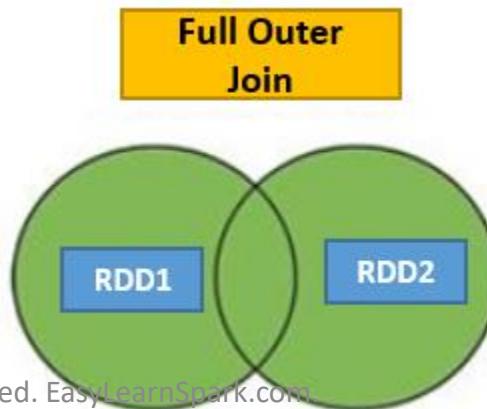
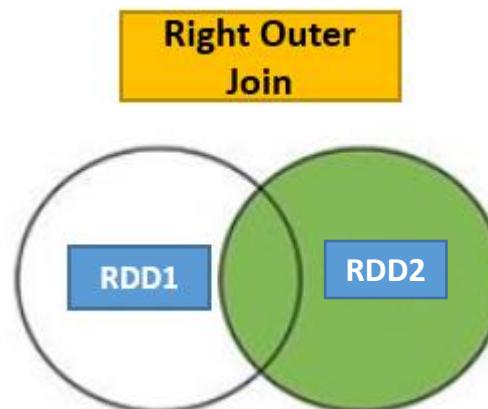
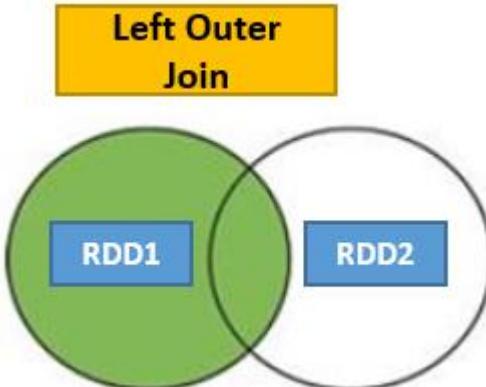
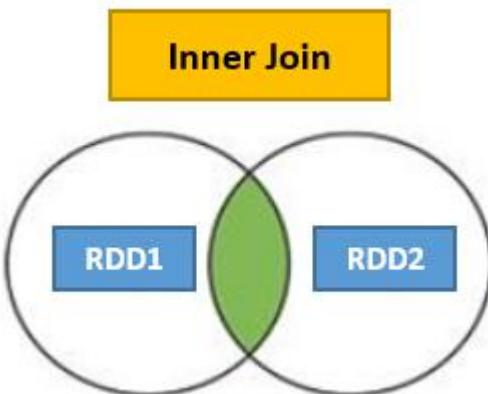
Join Transformations

Joining datasets

- Join
- leftOuterJoin
- rightOuterJoin
- fullOuterJoin

Join: `join(other, numPartitions=None)`

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.



```
ord = sc.textFile('practice/retail_db/orders')
ordItems = sc.textFile('practice/retail_db/order_items')
```

PS: Find the subtotal for each ORDER_CUSTOMER_ID.

```
ordMap=ord.map(lambda x : (x.split(',')[0],x.split(',')[2]))
ordItemsMap=ordItems.map(lambda x : (x.split(',')[1],x.split(',')[4]))
findSubtotalForCust = ordMap.join(ordItemsMap)
findSubtotalForCust.map(lambda x : x(1)(0)+','+x(1)(1)).first()
findSubtotalForCust.map(lambda x : str(x(1)(0))+','+str(x(1)(1))).first()
```

cogroup:

cogroup(other, numPartitions=None):

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples.

Ex –

```
x = sc.parallelize((("a", 1), ("b", 4)))
y = sc.parallelize((("a", 2)))
xy = x.cogroup(y)
for i,j in list(xy.take(5)) : print(i + ' ' + str(map(list,j)))
```

cartesian:

cartesian(other) : Perform a cross join.

Ex -

```
rdd = sc.parallelize((1,3,2))  
sorted(rdd.cartesian(rdd).collect())
```

Aggregation Operations (Total)

Aggregation Operations:

There are several APIs to perform aggregation Operations.

- Total aggregations – **reduce, count (Actions)**
- By Key aggregations – **reduceByKey, aggregateByKey, groupByKey, countByKey (Transformations)**

Total aggregations:

reduce(f): Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

count(): Return the number of elements in this RDD.

Ex –

```
ord = sc.textFile('practice/retail_db/orders')
ordItems = sc.textFile('practice/retail_db/order_items')
```

Count the number of orders which are closed.

```
ord.filter(lambda x : x.split(',')[3]).count()
```

Find the total quantity sold for Order ID 1-10.

```
from operator import add
ordItems.filter(lambda x : int(x.split(',')[1]) < 11).map(lambda x : float(x.split(',')[4])).reduce(lambda x,y : x+y)
ordItems.filter(lambda x : int(x.split(',')[1]) < 11).map(lambda x : float(x.split(',')[4])).reduce(add)
```

For a given order 10 find the maximum subtotal out of all orders.

```
ordItems.filter(lambda x : int(x.split(',')[1]) == 10).map(lambda x : x.split(',')[4]).reduce(lambda a,b : a if (float(a.split(',')[0]) > float(b.split(',')[0])) else b)
ordItems.filter(lambda x : int(x.split(',')[1]) == 10).map(lambda x : x.split(',')[4]).reduce(max)
```

Shuffling and Combiner

Shuffling and Combiner ?

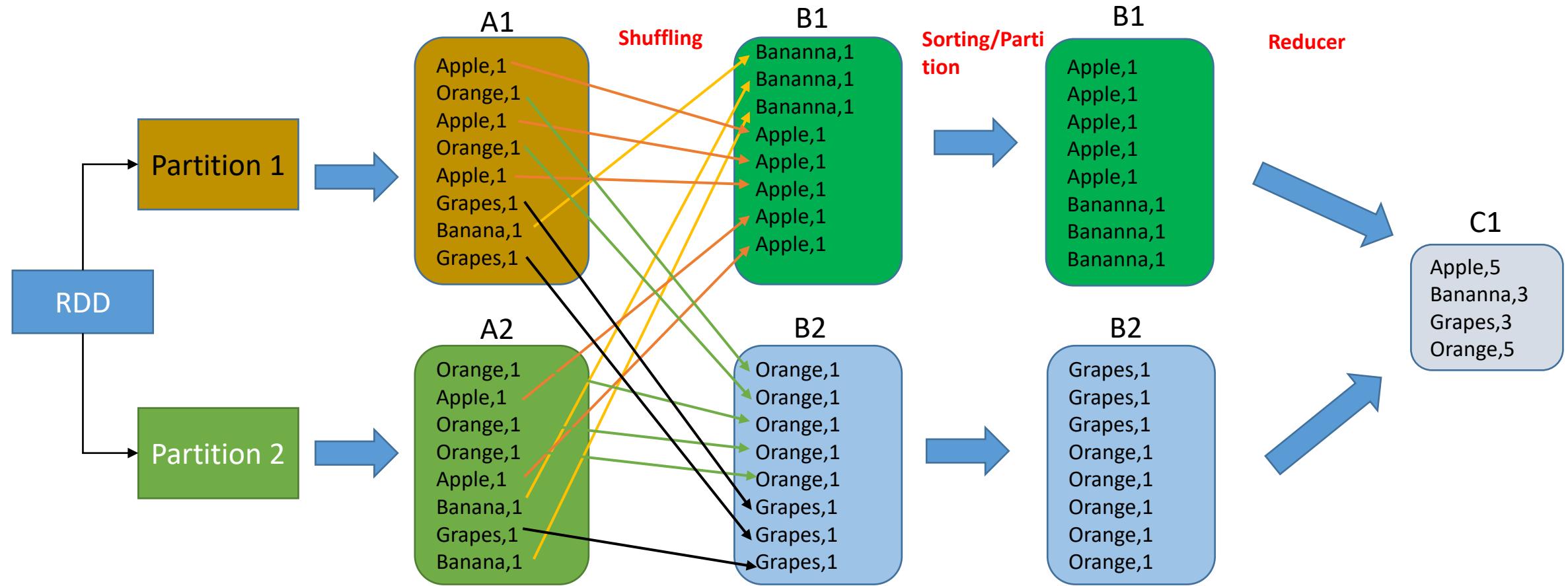
Shuffle:

- Shuffling is a process of redistributing data across partitions or even nodes.
- Shuffle operation would create a new stage.
- Based on data size we may reduce or increase the number of partitions using configuration `spark.sql.shuffle.partitions` or through codes like `repartition` and `coalesce`.
- Costly operation as it involves disk I/O, Network I/O and data serialization/de-Serilization.
- Spark shuffling triggers for transformation operations like `groupByKey()`, `reduceByKey()`, `join()`, `union()`, `cogroup`, `groupByKey()` etc.
- `distinct` creates a shuffle.
- `Count` and `countByKey` does not create any shuffle.
- *Avoid shuffling at all cost. If Shuffling is absolutely necessary, use combiner.*
- *Out of 3 main key aggregation APIs, the `groupByKey` does not use a combiner and so should be avoided. The `reduceByKey` and `aggregateByKey` use the combiner and should be preferred.*

Combiner:

- It computes the intermediate values for each partition to avoid partial shuffling.

Shuffling with out Combiner



What Happens in Shuffle Stage?

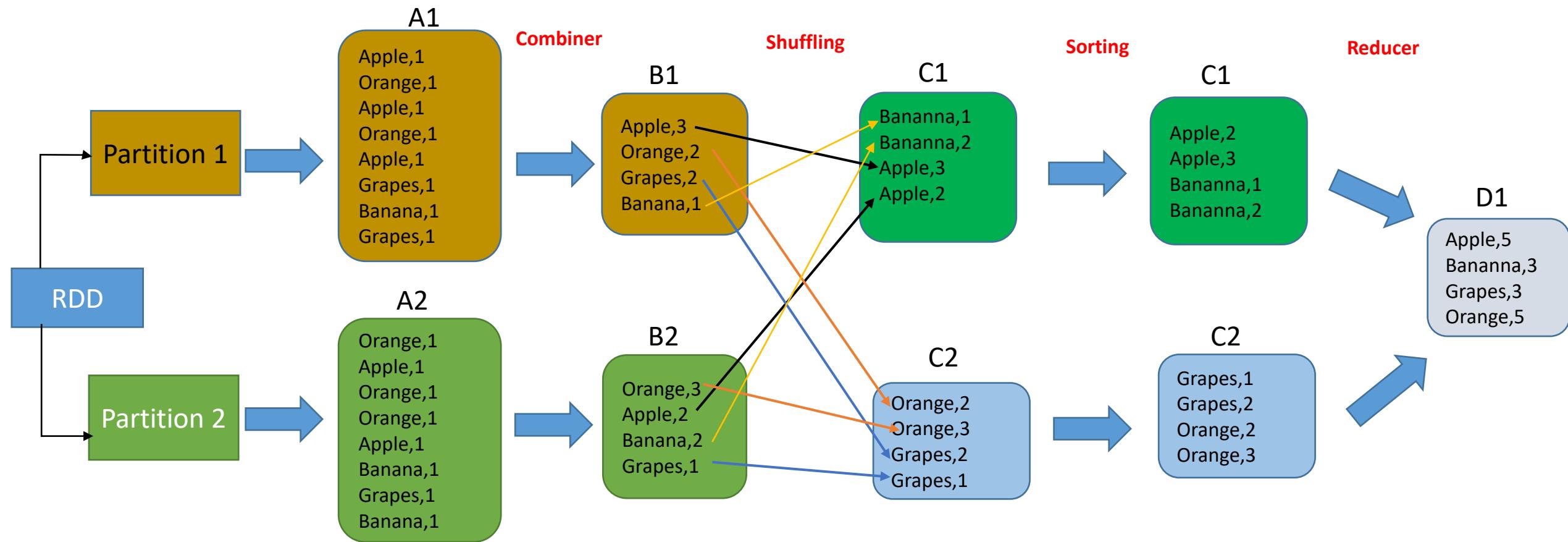
In shuffle Stage,

1. Spark runs the map tasks on all partitions which groups all values for a single key.
2. Then the results of the map tasks are kept in memory.
3. When results do not fit into memory, Spark stores the data into a disk. So there is I/O and data serialization involved.
4. So Shuffle generates a large number of intermediate files on disk. These files are preserved until the corresponding RDDs are no longer used and are then garbage collected.

We can also mention a temporary storage directory in the spark.local.dir while configuring the sparkcontext.

5. Finally it runs the reduce tasks on each partition based on key.

Shuffling with Combiner



If you have a performance issues, look into the code if there is a shuffle happening.

toDebugString returns “A description of this RDD and its recursive dependencies for debugging.” It includes possible shuffles.

```
>>> a = sc.parallelize([1,2,3]).distinct()
>>> print(a.toDebugString())
(2) PythonRDD[27] at RDD at PythonRDD.scala:53 []
|  MapPartitionsRDD[26] at mapPartitions at PythonRDD.scala:133 []
|  ShuffledRDD[25] at partitionBy at NativeMethodAccessorImpl.java:0 []
+- (2) PairwiseRDD[24] at distinct at <stdin>:1 []
   |  PythonRDD[23] at distinct at <stdin>:1 []
   |  ParallelCollectionRDD[22] at parallelize at PythonRDD.scala:195 []
```

Use `explain()` for DataFrames.

Aggregation Operations (Key)

Key Aggregations:

groupByKey():

aggregateByKey():

reduceByKey():

countByKey():

groupByKey(numPartitions=None, partitionFunc)

- Can be used for aggregations but should be given low priority as it does not use the combiner.
- When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- If grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
- The number of reduce tasks is configurable through an optional argument – numPartitions.

Ex –

```
ordItems = sc.textFile('practice/retail_db/order_items')
```

For each product, find its aggregated revenue.

```
ordGrp = ordItems.map(lambda x : (int(x.split(',')[2]),float(x.split(',')[4]))).groupByKey()
```

```
result = ordGrp.mapValues(sum).collect()
```

OR

```
result = ordGrp.map(lambda x : (x[0],sum(x[1]))).collect()
```

| ORDER_ITEMS |
|---------------------|
| order_item_id |
| order_item_order_id |
| order_item_prod_id |
| order_item_quantity |
| order_item_revenue |
| order_item_price |

reduceByKey(func, numPartitions=None, partitionFunc)

- When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
- Like in groupByKey, the number of reduce tasks is configurable through an optional argument - numPartitions.
- It uses Combiner. Associative reduction.

Ex-

```
from operator import add
rdd = sc.parallelize((("a", 1), ("b", 1), ("a", 1)))
sorted(rdd.reduceByKey(add).collect())
```

Find the total revenue sold for each order.

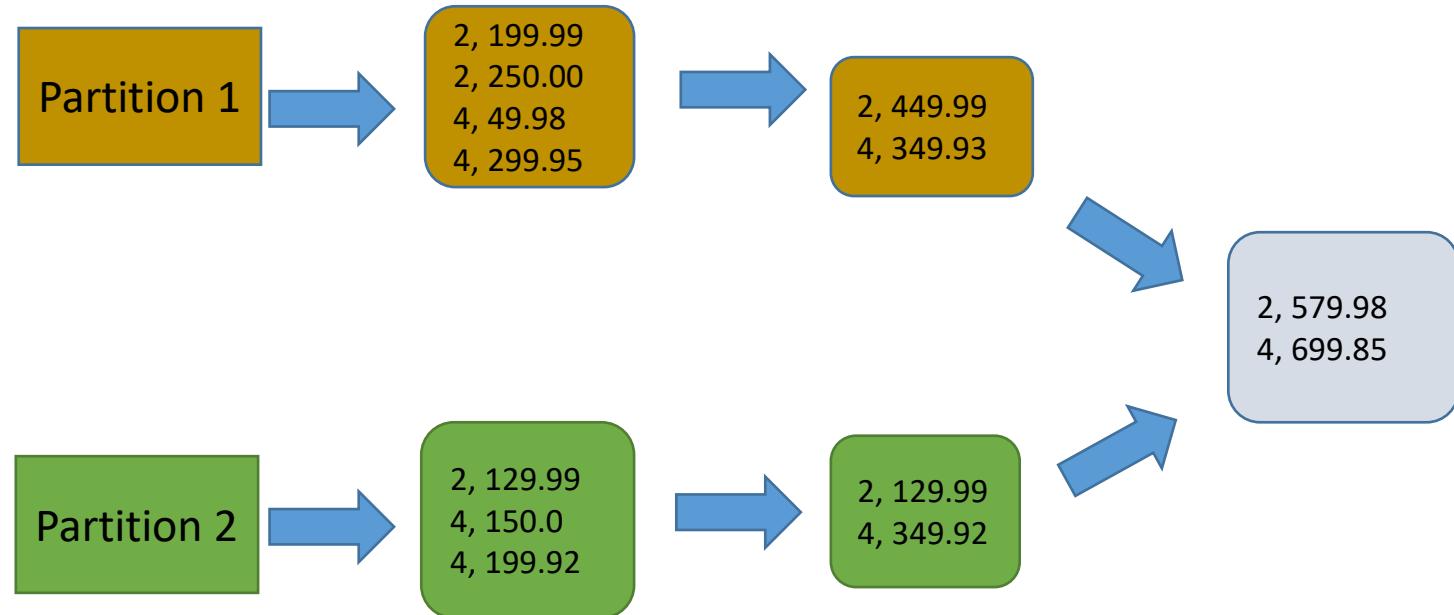
```
ordItems = sc.textFile('practice/retail_db/order_items')
ordItems.map(lambda x : (int(x.split(',')[1]),float(x.split(',')[4]))).reduceByKey(lambda x,y :
OR
ordItems.map(lambda x : (int(x.split(',')[1]),float(x.split(',')[4]))).reduceByKey(add).collect()
```

Find the maximum revenue for each order.

```
ordItems.map(lambda x : (int(x.split(',')[1]),x)).reduceByKey(lambda a,b : a if (float(a.split(',')[4]) > float(b.split(',')[4])) else b).collect()
```

| ORDER_ITEMS |
|---------------------|
| order_item_id |
| order_item_order_id |
| order_item_prod_id |
| order_item_quantity |
| order_item_revenue |
| order_item_price |

Associative Reduction:



aggregateByKey(zeroValue, seqOp, combOp, (numPartitions)):

- First aggregate elements in each partition and then aggregating results of all partition to get the final result and the result could be any type than the type of your RDD.
- 3 mandatory arguments:
 - ✓ Zero Value: Initial value to initialize the accumulator. Use 0 for integer and NULL for collections.
 - ✓ SeqOp: Function used to accumulate the results of each partition, and stores the running accumulated result to U. $(U,T) \Rightarrow U$.
 - ✓ CombOp: Function is used to combine results of all partitions U.

Partition 1

2 → ("Joseph", 200)
2 → ("Jimmy", 250)
4 → ("Joseph", 150)
4 → ("Ram", 200)
7 → ("Joseph", 300)

SeqOp : Max Revenue & Count

(2, 250, 2)
(4, 200, 2)
(7, 300, 1)

Partition 2

2 → ("Tina", 130)
4 → ("Jimmy", 50)
4 → ("Tina", 300)
7 → ("Tina", 200)
7 → ("Jimmy", 80)

SeqOp : Max Revenue & Count

(2, 130, 1)
(4, 300, 2)
(7, 200, 2)

CombOp : Max Revenue & Count

(2, 250, 3)
(4, 300, 4)
(7, 300, 3)

Ex – 1

Find the maximum revenue for each Order.

```
ordItems=sc.parallelize([
(2,"Joseph",200), (2,"Jimmy",250), (2,"Tina",130), (4,"Jimmy",50), (4,"Tina",300),
(4,"Joseph",150), (4,"Ram",200), (7,"Tina",200), (7,"Joseph",300), (7,"Jimmy",80)],2)
```

#Create a Paired RDD

```
ordPair = ordItems.map(lambda x : (x[0],(x[1],x[2])))
```

#Initialize Accumulator

Zero Value: Zero value in our case will be 0 as we are finding Maximum Marks

```
zero_val=0
```

#Define Sequence Operation

Sequence operation : Finding Maximum revenue from each partition

```
def seq_op(accumulator, element):
```

```
    if(accumulator > element[1]):
```

```
        return accumulator
```

```
    else:
```

```
        return element[1]
```

#Define Combiner Operation

#Combiner Operation : Finding Maximum revenue from all partitions

```
def comb_op(accumulator1, accumulator2):
```

```
    if(accumulator1 > accumulator2):
```

```
        return accumulator1
```

```
    else:
```

```
        return accumulator2
```

```
aggr_ordItems = ordPair.aggregateByKey(zero_val, seq_op, comb_op)
```

```
for i in aggr_ordItems.collect(): print(i)
```

Ex – 2

Find the maximum revenue for each Order. Print customer name.

```
ordItems=sc.parallelize([
(2,"Joseph",200), (2,"Jimmy",250), (2,"Tina",130), (4,"Jimmy",50), (4,"Tina",300),
(4,"Joseph",150), (4,"Ram",200), (7,"Tina",200), (7,"Joseph",300), (7,"Jimmy",80)],2)
```

```
#Create a Paired RDD
```

```
ordPair = ordItems.map(lambda x : (x[0],(x[1],x[2])))
```

```
#Initialize Accumulator
```

```
# Zero Value: Zero value in our case will be 0 as we are finding Maximum Marks
```

```
zero_val=(',',0)
```

```
#Define Sequence Operation
```

```
# Sequence operation : Finding Maximum revenue from each partition
```

```
def seq_op(accumulator, element):
```

```
    if(accumulator[1] > element[1]):
```

```
        return accumulator
```

```
    else:
```

```
        return element
```

```
#Define Combiner Operation
```

```
#Combiner Operation : Finding Maximum revenue from all partitions
```

```
def comb_op(accumulator1, accumulator2):
```

```
    if(accumulator1[1] > accumulator2[1]):
```

```
        return accumulator1
```

```
    else:
```

```
        return accumulator2
```

```
aggr_ordItems = ordPair.aggregateByKey(zero_val, seq_op, comb_op)
```

```
for i in aggr_ordItems.collect(): print(i)
```

Ex – 3

Sum up all revenue and number of records for each order.

```
ordItems=sc.parallelize([
(2,"Joseph",200), (2,"Jimmy",250), (2,"Tina",130), (4,"Jimmy",50), (4,"Tina",300),
(4,"Joseph",150), (4,"Ram",200), (7,"Tina",200), (7,"Joseph",300), (7,"Jimmy",80)],2)
```

```
#Create a Paired RDD
```

```
ordPair = ordItems.map(lambda x : (x[0],(x[1],x[2])))
```

```
#Initialize Accumulator
```

```
# Zero Value: Zero value in our case will be 0 as we are finding Maximum Marks
```

```
zero_val=(0,0)
```

```
#Define Sequence Operation
```

```
# Sequence operation : Sum up all revenue and number of records per partition.
```

```
def seq_op(accumulator, element):
```

```
    return (accumulator[0] + element[1], accumulator[1] + 1)
```

```
#Define Combiner Operation
```

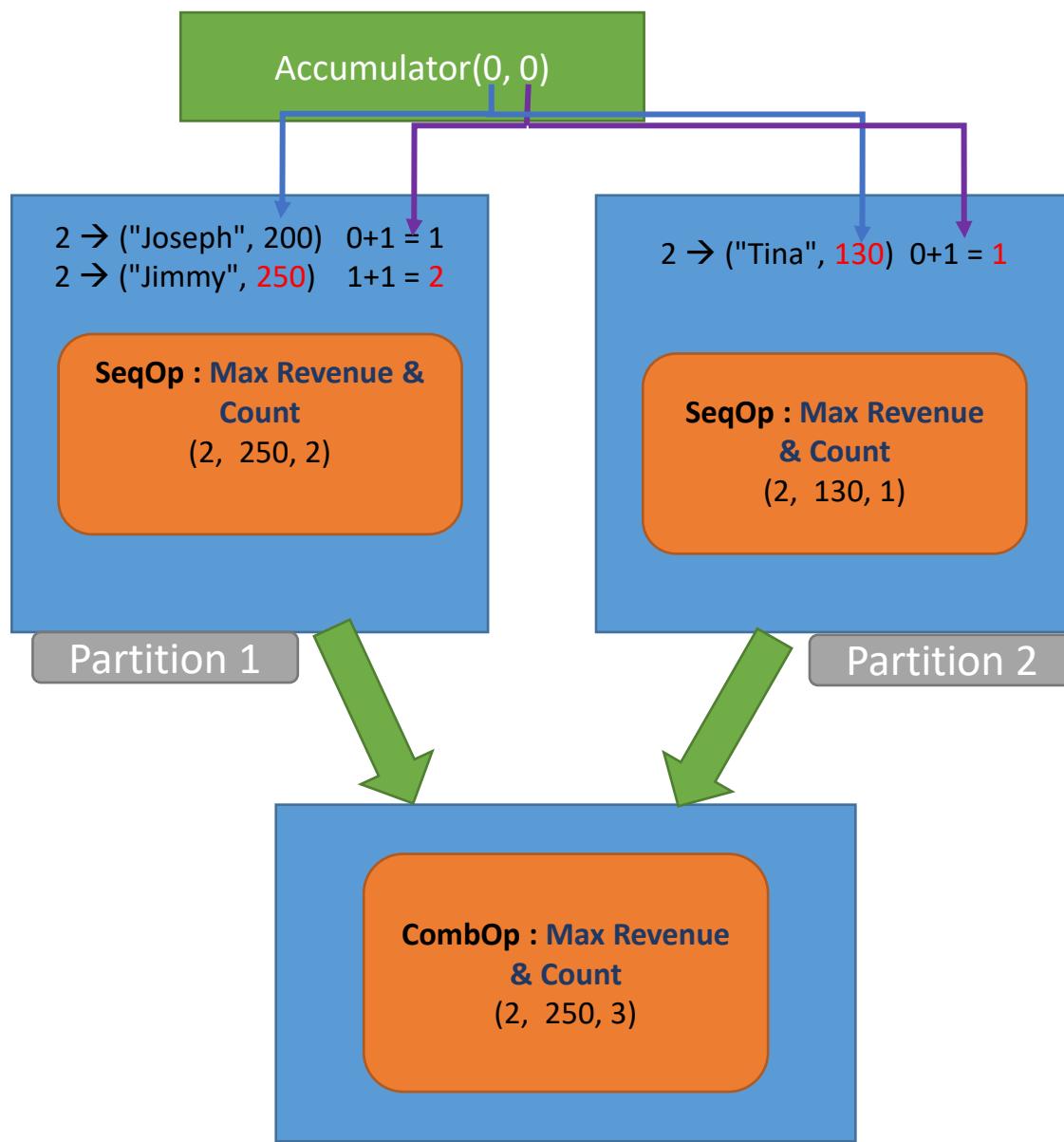
```
#Combiner Operation : Sum up all revenue and number of records for all partition.
```

```
def comb_op(accumulator1, accumulator2):
```

```
    return (accumulator1[0] + accumulator2[0], accumulator1[1] + accumulator2[1])
```

```
aggr_ordItems = ordPair.aggregateByKey(zero_val, seq_op, comb_op)
```

```
for i in aggr_ordItems.collect(): print(i)
```



```
def seq_op(accum, element):
    return (accum[0] + element[1], accum[1] + 1)

def comb_op(accum1, accum2):
    return (accum1[0] + accum2[0], accum1[1] + accum2[1])
```

Annotations for the code:

- Max Revenue** points to the addition of $accum[0]$ and $element[1]$ in the `seq_op` function.
- Count** points to the addition of $accum[1]$ and 1 in the `seq_op` function.

Difference Between ReduceByKey Vs AggregateByKey:

- aggregateByKey() is logically same as reduceByKey() but it lets you return result in different type.
- In another words, it lets you have an input as type x and aggregate result as type y.

Ex - For example (1,2),(1,4) as input and (1,"six") as output. It also takes zero-value that will be applied at the beginning of each key.

countByKey():

- Only available on RDDs of type (K, V). Returns a (K, Int) pairs with the count of each key.
- Returns a Collection Dictionary.
- No shuffle.

Ex - Count number oF orders per each status.

```
ord = sc.textFile('practice/retail_db/orders')
ordPair = ord.map(lambda x : (x.split(',')[3],1))
countByStatus = ordPair.countByKey()
for i in countByStatus.items : print(i)
for i in countByStatus.keys : print(i)
for i in countByStatus.keys : print(i)
```

| Orders |
|-------------------|
| Order Id |
| Order Date |
| Order Customer Id |
| Order Status |

Sorting Transformations

sortByKey(ascending=True, numPartitions=None, keyfunc=<function <lambda>>)

When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.

```
ord = sc.textFile('practice/retail_db/orders')
```

Ex-1 (Sort orders using customer id.)

```
ordPair = ord.map(lambda x : (int(x.split(',')[2]),x))
ordSort = ordPair.sortByKey(ascending=False)
for i in ordSort.take(10) : print(i)
```

Ex-2 (Sort orders using customer and status.)

```
ordPair = ord.map(lambda x : ((int(x.split(',')[2]), x.split(',')[3]),x))
ordSort = ordPair.sortByKey(ascending=False)
for i in ordSort.take(10) : print(i)
```

Ranking

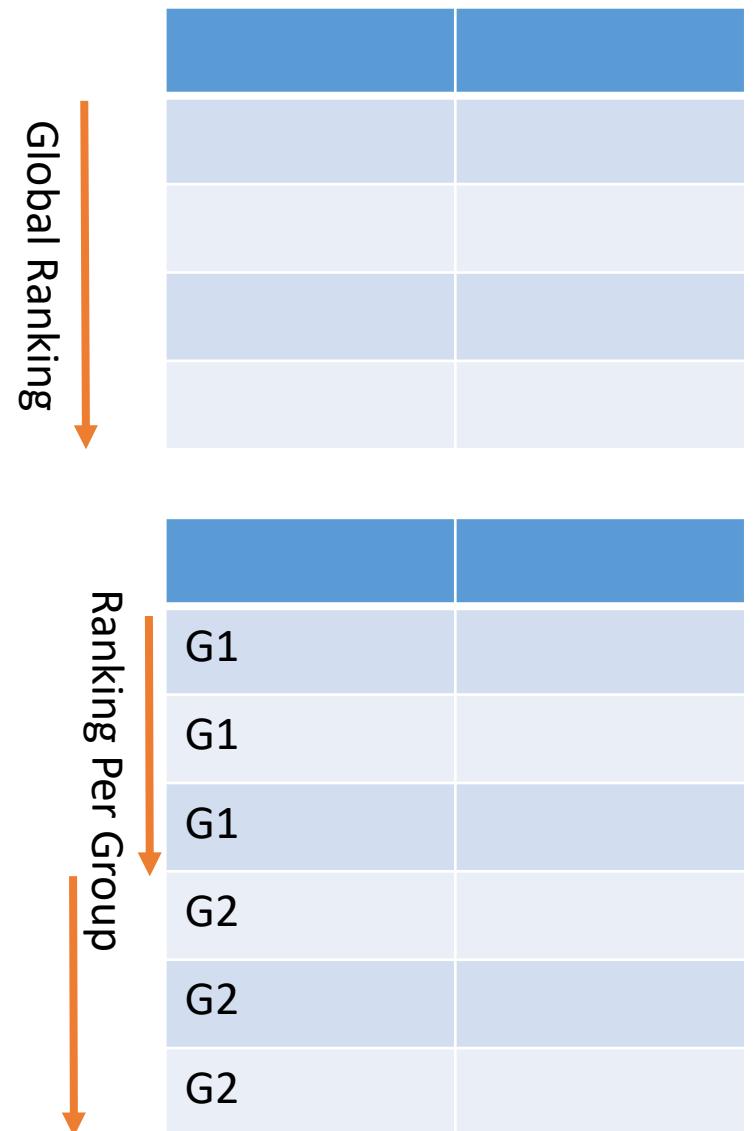
Global Ranking or Ranking per Group

Global Ranking :

- sortByKey and take
- takeOrdered or top

Ranking Per Group:

- Getting ranking per group is a bit complex but important to know.
- Per-key or Per group ranking can be achieved using
 - groupByKey with flatMap
 - Python Knowledge like sorted function, list etc

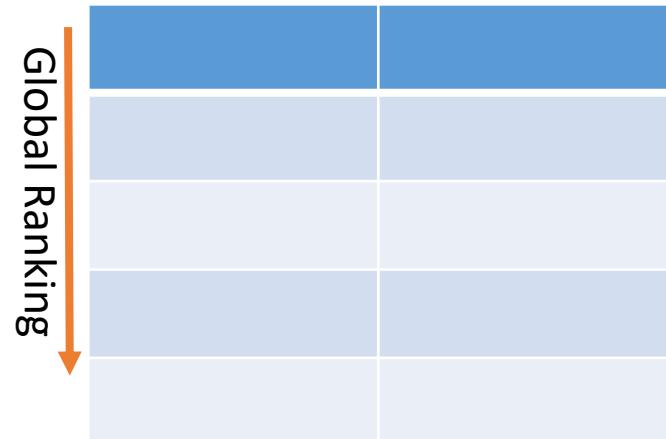


Global Ranking

Using sortByKey and take:

Ex – Top five products with highest prices.

```
prod = sc.textFile('practice/retail_db/products')  
prodPair = prod.map(lambda x : (float(x.split(',')[4]),x))  
prod = prod.filter(lambda x : x.split(',')[4] == "").count()  
prodPair = prod.map(lambda x : (float(x.split(',')[4]),x))  
top5Products = prodPair.sortByKey(False).take(5)
```



Using takeOrdered or top:

takeOrdered(num, key=None)

Get the N elements from an RDD ordered in ascending order or as specified by the optional key function.

Ex-

```
sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7]).takeOrdered(6)
```

Ex-

```
sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7]).takeOrdered(6, key=lambda x : -x)
```

Ex-

```
top5Products1 = prod.takeOrdered(5, lambda k: -float(k.split(',')[4]))
```

Ranking Per Group:

Ex –

Top 2 Products with highest Prices per Category.

```
prod = sc.textFile('practice/retail_db/products')

prodF = prod.filter(lambda x : (int(x.split(',')[1]) in [2,3,4]) and (int(x.split(',')[0]) in [1,2,3,4,5,25,26,27,28,29,49,50,51,52,53]))

prodGroupBy = prodF.map(lambda line : ( int(line.split(',')[1]), line)).groupByKey()

first = prodGroupBy.first()

sorted(first[1],key = lambda x : float(x.split(',')[4]),reverse=True)

top2ProductsByPrice= productsGroupBy.flatMap(lambda x: sorted(x[1], key=lambda k:float(k.split(",")[4]), reverse=True)[:2])
```



| | |
|----|--|
| | |
| G1 | |
| G1 | |
| G1 | |
| G2 | |
| G2 | |
| G2 | |

Set Transformations

union(other)

- A union will get all the elements from both the data sets.
- In the case of a union, it will not get distinct elements. Apply distinct, if you only want to get distinct elements after union operation.
- When we use set operations such as union and intersect, data should have a similar structure (Same Columns and Types).

Ex – Number of customers placed order in July or Aug Month.

```
ord = sc.textFile('practice/retail_db/orders')
```

```
julyOrd = ord.filter(lambda x : str(x.split(',')[1].split('-')[1]) == '07').map(lambda x : x.split(',')[2])
```

```
augOrd = ord.filter(lambda x : str(x.split(',')[1].split('-')[1]) == '08').map(lambda x : x.split(',')[2])
```

```
julyAugOrders = julyOrd.union(augOrd).distinct().count()
```

Intersection(other)

- Return the intersection of this RDD and another one.
- The output will not contain any duplicate elements, even if the input RDDs did.

Ex – Orders applied both in July and Aug datamonth.

```
julyAugCommonOrders=julyOrd.intersection(augOrd).count()
```

Ex – Check if duplicates are reported.

```
rdd1=sc.parallelize([1,2,3,3,3])  
rdd2=sc.parallelize([1,3,5])  
rdd1.intersection(rdd2).collect()
```

distinct(numPartitions=None)

- Return a new RDD containing the distinct elements in this RDD.

Ex-

```
rdd1.distinct().collect()
```

subtract (other, numPartitions=None)

- Return each value in left RDD that is not contained in right RDD.

Ex-

```
rdd2.subtract(rdd1).collect()
```

Sampling Transformations

```
sample(withReplacement, fraction, seed=None)
```



Transformations

- To get random sample records from the RDD.
- withReplacment: True or False. With True, Same result can be produced more than once.
- Fraction: Between 0 to 1. 0.3 means 30%. Does not guarantee the exact 30% of the records.
- Seed: Reproduce the same sample.

```
rdd = sc.parallelize(range(100), 4)
```

```
rdd.sample(seed=10,fraction=0.1,withReplacement=False).collect()
```

```
takeSample(withReplacement, num, seed=None)
```



Actions

- Return a fixed-size sampled subset of this RDD.
- 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.
- 'num' is exact.

```
rdd.takeSample(seed=10,num=10,withReplacement=True)
```

RDD Repartition & Coalesce

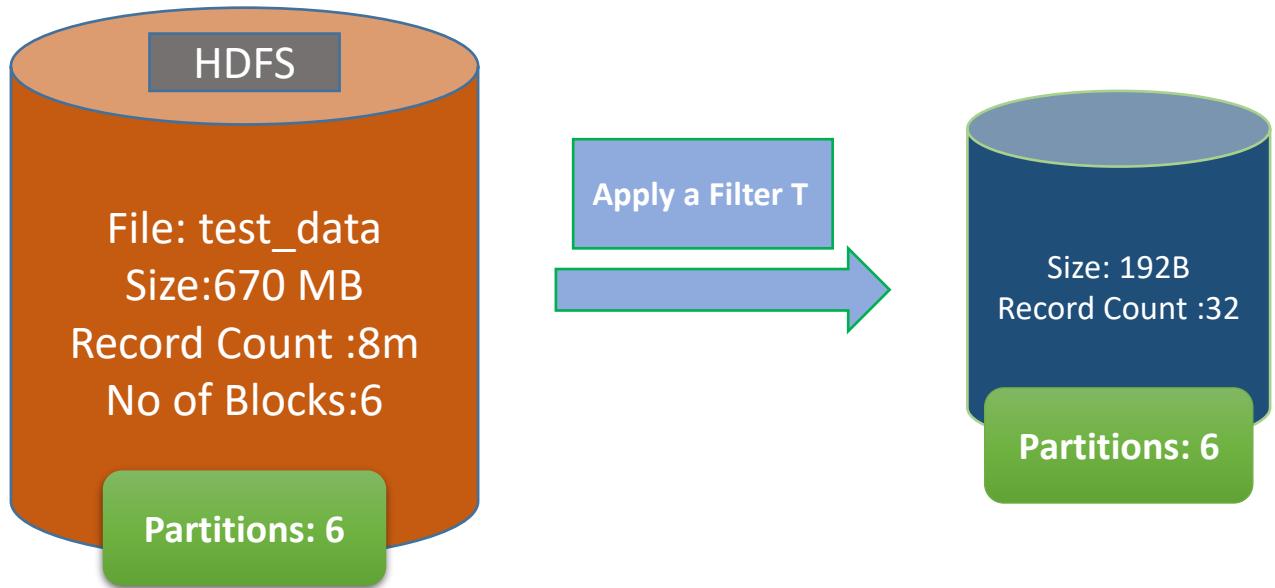
What is Partition ?

- Datasets are huge in size and they cannot fit into a single node and so they have to be partitioned across different nodes or machines.
- Partition in spark is basically an atomic chunk of data stored on a node in the cluster. They are the basic units of parallelism.
- One partition can not span over multiple machines.
- Spark automatically partitions RDDs/DataFrames and distributes the partitions across different nodes.
- We can also configure the optimal number of partitions. Having too few or many partitions is not good.
- How Spark does the default Partitioning of Data ?

Spark checks the HDFS Block size. The HDFS Block size for Hadoop 1.0 is 64mb and Hadoop 2.0/YARN is 128MB. It creates one partition for each block size.

Ex- We have a file of 500MB, so 4 partitions would be created.

- At times programmers are required to change the number of partitions based on the requirements of the application job. The change can be to increase the number of partitions or decrease the number of partitions.
- So we would either apply the repartition or coalesce.



```
### Create some Dump data for testing
df = spark.range(1000000)
df = df.select(df.id,df.id*2,df.id*3)
df = df.union(df)
```

```
### Convert DataFrame to RDD.
RDD = df.rdd.map(lambda x : str(x[0]) + ',' + str(x[1]) + ',' + str(x[2]))
```

```
### Save the file at a HDFS Path
RDD.coalesce(1).saveAsTextFile('/user/test/test_data')
```

```
### Testing
rdd = sc.textFile('/user/test/test_data')
rdd.getNumPartitions()
```

```
#Apply a filter.
rdd1 = rdd.filter(lambda x : int(x.split(',')[0]) == 1)
rdd1.getNumPartitions()
```

Repartition

Repartition(*numPartitions*):

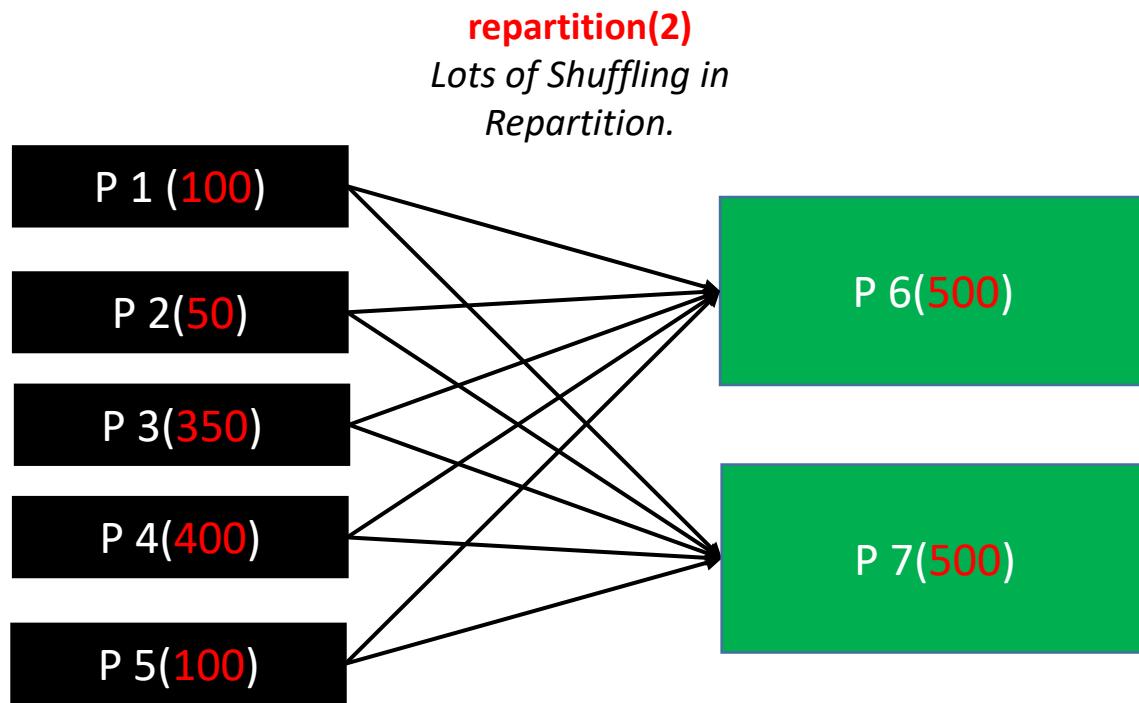
- Return a new RDD that has exactly *numPartitions* partitions.
- Create almost equal sized partitions.
- Can increase or decrease the level of parallelism.
- Spark performs better with equal sized partitions. If you need further processing of huge data, it is preferred to have equal sized partitions and so we should consider using repartition.
- Internally, this uses a shuffle to redistribute data from all partitions leading to very expensive operation. So avoid if not required.
- If you are decreasing the number of partitions, consider using **coalesce**, where the movement of data shuffling across the partitions is lower.

Ex –

```
ord = sc.textFile('practice/retail_db/orders')
ord.glom().map(len).collect()
```

```
ord = ord.repartition(5)
ord.glom().map(len).collect()
```

Repartition



Repartition and Sort

repartitionAndSortWithinPartitions(numPartitions=None, partitionFunc=<function portable_hash>, ascending=True, keyfunc)

- Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.
- Need a key value paid RDD.

Ex -

```
rdd = sc.parallelize(((9, ('a','z')), (3, ('x','f')), (6, ('j','b')), (4, ('a','b')), (8, ('s','b')), (1, ('a','b'))),2)
rdd2 = rdd.repartitionAndSortWithinPartitions(2, lambda x : x % 2, True)
rdd2.glm().collect()
```

Coalesce

coalesce(numPartitions, shuffle=False):

- Return a new RDD that is reduced into `numPartitions` partitions.
- Optimized version of repartition().
- No shuffling.
- Results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead each of the 100 new partitions will claim 10 of the current partitions.
- If a larger number of partitions is requested, it will stay at the current number of partitions.
- By Default Coalesce can be only used for decreasing the partitions. But by passing shuffle=True parameter it behaves like repartition and we can increase the partitions as well.

#Reduce number of partitions using coalesce

```
rdd = sc.textFile('practice/retail_db/orders')
rdd.getNumPartitions()
rdd.coalesce(1).getNumPartitions()
```

#Try increase number of partitions using coalesce

```
rdd.coalesce(5).getNumPartitions()
```

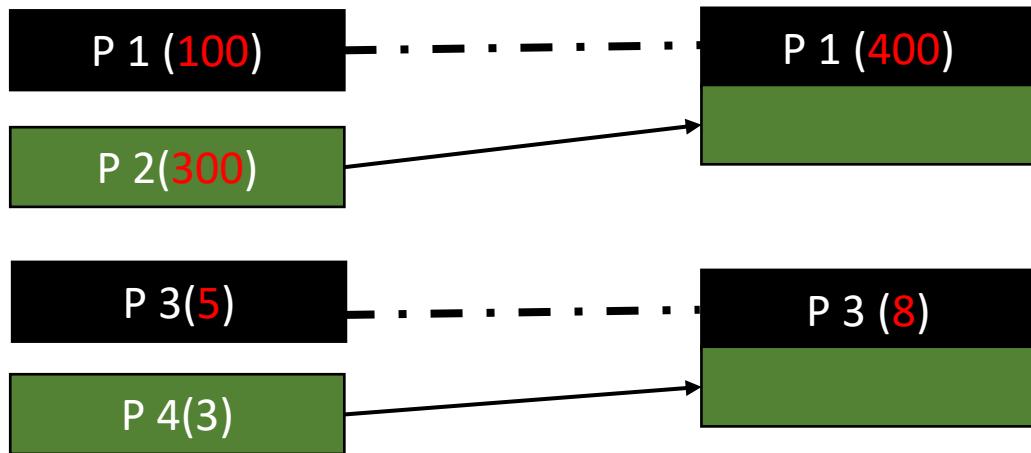
#Try increase number of partitions using coalesce and shuffle param

```
rdd.coalesce(5).getNumPartitions()
```

coalesce

coalesce(2)

No Shuffle in coalesce.



Repartition vs Coalesce

Repartition

1. Repartition does a full shuffle.
2. Preferable used to increase number of partitions.
3. Repartition creates new partitions and does a full shuffle.
4. Repartition results in roughly equal sized partitions.
5. Coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions.
6. It's critical to repartition or coalesce after running joining or filtering queries. If the data becomes smaller, consider using coalesce to merge partitions and data becomes larger consider using repartition to increase the number of partitions.

Coalesce

1. Coalesce avoids full shuffle.
2. Preferable used to decrease number of partitions.
3. Coalesce uses existing partitions to minimize the amount of data that's shuffled.
4. Coalesce results in partitions with different size of data.
5. Coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions.
6. It's critical to repartition or coalesce after running joining or filtering queries. If the data becomes smaller, consider using coalesce to merge partitions and data becomes larger consider using repartition to increase the number of partitions.

RDD Loading/Writing from HDFS

Reading File from HDFS into RDD:

- sc.textFile → Text File
- sc.sequenceFile → Sequence File

Writing data into HDFS From RDD:

- saveAsTextFile
- saveAsSequenceFile

RDD : Save as a Text File

`saveAsTextFile(path, compressionCodecClass=None)`

- Save this RDD as a text file.
- The list of supported Codecs will be found at core-site.xml.

Ex – Find the number of customers placed order in July or Aug Month. Store the output at HDFS as a text format.

Create 5 files. Compression format should be bzip2.

```
ord = sc.textFile('practice/retail_db/orders')
julyOrd = ord.filter(lambda x : str(x.split(',')[1].split('-')[1]) == '07')
augOrd = ord.filter(lambda x : str(x.split(',')[1].split('-')[1]) == '08')
julyAugOrders = julyOrd.union(augOrd).distinct()
julyAugOrders.coalesce(5).saveAsTextFile('practice/retail_db/dump/julyAugOrders',
compressionCodecClass='org.apache.hadoop.io.compress.BZip2Codec')
```

RDD : Save into a SequenceFile

saveAsSequenceFile(path, compressionCodecClass=None)

- A sequence File is a collection of tuple of Key-Value Pair.
- Its flat binary file type that serves as a container for data to be used in Hadoop distributed computing projects and Map Reduce.

#Extract rdd to sequence file

Ex – Find the number of customers placed order in July or Aug Month. Store the output at HDFS as a sequence file format. Create 1 file. No Compression is required.

```
JulyAugOrdPair = julyAugOrders.map(lambda r : (r.split(',')[0],r))
```

```
JulyAugOrdPair.coalesce(1).saveAsSequenceFile('practice/retail_db/dump/julyAugOrders')
```

#Keys can also be None if you do not have a suitable key.

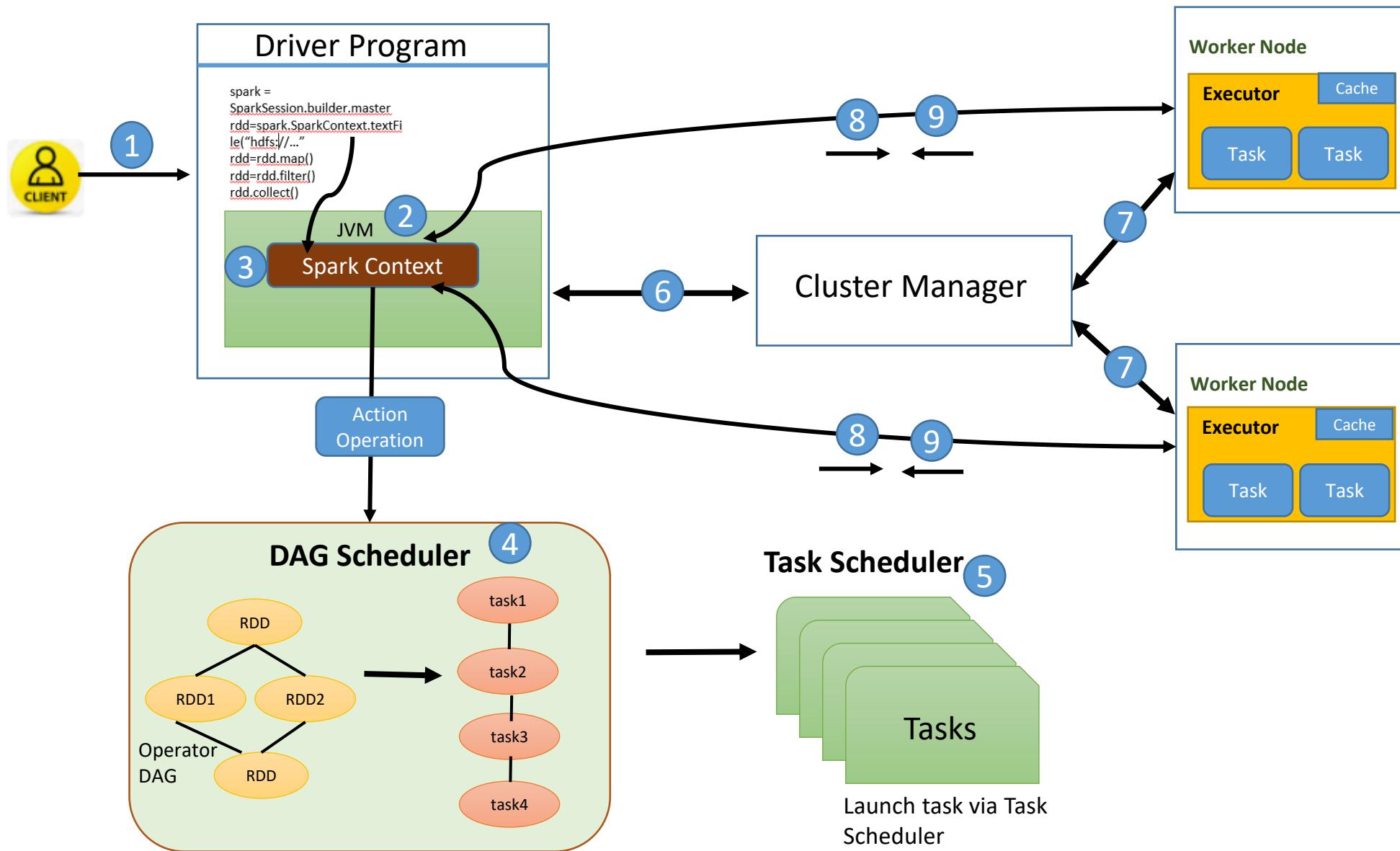
```
JulyAugOrdPair = julyAugOrders.map(lambda r : (None,r))
```

#Load a sequence file into rdd

```
rdd = sc.sequenceFile('practice/retail_db/dump/julyAugOrders')
```

Spark Cluster Execution Architecture

Spark Execution Architecture



- Follows master-slave architecture.
- Client submits user application code to Driver. (1)
- JVM Is created. (2)
- Spark Context is created in the JVM of driver program. Only one active SC per JVM. (3)
- Driver implicitly converts the user code into logically DAG(Directed Acyclic Graph) using DAG scheduler. (4)
 - DAG Scheduler performs optimizations such as pipelining transformations and then it converts the logical graph DAG into Physical executing plan with many stages.
 - After creating physical executing plans, it creates Physical executing units called tasks under each stage.
- The stages pass on to Task Scheduler. It launches task through cluster manager. (5)
- Now driver via Spark Context talks to the cluster manager and negotiates resources. It request for worker nodes and executors in the cluster. (6) (7)
 - Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN).
 - Cluster Manager allocate resources and instruct executors to execute the job.
 - Also track the submitted jobs and report back the status of the job to the driver.
- Driver sends application code and dependencies(defined by jar or Python files passed to the SparkContext) to executors. (8)
- Finally driver also send the tasks to the executors to run.
- Job resources which we are trying to pass as part of execution can be cached at Worker Nodes. One of the job resource can be our code itself.
- All the executors start to register themselves with the drivers so that driver will have a complete view of the executors.
- Executors now start executing the tasks that are assigned by the driver program.
- When application is running, the driver program will monitor the set of executors that runs. Driver also schedules the future tasks based on data placement.
- After execution, the result returns back to the Spark Context. (9)

Cluster Manager Types:

The system currently supports several cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.
- Hadoop YARN – the resource manager in Hadoop 2.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

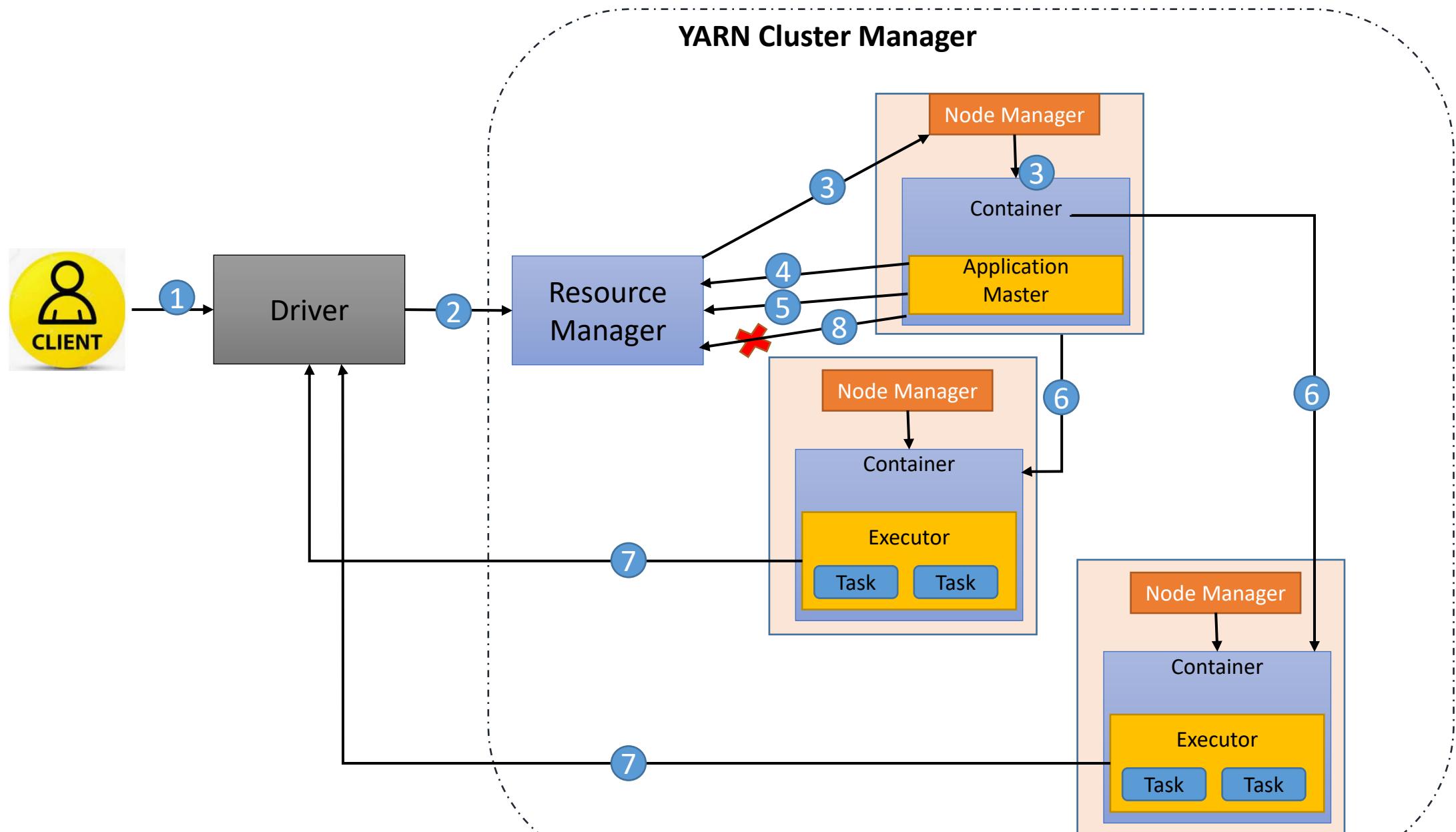
YARN Cluster Manager

YARN as Spark Cluster Manager:

- Yet Another Resource Negotiator for Hadoop 2.x.
- Cluster Management – Used for resource allocation and Scheduling.
- It has 3 major components –
 - ✓ Resource Manager
 - ✓ Node Manager
 - ✓ Application Master

Flow:

1. Client submit the Spark application. Driver instantiates SparkContext.
2. Driver talks to the cluster manager(YARN) and negotiates resources.
3. The YARN resource manager search for a Node Manger which will, in turn, launch an ApplicationMaster for the specific job in a container.
4. The ApplicationMaster registers itself with the resource Manager.
5. The ApplicationMaster negotiates containers for executors from the ResourceManager. Can request for more resources from RM.
6. The ApplicationMaster notifies the Node Managers to launch the containers and executors. Executor then executes the tasks.
7. Driver communicates with executors to coordinate the processing of tasks of an application.
8. Once the tasks are complete, ApplicationMaster un-registers with the Resource Manager.



Resource Manager:

- Runs on Master Node.
- Manages the resources used across the cluster.
- Two components – Scheduler and Application Manager.

Scheduler - Performs scheduling based on the requirement of resources by the applications.

Application Manager: It manages the running of Application Master and restart it on its failure. Also it is responsible to accept the submission of jobs.

Node Manager:

- Runs on all Worker Nodes.
- Launches and monitor containers which are assigned by RM.
- Responsible for the execution of the task in each data node.

Containers:

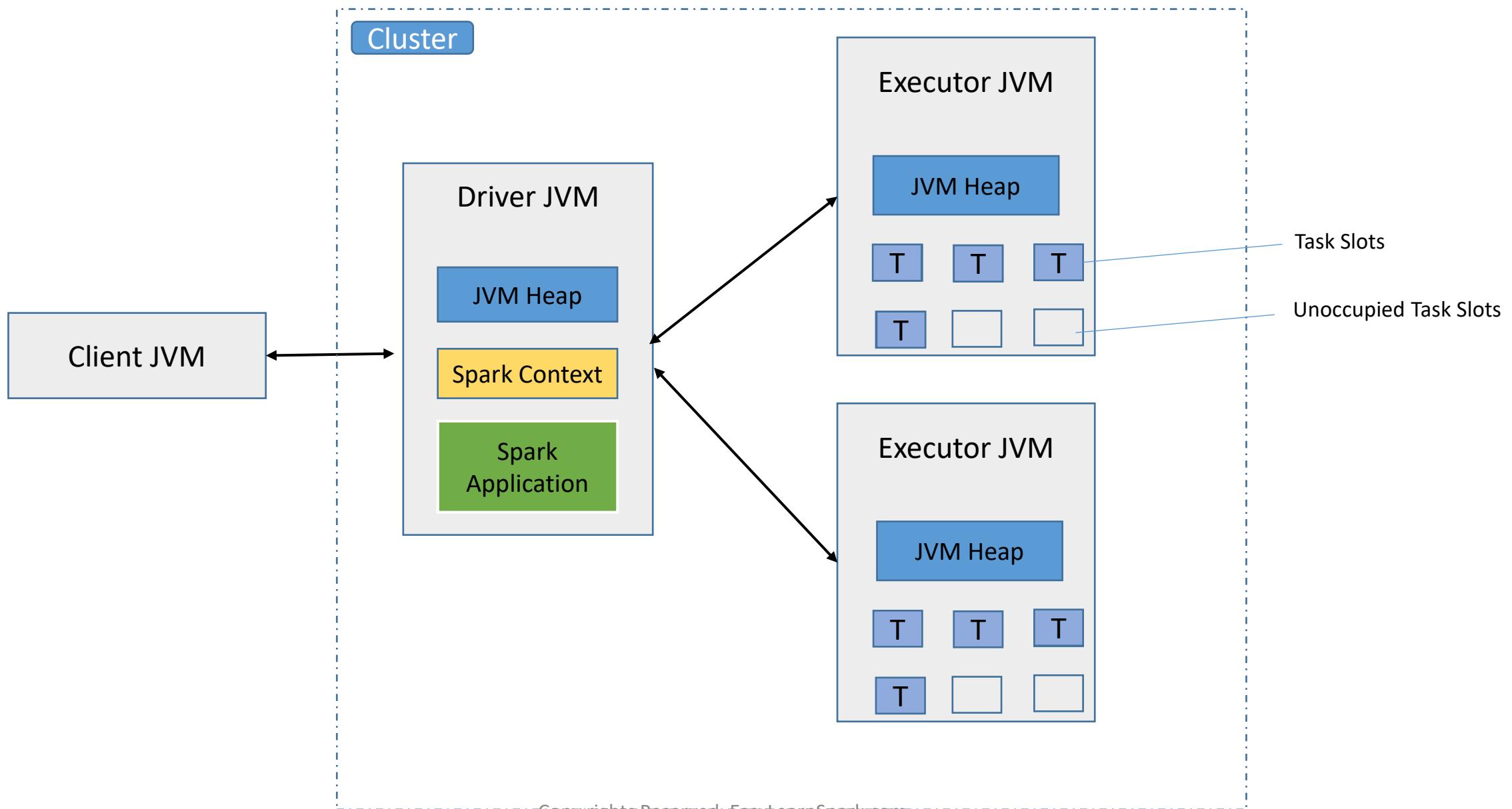
- Are set of resources like RAM, CPU, memory etc on a single node and they are scheduled by RM and monitored by NM.

Application Master:

- An individual ApplicationMaster is assigned for each job by RM.
- Its chief responsibility is to negotiate the resources from the RM. It works with the Node Manager to monitor and execute the tasks.

JVM Processes

Where JVMs are created during execution across the Clusters :



Terms used in Spark Execution Framework:

Application: User application built on Spark.

Job: A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect).

Stage: Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

Task: A unit of work that will be sent to one executor.

Application Jar: A jar containing the user's Spark application codes.

Driver Program: The process running the main() function of the application and creating the SparkContext.

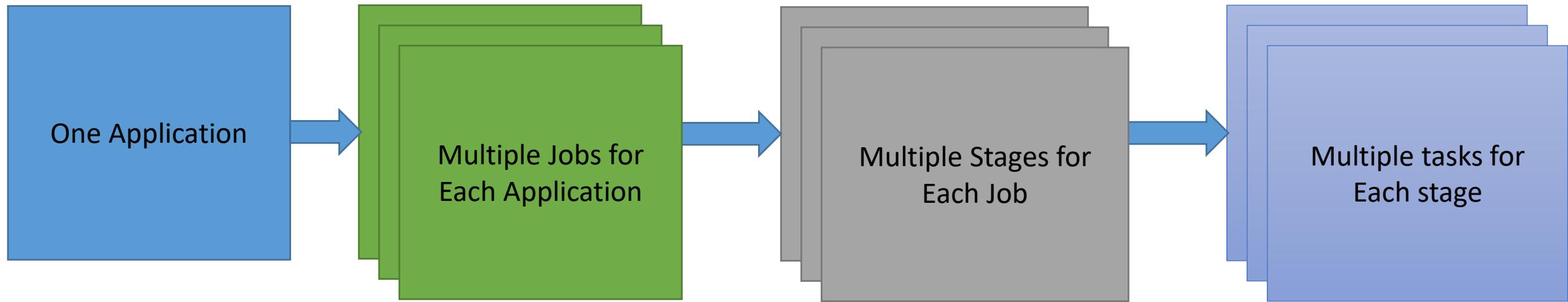
Cluster Manager: An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN). In YARN it is Resource Manager + Application Master.

Deploy Mode: Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.

Worker Node: Any node that can run application code in the cluster.

Executor: A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.

Cache: Cache is nothing but the job resources which we are trying to pass as part of execution. One of the job resource can be our code itself. Typically the code will be compiled in the jar file and jar file will be passed as cache and it will be cached into the executors. Tasks will be using this cache at the run time so that it can apply the logic onto the data while data is being processed. We can also persist or cache the RDDs.

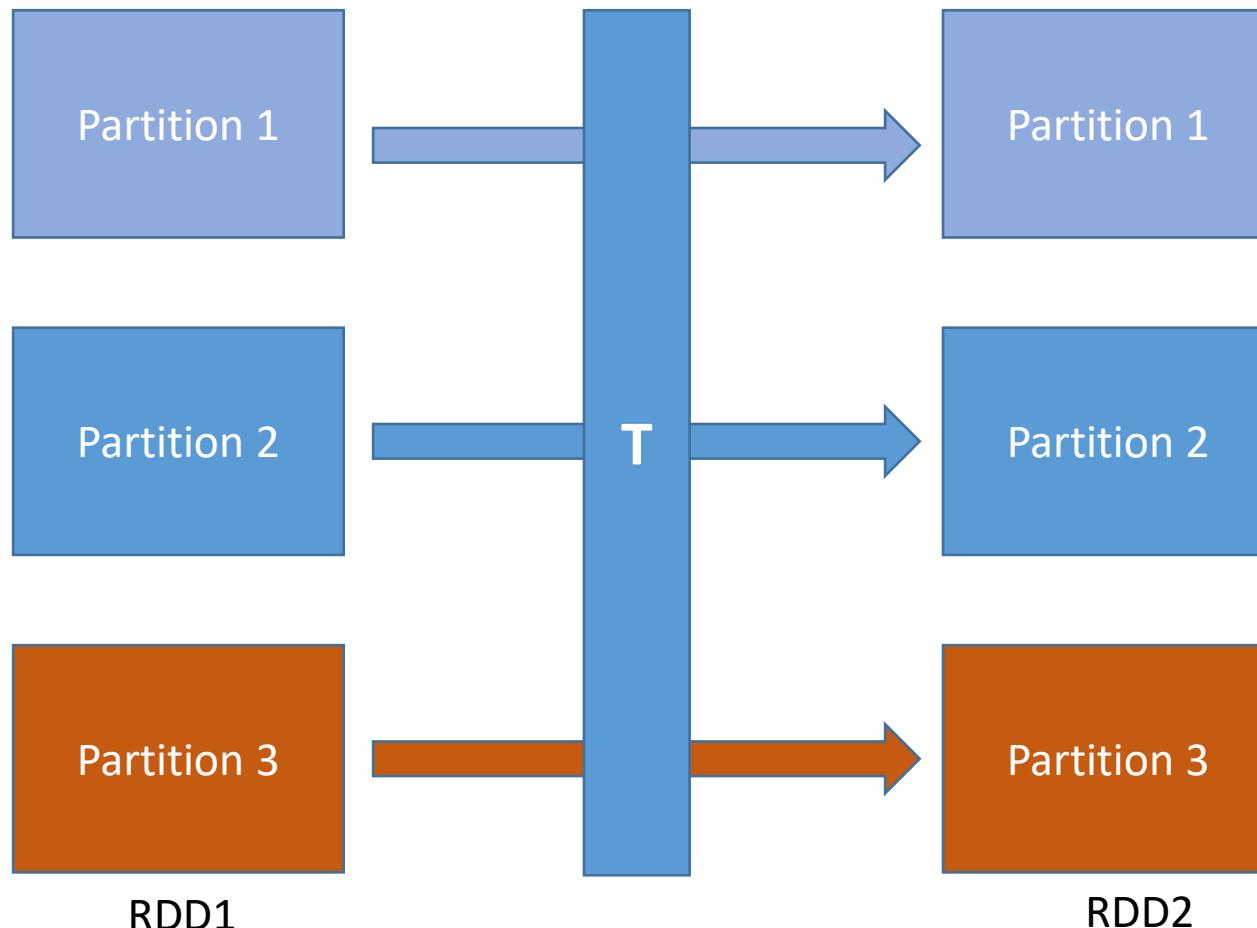


Narrow and Wide Transformations

Narrow and Wide Transformations

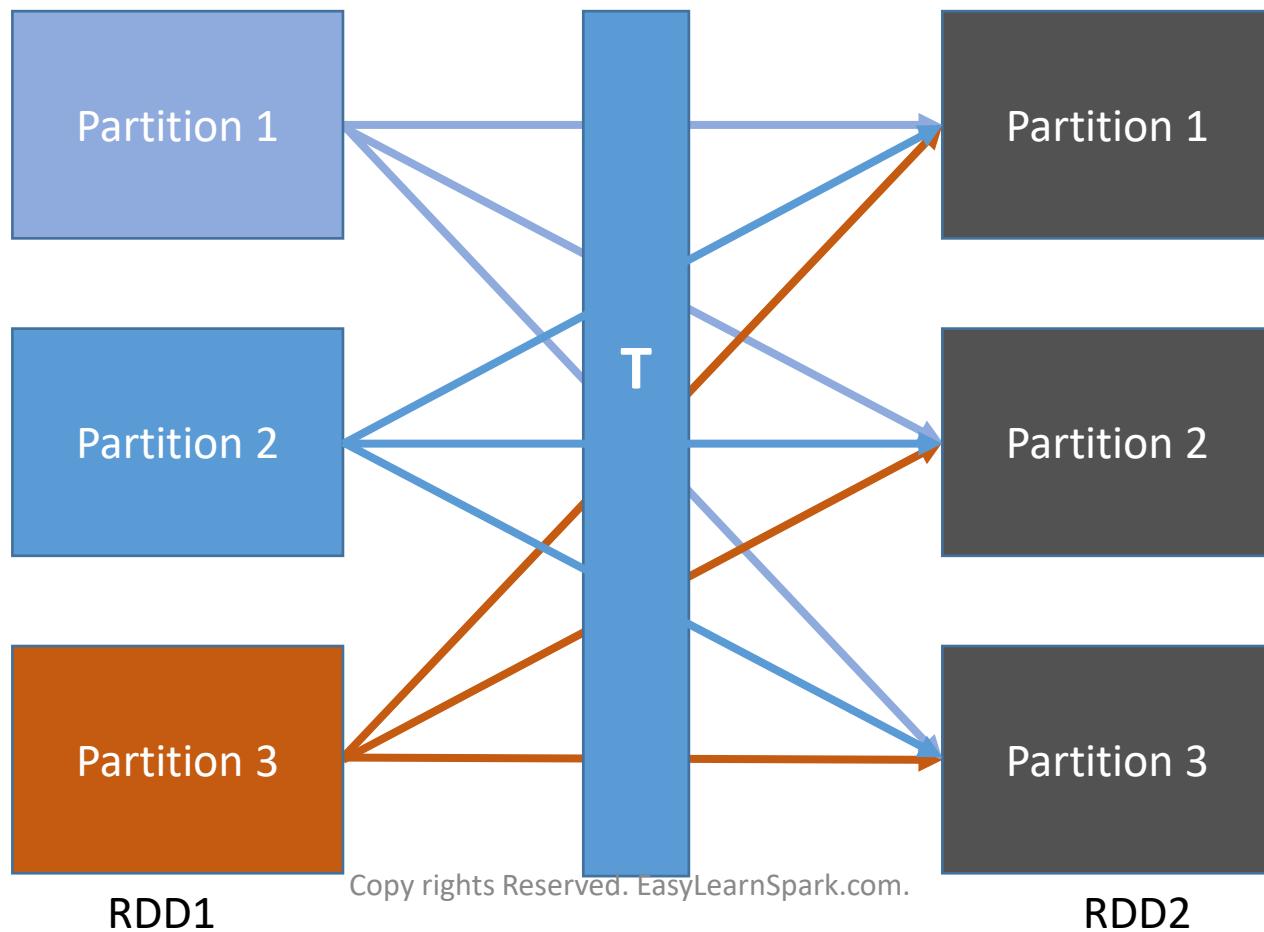
Narrow Transformations:

- These types of transformations convert each input partition to only one output partition.
- Spark merges all narrow transformations into one stage.
- Fast.
- No data shuffle.
- Ex- `map()`, `filter()`



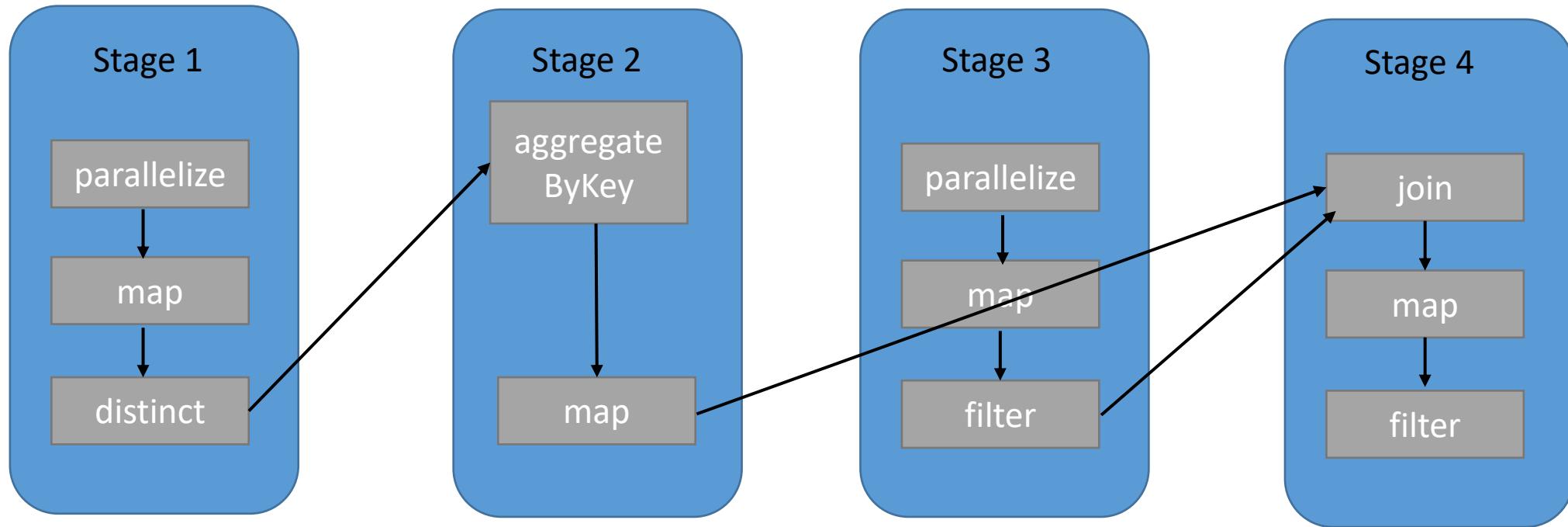
Wide Transformations:

- This type of transformation will have input partitions contributing to many output partitions.
- Each Wide Transformation creates a new stage.
- Slow compared to Narrow.
- Data shuffle.
- Ex- `groupByKey()`, `aggregateByKey()`, `join`, `distinct()`, `repartition()` etc.



DAG

DAG Graph



DAG

DAG Stands for **Directed Acyclic Graph**.

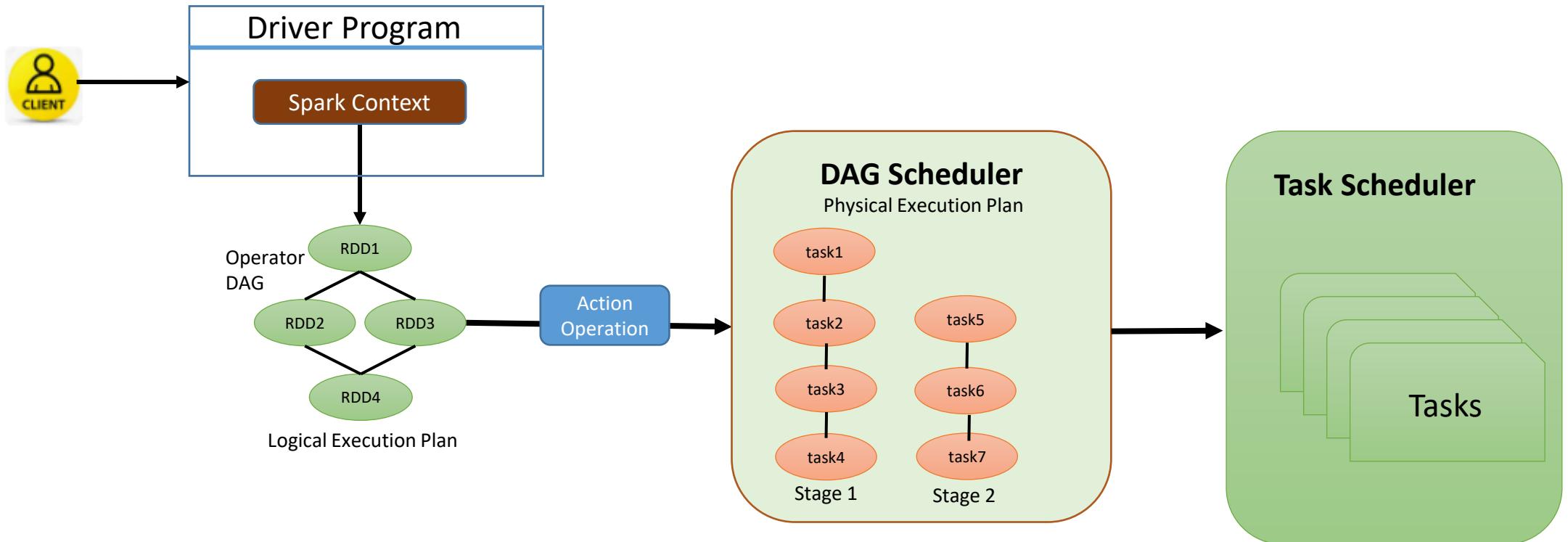
Directed → Directly connected to one node to another.

Acyclic → There is no cycle or loop. So it is in line and we can not go back to its original position.

Graph → It has Vertices and Edges. Vertices indicates RDDs and edges refers to the operations on the RDD.

These all represented as a graph.

DAG Scheduler



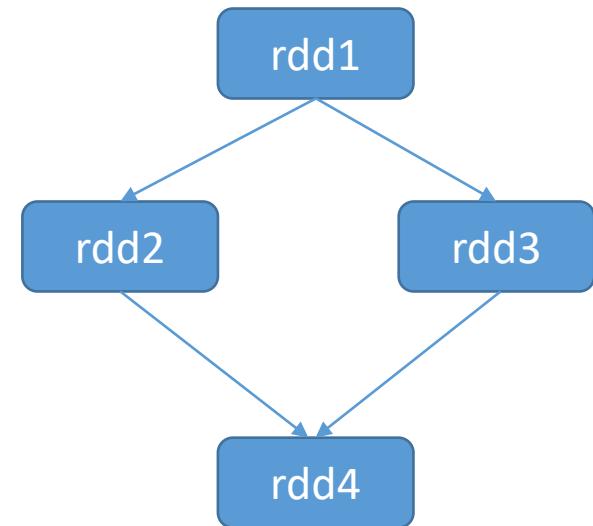
Steps to build a DAG

1. User submits a application job to spark.
2. Drivers takes the application and create a Spark Context to process the application.
3. Spark Context identifies all the T and A operations present in the application.
4. All the operations are arranged in a logical flow of operations called DAG (Logical Execution Plan).
5. It stops here if SC doesn't find any A Operations.
6. If it identifies an A operations, spark submit the Operator DAG to DAG scheduler.
7. DAG Scheduler converts the Logical Execution plan into Physical Execution plan and creates stages and tasks. Here Narrow T are fused together into one stage. Wide T involving shuffle process creates new stages.
8. DAG scheduler bundles all the tasks and send it Task Scheduler which then submit the job to cluster manager for execution.

RDD Lineage

RDD Lineage:

- Each RDD maintains a pointer to one or more parent along with metadata about what type of relationship it has with the parent.
- Ex - if we call `rdd2=rdd1.map()`, the `rdd2` keeps a reference to its parent `rdd1` and this is called RDD lineage.
- Print the RDD lineage information using `toDebugString()` API.



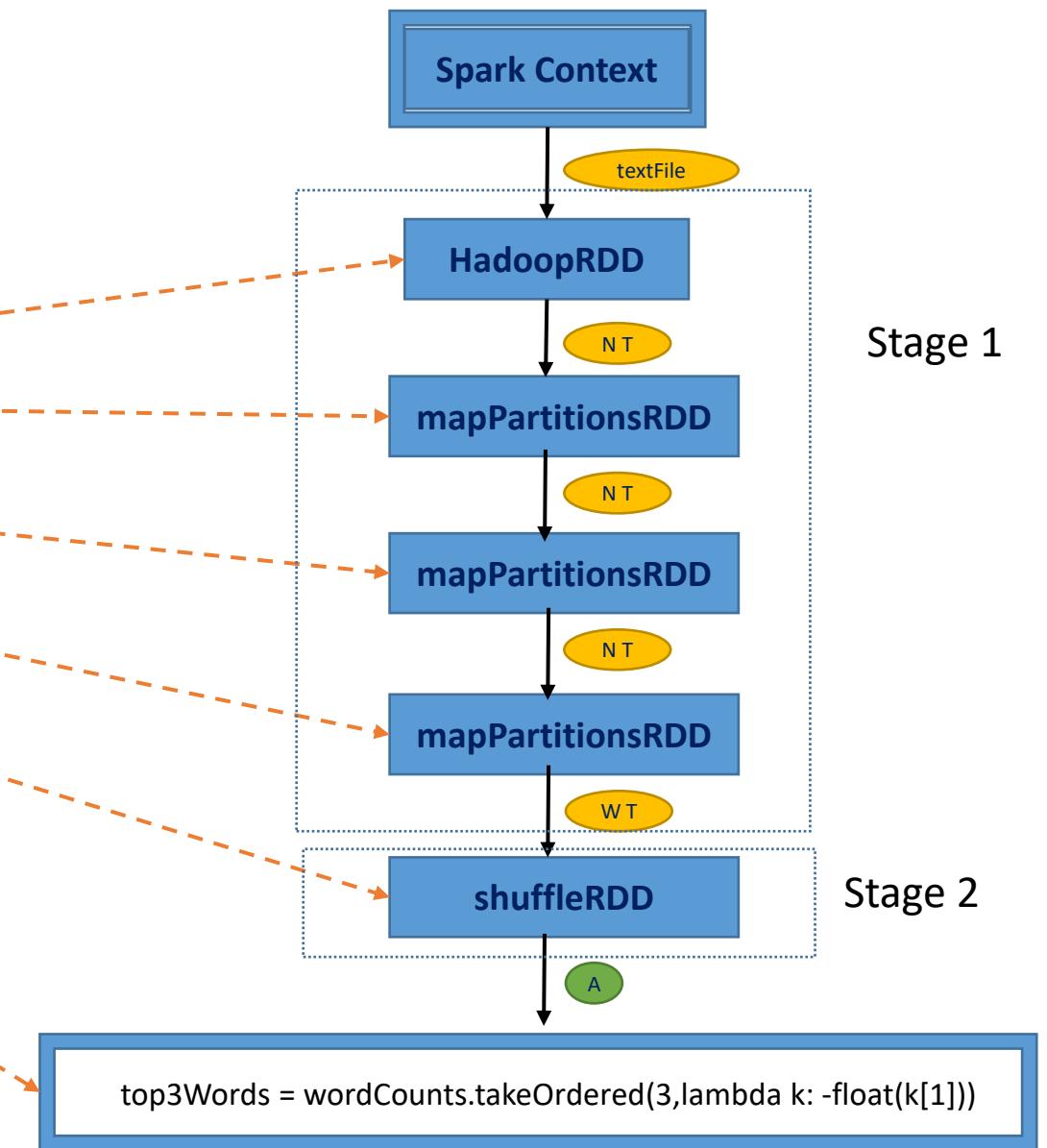
RDD Lineage Graph

`toDebugString(self)` :

- Displays Logical Execution Plan.
- We can learn about a RDD Lineage Graph using API `toDebugString`.
- Displays the description of this RDD and its recursive dependencies for debugging.

Word Count Program:

```
text_file = sc.textFile('practice/retail_db/word')  
wordCounts = text_file.flatMap(lambda line: line.split(',')) \  
.filter(lambda x : x.isdigit() == False) \  
.map(lambda word: (word, 1)) \  
.reduceByKey(lambda a, b: a + b)  
top3Words = wordCounts.takeOrdered(3,lambda k: -float(k[1]))  
  
for i in rdd.toDebugString().split("\n") : print(i)
```

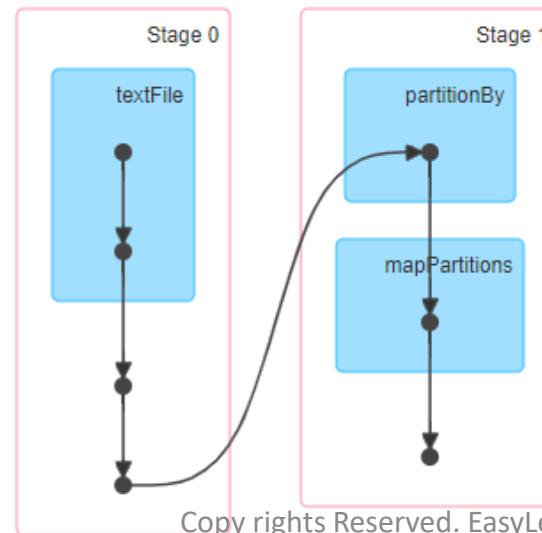


```
>>> for i in wordCounts.toDebugString().split("\n") : print(i)
...
(2) PythonRDD[6] at RDD at PythonRDD.scala:53 []
|  MapPartitionsRDD[5] at mapPartitions at PythonRDD.scala:133 []
|  ShuffledRDD[4] at partitionBy at NativeMethodAccessorImpl.java:0 []
+- (2) PairwiseRDD[3] at reduceByKey at <stdin>:4 []
   |  PythonRDD[2] at reduceByKey at <stdin>:4 []
   |  practice/retail_db/word MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0 []
   |  practice/retail_db/word HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []
```

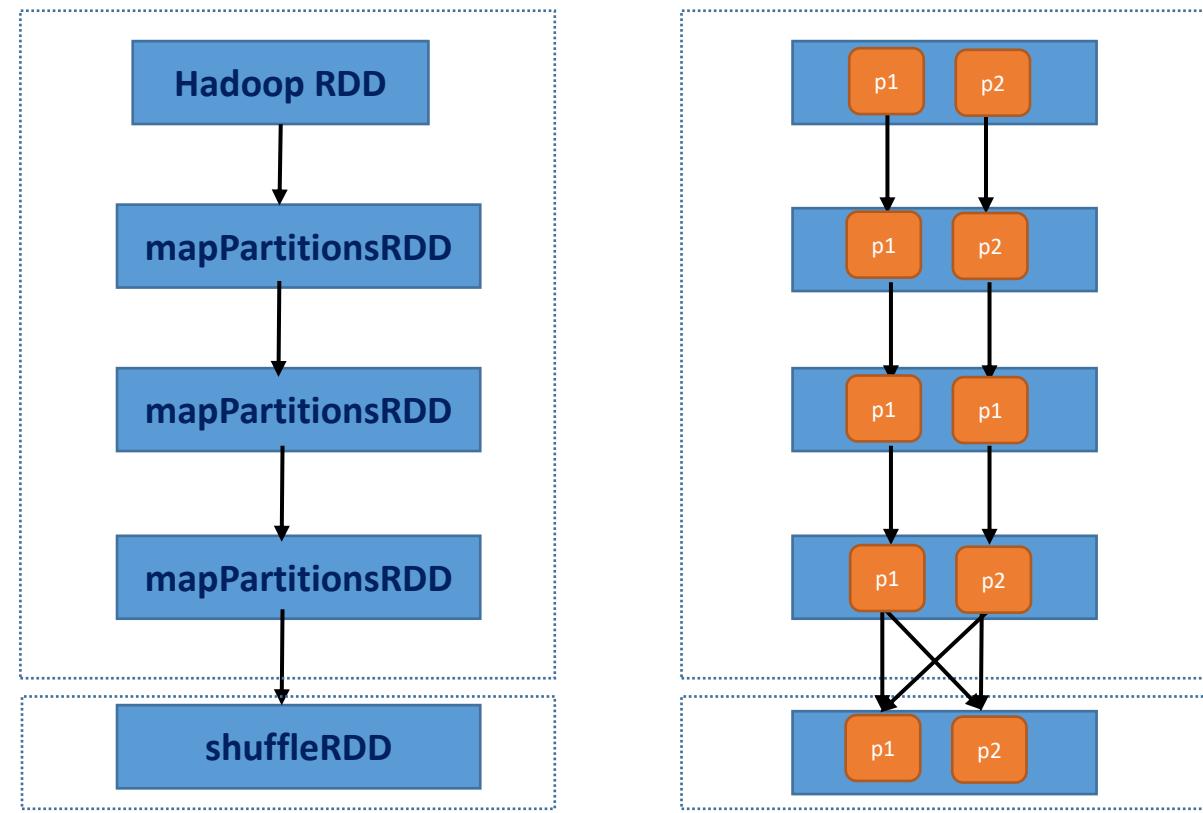
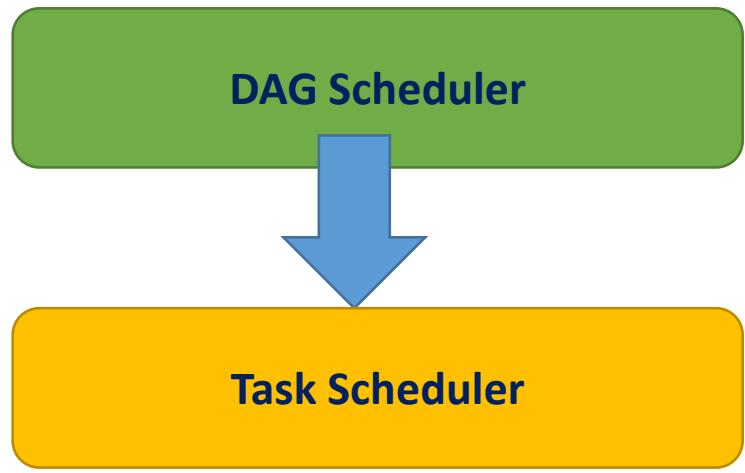
Logical Execution Plan



▼ DAG Visualization



Physical Execution Plan



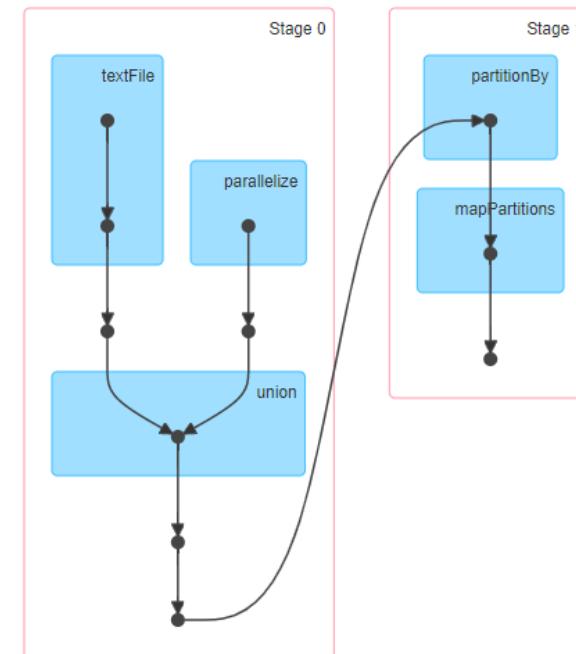
DAG Scheduler to Task Scheduler:

- DAG Scheduler submit the stages into task scheduler.
- The number of tasks submitted depends on the number of partitions present in the textFile.
- For ex – imagine we have 3 partitions, then there will be 3 set of tasks created and submitted in parallel provided there are enough cores.

Parallel Execution of Stages:

- The stages which are not dependent on each other may be submitted to the cluster for execution in parallel.

```
text_file = sc.textFile('practice/retail_db/orders').map(lambda x : (int(x.split(',')[0]),x.split(',')[3]))  
rdd = sc.parallelize([1,2,3,4,5,6,7]).map(lambda x : (x,1*x) )  
joined = text_file .join(rdd)  
joined.count()
```



How to find the Resource Manager URL

- ✓ Go to /etc/hadoop/conf
- ✓ Open yarn-site.xml
- ✓ Find the resource Manager url by searching name ‘yarn.resourcemanager.webapp.address’
- ✓ Default port : 8088

RDD Persistence

- One of the most important capabilities in Spark is persisting (or caching) a dataset in memory.
- When we persist an RDD, each node stores any partitions of it that it computes in memory and reuses them.
- We can mark an RDD to be persisted using the `persist()` or `cache()` methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
- Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.
- Each persisted RDD can be stored using different storage level.
- These levels are set by passing a `StorageLevel` object

StorageLevel(*useDisk*, *useMemory*, *useOffHeap*, *deserialized*, *replication=1*)

- Data can be stored either in Disk or Memory or off-heap memory or any of these combinations.
- Data can be stored in Serialized or deserialized. Serilization is a way to convert a object in memory to series of bits. The deserialization is the process of bringing those bits into memory as an object. Whenever we are talking about 'deserialized' RDD/DF, we are always referring to RDD/DFs in memory.
- Off-Heap Memory is a segment of memory lies outside the JVM, but is used by JVM for certain use-cases. Off-Heap memory can be used by Spark explicitly as well to store serialized data-frames and RDDs.
- Use the replicated storage levels if you want fast fault recovery.

- **DISK_ONLY** = StorageLevel(True, False, False, False, 1)
- **DISK_ONLY_2** = StorageLevel(True, False, False, False, 2)
- **MEMORY_AND_DISK** = StorageLevel(True, True, False, False, 1)
- **MEMORY_AND_DISK_2** = StorageLevel(True, True, False, False, 2)
- **MEMORY_AND_DISK_SER** = StorageLevel(True, True, False, False, 1)
- **MEMORY_AND_DISK_SER_2** = StorageLevel(True, True, False, False, 2)
- **MEMORY_ONLY** = StorageLevel(False, True, False, False, 1)
- **MEMORY_ONLY_2** = StorageLevel(False, True, False, False, 2)
- **MEMORY_ONLY_SER** = StorageLevel(False, True, False, False, 1)
- **MEMORY_ONLY_SER_2** = StorageLevel(False, True, False, False, 2)
- **OFF_HEAP** = StorageLevel(True, True, True, False, 1)

Default
Storage

persist(storageLevel=StorageLevel(False, True, False, False, 1))

- Set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet.
- If no storage level is specified defaults to →MEMORY_ONLY.

```
rdd = sc.parallelize(("b", "a", "c"))
```

```
rdd.persist().is_cached
```

```
rdd.getStorageLevel()
```

```
from pyspark import StorageLevel
```

```
rdd.persist( StorageLevel.MEMORY_AND_DISK_2 )
```

```
rdd.getStorageLevel()
```

```
print(rdd1.getStorageLevel())
```

unpersist():

- Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.
- Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion.
- If you would like to manually remove an RDD instead of waiting for it to fall out of the cache, use the RDD.unpersist() method.

```
rdd.persist()
```

```
rdd.is_cached
```

```
rdd.unpersist()
```

```
rdd.is_cached
```

Usage and trade-offs:

- Spark's storage levels are meant to provide different trade-offs between memory usage and CPU efficiency.
- If your RDDs fit comfortably with the default storage level (MEMORY_ONLY), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- When amount large amount of dataset or dataset operations are very expensive or memory is not sufficient to hold the data, spill it into disk.
- Use the replicated storage levels if you want fast fault recovery

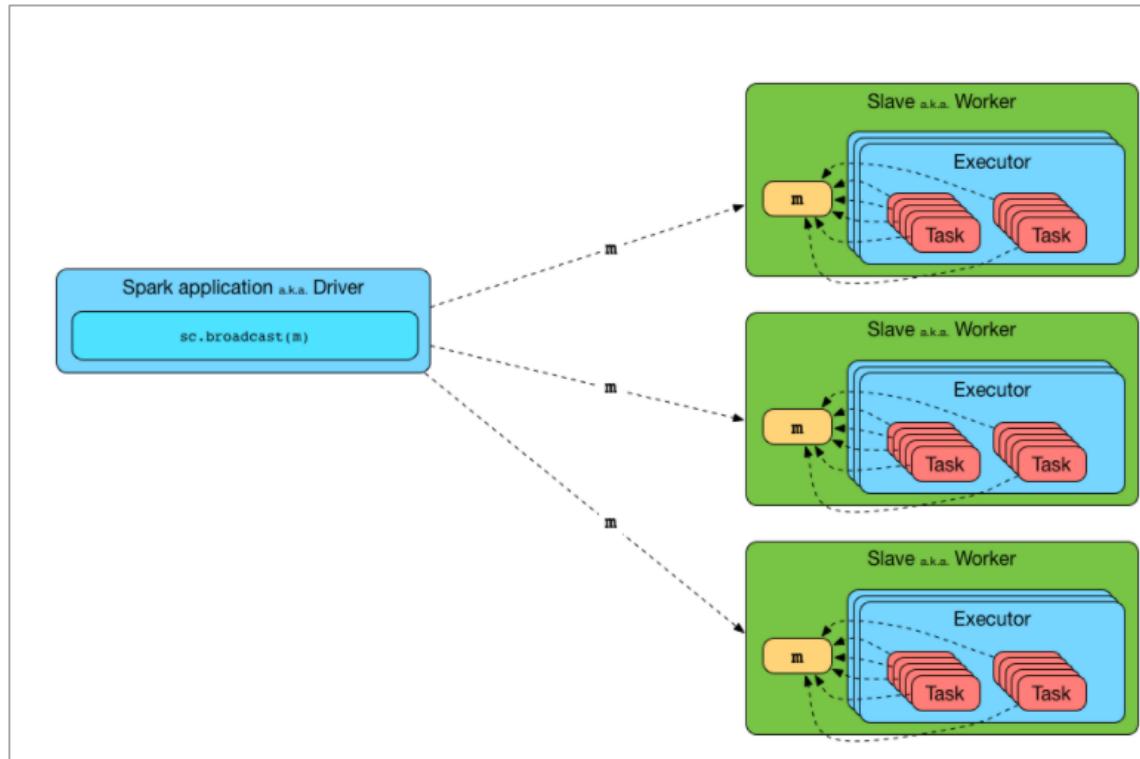
Spark Shared Variables

Shared Variables

- Shared variables are the variables that are required to be used by functions and methods in parallel.
- Shared variables can be used in parallel operations.
- Spark provides two types of shared variables –
 - ✓ Broadcast
 - ✓ Accumulator

Broadcast Variables

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- Immutable and cached on each worker nodes only once.
- Efficient manner to give a copy of a large dataset to each node.
- They should fit in memory.



When to use Broadcast Variable:

- For processing, the executors need information regarding variables or methods. This information is serialized by Spark and sent to each executor and is known as CLOSURE.
- If we have a huge array that is accessed from spark CLOSURES, for example - if we have 5 nodes cluster with 100 partitions (20 partitions per node), this Array will be distributed at least 100 times (20 times to each node). If you broadcast it will be distributed once per node using efficient p2p protocol.

What not to do:

- Once we broadcasted the value to the nodes, we shouldn't make changes to its value to make sure each node have exact same copy of data. The modified value might be sent to another node later that would give unexpected results.

Hands-on:

--Broadcast a Dictionary

```
days={"sun": "Sunday", "mon" : "Monday", "tue":"Tuesday"}
```

```
bcDays = spark.sparkContext.broadcast(days)
```

```
bcDays.value
```

```
bcDays.value('sun')
```

--Broadcast a list

```
numbers = (1,2,3)
```

```
broadcastNumbers=spark.sparkContext.broadcast(numbers)
```

```
broadcastNumbers.value
```

```
broadcastNumbers.value(0)
```

PS: Convert the 3 bytes days into full days.

```
Input data= (("James","Smith","USA","mon"),
            ("Michael","Rose","USA","tue"),
            ("Robert","Williams","USA","sun"),
            ("Maria","Jones","USA","tue")
            )
```

Solution:

```
days={"sun": "Sunday", "mon" : "Monday", "tue":"Tuesday"}
```

```
bcDays = spark.sparkContext.broadcast(days)
```

```
data = (("James","Smith","USA","mon"),
        ("Michael","Rose","USA","tue"),
        ("Robert","Williams","USA","sun"),
        ("Maria","Jones","USA","tue")
        )
```

```
rdd = spark.sparkContext.parallelize(data)
```

```
def days_convert(dict):
    return bcDays.value(dict)
```

```
input = rdd.map(lambda x: (x(0),x(1),x(2),x(3))).collect()
```

```
output = rdd.map(lambda x: (x(0),x(1),x(2),days_convert(x(3)))).collect()
```

Accumulator Variables

- Accumulator is a shared variable to perform sum and counter operations.
- These variables are shared by all executors to update and add information through associative or commutative operations.

Commutative -> $f(x, y) = f(y, x)$

Ex : $\text{sum}(5, 7) = \text{sum}(7, 5)$ 

Associative -> $f(f(x, y), z) = f(f(x, z), y) = f(f(y, z), x)$

Ex : $\text{sum}(\text{multiply}(5, 6), 7) = \text{sum}(\text{multiply}(6, 7), 5)$ 
 $\text{sum}(\text{sum}(5, 6), 7) = \text{sum}(\text{sum}(6, 7), 5)$ 

Accumulator Variables

- Why it is needed as we can use the normal variables?

```
counter = 0
def f1(x):
    global counter
    counter += 1
rdd = spark.sparkContext.parallelize((1,2,3))
rdd.foreach(f1)
counter.value
```

The counter Variable will not added or changed, because when spark ships this code to every executor the variables become local to that executor. So the variable is updated for that executor but do not send it back to the driver. To avoid this problems, we need an accumulator. All the updates to accumulator variable in every executor is send it back to the driver.

```
### sparkContext.accumulator() is used to define accumulator variables
counter = spark.sparkContext.accumulator(0)
def f2(x):
...     global counter
...     counter.add(1) ### add() function is used to add/update a value in accumulator

rdd.foreach(f1)
counter.value ### Only accessed by Driver
```

Accumulator Variables

Lets check one more example:

```
accum=sc.accumulator(0)
rdd=spark.sparkContext.parallelize((1,2,3,4,5))
rdd.foreach(lambda x:accum.add(x))
print(accum.value) #Accessed by driver
```

`foreach()` is an action which is applied to each element of the rdd and then adding each element to accum variable.
`rdd.foreach()` is executed on workers and `accum.value` is called from driver.

Accumulator Variables

- Spark natively supports accumulators of numeric types (int, float) and programmers can add support for new custom types using AccumulatorParam class of PySpark.
- Accumulators do not change the lazy evaluation model of Spark. If they are being updated within an operation on an RDD, their value is only updated once that RDD is computed as part of an action.

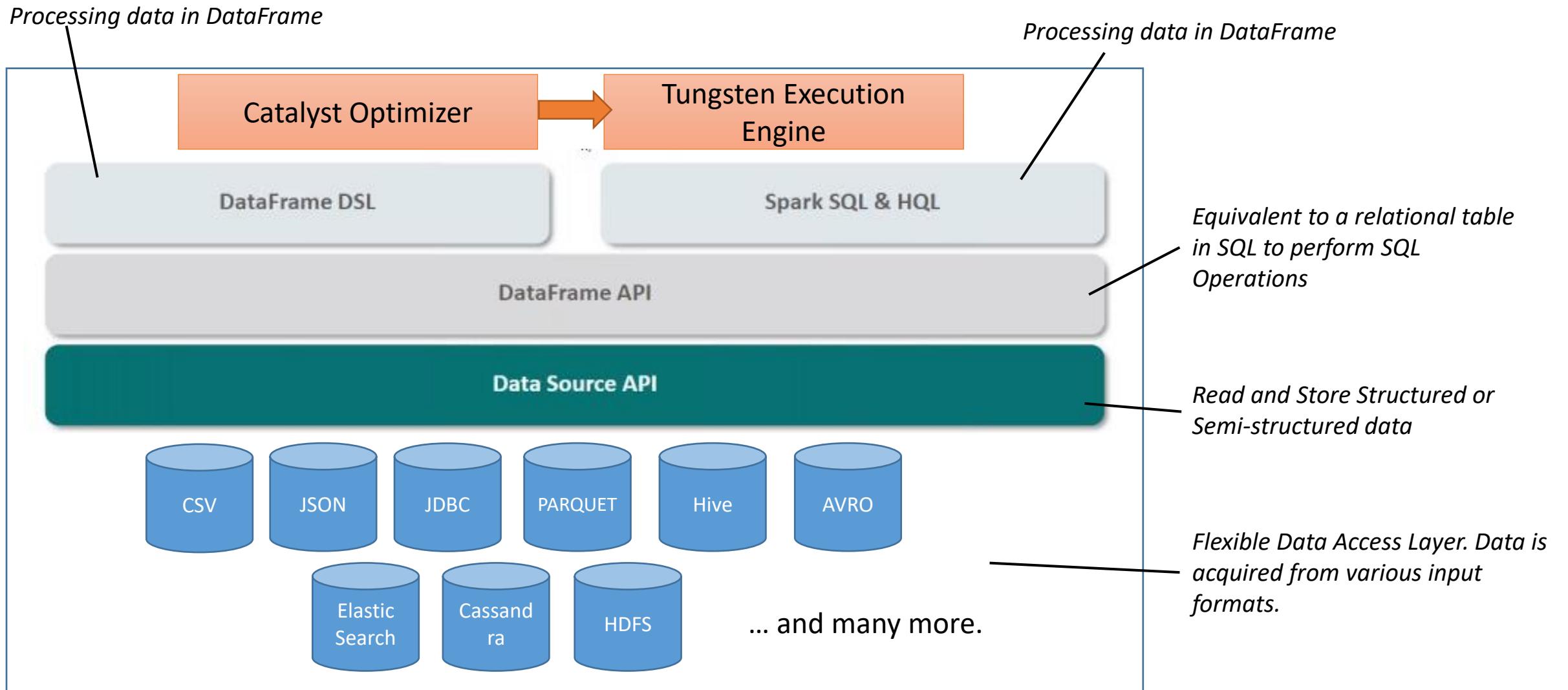
```
counter = spark.sparkContext.accumulator(0)  
def f2(x):  
    global counter  
    counter.add(1)  
rdd.map(f2)
```

- Computations inside transformations are evaluated lazily, so unless an action happens on an RDD the transformations are not executed. As a result of this, accumulators used inside functions like map() or filter() wont get executed unless some action happens on the RDD. Spark guarantees to update accumulators inside actions only once.

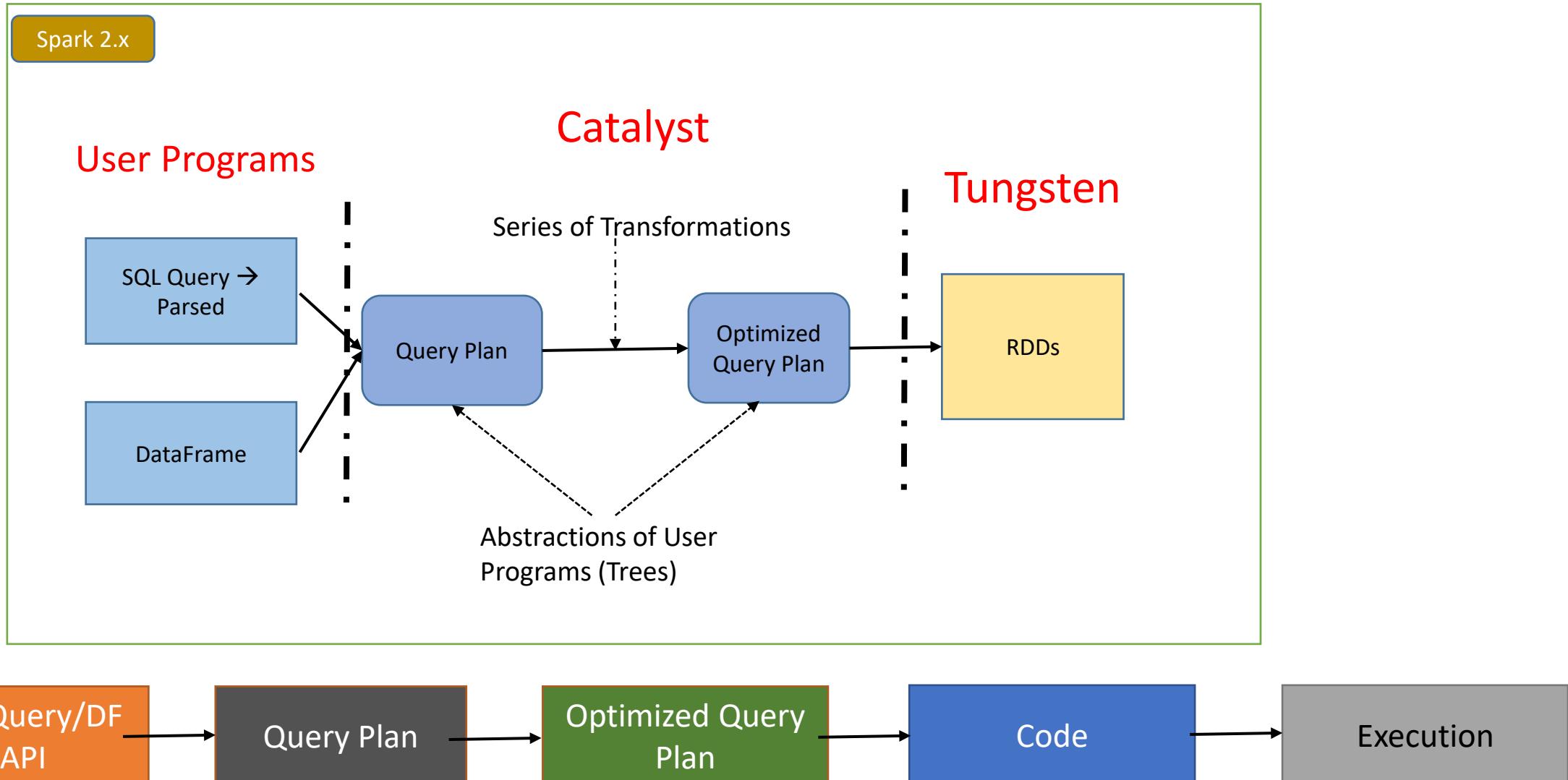
Always use accumulators inside actions ONLY (ex – foreach).

SparkSQL Architecture

Spark SQL Architecture



How Catalyst and Tungsten Works : An Overview



Reference: [databricks.com summit](https://databricks.com/summit)

Spark SQL Architecture

SQL, DataFrames are high-level APIs. It means the user programs just describe what data operations are needed without specifying how to execute these operations.

Optimizer: An optimizer can automatically find out the most efficient plan to execute a query.

Catalyst Optimizer:

- Spark SQL is designed with Catalyst Optimizer which is based on functional programming of Scala.
- Responsible to improve the performance of user programs (SQL Query/DataFrame APIs).
- It converts a query plan into optimized query plan.
- Two main Purpose:
 - ✓ Add new optimization techniques to solve big data problems.
 - ✓ Allows developers/spark community to implement and extend the optimizer with new features.
- Offers both rule-based and cost-based optimization (Spark 2.0).
 - ✓ Rule Based : How to execute the query from a set of defined rules.
 - ✓ Cost-Based: Generates multiple execution plans and chose the lowest cost plan.

Tungsten Execution Engine:

- Generates code and execute it in the cluster in the distributed fashion.
- The engine builds upon ideas from modern compilers and MPP databases.

Trees: Abstractions of Users Programs

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

*Identify
Expressions*



- An expression represents a new value, computed based on input values
 - e.g. $1 + 2 + t1.value$

Expression

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

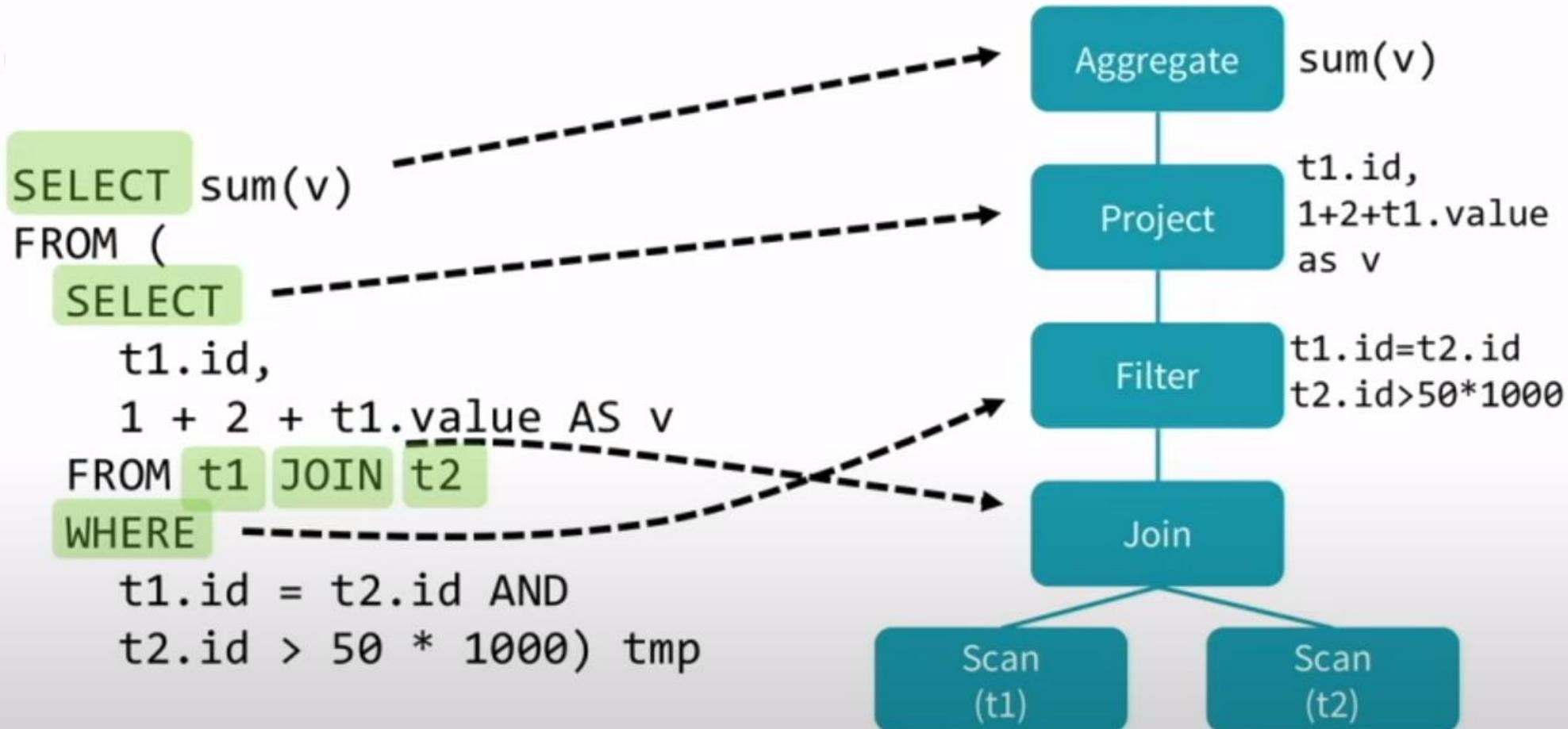
Query Plan

Describe data operation like aggregates, joins, filters etc. and these operations essentially generate a new dataset based on a input dataset.

```
SELECT sum(v)
FROM (
    SELECT
        t1.id,
        1 + 2 + t1.value AS v
    FROM t1 JOIN t2
    WHERE
        t1.id = t2.id AND
        t2.id > 50 * 1000) tmp
```

Reference: databricks.com summit

Query Plan

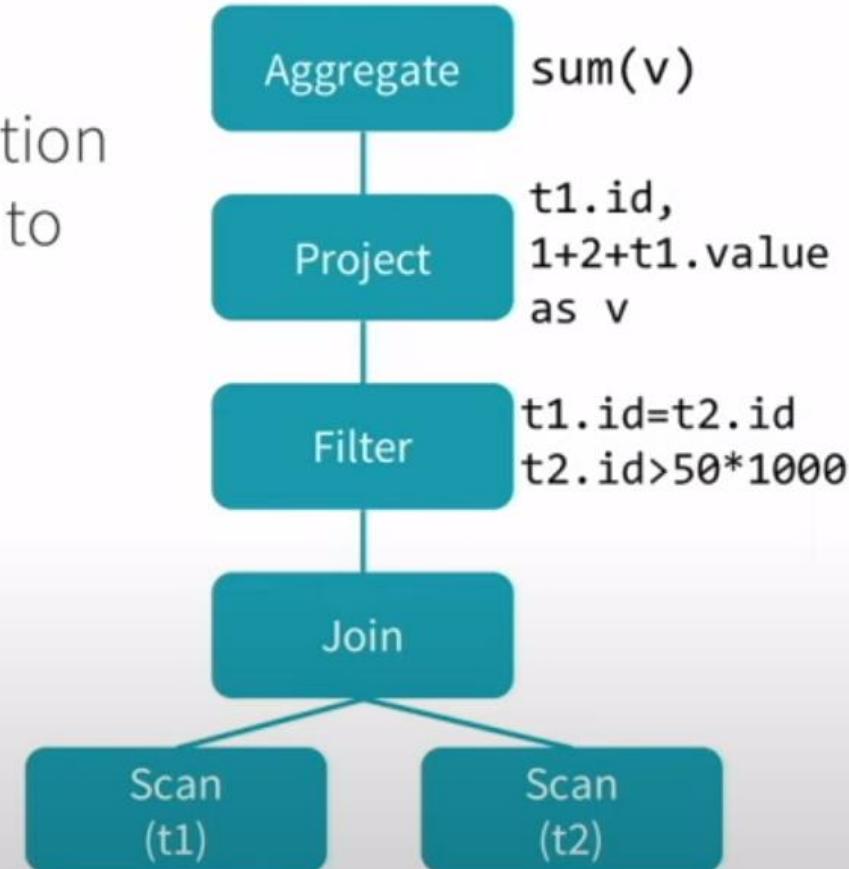


Reference: [databricks.com summit](https://databricks.com/summit)

Logical Query Plan

2 types - Logical Plan and Physical Plan

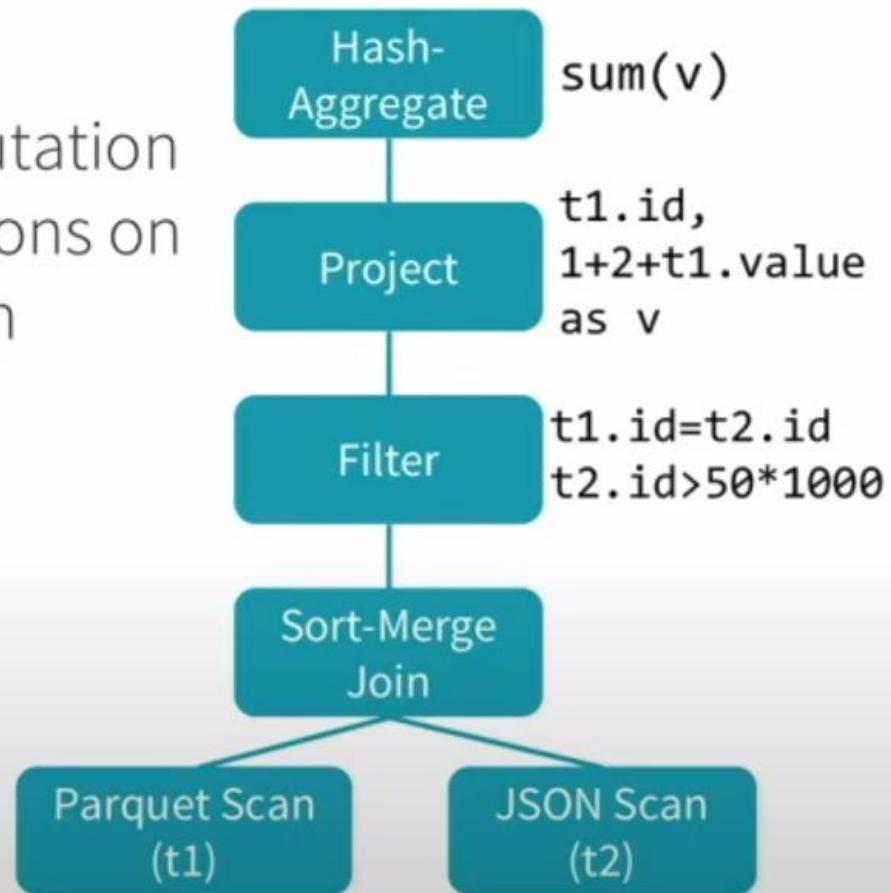
- A Logical Plan describes computation on datasets **without** defining how to conduct the computation



Reference: databricks.com summit

Physical Query Plan

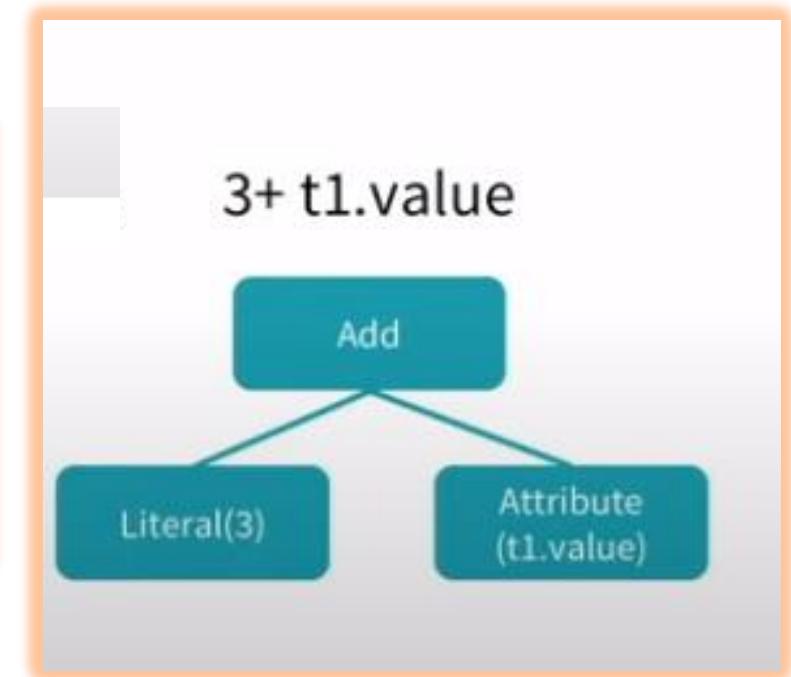
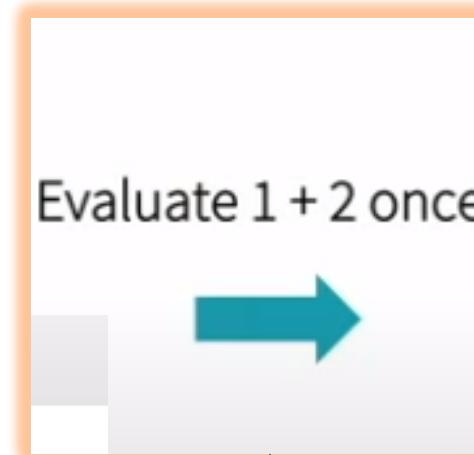
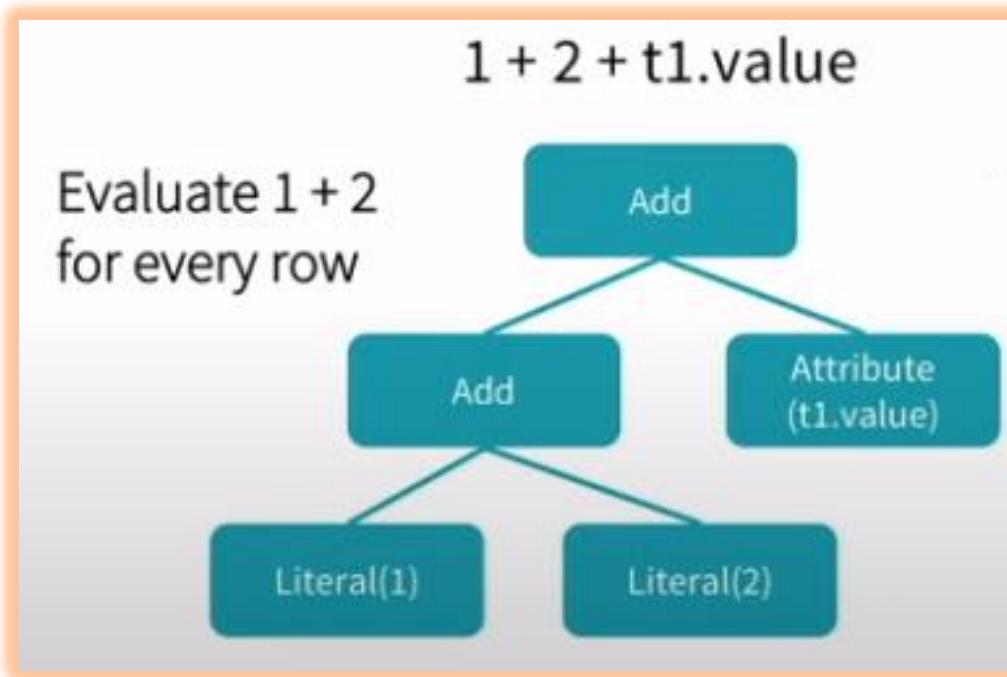
- A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation



Reference: [databricks.com summit](https://databricks.com/summit)

Transform

A Transform is a function associated with every tree to implement a single rule.

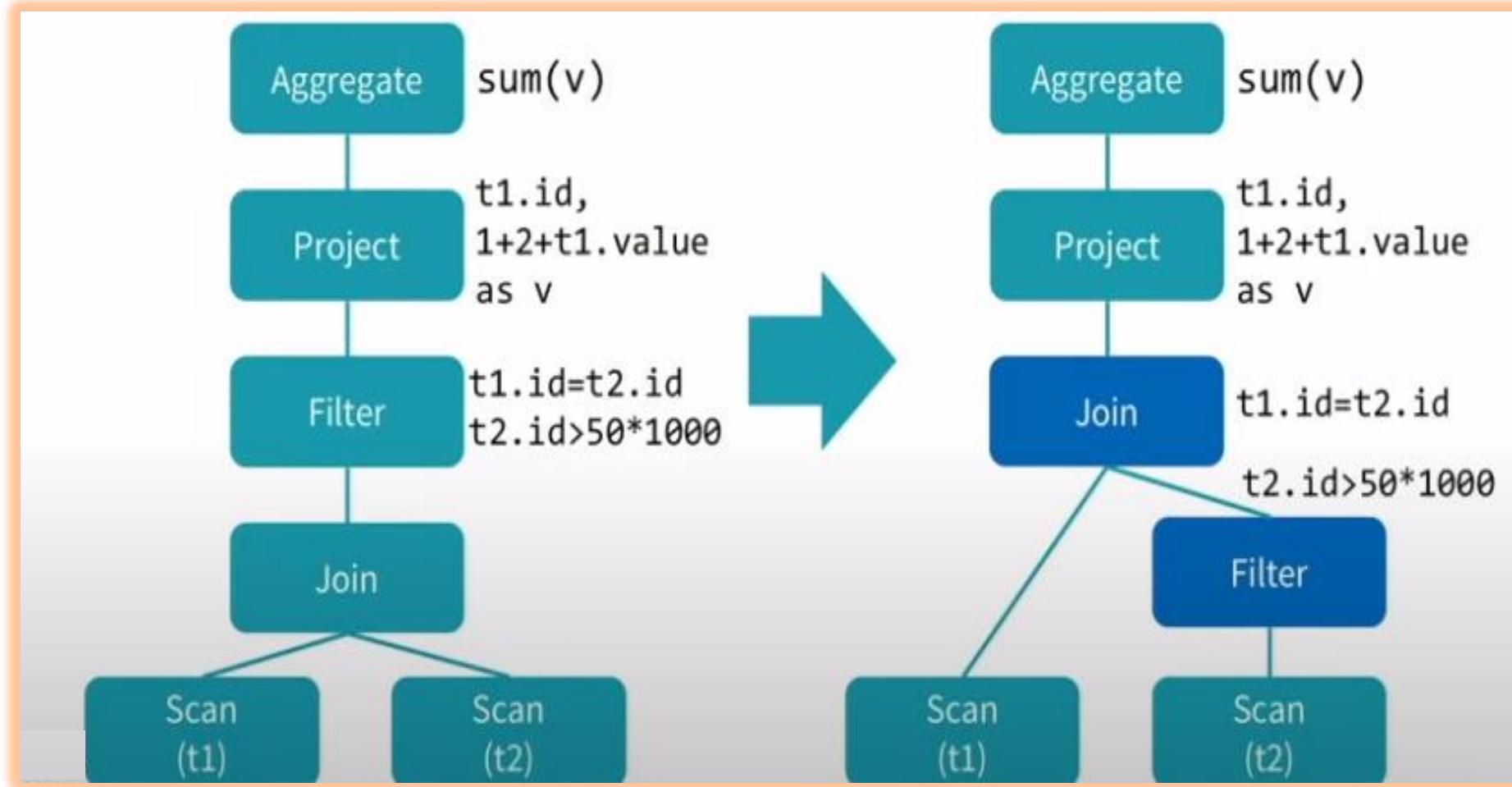


```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

Reference: databricks.com summit

Transform

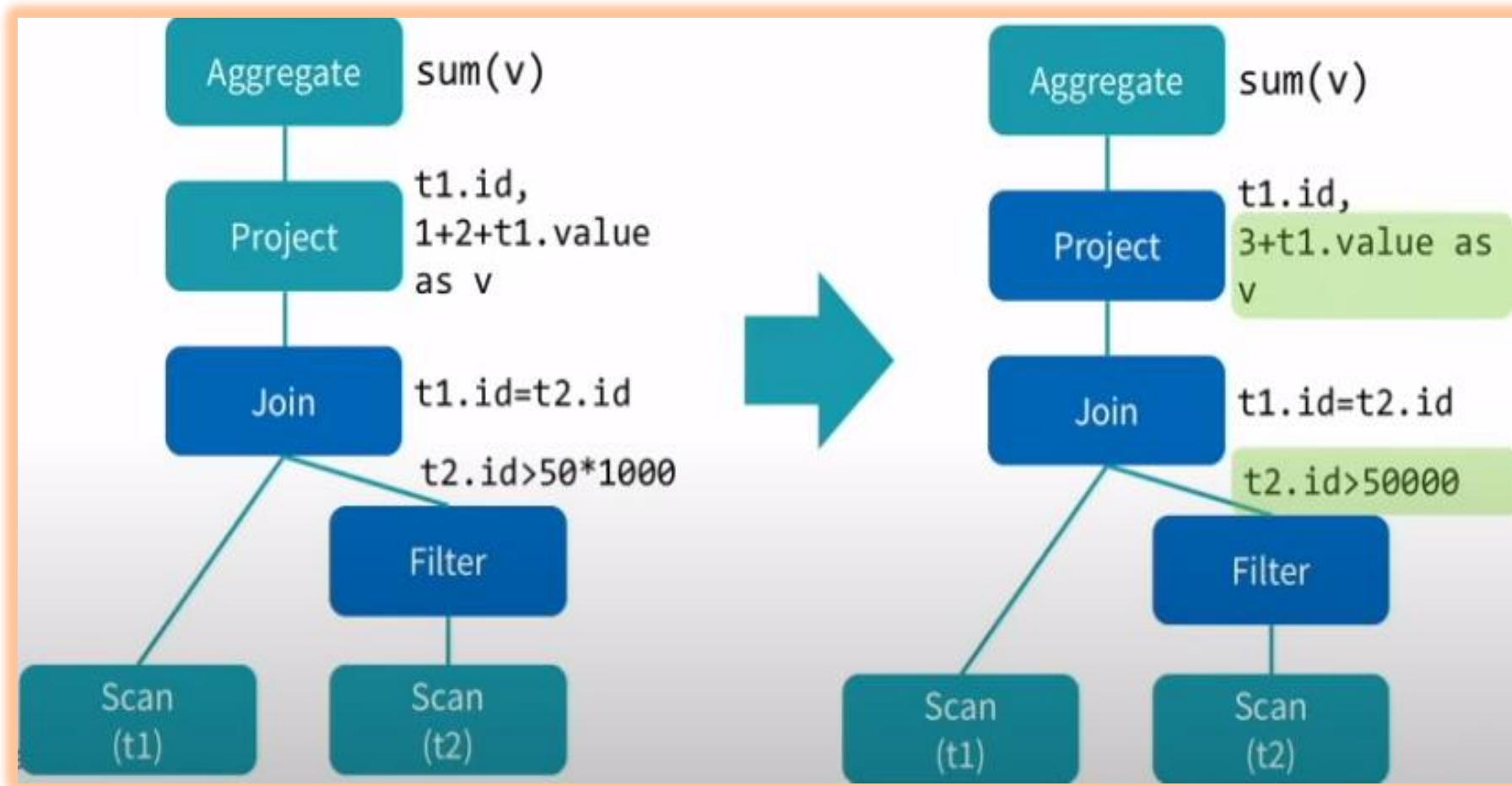
Predicate Pushdown



Reference: [databricks.com summit](https://databricks.com/summit)

Transform

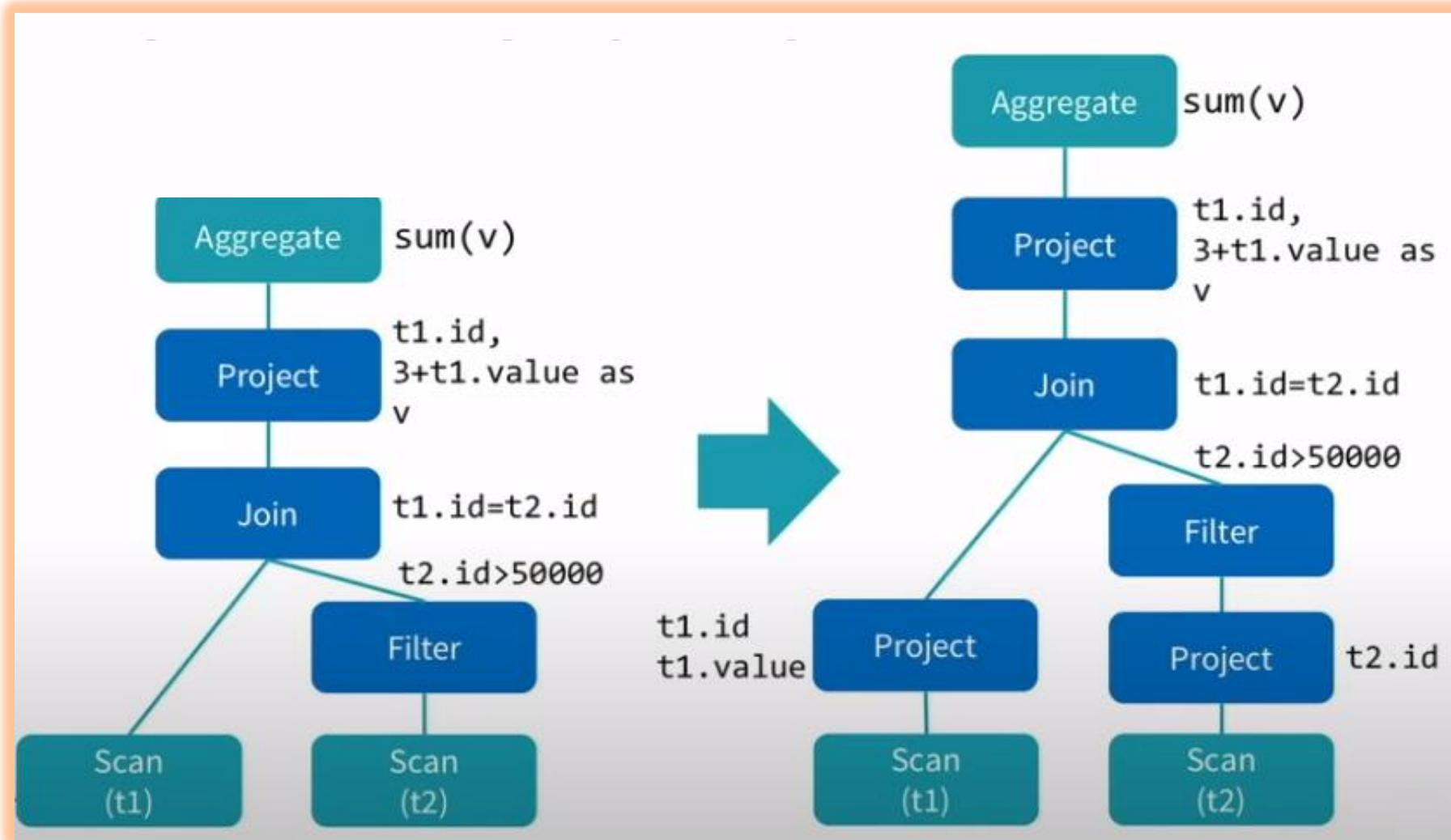
Constant Folding



Reference: [databricks.com summit](https://databricks.com/summit)

Transform

Column Pruning



Reference: [databricks.com summit](https://databricks.com/summit)

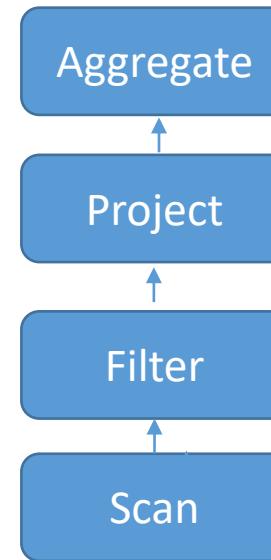
--SQL Query

```
select count(*) from orders  
where order_cust_id = 1000
```

OR

--DataFrame API

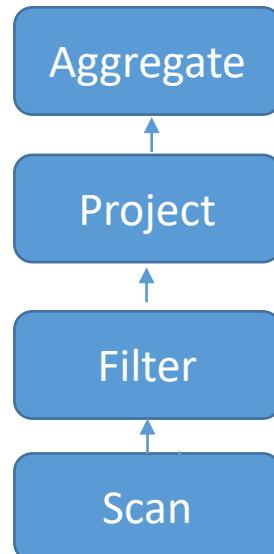
```
ord.select(ord.order_customer_id  
== 1000).count()
```



Volcano Iterator Model:

```
select count(*) from orders  
where order_cust_id = 1000
```

Each operator in the query plan would implement an iterator interface that takes in the records from the operator below it, tries to do some processing and output that record optionally to the operator above it.



```
class Filter(child: Operator, predicate: (Row => Boolean))  
  
  extends Operator {  
  
    def next(): Row = {  
  
      var current = child.next()  
  
      while (current != null && !predicate(current)) {  
  
        current = child.next()  
  
      }  
  
      return current  
    }  
  }
```

Reference: databricks.com summit

Volcano Iterator Model:

- Spark 1.6
- Each operator in the query plan would implement an iterator interface that takes in the records from the operator below it, tries to do some processing and output that record optionally to the operator above it.
- Advantages:
 - ✓ Each operator is independent from each other. So it is easy to write when we introduce a new operator and we no need to worry about how it interacts with all the other operators.
 - ✓ No need to worry about the operators before it or after it.
- Disadvantages:
 - ✓ Too many virtual function calls. Since we are agnostic to the operator that is below, we have no idea where the input data is coming from.
 - ✓ Extensive memory access: Each operator has to write the intermediate row that its trying to send to the upstream operator into memory. So there is memory read or writes bottleneck.
 - ✓ Can't take advantage of modern CPU features like SIMD, pipelining, prefetching etc.

Whole-stage Code Generation – Spark 2.0 Tungsten Engine

- Idea is to make Spark as a Compiler (Spark would try to generate code that look like hand optimized code)
- Fusing Operators together so the generator code looks like hand optimized code.
 - ✓ Identify chain of operators (“stages”)
 - ✓ Compile each stage into a single function
 - ✓ Better performance as if hand built system just to run your given query.

Aggregate (~ count)

Project (No Column to project)

Filter (~ Filter Condition)

Scan (~For Loop)



```
var count = 0
for (order_cust_id in orders) {
    if (order_cust_id == 1000) {
        count += 1
    }
}
```

Reference: databricks.com summit

Volcano vs Hand Optimized Code (Whole-stage):

- **No Function Virtual Dispatches:** In the Volcano model, to process a tuple would require calling the next() function at least once. These function calls are implemented by the compiler as virtual function dispatches. The hand-written optimized code, on the other hand, does not have a single function call.
- **Intermediate data memory vs CPU registers:** In the Volcano model, each time an operator passes a tuple to another operator, it requires putting the tuple in memory (function call stack). In the hand-written version, by contrast, the compiler (JVM JIT in this case) actually places the intermediate data in CPU registers. Again, the number of cycles it takes the CPU to access data in memory is orders of magnitude larger than in registers.
- **Modern CPU Features:** Volcano model is unable to use modern CPU features due to its complex function call graphs. Modern compilers are incredible efficient when compiling and executing simple for loops.

Benchmark

Lets check the performance with an example – Add one billion Numbers ?

```
def benchmark(version):
    start = time()
    spark.range(1000 * 1000 * 1000).select(sum("id")).show()
    end = time()
    elapsed = end-start
    print(elapsed)
```

Spark 1.6:

```
spark.conf.set("spark.sqlcodegen.wholeStage",False)
```

```
benchmark('1.6')
```

Total Time: 10.4 secs

Spark 2.0:

```
spark.conf.set("spark.sqlcodegen.wholeStage",True)
```

```
benchmark('2.0')
```

Total Time: 0.4 secs

Understanding the Execution Plan

explain(extended=False):

- Prints the (logical and physical) plans to the console for debugging purpose.
- Extended is a Boolean parameter.
default : False. If False, it prints only the physical plan.
If True, it prints all – Parsed Logical Plan, Analyzed Logical Plan, Optimized Logical Plan and Physical Plan.
- Explain function is extended for whole-stage code generation.
- When an operator has a star around it (*), whole-stage code generation is enabled. In the following example, Range, Filter, and the two Aggregates are both running with whole-stage code generation. Exchange does not have whole-stage code generation because it is sending data across the network.

Ex –

```
spark.conf.set("spark.sqlcodegen.wholeStage",True)
spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain()
```

Operator Benchmarks : Processing Cost/row in ns.

| primitive | Spark 1.6 | Spark 2.0 | |
|--------------------------------------|-----------|-----------|--------------------------------|
| filter | 15 | 1.1 | 5-30x Speedups |
| sum w/o group | 14 | 0.9 | |
| sum w/ group | 79 | 10.7 | |
| hash join | 115 | 4 | |
| Sort (8-bit entropy) | 620 | 5.3 | 10-100x Speedups |
| Sort (64-bit entropy) | 620 | 40 | |
| Sort-merge join | 750 | 700 | Shuffling still the bottleneck |
| Parquet Decoding (single int column) | 120 | 13 | 10x Speedups |

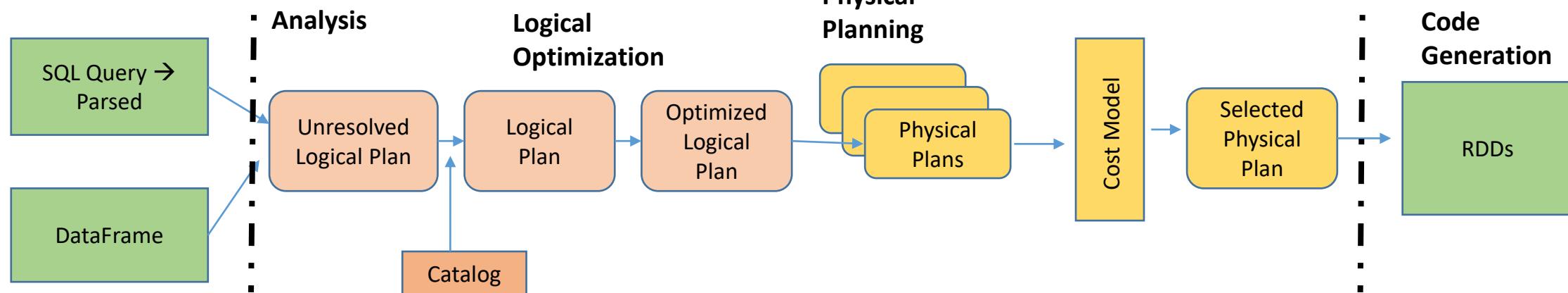
Reference: [databricks.com summit](https://databricks.com/summit)

**Let's put it all
together**

User Programs

Catalyst

Tungsten



Spark SQL Architecture

Spark SQL:

It is a module used for structured and semi structured data processing.

Flexible Data Access:

The bottom layer in the architecture is the flexible data access. Data is acquired from various input formats like CSV, JSON, Parquet, any database with JDBC, Hive etc.

DataSource API:

Used to read and store structured and semi-structured data into Spark SQL.

DataSource API then fetches the data which is then converted into a DataFrame API.

DataFrame API:

Equivalent to a relational table in SQL to perform SQL Operations.

Distributed collection of data organized into named Columns.

Data is stored in partitions.

DataFrame DSL/DataFrame SQL/HQL:

Data is processed.

Catalyst Optimizer:

It converts a query plan into optimized query plan.

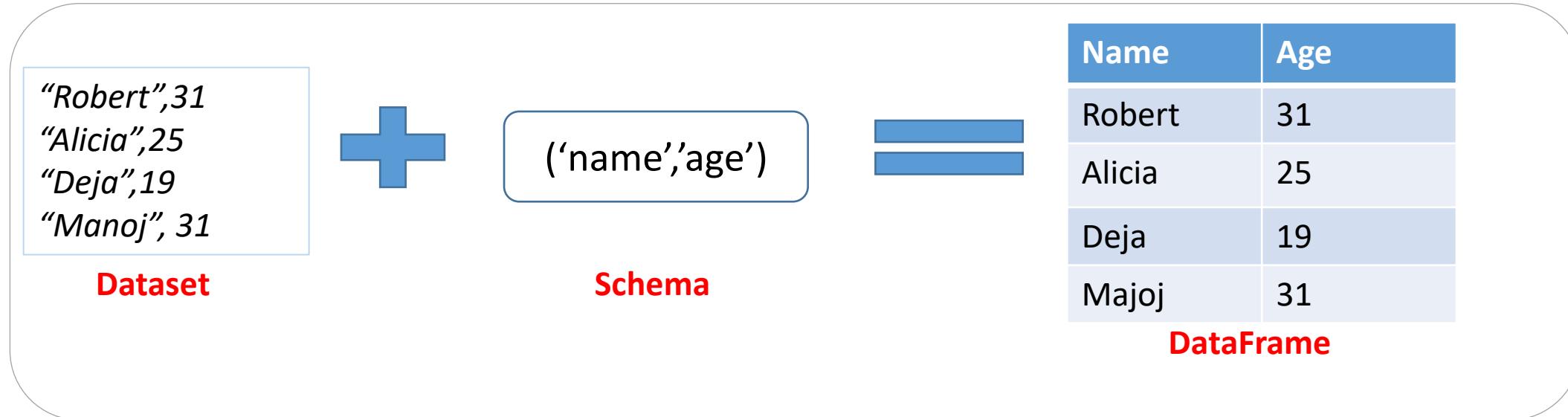
Tungsten:

Takes the optimized query plan from Catalyst and generates code and execute in the cluster in a distributed fashion.

DataFrame Fundamentals

What is a DataFrame ?

- DataFrame is a Dataset organized into named columns/rows.



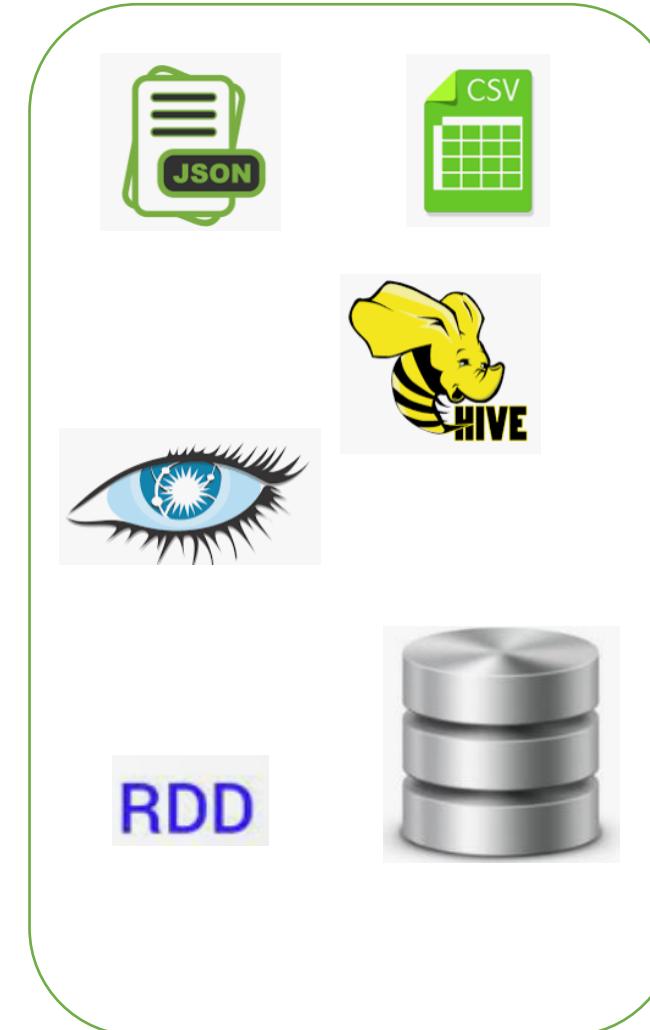
- Conceptually equivalent to RDBMS Table/Python data frame

+

Richer Optimizations.

DataFrame Sources ?

- Structured data files (CSV, JSON, AVRO, PARQUET etc)
- Hive
- Cassandra
- Python Data frame
- RDBMS Databases
- RDDs



DataFrame Features

DataFrame Features ?

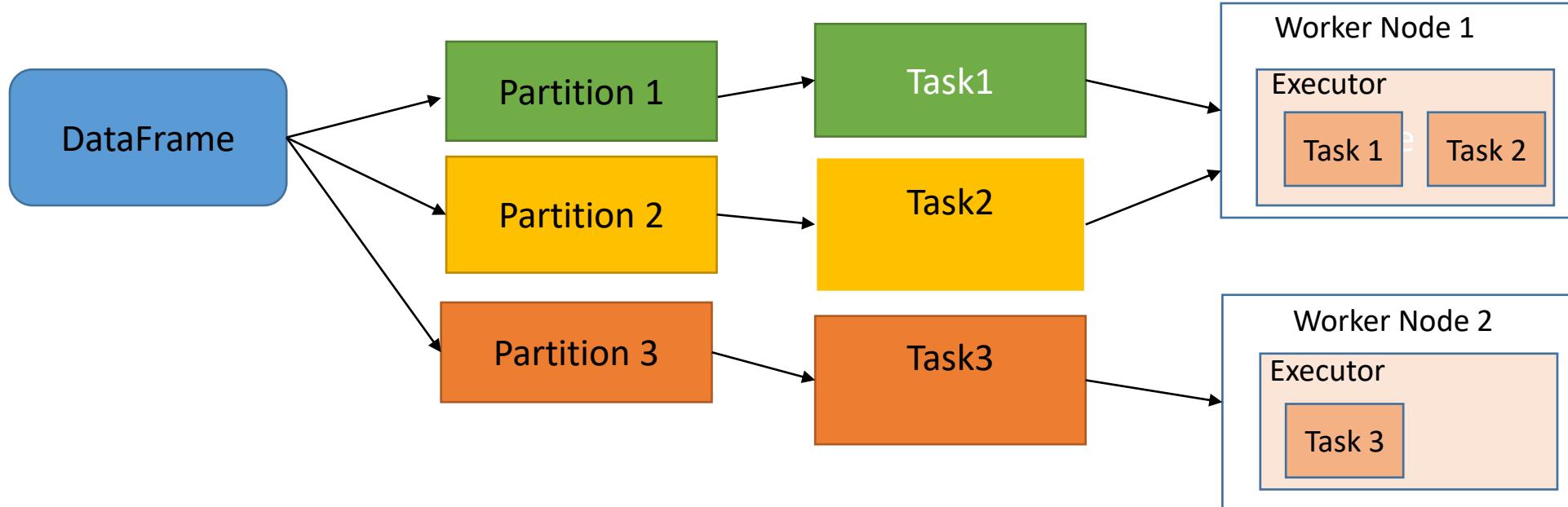
1. Distributed
2. Lazy Evaluation
3. Immutability
4. Used across the Spark Ecosystem
5. Polyglot
6. Work on Huge collection of dataset
7. Support both Structured and Semi-Structured Data

DataFrame Features - Distributed

DataFrame Features

1. DataFrame is Distributed.

- Like RDD, DataFrame is also distributed.
- Supports HA and FT.



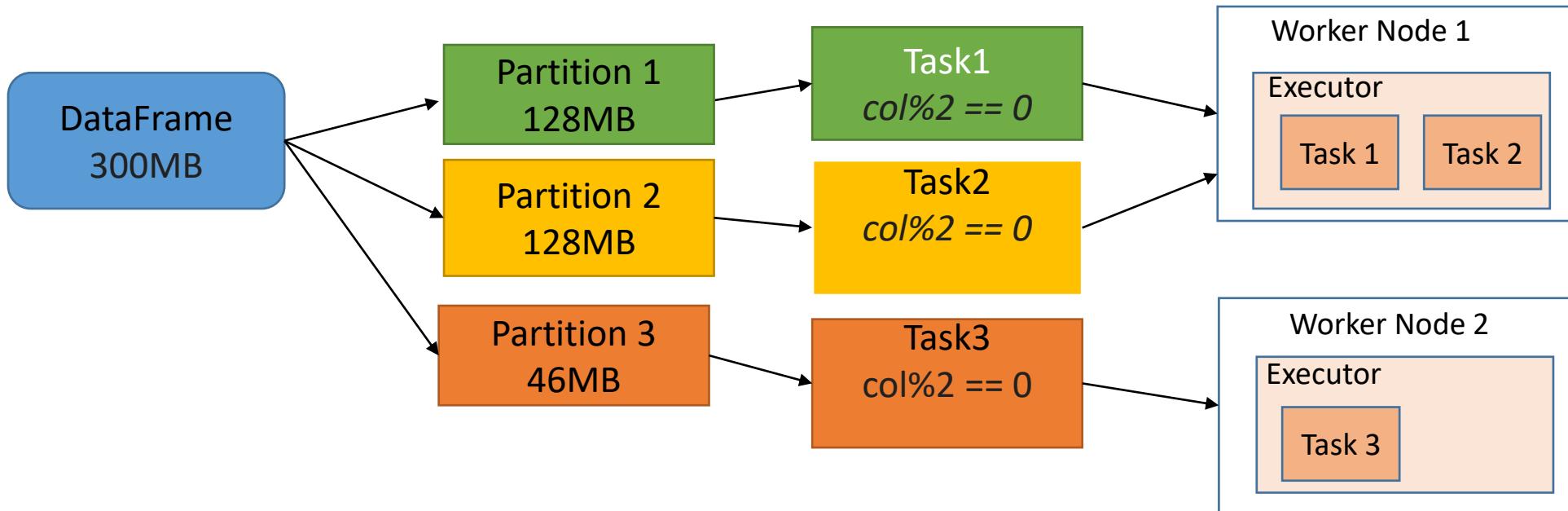
Hands-on 1



DataFrame Partitions Example.txt

1. DataFrame is Distributed.

Ex – Find out the even numbers.

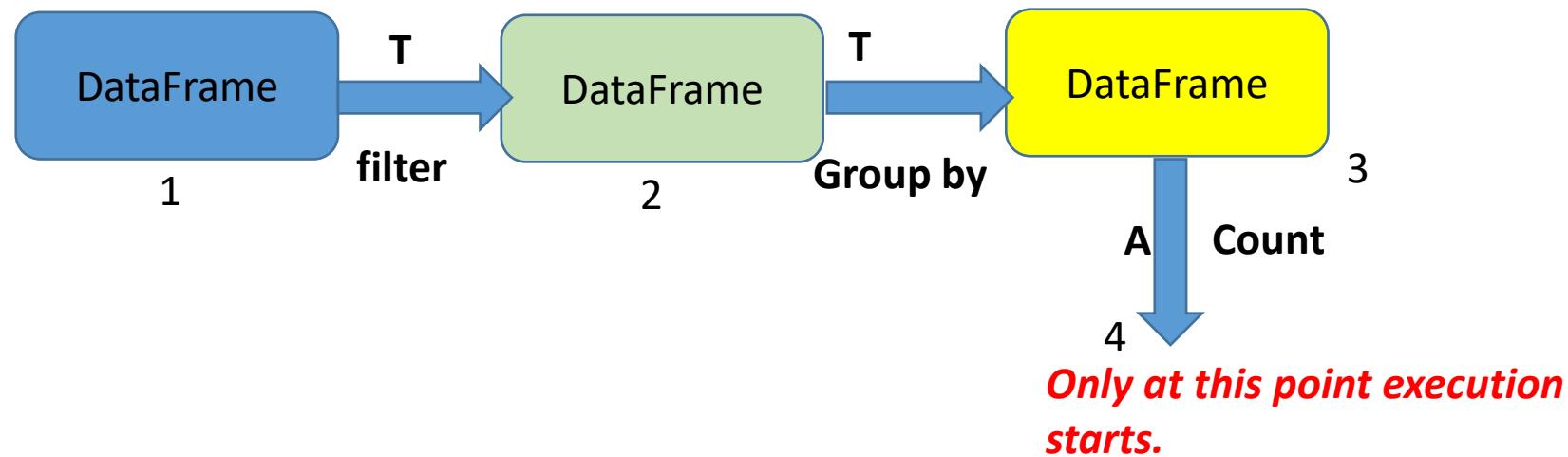


DataFrame Features – Lazy Evaluation

DataFrame Features Contd ...

2. Lazy Evaluation:

Each Transformation is a Lazy Operation. Evaluation is not started until an action is triggered.



DataFrame Features Contd ...

Example – Find out 10 sample records having a string “Robert” from a file(1TB).

With Out Lazy Evaluation:

- Step 1 Load the 1TB File
- Step 2 Perform full scan to find out all Records having “Robert”
- Step 3 Retrieve 10 Records

With Lazy Evaluation:

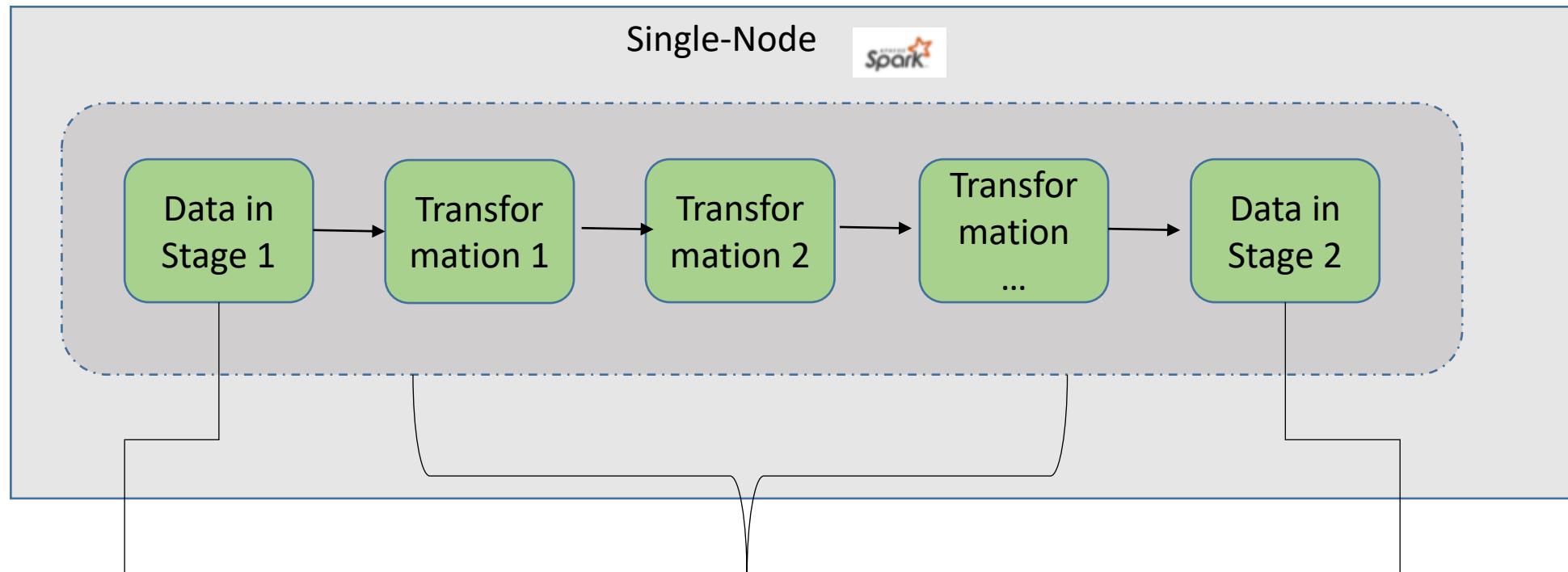
- Step 1 Wait for an Action and then Load the 1TB File
- Step 2 Filter out first 10 records having “Robert” .

DataFrame Features - Immutability

DataFrame Features Contd ...

3. Immutability:

DataFrames are considered to be “Immutable Storage”.

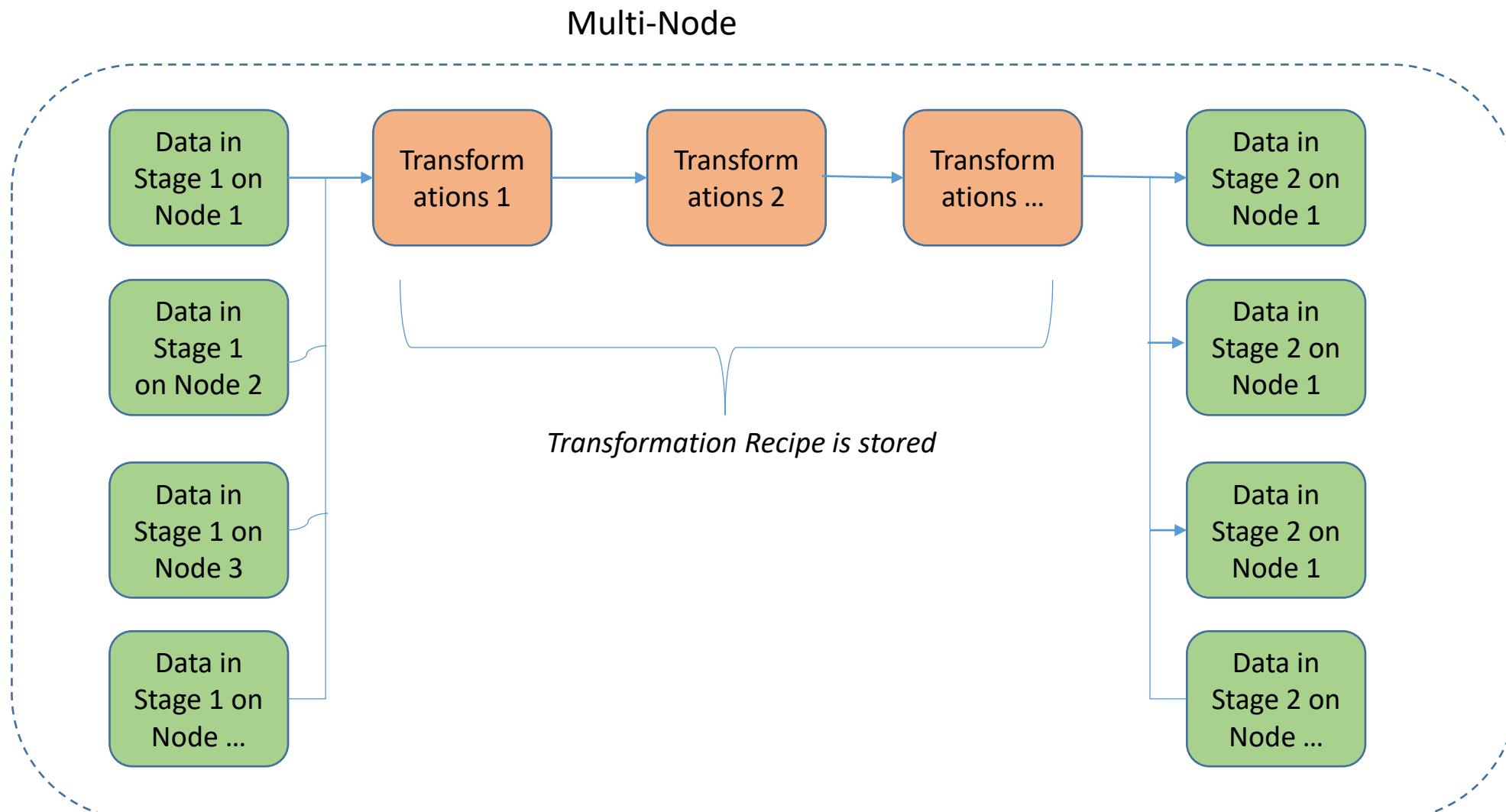


Data Stored
Immutably

The Transformation Recipe is Stored
*Ex – Logic to print all the even
numbers is $col \% 2 == 0$*

Data is not
Stored

DataFrame Features Contd ...

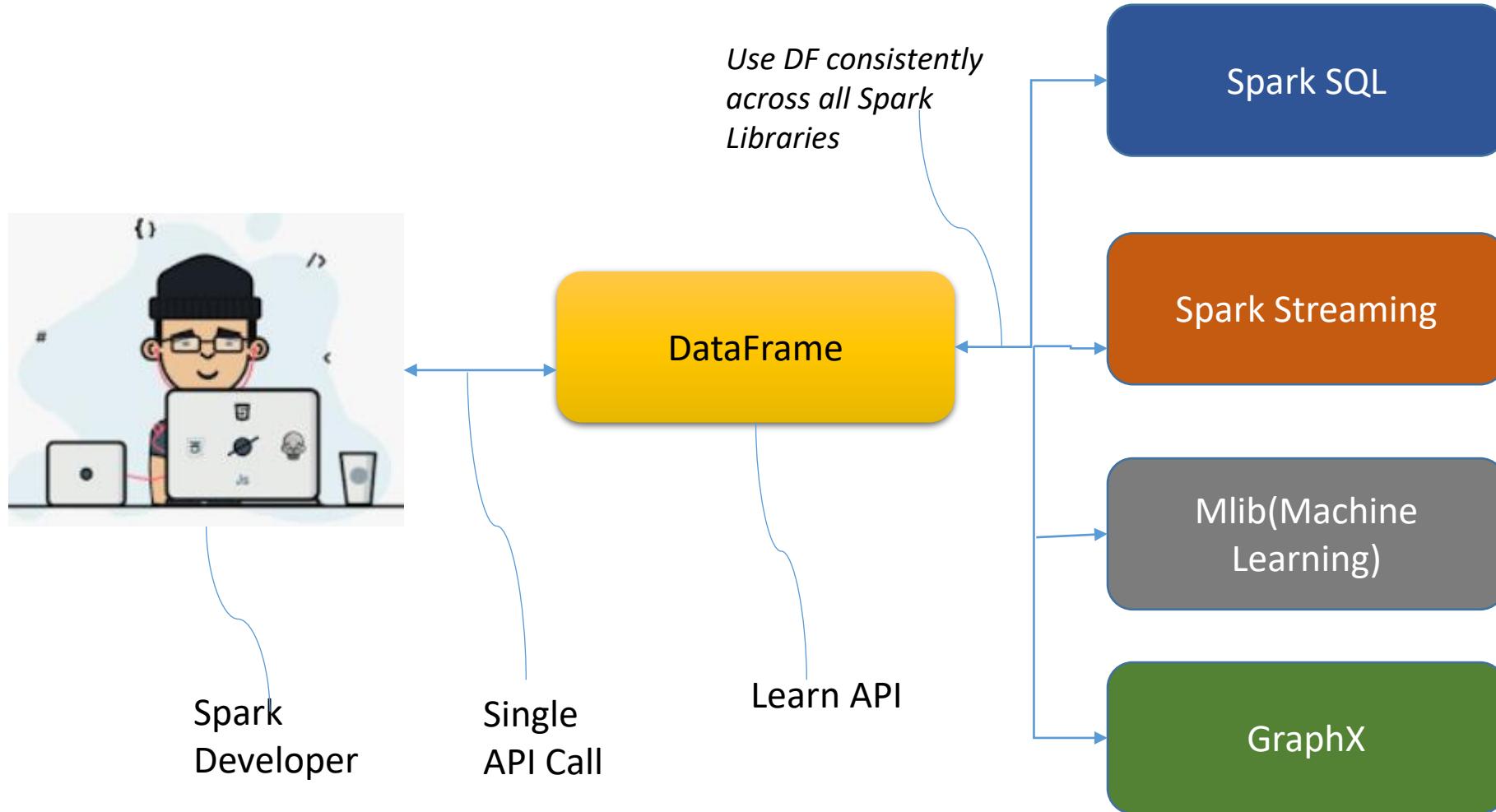


DataFrame Features – Used Across Spark Ecosystem

DataFrame Features Contd ...

4. Used across the Spark Ecosystem

DataFrame is a unified API across all libraries in Spark.



DataFrame Features Contd ...

5. Polyglot:

Support multiple Languages - Scala, Python, Java, R.

6. Works on Huge collection of dataset, feasible to work with a **wide** file.

7. Supports both Structured and Semi structured data (JSON, XML etc).

Hands-On

✓ Common functions on Data Frames

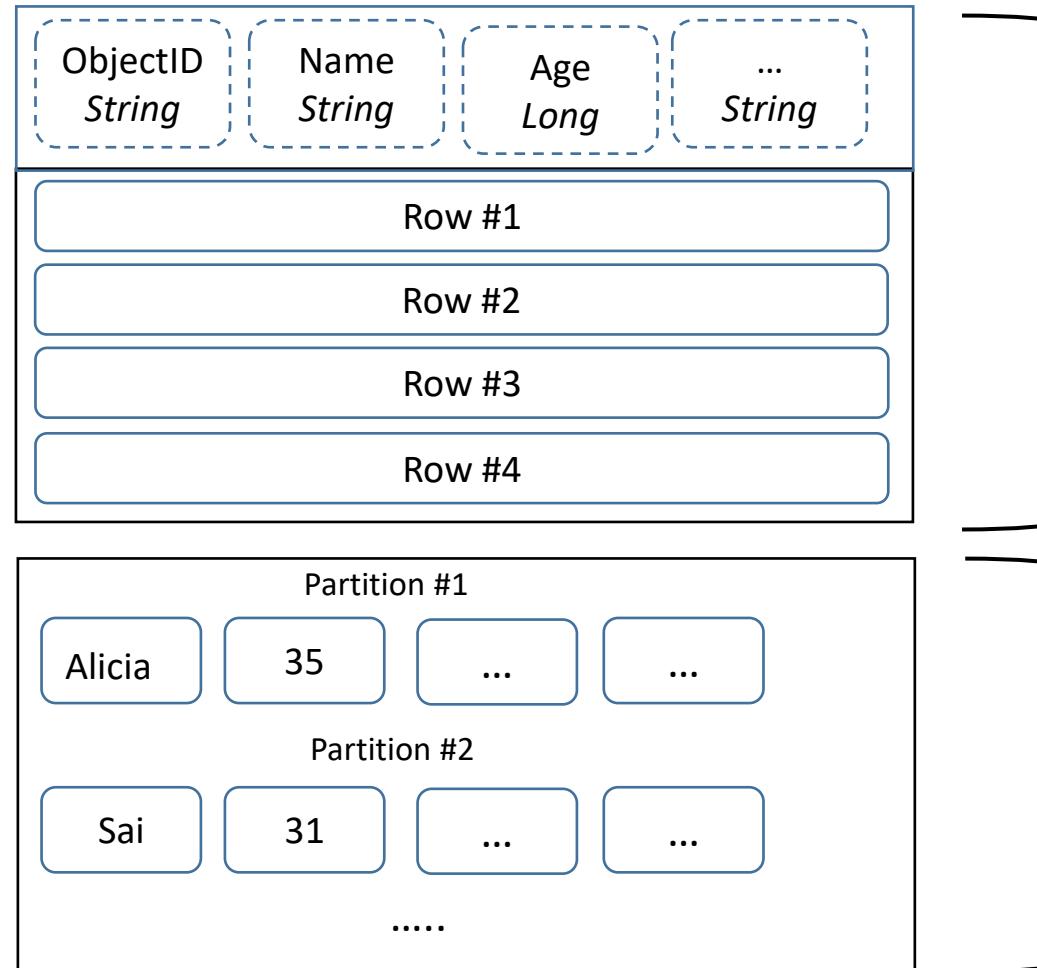
- `printSchema` – to print the column names and data types of data frame
- `show` – to preview data (default 20 records)
- `describe` – to understand characteristics of data
- `count` – to get number of records
- `collect` – to convert data frame into Array
- `type`
- `dtypes`

| Name | Age |
|--------|-----|
| Robert | 31 |
| Alicia | 25 |
| Deja | 19 |
| Majoj | 31 |

DataFrame Organization

DataFrame Organization of Data

- A DataFrame has 3 levels to organize and process its data - Schema, Storage and API.
- Schema:
 - ✓ DataFrame is Implemented as a dataset of rows.
 - ✓ Each column is named and typed.
- Storage:
 - ✓ Storage is distributed and data is stored in partitions.
 - ✓ Storage in Memory and Disc or off-heap or any of these 3 combinations.
- API
 - ✓ Used to process the data.



DataFrame Organization of Data

`StorageLevel(useDisk, useMemory, useOffHeap, deserialized, replication=1)`

- Data can be stored either in Disk or Memory or off-heap memory or any of these combinations.
- Off-Heap Memory is a segment of memory lies outside the JVM, but is used by JVM for certain use-cases. Off-Heap memory can be used by Spark explicitly as well to store serialized data-frames and RDDs.
- Data can be stored in Serialized or deserialized. Serialization is a way to convert a java object in memory to series of bits. The deserialization is the process of bringing those bits into memory as an object. Whenever we are talking about 'deserialized' RDD/DF we are always referring to RDD/DFs in memory.
- Use the replicated storage levels if you want fast fault recovery.

- **DISK_ONLY** = StorageLevel(True, False, False, False, 1)
- **DISK_ONLY_2** = StorageLevel(True, False, False, False, 2)
- **MEMORY_AND_DISK** = StorageLevel(True, True, False, False, 1)
- **MEMORY_AND_DISK_2** = StorageLevel(True, True, False, False, 2)
- **MEMORY_AND_DISK_SER** = StorageLevel(True, True, False, False, 1)
- **MEMORY_AND_DISK_SER_2** = StorageLevel(True, True, False, False, 2)
- **MEMORY_ONLY** = StorageLevel(False, True, False, False, 1)
- **MEMORY_ONLY_2** = StorageLevel(False, True, False, False, 2)
- **MEMORY_ONLY_SER** = StorageLevel(False, True, False, False, 1)
- **MEMORY_ONLY_SER_2** = StorageLevel(False, True, False, False, 2)
- **OFF_HEAP** = StorageLevel(True, True, True, False, 1)

Default
Storage

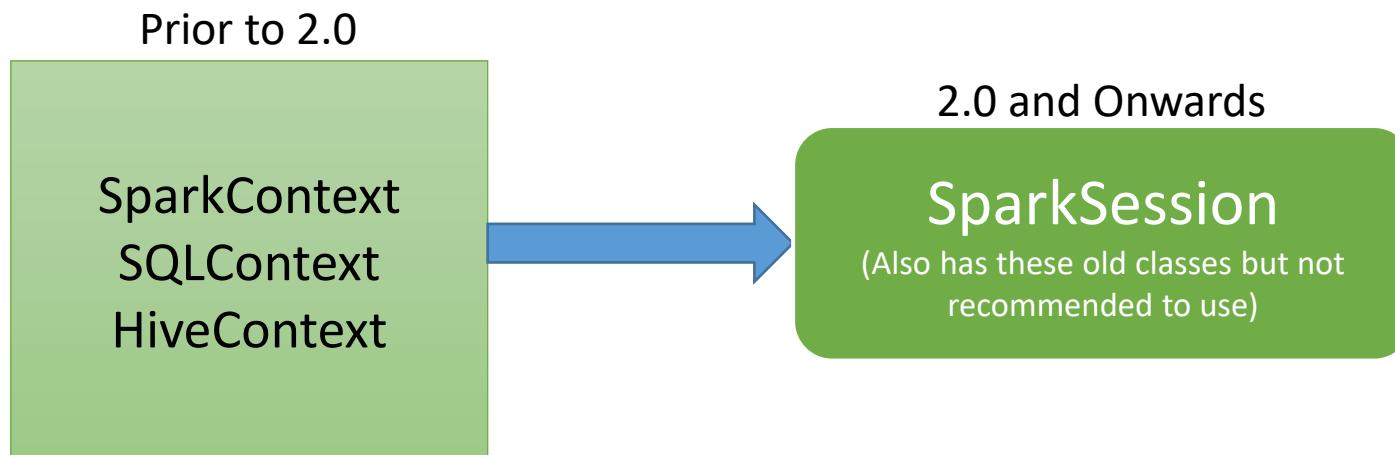
DataFrame Organization of Data

```
from pyspark import StorageLevel  
df = spark.range(10)  
df.rdd.persist().getStorageLevel()  
df.rdd.persist(StorageLevel.MEMORY_AND_DISK_2).getStorageLevel()
```

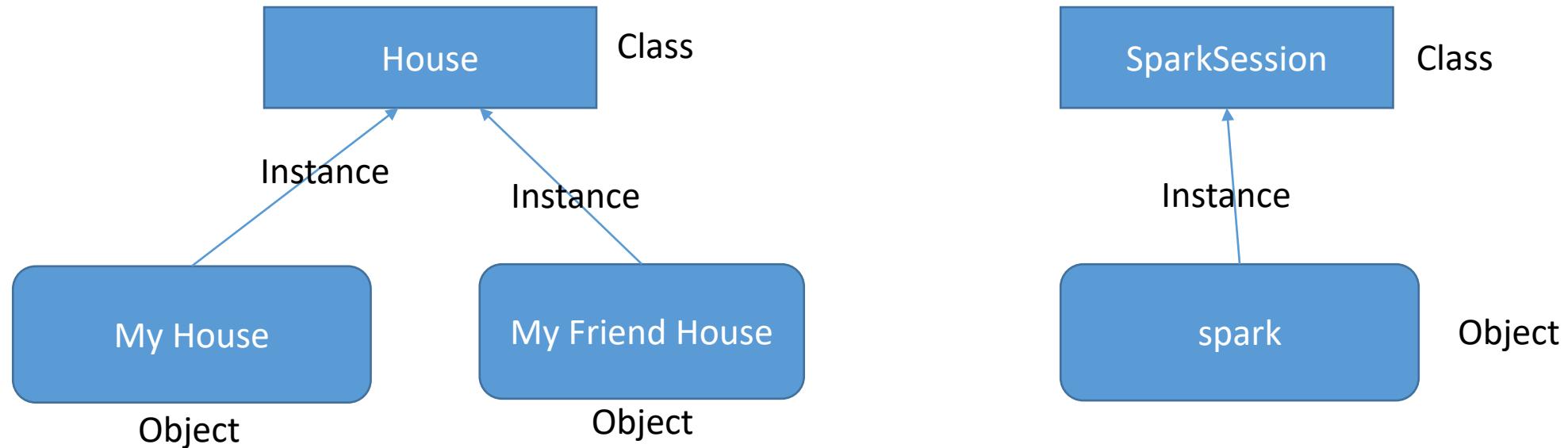
Introduction To Spark Session

SparkSession : The Entry point Spark 2.0 Onwards

- In Spark 2.0, SparkSession is the new entry point to work with RDD, DataFrame and all other functionalities.
- Prior to 2.0, SparkContext used to be an entry point.
- Almost all the APIs available in SparkContext, SQLContext, HiveContext are now available in SparkSession.
 - SparkContext:** Entry point to work with RDD, Accumulators and broadcast variables (< Spark 2.0).
 - SQLContext:** Used for initializing the functionalities of Spark SQL (< spark 2.0).
 - HiveContext:** Super set of SQLContext (< spark 2.0).
- By Default, Spark Shell provides a “spark” object which is an instance of SparkSession class.



Spark Session : The Entry point Spark 2.0 Onwards



Spark Session : Spark Object & spark-submit

Spark Session : Create

```
from pyspark.sql import SparkSession  
  
spark = SparkSession \  
    .builder \  
    .master('yarn') \  
    .appName("Python Spark SQL basic example") \  
    .getOrCreate()
```

Master can be yarn, mesos, Kubernetes or local(x) , x > 0

How to Run :

1. Organize the folders and create a python file under bin folder.
2. Write above codes in the .py file.
3. Execute the file using spark-submit command.

```
spark2-submit \  
/devl/example1/src/main/python/bin/basic.py
```

Spark Session : spark-submit

Spark-submit is a utility to run a pyspark application job by specifying options and configurations.

```
spark-submit \
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key<=<value> \
--driver-memory <value>g \
--executor-memory <value>g \
--executor-cores <number of cores> \
--jars <comma separated dependencies> \
--packages <package name> \
--py-files \
<application> <application args>
```

Spark Session : spark-submit

--master : Cluster Manager (yarn, mesos, Kubernetes, local, local(k))

local – Use local to run locally with one worker node.

local(k) – Specify k with the number of cores you have locally, this runs application with k worker threads.

--deploy-mode: Either cluster or client

Cluster: Driver runs on one of the worker nodes and you can see the code as a driver on the spark UI of your application. We cant see the logs on the terminal. Logs available only in the UI or the yarn CLI.

yarn logs -applicationId application_1622930712080_16253

Mainly used for production jobs.

Client: Driver runs locally where we submit the application.

See the logs on the terminal.

Mainly used for interactive or debugging purpose.

Spark Session : spark-submit

--conf: We can provide runtime configurations, shuffle parameters, application configurations using --conf.

Ex: --conf spark.sql.shuffle.partitions = 300

This configures the number of partitions that are used when shuffling data for joins or aggregations.

<https://spark.apache.org/docs/latest/sql-performance-tuning.html>

--conf spark.yarn.appMasterEnv.HDFS_PATH="practice/retail_db/orders"

We can set environment variables like this when spark is running on yarn.

<https://spark.apache.org/docs/latest/running-on-yarn.html#configuration>

Spark Session : spark-submit

--driver-memory : Amount of memory to allocate for a driver (Default: 1024M).

--executor-memory : Amount of memory to use for the executor process.

--executor cores : Number of CPU cores to use for the executor process.

Spark Session : spark-submit

--jars: Dependency .jar files.

Ex : --jars /devl/src/main/python/lib/ojdbc7.jar, fil2.jar, file3.jar

--packages: Pass the dependency packages.

Ex : --packages org.apache.spark:spark-avro_2.11:2.4.4

--py-files: Use --py-files to add .py and .zip files. File specified with --py-files are uploaded to the cluster before it run the application.

Ex - --py-files file1.py, file2.py,file3.zip

Spark Session : spark-submit

```
spark-submit \
--master "yarn" \
--deploy-mode "client"\
--conf spark.sql.shuffle.partitions = 300 \
--conf spark.yarn.appMasterEnv.HDFS_PATH="practice/retail_db/orders"
--driver-memory 1024M \
--executor-memory 1024M \
--num-executors 2 \
--jars --jars /devl/src/main/python/lib/ojdbc7.jar, fil2.jar, file3.jar \
--packages org.apache.spark:spark-avro_2.11:2.4.4 \
--py-files file1.py, file2.py,file3.zip \
/dev/example1/src/main/python/bin/basic.py arg1 arg2 arg3
```

Spark Session : Commonly Used Functions

Spark Session : Commonly Used Functions

- **version:**
Returns Spark version where your application is running.
- **range():**
This creates a DataFrame with a range of values.
- **createDataFrame() :**
This creates a DataFrame from a collection(list, dict), RDD or Python Pandas.
- **sql() :**
Returns a dataframe representing the result of a given query.
- **table():**
Returns the specified table as dataframe.
- **sparkContext:**
Returns sparkContext
- **conf():**
Runtime configuration (get and set).
Ex- `spark.sql.shuffle.partitions`
- **read():** Used to load a dataframe from external storage systems.
- **udf() :** Dedicated section for this.
- **newSession()**
- **stop() :** Stop the underlying SparkContext.
- **catalog()**

SparkSession - Version

Spark Session : version Method

- **version:** Returns Spark version where your application is running.

```
spark.SparkContext.version  
sc.version
```

SparkSession - range

Spark Session : range() Method

- **range():**

This creates a DataFrame with a range of values.

Ex- 1 :

```
df = spark.range(1,10,2)
```

Ex-2:

```
df =spark.range(10)
```

SparkSession - createDataFrame

Spark Session : createDataFrame() Method

- **createDataFrame()** :

This creates a DataFrame from a collection(list, dict), RDD or Python Pandas.

Ex-1 : Using Python List

```
Ist = ('Robert',35),('James',25)
spark.createDataFrame(data=Ist),
df = spark.createDataFrame(data=Ist,schema=('Name','Age')) ##With Out Schema
df = spark.createDataFrame(data=Ist,schema=('Name','Age')) ##With Schema
```

Ex-2 : Using Python Dict

```
dict = {"name":"robert","age":25}, {"name" : "james","age" : 31}
df = spark.createDataFrame(dict)
```

Ex-3: Using RDD

```
rdd = sc.parallelize(Ist)
df = spark.createDataFrame(data=rdd,schema=('name string, age long'))
```

Ex-4: Using Row in RDD

```
#Row is used to create row Object using named arguments.
from pyspark.sql import Row
rdd = sc.parallelize((Row(name='James',age=31),Row(name='Robert',age=55)))
df = spark.createDataFrame(data=rdd)
```

Spark Session : createDataFrame() Method

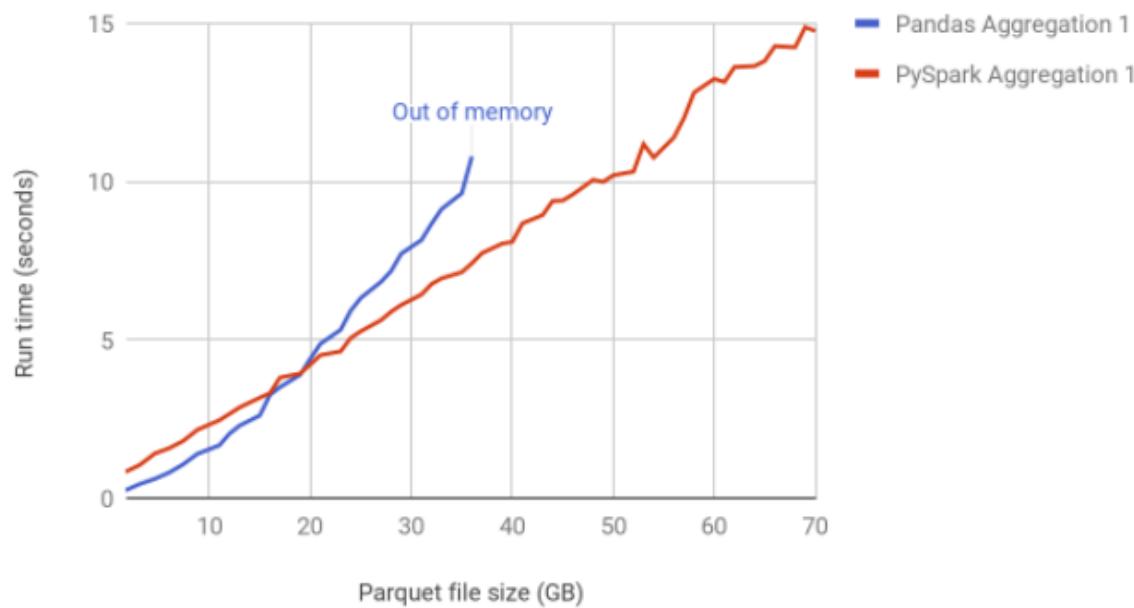
- Ex-5 : Using Python Pandas DataFrame

Pandas dataframe is a two dimensional structure with named rows and columns. So data is aligned in a tabular fashion in rows and columns

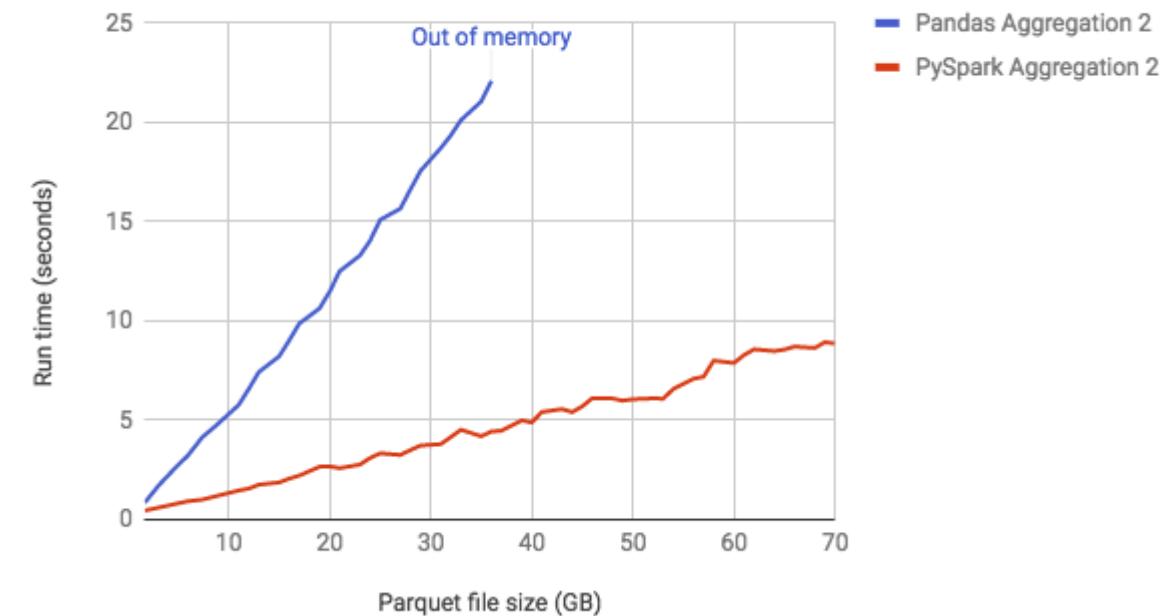
```
import pandas as pd  
data = ('tom', 10), ('nick', 15), ('juli', 14))  
df_pandas = pd.DataFrame(data,columns=['Name','Age'])  
df = spark.createDataFrame(data=df_pandas)
```

Performance: Pandas DataFrame Vs Spark DataFrame

Pandas VS PySpark: aggregation query 1



Pandas VS PySpark: aggregation query 2



Source: www.databricks.com

Spark Session : sql() Method

- **sql()**:

Returns a dataframe representing the result of a given query.

Ex-1:

```
Ist1 = ('Robert',35),('James',25)  
Ist2= ('Robert',101),('James',102))
```

```
df_emp = spark.createDataFrame(data=Ist1,schema=('EmpName','Age'))  
df_emp.createOrReplaceTempView("dept")
```

```
df_dept = spark.createDataFrame(data=Ist2,schema=('EmpName','DeptNo'))  
df_dept.createOrReplaceTempView("dept")
```

```
df_joined = spark.sql (""" select e.name,e.age,d.dept from emp e join dept d where e.name = d.name """)
```

createOrReplaceTempView("table1") //Creates the view in the current database and valid for only one session.

createOrReplaceGlobalTempView("table1") //Creates the views in global_temp database.
// Valid across all sessions of an application.

Ex-2: Using Hive Table

```
df = spark.sql (""" select * from emp """)
```

Spark Session : table() Method

- **table()** :

Returns the specified table as dataframe.

Ex-1:

```
Ist1 = ('Robert',35),('James',25)
df_emp = spark.createDataFrame(data=Ist1,schema=('EmpName','Age'))
df_emp.createOrReplaceTempView("emp")

df_op = spark.table("emp")
sorted(df_op.collect()) == sorted(df_emp.collect())
```

Spark Session : conf() Method

- **conf():**

We can provide runtime configurations, shuffle parameters, application configurations using –conf or spark.conf().

Ex-1: System Defined

```
spark.conf.get("spark.sql.session.timeZone"  
spark.conf.get('spark.sql.shuffle.partitions')  
spark.conf.set('spark.sql.shuffle.partitions',300) # configures the number of partitions that are used  
when shuffling data for joins or aggregations.
```

<https://spark.apache.org/docs/latest/sql-performance-tuning.html>

Ex-2 : Spark Running on YARN Environment Variables

```
spark.conf.set('spark.yarn.appMasterEnv.HDFS_PATH','practice/retail_db/orders')  
--conf spark.yarn.appMasterEnv.HDFS_PATH #From spark-submit
```

<https://spark.apache.org/docs/latest/running-on-yarn.html#configuration>

Spark Session : read() Method

- **read():** Interface used to load a dataframe from external storage systems.

Load a csv File:

Load a Text File:

Load a orc File:

Load a Parquet File:

Load a json File:

Load a avro File:

Read a hive Table:

Read a JDBC:

Spark Session : read – csv

Load a csv File:

```
Ex-1: df = spark.read.load(path='practice/retail_db/orders', format='csv', \  
schema=('order_id int,order_date string,order_customer_id int,order_status string'))
```

```
Ex-2: df = spark.read.load(path='practice/retail_db/orders', format='csv',inferSchema=True)
```

```
Ex-3: df = spark.read.load(path='practice/retail_db/orders', format='csv',header=True)
```

Ex-4:

```
df=spark.read.load('practice/retail_db/testSpace.txt',format='csv',sep=',',ignoreLeadingWhiteSpace=True,ignoreTrailingWhiteSpace=True)
```

Spark Session : read – text

Load a text File:

- Use text file where there is fixed length.
- Default field name is ‘value’.
- Also you may load into rdd.
- Convert to dataframe using toDF() and Row.

```
df = spark.read.load('practice/retail_db/orders',format='text')
```

--Read the whole text file into a single line.

```
df =spark.read.load('practice/retail_db/orders',format='text', wholeText=True)
```

Spark Session : read – orc/parquet

- **Load a orc File**

```
df = spark.read.load('practice/retail_db/orders_orc',format='orc')
```

- **Load a Parquet File**

```
df = spark.read.load('practice/retail_db/orders_parquet',format='parquet')
```

CSV, JSON and AVRO are Row-based File formats. Sample data in CSV:

ID,FIRST_NAME,AGE

1, Matthew, 19

2, Joe, 25

ORC,PARQUET are Column-based File formats.

ID/INT/3:1,2

FIRST_NAME/STRING/11:Matthew,Joe

AGE/INT/6:19,25

Spark Session : read – json

Load a json File:

```
df = spark.read.load('practice/retail_db/orders_json',format='json')
```

Spark Session : read – avro

Load a avro File:

- Avro Is a third party file format. We need to import its package or jar file while launching a spark-submit application or pyspark shell. spark by default does not support it.

```
pyspark2 --master yarn \  
--packages org.apache.spark:spark-avro_2.11:2.4.4
```

```
df = spark.read.load('practice/retail_db/orders',format='avro')
```

Spark Session : read() Method

Load a Hive Table: If Hive and Spark are integrated, we can create data frames from data in Hive tables or run Spark SQL queries against it.

```
spark.sql("SELECT * FROM <db>.<table_name>").show()  
spark.table("<db>.<table_name>").show()
```

Spark Session : read() Method

Load a JDBC Table:

- Make sure JDBC jar file is registered using –packages or –jars while launching pyspark or spark-submit
- Typical jdbc files are located at /usr/share/java folder. You may keep it there or copy it to your project lib folder.

pyspark2 ---jars <jdbc driver jar file?

Ex-1: Table

```
df= spark.read.format("jdbc") \
.option("url", "jdbc:oracle:thin:@xxxx-xxx-xxxx:1521/xxx") \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("dbtable","ORDERS" ) \
.option("user", "someUser") \
.option("password", "somePsw") \
.load()
```

Spark Session : read() Method

Ex-2: Query

```
df= spark.read.format("jdbc") \
.option("url", "jdbc:oracle:thin:@xxxx-xxx-xxxx:1521/xxx") \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("dbtable","(SELECT * FROM T_EMP WHERE ID=1) query" ) \
.option("user", "someUser") \
.option("password", "xxx") \
.load()
```

Spark Session : read() Method

Ex-3: Partition

Partitioning can be done only on numeric or date fields. If there is no numeric field generate it. For ex- Use ROWNUM to generate dummy numeric fields in Oracle Database. Define partitionColumn, numPartitions, lowerBound, upperBound.

```
df= spark.read.format("jdbc") \
.option("url", "jdbc:oracle:thin:@xxxx-xxx-xxxx:1521/xxx") \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("dbtable","ORDERS" ) \
.option("partitionColumn","ORDER_ID") \
.option("lowerBound", "500") \
.option("upperBound", "1000") \
.option("numPartitions","5") \
.option("user", "someUser") \
.option("password", "somePassword") \
.load()
```

Spark Session : read() Method

lowerBound=500

upperbound=1000

numPartitions=5

So stride is = (upperbound-lowerbound)/numPartitions = 100.

Total Records in the Table = 1509

Partition 1: (599 Records)

First 499 Records

+

Select * from orders where order_id between (500,599) → 100 records

Partition 2: (100 Records)

Select * from orders where order_id between (600,699) → 100 records

Partition 3: (100 Records)

Select * from orders where order_id between (700,799) → 100 records

Partition 4: (100 Records)

Select * from orders where order_id between (800,899) → 100 records

Partition 5: (610 Records)

Select * from orders where order_id >=900

Spark Session : read() Method

Ex-4: Partition with out Numeric field

```
df= spark.read.format("jdbc") \
.option("url", "jdbc:oracle:thin:@xxxx-xxx-xxxx:1521/xxx") \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("dbtable", "(select t1.* , cast(ROWNUM as number(5)) as num_rows from (select * from orders) t1) oracle_table1") \
.option("partitionColumn", "num_rows") \
.option("lowerBound", "500") \
.option("upperBound", "1000") \
.option("numPartitions","10") \
.option("user", "someUser") \
.option("password", "somePassword") \
.load()
```

Spark Session : spark.udf

- UDFs are the User Defined Functions. Spark UDFs are similar to RDBMS User Defined Functions.
- If there is a need of a function and pyspark build-in features don't have this function, then you can create a udf and use it in DataFrames and Spark SQLs.
- UDFs are error-prune and so should be designed carefully. First check if similar function is available in pyspark functions library(pyspark.sql.functions). If not designed properly, we would come across optimization and performance issues.
- We can use UDFs both in DataFrame and Spark SQL.
 1. For Spark SQL, create a python function/udf and register it using `spark.udf.register` method.
 2. For DataFrame, create a udf by wrapping under `@udf` or `udf()` function.

Ex – 1 (Create a udf, use it in DataFrame and register for spark sql)

```
import string
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType,IntegerType

@udf(returnType=StringType())
def initCap(str):
    finalStr=""
    ar = str.split(" ")
    for word in ar:
        finalStr= finalStr + word(0:1).upper() + word(1:len(word)) + " "
    return string.strip(finalStr)
```

DataFrame:

```
df.select(df.name, initCap(df.name)).show()
```

Spark Sql:

```
spark.udf.register("initcap1", initCap)
spark.sql(""" select emp_name, initcap1(emp_name) from default.emp """).show()
```

Ex-2: (Using Python Function and Register in Spark sql)

```
def convertCap(str):
    finalStr=""
    ar = str.split(" ")
    for word in ar:
        finalStr= finalStr + word(0:1).upper() + word(1:len(word)) + " "
    return string.strip(finalStr)
```

Spark Sql:

```
spark.udf.register(" initcap ", convertCap)
Spark.sql(" " " select emp_name,initcap(emp_name) from default.emp " " ")
```

Ex-3: (Using Python Lambda Function and Use it in Spark Sql)

```
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import udf
slen = udf(lambda s: len(s), IntegerType())
spark.udf.register("slen", slen)
spark.sql("SELECT slen('test')").collect()
```

Spark Session : newSession()

spark.newSession():

- Returns a new SparkSession as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache.
- Ex- The registered udfs will not be visible to the new session.

Ex- 1 (Using udf) → Different SparkContext

```
new_spark = spark.newSession()
```

```
import string
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType,IntegerType

@udf(returnType=StringType())
def initCap(str):
    finalStr=""
    ar = str.split(" ")
    for word in ar:
        finalStr= finalStr + word(0:1).upper() + word(1:len(word)) + " "
    return string.strip(finalStr)
```

```
spark.udf.register("initcap1", initCap)
spark.sql(""" select emp_name, initcap1(emp_name) from default.emp """).show()
```

```
spark.sql(""" select emp_name, initcap1(emp_name) from default.emp """).show()
```

Ex-2: (Using Table data) → Shared SparkContext

```
new_spark = spark.newSession()
```

```
spark.sql (""" create table student(name int) """);  
spark.sql (""" insert into student values (1) """);
```

```
spark.sql (""" select count(*) from student """).show()  
new_spark.sql (""" select count(*) from student """).show()
```

Spark Session : stop()

spark.stop(): To stop the underlying spark context.

Spark Session : catalog()

- Catalog is introduced in spark 2.0 which is a standard API for accessing metadata in Spark SQL.
- This works both for Spark Sql and Hive Metadata.
- Below are different methods in catalog for extracting important information.

Database Functions:

currentDatabase
listDatabases
setCurrentDatabase

View Functions:

dropGlobalTempView
dropTempView

Table Functions:

listColumns
listTables
cacheTable
isCached
uncacheTable
clearCache
recoverPartitions
refreshTable
refreshByPath

Function based functions:

listFunctions
registerFunction (= spark.udf.register)

Data Types

Supported Data Types:

Below are some of the important data types supported in both Data Frame and Spark SQL. For full list, please visit <https://spark.apache.org/docs/latest/sql-ref-datatypes.html>.

```
from pyspark.sql.types import *
```

| Numeric Types | |
|---------------------|---|
| IntegerType() | 4-byte signed integer numbers |
| FloatType() | 4-byte single-precision floating point numbers |
| DoubleType() | 8-byte double-precision floating point numbers |
| String Types | |
| StringType() | Character String Values |
| VarcharType(length) | Variant of StringType with Length limitation |
| CharType(length) | Variant of VarcharType with Fixed Length |
| Boolean Types | |
| BooleanType () | Boolean Values (True or False. Also can have Null Values) |
| Binary Type | |
| BinaryType () | Byte Sequence Values |

Supported Data Types:

| Date Types | |
|-----------------|---|
| TimestampType() | year, month, day, hour, minute, second, time zone |
| DateType () | year, month, day |

Complex Type

ArrayType (elementType,containsNull)

MapType (keyType, valueType,valueContainsNull)

StructType (fields)

Supported Data Types:

```
from pyspark.sql.types import StructType,StructField, StringType, IntegerType, DateType
```

Ex-1

```
schema = StructType([
    StructField("name",StringType(),True),
    StructField("id", IntegerType(),True),
])
```

```
data=(("James",1),
      ("Robert",2),
      ("Maria",3)
     )
```

```
df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show(truncate=False)
```

Supported Data Types:

Ex-2 (Map Types)

```
schema=MapType(StringType(),StringType())
```

```
schema = StructType((  
    StructField('name', StringType(), True),  
    StructField('properties', MapType(StringType(),StringType()),True)  
))
```

```
d = ( ('James',{'hair':'black','eye':'brown'}),  
      ('Michael',{'hair':'brown','eye':None}),  
      ('Robert',{'hair':'red','eye':'black'})  
 )
```

```
df_map= spark.createDataFrame(data=d, schema = schema)  
df_map.printSchema()  
df_map.show(truncate=False)  
df_map.select(df_map.properties).show(truncate=False)  
df_map.select(df_map.properties('eye')).show(truncate=False)
```

Supported Data Types:

Ex-3 (Array Types)

```
schema=ArrayType(IntegerType())
```

```
schema = StructType((  
    StructField('name', StringType(), True),  
    StructField('mobileNumbers', ArrayType(IntegerType()),True)  
))
```

```
d = ( ('James',(123,456,789)),  
      ('Michael',(234,456,678)),  
      ('Robert',(168,89,190))  
 )
```

```
df_arr = spark.createDataFrame(data=d, schema = schema)  
df_arr.printSchema()  
df_arr.show(truncate=False)  
df_arr.select(df_arr.mobileNumbers(1)).show()
```

Supported Data Types:

What we have been so far ? → Aliases used in Spark Sql

For Ex-

IntegerType → int, integer

StringType → string

BooleanType → boolean

Supported Data Types:

Special Values:

- None (Null)
- Inf, -Inf (FloatType or DoubleType. Infinity)
- NaN (FloatType or DoubleType. Non a Number)

NaN = NaN returns True.

In aggregations, all NaN values are grouped together.

Ex – spark.sql ("""" SELECT float('NaN') AS col """").show()
spark.sql("""" SELECT double('NaN') = double('NaN') AS col """").show()

```
spark.sql ("""" CREATE TABLE test (c1 int, c2 double) """")  
spark.sql ("""" INSERT INTO test VALUES (1, double(10)) """")  
spark.sql ("""" INSERT INTO test VALUES (2, double(10)) """")  
spark.sql ("""" INSERT INTO test VALUES (3, double('NaN')) """")  
spark.sql ("""" INSERT INTO test VALUES (4, double('NaN')) """")  
spark.sql ("""" INSERT INTO test VALUES (5, double('NaN')) """")  
spark.sql ("""" SELECT c2,count(*) FROM TEST GROUP BY C2"""").show()
```

DataFrame Rows

DataFrame is a Dataset organized into named columns/rows.

| Name | Age |
|--------|-----|
| Robert | 31 |
| Alicia | 25 |
| Deja | 19 |
| Majoj | 31 |

Row

- Represented as a record/row in DataFrame.
- We can create a Row object by using named arguments, or create a custom Row like class.
- Available in pyspark.sql.Row

1. Row Object:

```
from pyspark.sql import Row
```

```
row = Row(name="Alice", age=11)
```

```
row.name
```

```
'name' in row
```

```
'Alice' in row.name
```

```
lst=(Row(name="Alice",age=11), Row(name="Robert",age=35),Row(name="James",age=33))
```

```
rdd = sc.parallelize(lst)
```

```
for i in rdd.collect(): print (str(i.age) + ' ' + i.name)
```

```
df = spark.createDataFrame(lst)
```

Row

2. Custom Class from Row

```
Person = Row("name", "age")
```

```
p1=Person("James", 40)
```

```
p2=Person("Alice", 35)
```

```
print(p1.name +",""+p2.name)
```

```
Ist=(Person("Alice",11), Person("Robert",35),Person("James",33))
```

```
rdd=sc.parallelize(Ist)
```

```
for i in rdd.collect() : print i.name
```

```
df = spark.createDataFrame(Ist)
```

Row

Row Methods:

count(): Return number of occurrences of value

```
person = Row(name="Alice", age=11,username="Alice")  
person.count("Alice")
```

index(): Return first index of value.

```
person.index(11)
```

asDict(): Resturn as a Dict

```
person.asDict()
```

DataFrame Columns

Column

- A column in a DataFrame.
- Available in `pyspark.sql.Column`

```
ord = spark.read.load('practice/retail_db/orders',format='csv',sep=',',schema=('order_id int,order_date timestamp,  
order_cust_id int,order_status string') )
```

1. Select a Column

```
df.order_id or df("order_id")  
ord.select(col("*")).show() #import pyspark.sql.functions import col
```

2. Give a alias name to a column

```
alias()  
Ex - df.select(df.order_id.alias("orderId")).show(5)
```

3. Order a Column

```
asc()  
asc_nulls_first()  
asc_nulls_last()  
desc()  
desc_nulls_first()  
desc_nulls_last()
```

```
Ex - ord.orderBy(ord.order_status.asc()).select(ord.order_status).distinct().show()
```

Column

4.cast() : Convert type of a column. asType() is alias of cast().

PS: Convert order_id column from Integer Type to String Type.

```
ord.select(ord.order_id.cast("string"))
```

5. between():

PS: Print all the orders between 10 and 20.

```
ord(ord.order_id.between(10,20)).show()  
ord.where(ord.order_id.between(10,20)).show()
```

6. contains(), startswith,endswith(),like(),rlike()

PS: Print all the orders with Status CLOSED.

```
ord.where(ord.order_status.contains('CLOSED')).show()
```

PS: Print all the status with alphabets 'LO.'

```
ord.where(ord.order_status.like('%LO%')).show()
```

7.isin(): Multiple values.

PS: Print all the status with CLOSED or PENDING Orders.

```
ord.where(ord.order_status.isin('CLOSED','PENDING')).select(ord.order_status).distinct().show()
```

Column

8. eqNullSafe() : Equality test that is safe for null values.

Assignments in pyspark do not check the Null or None Values. To check them use this function.

```
from pyspark.sql import Row  
df1 = spark.createDataFrame(( Row(id=1, value='foo'), Row(id=2, value=None) ))  
df1.select( df1('value') == 'foo', df1('value').eqNullSafe('foo'), df1('value').eqNullSafe(None)).show()  
isNull(), isNotNull()
```

9. substr

PS: Find Number of completed orders in the year 2013.

```
ord.where((ord.order_date.substr(1,4).contains('2013')) & (ord.order_status.contains('CLOSED') )).count()  
ord.where((ord.order_date.substr(1,4) == '2013') & (ord.order_status == 'CLOSED' )).count()
```

10.getField() : gets a field by name in a StructField

Ex - #create a struct field

```
df1 = spark.createDataFrame((Row(r=Row(a1=1, a2="b"))))  
df1.select(df.r.getField("a2")).show()  
df1.select(df.r.a).show()
```

11. getItem(): An expression that gets an item at position ordinal out of a list, or gets an item by key out of a dict.

Ex- df = spark.createDataFrame(((1, 2), {"key": "value"}), ("lst", "dict"))

```
df.select(df.lst.getItem(0), df.dist.getItem("key")).show()
```

```
df.select(df.lst(1)).show()
```

Column

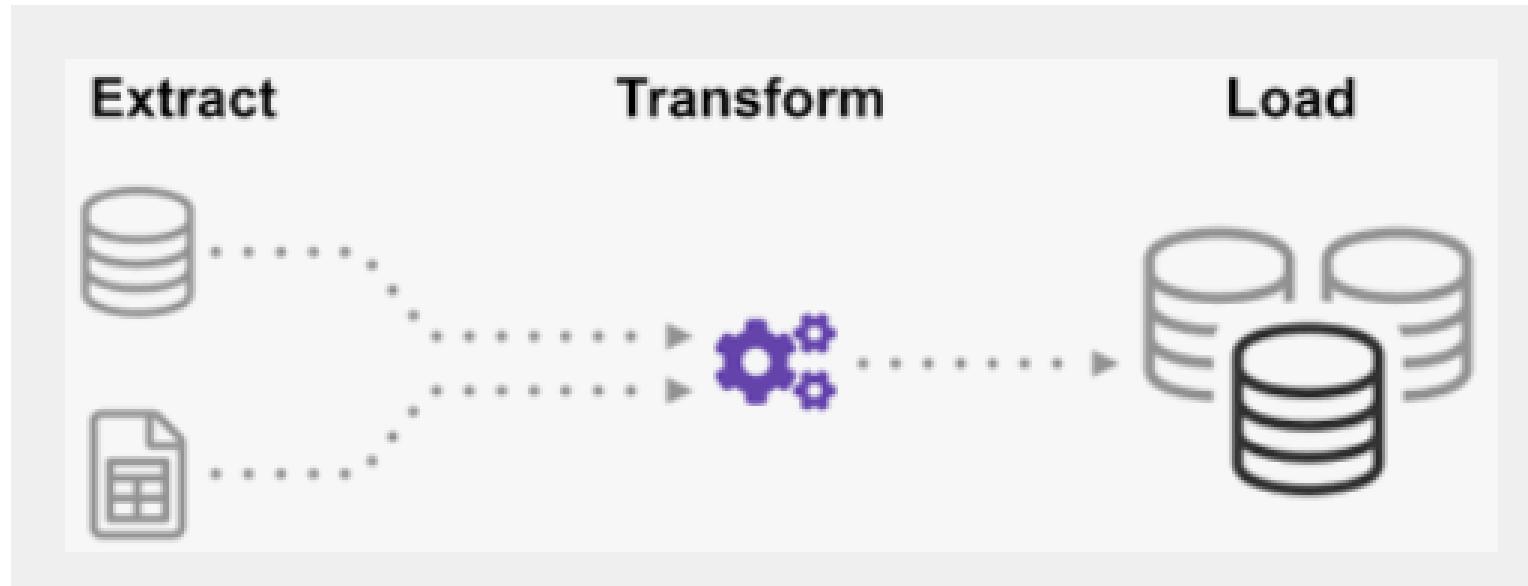
12. when(), otherwise() :

Kind of if-else statements in SQL. Using this, we can check multiple conditions in sequence and returns a value when the first condition is met.

Ex –

```
from pyspark.sql.functions import when  
ord.select(ord.order_status,  
when(ord.order_status == 'PENDING_PAYMENT', 'PP')  
.when(ord.order_status == 'CLOSED', 'CL')  
.when(ord.order_status == 'COMPLETE', 'CO')  
.when(ord.order_status == 'PROCESSING', 'PR')  
.otherwise(ord.order_status).alias("order_status2")).show(10)
```

DataFrame Transformations and Extractions



Transformations

- DataFrame APIs
 - ✓ selection
 - ✓ filter
 - ✓ sort
 - ✓ set
 - ✓ join
 - ✓ aggregation
 - ✓ groupBy
 - ✓ window
 - ✓ sample

Part1

- DataFrame Built-in Functions

- ✓ New Column
- ✓ Encryption
- ✓ String
- ✓ RegExp
- ✓ Date
- ✓ Null
- ✓ Collection
- ✓ Na
- ✓ Math & Statistics
- ✓ Explode & Flatten
- ✓ Formatting
- ✓ Json
- ✓ Other

Part2

Extraction

- csv
- text
- parquet
- orc
- avro
- json
- hive
- jdbc

Part3

Part 1

DataFrame APIs

DataFrame APIs

- ✓ selection
- ✓ filter
- ✓ sort
- ✓ set
- ✓ join
- ✓ aggregation
- ✓ groupBy
- ✓ window
- ✓ sample

DataFrame APIs : Selection APIs

```
### Prepare Data
ord = spark.read.load('retail_db/orders',sep=',',format='csv',schema=('order_id int,order_date timestamp,
order_customer_id int,order_status string'))

data=([('Robert',35,40,40),('Robert',35,40,40),('Ram',31,33,29),('Ram',31,33,91)])
emp = spark.createDataFrame(data=data,schema=('name','score1','score2','score3'))
```

Selection or Projection APIs

- **Select(*cols)**
 - ✓ Select one or more columns.
`ord.select(ord.order_id,'order_id',"order_id", (ord.order_id +100).alias("order10")).show()`
 - ✓ Can apply necessary functions on the selected columns.
`ord.select(lower(ord.order_status)).show()`
- **selectExpr(*expr)**
 - ✓ This is a variant of select that accepts SQL expressions.
`ord.selectExpr('substring(order_date,1,10) as order_month').show()`
`ord.select(substring(ord.order_date,1,4).alias('order_year')).show()`
 - ✓ If we want to use any functions available in SQL but not in Spark Built-in functions, then we can use selectExpr.
stack(n, expr1, ..., exprk) - Separates expr1, ..., exprk into n rows.
`df.selectExpr("stack(3,1,2,3,4,5,6)").show()`

Selection or Projection APIs

- **withColumn(colName, col)**
 - ✓ Applied transformation to only selected columns.
 - ✓ The first argument is a alias name. If we give a alias name same as a column name, the transformations will apply on the same column.
 - ✓ Otherwise a new column will be formed. Avoid giving alias name same as column name.
`ord.withColumn('order_month',substring(orderDF.order_date,1,10)).show()`
- **withColumnRenamed(existingCol, newCol)**
 - ✓ Rename Existing Column.
`ord.withColumnRenamed('order_id','order_id1').show()`

Selection or Projection APIs

- **drop(*cols)**
 - ✓ Drop a column.
`order = ord.drop('order_id','order_date')`
- **dropDuplicates(subset=None)**
 - ✓ Drop duplicate rows.
 - ✓ Optionally can consider only subset of columns.
`emp.dropDuplicates().show()`
`emp.dropDuplicates(("name","score1","score2")).show()`

DataFrame APIs : Filter APIs

```
### Prepare Data  
ord = spark.read.load('retail_db/orders',sep=',',format='csv',schema=('order_id int,order_date timestamp,  
order_customer_id int,order_status string'))
```

Filter APIs

- **filter(condition): (Its alias 'where')**

- ✓ Filter rows using a given condition.
- ✓ use '&' for 'and'. '|' for 'or'. (boolean expressions)
- ✓ Use column function isin() for multiple search.
- ✓ Or use IN Operator for SQL Style syntax.

```
ord.where((ord.order_id > 10) & (ord.order_id < 20)).show()
```

--Using isin()

```
ord.where(ord.order_status.isin('COMPLETE','CLOSED')).show()
```

--Using IN Operator

```
ord.where("order_status IN ('COMPLETE','CLOSED') ").show()
```

DataFrame APIs : Sort APIs

```
### Prepare Data
ord = spark.read.load('retail_db/orders',sep=',',format='csv',schema=('order_id int,order_date timestamp,
order_customer_id int,order_status string'))

data=([('a',1),('d',4),('c',3),('b',2),('e',5))
df = spark.createDataFrame(data=data,schema='col1 string,col2 int')
```

Sorting APIs

- **sort() or orderBy():**

- ✓ Sort specific column(s).

`ord.sort(ord.order_date.desc(), ord.order_status.asc()).show()`

`ord.sort(ord.order_date, ord.order_status, ascending=(0,1)).show() #1 Ascending, 0 Descending`

- **sortWithinPartitions:**

- ✓ At time, we may not want sort globally, but with in a group. In that case we can use sortWithinPartitions.

`df.sortWithinPartitions(df.col1.asc(), df.col2.asc()).show()`

DataFrame APIs : Set Operators

Set Operator APIs

- **union() and unionAll():**
 - ✓ Same and contains duplicate values.
 - ✓ Use distinct after union or unionAll to remove duplicates .
- **unionByName():**
 - ✓ The difference between this function and :func:`union` is that this function resolves columns by name (not by position)

```
df1 = spark.createDataFrame(data=([('a',1),('b',2)]),schema=['col1 string,col2 int'])
df2 = spark.createDataFrame(data=([(2,'b'),(3,'c')]),schema=['col2 int,col1 string'])
df1.union(df2).show()
df1.unionByName(df2).show()
```
- **intersect():** Containing rows in both DataFrames. Removed duplicates.
- **intersectAll():** Same as intersect. But retains the duplicates.

```
df1 = spark.createDataFrame(data=[('a',1),('a',1),('b',2)],schema=['col1 string,col2 int'])
df2 = spark.createDataFrame(data=[('a',1),('a',1),('c',2)],schema=['col1 string,col2 int'])
df1.intersect(df2).show()
df1.intersectAll(df2).show()
```
- **exceptAll():** Rows present in one DataFrame but not in another.

```
df1.exceptAll(df2).show()
```

DataFrame APIs : Join

Join APIs

| Join Type | ~ SQL Join |
|---------------------------------------|------------------|
| inner | INNER JOIN, JOIN |
| outer, full, fullouter, full_outer | FULL OUTER JOIN |
| left, left_outer, leftouter | LEFT JOIN |
| right, right_outer, rig htouter | RIGHT JOIN |
| cross | CROSS JOIN |
| left_anti, leftanti | |
| leftsemi, left_semi | |

PT: Semi joins performs better than inner joins. Use them wherever possible.

Join APIs

- **join** (otherDF, on=None, how=None)

on: *Joining Column*

how: 'inner', 'outer', 'full', 'fullouter', 'full_outer', 'leftouter', 'left', 'left_outer', 'rightouter', 'right', 'right_outer', 'leftsemi', 'left_semi', 'leftanti', 'left_anti', 'cross'.

```
df1 = spark.createDataFrame(data=((1,'Robert'),(2,'Ria'),(3,'James')),schema='empid int,empname string')
```

```
df2 = spark.createDataFrame(data=((2,'USA'),(4,'India')),schema='empid int,country string')
```

```
df1.join(df2,df1.id == df2.id,'inner').select(df1.id,df2.country).show()
```

- **crossJoin(self, other)**

- **self Join**

```
df1 = spark.createDataFrame(data=((1,'Robert',2),(2,'Ria',3),(3,'James',5)),schema='empid int,empname string,managerid int')
```

```
df1.alias("emp1").join(df1.alias("emp2"),col("emp1.managerid") == col("emp2.empid"),'inner').select(col("emp1.empid"),col("emp1.empname"),col("emp2.empid").alias("managerid"),col("emp2.empname").alias("managaer_name")).show()
```

Use of col(): Sometimes we need to use the column name which is the alias of a withColumn. In that case we need to refer the column name as col(column_name).

```
pyspark.sql.functions import col
```

Join APIs

- **Multi Column Join**

```
df1 = spark.createDataFrame(data=((1,101,'Robert'),(2,102,'Ria'),(3,103,'James'))),schema='empid int,deptid  
int,empname string')  
df2 = spark.createDataFrame(data=((2,102,'USA'),(4,104,'India'))),schema='empid int,deptid int,country string')  
df1.join(df2,(df1.empid == df2.empid) & (df1.deptid == df2.deptid)).show()
```

- **Multi DataFrame Join**

```
df1 = spark.createDataFrame(data=((1,'Robert'),(2,'Ria'),(3,'James'))),schema='empid int,empname string')  
df2 = spark.createDataFrame(data=((2,'USA'),(4,'India'))),schema='empid int,country string')  
df3 = spark.createDataFrame(data=((1,'01-jan-2021'),(2,'01-feb-2021'),(3,'01-mar-2021'))),schema='empid  
int,joindate string')  
df1.join(df2,df1.empid==df2.empid).join(df3,df1.empid == df3.empid).show()
```

DataFrame APIs : Aggregation

Aggregation APIs

- **summary**

```
df.summary("count", "min", "25%", "75%", "max").show()
```

```
df.select("age", "name").summary("count").show()
```

- **avg, max, min**

```
ordItems.select(avg(ordItems.price)).show()
```

- **sum, sumDistinct**

```
ordItems.select(sum(ordItems.price), sumDistinct(ordItems.price)).show()
```

- **count, countDistinct**

```
ordItems.select(count(ordItems.order_item_product_id),countDistinct(ordItems.order_item_product_id)).show()
```

- **first, last**

```
ordItems.sort(ordItems.price.asc()).select(first(ordItems.price)).show()
```

- **collect_set, collect_list**

```
df = spark.createDataFrame(((1,100),(2,150),(3,200),(4,50),(5,50)),schema='id int,salary int')
```

```
df.select(collect_list(df.salary)).show(truncate=False)
```

- **skewness**

- **variance**

- **stddev**

DataFrame APIs : groupBy

GroupBy API

When we apply groupBy on a DataFrame Column, it returns GroupedData object. It has below aggregate functions:

*avg(), mean()
count()
min()
max()
sum()
agg() →For multiple aggregations at once
pivot()
apply()*

```
data = (("James","Sales","NY",9000,34),  
        ("Alicia","Sales","NY",8600,56),  
        ("Robert","Sales","CA",8100,30),  
        ("Lisa","Finance","CA",9000,24),  
        ("Deja","Finance","CA",9900,40),  
        ("Sugie","Finance","NY",8300,36),  
        ("Ram","Finance","NY",7900,53),  
        ("Kyle","Marketing","CA",8000,25),  
        ("Reid","Marketing","NY",9100,50))  
schema=("empname","dept","state","salary","age")  
df = spark.createDataFrame(data=data,schema=schema)
```

```
df.groupBy(df.dept)  
<pyspark.sql.group.GroupedData object at 0x7f68eaead690>
```

GroupBy API

Using avg(),sum(),min(),max(),count(),agg()

Ex-1 (Using 1 column)

```
df.groupBy(df.dept).avg("salary").show()
```

Ex-2 (Multiple Columns)

```
df.groupBy(df.dept,df.state).min("salary","age").show()
```

Ex-3 (Using Agg() many aggregations)

```
df.groupBy(df.dept).agg(min("salary").alias('min_salary'),  
                         max("salary").alias('max_salary'),  
                         avg("salary").alias('avg_salary'))  
                         .show()
```

Ex-4 (Using filter or where)

```
df.where(df.state == 'NY').groupBy(df.dept).agg(min("salary").alias('min_salary')).where(col("min_salary") > 8000).show()
```

GroupBy API

Using pivot():

Transpose rows into columns.

Ex-

```
df_t = df.groupBy(df.dept).pivot("state").sum("salary")
```

| | dept | state | sum(salary) |
|--|-----------|-------|-------------|
| | Finance | NY | 16200 |
| | Marketing | NY | 9100 |
| | Sales | CA | 8100 |
| | Marketing | CA | 8000 |
| | Finance | CA | 18900 |
| | Sales | NY | 17600 |



| | dept | CA | NY |
|--|-----------|-------|-------|
| | Sales | 8100 | 17600 |
| | Finance | 18900 | 16200 |
| | Marketing | 8000 | 9100 |

GroupBy API

Using unpivot:

Ex-

There is no such function as unpivot. We can do it using stack() function in a selectExpr.

stack(n, expr1, ..., exprk) - Separates expr1, ..., exprk into n rows. Uses column names col0, col1, etc. by default unless specified otherwise.

```
spark.sql("''' select stack(3,1,2,3,4,5,6) '''").show()
```

```
spark.sql("''' select dept, stack(2,'CA',CA,'NY',NY) as (state,salary) from d '''").show()
```

```
df_t.selectExpr("dept","stack(2,'CA',CA,'NY',NY) as (state,salary)").show()
```

| | dept | CA | NY |
|-----------|-------|-------|----|
| Sales | 8100 | 17600 | |
| Finance | 18900 | 16200 | |
| Marketing | 8000 | 9100 | |



| | dept | state | salary |
|-----------|------|-------|--------|
| Sales | CA | 8100 | |
| Sales | NY | 17600 | |
| Finance | CA | 18900 | |
| Finance | NY | 16200 | |
| Marketing | CA | 8000 | |
| Marketing | NY | 9100 | |

GroupBy API

/* Not Recorded */

Using apply(pandas_udf):

- Takes a pandas udf and apply it to the current dataframe and returns a dataframe.
- Pandas_udf is available in pyspark.sql.functions
- pandas_udf(f=None, returnType=None, functionType=None)
 - f → Function. Optional. User Defined Function.
 - returnType → Optional. The return type of user defined function.
 - functionType → Optional. A value in pyspark.sql.functions.pandasUDFType. Default:scalar. Exists for compatibility.

Ex-

```
from pyspark.sql.functions import pandas_udf, PandasUDFType
df = spark.createDataFrame(((1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)),("id", "v"))
@pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP) # doctest: +SKIP
def normalize(pdf):
    v = pdf.v
    return pdf.assign(v=(v - v.mean()) / v.std())

df.groupby("id").apply(normalize).show()
```

DataFrame APIs : window

Window Functions

- Window Functions operates on a group of rows and return a single value for each input row.
- Main Package: pyspark.sql.window. It has two classes Window and WindowSpec
- Window class has APIs such as partitionBy, orderBy, rangeBetween, rowsBetween.
- WindowSpec class defines the partitioning, ordering and frame boundaries. It has also above 4 APIs.
- These APIs such partitionBy returns a WindowSpec object.

```
>>> spec = Window.partitionBy(df.dept)
>>> Window.partitionBy(df.dept)
<pyspark.sql.window.WindowSpec object at 0x7f88d8ef3910>
```

- To perform a window function, we will have to partition the data using Window.partitionBy.
- Lets see 3 types of Window Functions:
 - ✓ Ranking
 - ✓ Analytical
 - ✓ Aggregate

Window Functions

| empname | dept | state | salary | age |
|---------|-----------|-------|--------|-----|
| James | Sales | NY | 9000 | 34 |
| Alicia | Sales | NY | 8600 | 56 |
| Robert | Sales | CA | 8100 | 30 |
| Lisa | Finance | CA | 9000 | 24 |
| Deja | Finance | CA | 9900 | 40 |
| Sugie | Finance | NY | 8300 | 36 |
| Ram | Finance | NY | 7900 | 53 |
| Kyle | Marketing | CA | 8000 | 25 |
| Reid | Marketing | NY | 9100 | 50 |

Aggregate Function



| dept | sum(salary) |
|-----------|-------------|
| Sales | 25700 |
| Finance | 35100 |
| Marketing | 17100 |

| empname | dept | state | salary | age |
|---------|-----------|-------|--------|-----|
| James | Sales | NY | 9000 | 34 |
| Alicia | Sales | NY | 8600 | 56 |
| Robert | Sales | CA | 8100 | 30 |
| Lisa | Finance | CA | 9000 | 24 |
| Deja | Finance | CA | 9900 | 40 |
| Sugie | Finance | NY | 8300 | 36 |
| Ram | Finance | NY | 7900 | 53 |
| Kyle | Marketing | CA | 8000 | 25 |
| Reid | Marketing | NY | 9100 | 50 |

Analytical Function



| empname | dept | state | salary | age | sum_salary_dept |
|---------|-----------|-------|--------|-----|-----------------|
| James | Sales | NY | 9000 | 34 | 25700 |
| Alicia | Sales | NY | 8600 | 56 | 25700 |
| Robert | Sales | CA | 8100 | 30 | 25700 |
| Deja | Finance | CA | 9900 | 40 | 35100 |
| Sugie | Finance | NY | 8300 | 36 | 35100 |
| Ram | Finance | NY | 7900 | 53 | 35100 |
| Lisa | Finance | CA | 9000 | 24 | 35100 |
| Kyle | Marketing | CA | 8000 | 25 | 17100 |
| Reid | Marketing | NY | 9100 | 50 | 17100 |

Window Functions

```
--Prepare Input Data
data = (("James","Sales","NY",9000,34),
        ("Alicia","Sales","NY",8600,56),
        ("Robert","Sales","CA",8100,30),
        ("John","Sales","AZ",8600,31),
        ("Ross","Sales","AZ",8100,33),
        ("Kathy","Sales","AZ",1000,39),
        ("Lisa","Finance","CA",9000,24),
        ("Deja","Finance","CA",9900,40),
        ("Sugie","Finance","NY",8300,36),
        ("Ram","Finance","NY",7900,53),
        ("Satya","Finance","AZ",8200,53),
        ("Kyle","Marketing","CA",8000,25),
        ("Reid","Marketing","NY",9100,50)
      )
schema=("empname","dept","state","salary","age")
df = spark.createDataFrame(data=data,schema=schema)
```

Window Functions

- **Ranking Window Functions:** Used to provide a ranking to the result within a partition.
 - ✓ `row_number()` : Sequential Row Number.
 - ✓ `rank()` : Ranks but gaps when ties.
 - ✓ `dense_rank()` : Ranks without any gaps.
 - ✓ `percent_rank`: Relative rank (i.e. percentile) of rows within a window partition. First row is always 0 and last row is always 1.
 - ✓ `ntile()` : returns the ntile group id (from 1 to n inclusive) in an ordered window partition. For example, if n is 4, the first quarter of the rows will get value 1, the second quarter will get 2, the third quarter will get 3, and the last quarter will get 4.
 - ✓ `cume_dist()`: Returns the cumulative distribution of values within a window partition,
i.e. the fraction of rows that are below the current row.

Ex –

```
from pyspark.sql.window import *
from pyspark.sql.functions import *
spec = Window.partitionBy("dept").orderBy("salary")
df.select(df.dept,df.salary) \
    .withColumn("row_number",row_number().over(spec))\
    .withColumn("rank",rank().over(spec))\
    .withColumn("dense_rank",dense_rank().over(spec))\
    .withColumn("percent_rank",percent_rank().over(spec))\
    .withColumn("cume_dist",cume_dist().over(spec))\
    .withColumn("ntile",ntile(3).over(spec))\
    .show()
```

Window Functions

- Analytical Window Functions:

- ✓ `lag()` : Return offset row value before the current row value.
- ✓ `lead()` : Return offset row value after the current row value.

Ex -

```
df.select(df.dept,df.salary) \  
.withColumn("lag_prev_sal",lag("salary",1,0).over(spec)) \  
.withColumn("lead_next_sal",lead("salary",1,0).over(spec)) \  
.show()
```

Window Functions

- Aggregate Window Functions:

- ✓ avg
- ✓ sum()
- ✓ max()
- ✓ min()
- ✓ count()
- ✓ first()
- ✓ last()

Ex –

```
spec = Window.partitionBy("dept")
```

```
df.select(df.dept,df.salary) \  
.withColumn("sum_sal",sum("salary").over(spec)) \  
.withColumn("max_sal",max("salary").over(spec)) \  
.withColumn("min_sal",min("salary").over(spec)) \  
.withColumn("avg_sal",avg("salary").over(spec)) \  
.withColumn("count_sal",count("salary").over(spec)) \  
.show()
```

Ex –

```
spec = Window.partitionBy("dept").orderBy("salary")
```

```
df.select(df.dept,df.salary) \  
.withColumn("first_sal",first("salary").over(spec)) \  
.withColumn("last_sal",last("salary").over(spec)) \  
.show()
```

Window Functions

- **rangeBetween:**
 - ✓ Takes two argument (start,end) to define frame boundaries.
 - ✓ Default : unboundedPreceding and unboundedFollowing.
 - ✓ Both `start` and `end` are relative from the current row. For example, "0" means "current row", while "-1" means one off before the current row, and "5" means the five off after the current row.
 - ✓ Recommend to use ``Window.unboundedPreceding``, ``Window.unboundedFollowing``, and ``Window.CurrentRow`` to specify special boundary values, rather than using integral values directly.

```
spec=Window.partitionBy(df.dept).\  
orderBy(df.salary). \  
rangeBetween(Window.unboundedPreceding, Window.unboundedFollowing)
```

```
spec=Window.partitionBy(df.dept).\  
orderBy(df.salary). \  
rangeBetween(Window.CurrentRow, Window.unboundedFollowing)
```

```
spec=Window.partitionBy(df.dept).\  
orderBy(df.salary). \  
rangeBetween(Window.CurrentRow,500)
```

```
df.select(df.dept,df.salary).withColumn("sum_sal",sum("salary").over(spec1)).show()
```

Window Functions

- **rowsBetween:**
 - ✓ Takes two argument (start,end) to define frame boundaries.
 - ✓ Default : unboundedPreceding and unboundedFollowing.
 - ✓ Both `start` and `end` are relative from the current row. For example, "0" means "current row", while "-1" means one off before the current row, and "5" means the five off after the current row.
 - ✓ Recommend to use ``Window.unboundedPreceding``, ``Window.unboundedFollowing``, and ``Window.CurrentRow`` to specify special boundary values, rather than using integral values directly.

```
spec1=Window.partitionBy(df.dept).\  
orderBy(df.salary). \  
rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
```

```
spec1=Window.partitionBy(df.dept).\  
orderBy(df.salary). \  
rowsBetween(Window.CurrentRow, Window.unboundedFollowing)
```

```
spec1=Window.partitionBy(df.dept).\  
orderBy(df.salary). \  
rowsBetween(Window.CurrentRow, 2)
```

```
df.select(df.dept,df.salary).withColumn("sum_sal",sum("salary").over(spec1)).show()
```

Window Functions

rangeBetween(Window.unboundedPreceding, Window.unboundedFollowing)

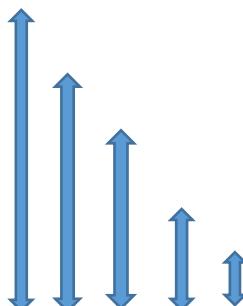
rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)

| dept | salary | sum_sal |
|---------|--------|---------|
| Finance | 7900 | 43300 |
| Finance | 8200 | 43300 |
| Finance | 8300 | 43300 |
| Finance | 9000 | 43300 |
| Finance | 9900 | 43300 |

rangeBetween(Window.currentRow, Window.unboundedFollowing)

rowsBetween(Window.currentRow, Window.unboundedFollowing)

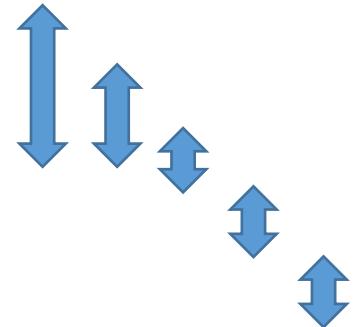
| dept | salary | sum_sal |
|---------|--------|---------|
| Finance | 7900 | 43300 |
| Finance | 8200 | 35400 |
| Finance | 8300 | 27200 |
| Finance | 9000 | 18900 |
| Finance | 9900 | 9900 |



Window Functions

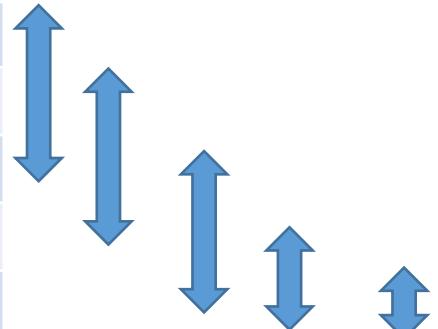
rangeBetween(Window.currentRow, 500)

| dept | salary | sum_sal |
|---------|--------|---------|
| Finance | 7900 | 24400 |
| Finance | 8200 | 16500 |
| Finance | 8300 | 8300 |
| Finance | 9000 | 9000 |
| Finance | 9900 | 9900 |



rowsBetween(Window.currentRow, 2)

| dept | salary | sum_sal |
|---------|--------|---------|
| Finance | 7900 | 24400 |
| Finance | 8200 | 25500 |
| Finance | 8300 | 27200 |
| Finance | 9000 | 18900 |
| Finance | 9900 | 9900 |



Window Functions

Prepare Input Data

```
--df DataFrame  
data = (("James","Sales","NY",9000,34),  
        ("Alicia","Sales","NY",8600,56),  
        ("Robert","Sales","CA",8100,30),  
        ("John","Sales","AZ",8600,31),  
        ("Ross","Sales","AZ",8100,33),  
        ("Kathy","Sales","AZ",1000,39),  
        ("Lisa","Finance","CA",9000,24),  
        ("Deja","Finance","CA",9900,40),  
        ("Sugie","Finance","NY",8300,36),  
        ("Ram","Finance","NY",7900,53),  
        ("Satya","Finance","AZ",8200,53),  
        ("Kyle","Marketing","CA",8000,25),  
        ("Reid","Marketing","NY",9100,50)  
    )  
schema=("empname","dept","state","salary","age")  
df = spark.createDataFrame(data=data,schema=schema)  
  
--df1 DataFrame  
df1 = spark.range(10)
```

DataFrame API: Sampling

`sample(withReplacement=None, fraction=None, seed=None):`

- To get random sample records from the dataset.
withReplacement: True or False. With True, Same result can be produced more than once.
Fraction: Between 0 to 1. 0.3 means 30%. Does not guarantee the exact 30% of the records.
Seed: Reproduce the same sample.

Ex-

```
dataset=spark.range(100)  
dataset.sample(withReplacement=True,fraction=0.07,seed=10).show()
```

`sampleBy(col, fractions, seed=None):`

- Returns a stratified sample without replacement based on the fraction given on each stratum.
- If a stratum is not specified, we treat its fraction as zero.

Ex-

```
from pyspark.sql.functions import col  
dataset = spark.range(0, 100).select((col("id") % 3).alias("key"))  
sampled = dataset.sampleBy("key", fractions={0: 0.1, 1: 0.2}, seed=0)  
sampled.groupBy("key").count().orderBy("key").show()
```

Part 2

DataFrame Built-in Functions

DataFrame Built-in Functions

- ✓ New Column
- ✓ Encryption
- ✓ String
- ✓ RegExp
- ✓ Date
- ✓ Null
- ✓ Collection
- ✓ Na
- ✓ Math & Statistics
- ✓ Explode & Flatten
- ✓ Formatting
- ✓ Json
- ✓ Other

DataFrame APIs : New Column Functions

New Columns Functions:

Window Ranking Functions(`dense_rank`, `rank`, `row_number` etc) :

`monotonically_increasing_id()`:

- A column that generates monotonically increasing 64-bit integers.
- The generated ID is guaranteed to be monotonically increasing and unique, but not consecutive.
- Use Case: Create a Primary Key/Unique column.

```
df.withColumn('id',monotonically_increasing_id()).show()
```

`lit()`:

- It creates a static column with value is provided.

```
df.withColumn('col',f.lit(10)).show()
```

- Can also be used to concat columns.

```
df.select(concat('salary',lit('|'),'age').alias('value')).show()
```

New Columns Functions:

expr(str):

- It takes SQL Expression as a string argument, executes the expression and returns a Column Type.
Ex - `df.withColumn('empname_len',expr("length(empname)").show())`
- We can use SQL-like functions that are not present in pyspark column type and built-in functions(`pyspark.sql.functions`).
For example – CASE WHEN, Concat operator || etc
`df.withColumn(age_desc",expr(" case when age > 50 then 'Senior' else 'Adult' end")).show()`
`df.withColumn('emp-dept',expr(" empname || '-' || dept")).show()`
- Arithmetic Expression
`df.select(expr(" age + 10 as age_10 ")).show()`
- Spark do not check this in the compile type like other DataFrame Operations.

spark_partition_id():

- Generates a column with partitions ids.

Ex-

```
df1 = spark.range(10)
df1.repartition(5)
df1=df1.repartition(5)
df1.select("id",spark_partition_id()).show()
```

New Columns Functions:

rand(seed):

- Generates a random column with independent and identically distributed (i.i.d.) samples from uniform distribution.

```
df.withColumn('random_col',rand(70)).show(truncate=False)
```

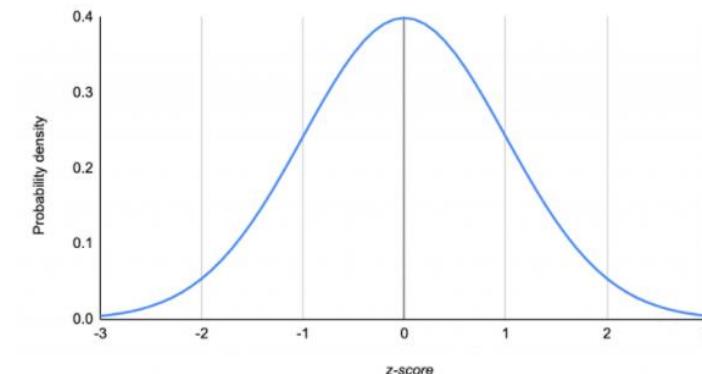
randn(seed):

- Generates a column with independent and identically distributed (i.i.d.) samples from the standard normal distribution.

Uniform Distribution



Standard Distribution



DataFrame APIs : Column Encryption Functions

Column Encryption Functions

sha1(col):

- Used for Encryption.
- Returns the hex string result SHA-1 Family.
- Only for string columns. Convert the int column to string.

```
df.select(df.age,sha1(df.age.cast('string'))).show(truncate=False)
```

sha2(col,numBits):

- Used for Encryption.
- Returns the hex string result of SHA-2 family.
- numBits : 0, 224, 256, 384, 512

```
df.select(df.age,sha2(df.age.cast('string'))).show(truncate=False)
```

hash(*cols):

- Any type of column or combination of columns.
- Calculates the hash code of given columns and return result as int column.
- May be used for Encryption.

```
df.select(df.age,hash(df.age)).show(truncate=False)
```

md5(cols):

- Calculates the MD5 digest and returns the value as a 32 character hex string.

```
df.select(df.age,md5(df.age)).show(truncate=False)
```

DataFrame APIs : String Functions

String Manipulation Functions

split(str, pattern) : Splits str around pattern (pattern is a regular expression).

Ex-1

```
df = spark.createDataFrame([('ab12cd23fe27kl',)), ('s',))  
df.select(split(df.s,'(0-9)+')).show(truncate=False)
```

Ex-2

```
ord.select(split(ord.order_date,'-')).show(truncate=False)
```

length(col): Computes the character length of string.

```
ordDF.select(length(ordDF.order_status)).distinct().show()
```

lower(col), upper(col), initcap(col):

ltrim(col),rtrim(col),trim(col):

lpad(col,len,pad), rpad(col,len,pad): Pad the string column to width `len` with `pad`.

reverse(col): Returns a reversed string.

repeat(col, n): Repeats a string column n times.

hex(col): Computes hex value of the given column.

String Manipulation Functions

concat(*cols): Concatenates multiple input columns together into a single column

```
ord.withColumn('IDStatus',concat(ord.order_id,ord.order_status)).show()
```

concat_ws(sep, *cols): Concatenates multiple input string columns together into a single string column, using the given separator.

```
ord.withColumn('IDStatus',concat_ws('-',ord.order_id,ord.order_status)).show()
```

substring(str, pos, len): Substring starts at `pos` and is of length `len` when str is String type.

```
ord.withColumn('orderYear',substring(ord.order_date,1,4)).show()
```

substring_index(str, delim, count): Returns the substring from string str before count occurrences of the delimiter delim. If count is positive, everything to the left of the final delimiter (counting from left) is returned. If count is negative, everything to the right of the final delimiter (counting from the right) is returned. substring_index performs a case-sensitive match when searching for delim.

```
ord.withColumn('sub',substring_index(ord.order_date,'-',1)).show()
```

instr(str, substr): Locate the position of the first occurrence of substr column in the given string.

Returns null if either of the arguments are null.

```
ord.withColumn('instr',instr(ord.order_status,'LO')).show()
```

String Manipulation Functions

locate(substr, str, pos=1): Locate the position of the first occurrence of substr in a string column, after position pos. The position is not zero based, but 1 based index. Returns 0 if substr could not be found in str.

```
ord.withColumn('instr',locate('00',ord.order_date,2)).show()
```

translate(srcCol, matching, replace): Translate any character in the `srcCol` by a character in `matching`.

```
df = spark.createDataFrame([('translate',)), ('col'))  
df.select(translate(df.col, "rnlt", "123")).show()
```

String Manipulation Functions

`overlay(src,replace,pos,len)`: *New in version 3.0.* Overlay the specified portion of `src` with `replace` starting from the byte position `pos` of `src` and proceeding for `len` bytes.

```
>>> df = spark.createDataFrame([ ("SPARK_SQL", "CORE") ], ("x", "y"))
>>> df.show()
+-----+---+
|      x |    y |
+-----+---+
| SPARK_SQL | CORE |
+-----+---+
>>> df.select(overlay("x", "y", 7).alias("overlaid")).show()
+-----+
| overlaid|
+-----+
| SPARK_CORE |
+-----+
```

DataFrame APIs : RegExp Functions

`regexp_extract(str, pattern, idx):`

We can extract a pattern.

str: string or column from which we want to extract data.

pattern: pattern-regex pattern.

idx: Group Number to extract

```
df=spark.createDataFrame(data=([('11ss1 ab',)),schema='str'))
```

```
df.select(df.str,regexp_extract(df.str,'(\d+)(\w+)(\s)((a-z)+)',1).alias('op')).show()
```

`regexp_replace(str, pattern, replacement):`

We can replace a column value with a string for another string or substring.

Returns empty string if does not match.

```
df.select(regexp_replace(lit('11ss1 ab'),'(\d+)','xx').alias('op')).show()
```

| Quantifier | Meaning |
|------------|---------------------------------------|
| \d | Matches digits 0-9. |
| \w | Matches alphabets and digits. |
| \s | Matches space. |
| . | Matches any character except newline. |
| ? | 0 or more character. |
| + | 1 or more character. |
| (a-zA-Z) | Anything in range a-z and A-Z. |

Use `when()` for conditional replace.

```
addr = ((1,"2625 Indian School Rd","Phoenix"),
(2,"1234 Thomas St","Glendale"))
```

```
df=spark.createDataFrame(address,("id","addr","city"))
```

```
df.withColumn('new_addr',
when(df.addr.endswith('Rd'),regexp_replace(df.addr,'Rd','Road')) \
.when(df.addr.endswith('St'),regexp_replace(df.addr,'St','Street')) \
.otherwise(df.addr)) \
.show(truncate=False)
```

rlike():

Not a dataframe function but a column function we can use to check if a pattern is found or not.

```
df.select(df.str.rlike('(\d+)').show()
```

DataFrame APIs : Date Functions

```
### Prepare Data
ord = spark.read.load('retail_db/orders',sep=',',format='csv',schema=('order_id int,order_date timestamp,
order_customer_id int,order_status string'))

ord_new = ord.withColumn('new_order_date',date_add(ord.order_date,50))
```

current_date():

Returns the current date.

df.select(current_date()).show()

current_timestamp():

next_day(date, dayOfWeek):

Returns the first date which is later than the value of the date column.

dayOfWeek - "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun".

ord.select(ord.order_date,next_day(ord.order_date,'Fri')).distinct().show()

last_day(date):

Returns the last day of the month which the given date belongs to.

ord.select(ord.order_date,last_day(ord.order_date)).distinct().show()

dayofweek(col):

dayofmonth(col):

dayofyear(col):

weekofyear(col):

second(col):

Extract the seconds of a given date as integer.

```
ord.select(current_timestamp(),second(current_timestamp())).distinct().show(truncate=False)
```

minute(col):

hour(col):

month(col):

quarter(col):

year(col):

months_between(date1, date2, roundOff=True)

The result is rounded off to 8 digits unless `roundOff` is set to `False`.

```
ord_new.select(ord_new.order_date,ord_new.new_order_date,months_between(ord_new.order_date,ord_new.new_order_date,roundOff=False)).show()
```

date_add(start, days) : Add Number Of Days.

```
ord.withColumn('new_order_date',date_add(ord.order_date,50)).show()
```

date_sub(start, days) : Subtract Number Of Days.

```
ord.withColumn('new_order_date',date_sub(ord.order_date,50)).show()
```

add_months(start, months): Add Number of months.

```
ord.withColumn('new_order_date',add_months(ord.order_date,3)).show()
```

datediff(end, start): Returns the number of days from `start` to `end`.

```
ord_new.select(datediff(ord_new.new_order_date,ord_new.order_date)).show()
```

date_trunc(format, timestamp)

Returns timestamp truncated to the unit specified by the format.

format: 'year', 'yyyy', 'yy', 'month', 'mon', 'mm', 'day', 'dd', 'hour', 'minute', 'second', 'week', 'quarter' .

```
ord.withColumn('new_order_date',date_trunc('yyyy',ord.order_date)).show()
```

date_format(date, format):

Converts a date/timestamp/string to a value of string in the format specified by the date.

```
ord.withColumn('new_order_date',date_format(ord.order_date,'yyyy/MM/dd')).show(5)
```

unix_timestamp(timestamp=None, format='yyyy-MM-dd HH:mm:ss'):

Convert time string with given pattern ('yyyy-MM-dd HH:mm:ss', by default) to Unix time stamp (in seconds), using the default timezone and the default locale, return null if fail.

```
ord.withColumn('new_order_date',unix_timestamp(ord.order_date)).show(5)
```

to_timestamp(col, format=None): Converts a Column into timestamp type.

```
df = spark.createDataFrame([('1997-02-28 10:30:00',)), ('t'))
```

```
df.select(to_timestamp(df.t).alias('dt')).show()
```

from_unixtime(timestamp, format='yyyy-MM-dd HH:mm:ss'):

Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format.

```
time_df = spark.createDataFrame(((1428476400,)), ('unix_time'))
time_df.select(from_unixtime('unix_time').alias('ts')).collect()
```

from_utc_timestamp(timestamp, tz):

This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE. This function takes a timestamp which is timezone-agnostic, and interprets it as a timestamp in UTC, and renders that timestamp as a timestamp in the given time zone.

```
df = spark.createDataFrame([('1997-02-28 10:30:00', 'JST')), ('ts', 'tz'))
df.select(from_utc_timestamp(df.ts, "PST").alias('local_time')).collect()
df.select(from_utc_timestamp(df.ts, df.tz).alias('local_time')).collect()
```

to_date(col, format=None):

Converts a string type or timestamp type into date type using the optionally specified format.

```
ord.select(ord.order_date,to_date(ord.order_date,'yyyy-mm-dd')).show()
```

to_timestamp(col, format=None):

```
df.select(df.t,to_timestamp(df.t).alias('dt')).show()
```

DataFrame APIs : Null Functions

###Prepare Data

```
df = spark.createDataFrame([('Robert',1, None,114.0), ('John',None, 2577,float('nan'))), ("name", "id","phone","stAdd"))
```

isnull(col): Returns true if the column is null.

```
df.select(isnull(df.id)).show()
```

isnan(col): Returns true if the column is NaN.

```
df.select(isnan(df.stAdd)).show()
```

nanvl(col1, col2): Returns col1 if it is not NaN, or col2 if col1 is NaN.

```
df.select(df.stAdd,df.phone,nanvl(df.stAdd,df.phone)).show()
```

coalesce(*cols): Returns the first column that is not null.

```
df.select(df.phone,df.stAdd,coalesce(df.phone,df.stAdd)).show()
```

DataFrame APIs : Collection Functions

emp1 DataFrame

```
data =([('Alicia','Joseph'),('Java','Scala','Spark'),{'hair':'black','eye':'brown'}), \
        ('Robert','Gee','Spark','Java'),{'hair':'brown','eye':None}), \
        ('Mike','Bianca','CSharp',{}),{'hair':'red','eye':''}), \
        ('John','Kumar',None,None), \
        ('Jeff','L',('1','2'),{})
```

```
schema = ('FirstName','LastName','Languages','properties')
```

```
emp1 = spark.createDataFrame(data=data,schema=schema)
```

emp2 DataFrame

```
data=([('Robert',35,40,40),('Ram',31,33,29),('John',95,89,91))]
```

```
schema = ('name','score1','score2','score3')
```

```
emp2= spark.createDataFrame(data=data, schema=schema)
```

emp3 DataFrame

```
emp3 =
```

```
spark.createDataFrame(data=([('John',(10,20,20),(25,11,10)),('Robert',(15,13,55),(5,None,29)),('James',(11,13,45),(5,89,79))),schema=('em  
pName', 'score_arr1', 'score_arr2'))
```

df DataFrame

```
df = spark.sql("SELECT array(struct(1, 'a'), struct(2, 'b')) as data")
```

size(col):

- Returns the length of the array or map stored in the column.
- Size is -1 for null elements.

```
emp1.select(size(emp1.Languages),size(emp1.properties)).show()
```

element_at(col,extraction):

- Returns element of array at given index in extraction if col is array.
- Returns value for the given key in extraction if col is map.

```
emp1.select(emp1.FirstName,element_at(emp1.Languages,2),element_at(emp1.properties,'eye')).show()
```

struct(*cols) :

- Create a new struct column.

```
emp_new = emp1.select(struct((emp1.FirstName,emp1.LastName)))
```

array(*cols):

- Creates a new array column.

```
emp_new = emp2.select(array(emp2.score1,emp2.score2,emp2.score3))
```

array_max(col), array_min(col):

- Returns maximum or minimum values of an array column.

```
emp3.select(array_max("score_arr1")).show()
```

array_distinct(col):

- Returns distinct values of an array column.

```
emp3.select(array_distinct("score_arr1")).show()
```

array_repeat(col,count):

- Repeated count times.

```
emp3.select(array_repeat("score_arr1",3)).show(truncate=False)
```

--Flatten the array and chose the distinct values

```
emp3.select(array_distinct(flatten(array_repeat("score_arr1",3)))).show(truncate=False)
```

slice(col,start,length)

- Returns an array containing all the elements in `col` from index `start` for length `length`.
- col is Array Type.

```
emp1.select(emp1.Languages,slice(emp1.Languages,3,1)).show()
```

array_position(col,value):

- Locates the position of the first occurrence of the given value in the given array.
- Starts with Index 1.

```
emp3.select(emp3.score_arr1,array_position(emp3.score_arr1,27)).show()
```

array_remove(col,element):

- Remove all elements that equal to element from the given array.

```
emp3.select(array_remove("score_arr1",20)).show()
```

array_sort(col):

- Sorts the input array in ascending order.
- The elements of the input array must be orderable.
- Null elements will be placed at the end of the returned array.

```
emp3.select("score_arr1",array_sort("score_arr1")).show()
```

sort_array(col,asc=True):

- Sorts the input array in ascending or descending order according to the natural ordering of the array elements.
- Null elements will be placed at the beginning of the returned array in ascending order or at the end of the returned array in descending order.

```
emp3.select(emp3.score_arr1,sort_array(emp3.score_arr1,asc=False)).show()
```

array_contains(col, value):

- Returns null if the array is null, true if the array contains the given value, and false otherwise.

```
emp3.select(array_contains(emp3.score_arr1,55)).show()
```

array_union(col1,col2):

- Returns an array of the elements in the union of col1 and col2 without duplicates.

```
emp3.select(array_union(emp3.score_arr1,emp3.score_arr2)).show(truncate=False)
```

array_except(col1, col2):

- Returns an array of the elements in col1 but not in col2 without duplicates.

```
emp3.select(array_except(emp3.score_arr1,emp3.score_arr2)).show(truncate=False)
```

array_intersect(col1, col2):

- Returns an array of the elements in the intersection of col1 and col2 without duplicates.

```
emp3.select(array_intersect(emp3.score_arr1,emp3.score_arr2)).show(truncate=False)
```

array_join(col, delimiter, null_replacement=None):

- Concatenates the elements of `column` using the `delimiter`.
- Null values are replaced with `null_replacement` if set, otherwise they are ignored.

```
emp3.select(emp3.score_arr2,array_join(emp3.score_arr2,'#',null_replacement="1")).show()
```

arrays_zip(*cols):

- Merge arrays. First element of array1 will be merged with first element of array 2 and so on.

```
emp3.select(arrays_zip(emp3.score_arr1,emp3.score_arr2)).show(truncate=False)
```

arrays_overlap(a1,a2):

- Returns true if the arrays contain any common non-null element;
- Returns null if both the arrays are non-empty and any of them contains a null element
- Returns false otherwise.

```
emp3.select(arrays_overlap(emp3.score_arr1,emp3.score_arr2)).show(truncate=False)
```

shuffle(col):

- Random shuffle an array.

```
emp3.select(emp3.score_arr1,shuffle(emp3.score_arr1)).show()
```

create_map(*cols):

- Create a new map column.

```
emp1.select(create_map(emp1.FirstName,emp1.LastName)).printSchema()
```

map_from_entries(col):

- col is array of paired structs.
- Function returns a map created from the given array of entries.

```
df.select(map_from_entries("data").alias("map")).show()
```

map_from_arrays(col1,col2):

- Creates a new map from two arrays.

```
emp3.select(map_from_arrays(emp3.score_arr1,emp3.score_arr2)).printSchema()
```

map_keys():

- Returns an unordered array containing the keys of the map.

```
emp1.select(map_keys(emp1.properties)).show()
```

map_values():

- Returns an unordered array containing the values of the map.

```
emp1.select(map_values(emp1.properties)).show()
```

map_concat(*cols):

- Returns the union of all the given maps.

```
emp1.select(map_concat(emp1.properties,emp1.properties)).show(truncate=False)
```

sequence(start, stop, step =1):

- Generate a sequence of integers from `start` to `stop`, incrementing by `step`.
- If `step` is not set, incrementing by 1 if `start` is less than or equal to `stop`, otherwise -1.

```
emp2.select(emp2.score1, emp2.score2,sequence(emp2.score1,emp2.score2).alias('new_col')).show(truncate=False)
```

DataFrame APIs : na Functions

na Functions

Na Functions are used to work with missing data.

`drop(how='any', thresh=None, subset=None):`

- Remove rows with NULL Values.
- 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 : 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.

```
Ex- data=([('Alice',80,10),('Bob',None,5),('Tom',50,50),(None,None,None),('Robert',30,35))  
schema='name string, age int, height int'  
df = spark.createDataFrame(data,schema)  
df.na.drop().show()
```

na Functions

`fill(value, subset=None):`

- Replace null values.
- value : Value to replace null values with.
- subset: Optional list of column names to consider.

`df.na.fill(50).show() #String Columns are ignored.`

`df.na.fill('Ram').show() #Non-String Columns are ignored.`

`df.na.fill({'age' : 50, 'name' : 'Ram' }).show()`

`df.na.fill(value=100,subset='height').show()`

`replace(to_replace, value=<no value>, subset=None)`

`df.na.replace(10,20).show()`

`df.na.replace({'Alice':'Alex','Bob': 'Cob'},subset='name').show()`

DataFrame APIs : Mathematics and Statistics Functions

Mathematics Functions

abs(col):

Compute the absolute value.

```
df1.select(df1.col1,abs(df1.col1)).show()
```

exp(col):

Computes the exponential of the given value.

```
df1.select(df1.col1,exp(df1.col1)).show()
```

factorial(col):

Computes the factorial of the given value.

```
df1.select(df1.col1,factorial(df1.col1)).show()
```

sqrt(col):

cbrt(col) :

pow(col,n):

```
df.select(df.age,sqrt(df.age),cbrt(df.age),pow(df.age,2)).show()
```

floor():

ceil():

```
df2.select(df2.col1,floor(df2.col1),ceil(df2.col1)).show()
```

Mathematics Functions

round(col,scale=0):

```
df2.select(df2.col1,round(df2.col1,2)).show()
```

trunc(col,format):

- Format : 'year', 'yyyy', 'yy' or 'month', 'mon', 'mm'

```
ord.select(ord.order_date,trunc(ord.order_date,'year')).show()
```

signum(): 1 if n > 0, 0 if n = 0, -1 if n < 0)

```
df2.select(df2.col1,signum(df2.col1)).show()
```

avg(col):

sum(col):

sumDistinct(col):

mean(col):

Count(col):

countDistinct(col):

min(col):

max(col):

Statistics Functions

corr(col1,col2):

Return Pearson Correlation Coefficient.

```
df.select(corr(df.salary,df.age)).show()
```

covar_pop(col1, col2):

Return population covariance.

```
df.select(covar_pop(df.salary,df.age)).show()
```

covar_samp(col1, col2):

Return sample covariance.

var_pop(col): Return population variance of the values in a group.

var_samp(col): Returns the unbiased variance of the values in a group.

variance(col): Returns the population variance of the values in a group.

stddev(col): Returns the unbiased sample standard deviation of the expression in a group.

stddev_pop(col): Returns population standard deviation of the expression in a group.

stddev_samp(col): Returns the unbiased sample standard deviation of the expression in a group.

DataFrame APIs : Explode and Flatten Functions

explode(col):

- Can be used for array and map.
- When an array is passed to this function, it creates a new column and each element of the array in a separate row.
- When a map is passed to this function, it creates two new columns one for key and one for value.
- This will ignore null elements.

```
data =([('Alicia',('Java','Scala'),{'hair':'black','eye':'brown'}),  
        ('Robert',('Spark','Java',None),{'hair':'brown','eye':None}),  
        ('Mike',('CSharp',{}),{'hair':'red','eye':None}),  
        ('John',None,None),  
        ('Jeff',('1','2'),{}))]
```

```
schema = ('empName','Languages','properties')
```

```
emp = spark.createDataFrame(data=data,schema=schema)
```

```
emp.select(emp.empName,explode(emp.Languages)).show()
```

```
emp.select(emp.empName,explode(emp.properties)).show()
```

explode_outer(col): In explode nulls are ignored, but in explode_outer nulls are reported.

```
emp.select(emp.empName,explode_outer(emp.Languages)).show()
```

posexplode(col): Explode with a separate position/index field. Ignores null values.

```
emp.select(emp.empName,posexplode(emp.Languages)).show()
```

```
emp.select(emp.empName,posexplode(emp.properties)).show()
```

posexplode_outer(col): Explode with a separate position/index field. Nulls are reported.

```
emp.select(emp.empName,posexplode_outer(emp.Languages)).show()
```

```
emp.select(emp.empName,posexplode_outer(emp.properties)).show()
```

flatten():

- Convert an Array of Array Column to a single Array Column.
 ArrayType(ArrayType(StringType)) → ArrayType(StringType)
- If a structure of nested arrays is deeper than two levels, only one level of nesting is removed.

Ex-

```
data =('Alicia',(['Java'],['Scala'],['Python'))),\
('Robert',([None],['Java'],['Hadoop']))
)
schema = ('empName','ArrayType')
emp = spark.createDataFrame(data=data,schema=schema)
emp.select(emp.empName,flatten(emp.ArrayType)).show()
```

Ex-

```
data =('Alicia',((1),(2))),\
('Robert',(None,(1)))
)
schema = ('empName','ArrayType')
emp = spark.createDataFrame(data=data,schema=schema)
emp.select(emp.empName,flatten(emp.ArrayType)).show()
```

DataFrame APIs : Formatting Functions

Prepare Data

```
ordItems=spark.read.load('retail_db/order_items',sep=',',format='csv',schema=('order_item_id int,order_item_order_id int,order_item_product_id int,quantity tinyint,subtotal float,price float'))  
df = spark.createDataFrame(((5, "hello")), ('a', 'b'))
```

format_number(col,d): Formats the number X to a format to d decimal places with HALF_EVEN round mode, and returns the result as a string.

```
ordItems.select(ordItems.subtotal,format_number(ordItems.subtotal,4)).show()
```

format_string(format, *cols):

Formats the arguments in printf-style and returns the result as a string column.

```
df.select(format_string('%d %s', df.a, df.b).alias('v')).show()
```

DataFrame API: json Functions

```
### Prepare Data  
data=((1,"""{"Zipcode":85016,"ZipCodeType":"STANDARD","City":"Phoenix","State":"AZ"}"""))  
df_map=spark.createDataFrame(data,("id","value"))
```

```
data = ((1, "(1, 2, 3)"))  
df_arr=spark.createDataFrame(data,("id","value"))
```

```
data=((1, """{"Zipcode":85016,"ZipCodeType":"STANDARD","City":"Phoenix","State":"AZ"}"""))  
df_struct=spark.createDataFrame(data,("id","value"))
```

`from_json(col, schema):`

Convert a JSON string into a Map Type or StructType or ArrayType.

Returns `null`, in the case of an unparseable string.

--Map Type Ex

```
schema=MapType(StringType(),StringType())
df_map_new = df_map.withColumn('map_column',from_json(df_map.value,schema))
df_map_new.printSchema()
```



--Array Type Ex

```
schema=ArrayType(IntegerType())
df_arr_new = df_arr.withColumn('arr_column',from_json(df_arr.value,schema))
df_arr_new.printSchema()
```

--Struct Type Ex

```
schema=StructType((StructField("Zipcode",IntegerType()),StructField("ZipCodeType",StringType()),StructField("city",StringType()),StructField("state",StringType())))
df_struct_new = df_struct.withColumn('struct_column',from_json(df_struct.value,schema))
df_struct_new.printSchema()
```

to_json(col)

Converts a column with StructType or ArrayType or MapType into a JSON string.

Throws an exception, in the case of an unsupported type.

--Using Map Type

```
df_map_new.select(to_json(df_map_new.map_column)).printSchema()
```

--Using Array Type

```
df_arr_new.select(to_json(df_arr_new.arr_column)).printSchema()
```

--Using Struct Type

```
df_struct_new.select(to_json(df_struct_new.struct_column)).printSchema()
```

Collection Type Column



Json String Column

json_tuple(col, *fields):

Extract the elements from JSON string column and create the result as a new columns.

```
df_map.select(json_tuple(col("value"), "Zipcode", "State")).toDF("Zip", "State").show()
```



schema_of_json(json_string):

Use schema_of_json() to create schema string from JSON string column.

```
schemaOfStr=spark.range(1) \
```

```
.select(schema_of_json(lit(''{"id":101, "name":"Robert","City":"Phoenix","State":"AZ"}'')))) \\\n.collect()(0)(0)
```



get_json_object(col,path):

Used to extract the JSON string based on path from the JSON column.

```
df_map.select(col("id"),get_json_object(col("value"),"$ZipCodeType").alias("ZipCodeType")) \\n.show(truncate=False)
```

DataFrame APIs : Other Aggregate Functions

```
### Prepare Data
data = (("James","Sales","NY",None,34),
        ("Alicia","Sales","NY",8600,56),
        ("Robert","Sales","CA",8100,30),
        ("John","Sales","AZ",8600,31),
        ("Ross","Sales","AZ",8100,33),
        ("Kathy","Sales","AZ",1000,39),
        ("Lisa","Finance","CA",9000,24),
        ("Deja","Finance","CA",9900,40),
        ("Sugie","Finance","NY",8300,36),
        ("Ram","Finance","NY",7900,53),
        ("Satya","Finance","AZ",8200,53),
        ("Kyle","Marketing","CA",8000,25),
        ("Reid","Marketing","NY",9100,50)
)
schema=("empname","dept","state","salary","age")
df = spark.createDataFrame(data=data,schema=schema)
```

first(col, ignorenulls=False) :

Returns the first value in a group.

Return the first non-null value it sees when ignoreNulls is set to true.

df.select(first(df.salary)).show()

last(col, ignorenulls=False):

greatest(*cols):

Returns the greatest value of the list of column names, skipping null values.

df.select(greatest(df.salary,df.age)).show()

least(*cols):

Returns the leastvalue of the list of column names, skipping null values.

skewness(col):

Returns the skewness of the values in a group.

df.select(skewness(df.salary)).show()

collect_list(col): Returns a list of objects with duplicates.

df.select(collect_list('age')).show(truncate=False)

DataFrame APIs : Other Misc Functions

###Prepare Data

```
df = spark.createDataFrame([('Robert',1, None,114.0), ('John',None, 2577,float('nan'))), ("name", "id","phone","stAdd"))
```

ascii(col):

- Computes the numeric value of the first character of the string column.

```
df.select(ascii(lit('a'))).show()
```

bin(col):

- Returns the string representation of the binary value of the given column.

```
df.select(df.phone,bin(df.phone)).show()
```

expr()

Part 3

DataFrame Extraction

- csv
- text
- parquet
- orc
- avro
- json
- hive
- jdbc

Repartition & Coalesce

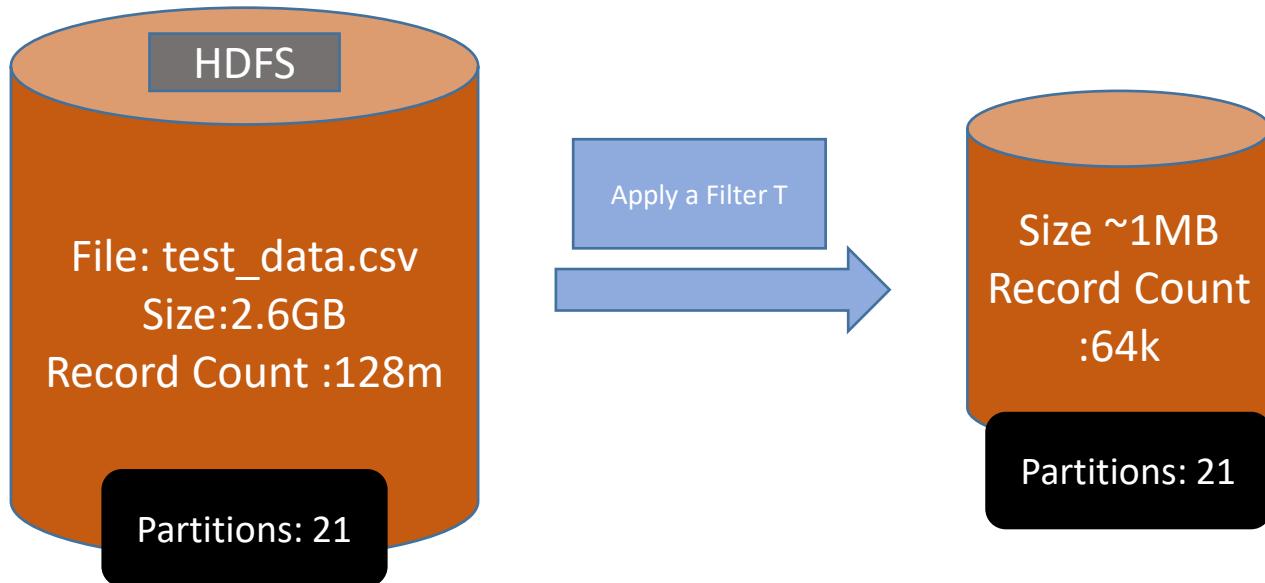
What is Partition ?

- Datasets are huge in size and so they cannot fit into a single node and so they have to be partitioned across different nodes.
- Partition in spark is basically an atomic chunk of data stored on a node in the cluster. They are the basic units of parallelism.
- One partition can not span over multiple machines.
- Spark automatically partitions RDDs/DataFrames and distributes the partitions across different nodes.
- We can also configure the optimal number of partitions. Having too few or many partitions is not good.
- How Spark does the default Partitioning of Data ?

Spark checks the HDFS Block size. The HDFS Block size for Hadoop 1.0 is 64mb and Hadoop 2.0/YARN is 128MB. It creates one partition for each block size.

Ex- We have a file of 500MB, so 4 partitions would be created.

- At times programmers are required to change the number of partitions based on the requirements of the application job. The change can be to increase the number of partitions or decrease the number of partitions.
- So we would either apply the repartition or coalesce.



```
### Create some Dump data for testing
df = spark.range(1000000)
df = df.select(df.id,df.id*2,df.id*3)
df = df.union(df)
```

```
### Save the file at a HDFS Path
df.write.save('/user/test/test_data.csv',format='csv')
```

```
### Testing
df_new = spark.read.load('/user/test/test_data.csv',format='csv',schema=('col1 int, col2 int, col3 int'))
df_new.rdd.getNumPartitions()
--- Apply Filter
df_filter = df_new.where(df_new.col1 < 501)
df_filter.rdd.getNumPartitions()
```

Repartition

Repartition(numPartitions, *cols):

- DataFrame is hash partitioned.
- Create almost equal sized partitions.
- Can increase or decrease the level of parallelism.
- Internally, this uses a shuffle to redistribute data from all partitions leading to very expensive operation. So avoid if not required.
- Spark performs better with equal sized partitions. If you need further processing of huge data, it is preferred to have equal sized partitions and so we should consider using repartition.
- If you are decreasing the number of partitions, consider using **coalesce**, where the movement of data across the partitions is lower.

```
df_filter.rdd.glom().map(len).collect()
```

```
df_filter = df_filter.repartition(5)
df_filter.rdd.glom().map(len).collect()
```

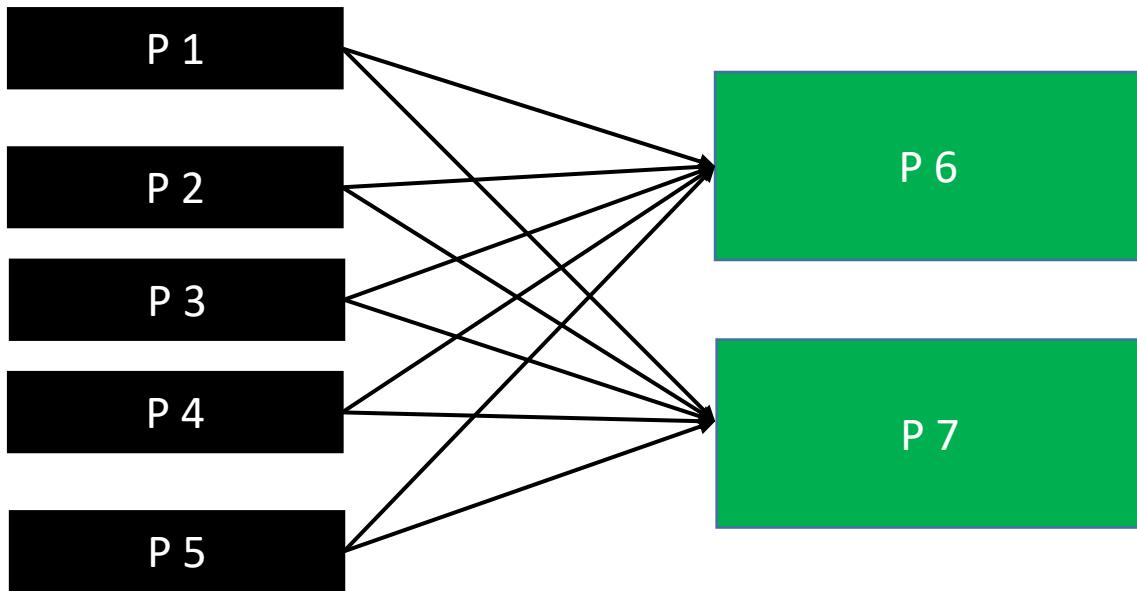
Ex-

```
data=([('Ram',30),('Raj',25),('James',30),('Joann',25),('Kyle',25),('Robert',30),('Reid',35),('Sam',35))
df = spark.createDataFrame(data=data,schema=('name','age'))
df = df.repartition('age')
df.rdd.getNumPartitions()
spark.conf.get("spark.sql.shuffle.partitions")
```

```
df = df.repartition(4,'age')
df.rdd.getNumPartitions()
```

Repartition

repartition(2)
Shuffle in Repartition



coalesce

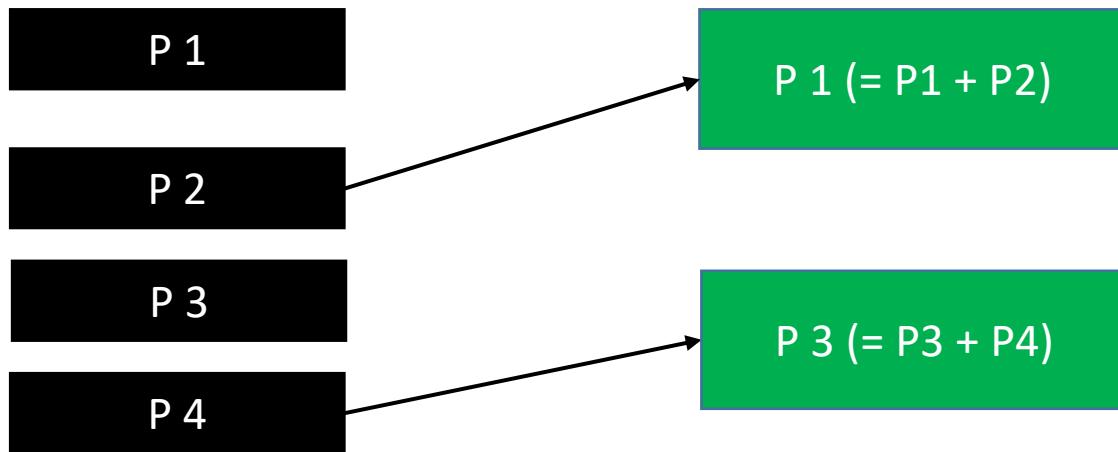
coalesce(numPartitions):

- Return a new dataframe that is reduced into numPartitions partitions.
- Optimized version of repartition().
- No shuffling.
- Results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead each of the 100 new partitions will claim 10 of the current partitions.
- If a larger number of partitions is requested, it will stay at the current number of partitions.

```
df_filter.rdd.glom().map(len).collect()  
df_filter = df.coalesce(5)  
df_filter.rdd.glom().map(len).collect()
```

coalesce

coalesce(2)
No Shuffle in coalesce



Repartition vs Coalesce

Repartition

1. Repartition does a full shuffle.
2. Preferable used to increase number of partitions.
3. Repartition creates new partitions and does a full shuffle.
4. Repartition results in roughly equal sized partitions.
5. Coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions. You'll usually need to repartition datasets after filtering a large data set. I've found repartition to be faster overall because Spark is built to work with equal sized partitions.
6. It's critical to repartition after running filtering queries. The number of partitions does not change after filtering, so if you don't repartition, you'll have way too many memory partitions. The more the filter reduces the dataset size, the bigger the problem.

Coalesce

1. Coalesce avoids full shuffle.
2. Preferable used to decrease number of partitions.
3. Coalesce uses existing partitions to minimize the amount of data that's shuffled.
4. Coalesce results in partitions with different size of data.
5. Coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions. You'll usually need to repartition datasets after filtering a large data set. I've found repartition to be faster overall because Spark is built to work with equal sized partitions.
6. It's critical to repartition after running filtering queries. The number of partitions does not change after filtering, so if you don't repartition, you'll have way too many memory partitions. The more the filter reduces the dataset size, the bigger the problem.

DataFrame Extraction

```
### Prepare Data  
ord = spark.read.load('practice/retail_db/orders',sep=',',format='csv',schema=('order_id int,order_date  
timestamp, order_customer_id int,order_status string'))
```

Extraction : csv File

- csv can take a lot of parameters. For complete list, use help command.
- Below are few important parameters.

csv Params:

- path
- mode : default 'error' or 'errorIfExists' : Throw exception.
 - 'append' :Append contents to existing data.
 - 'overwrite' : Overwrite existing data.
 - 'ignore' : Ignore the operation if data exists.
- compression (none, bzip2, gzip, lz4,snappy and deflate)
- sep (default ' , ')
- header (True or False. default – False)
- dateFormat (default – 'yyyy-MM-dd')
- timestampFormat (Default – 'yyyy-MM-dd'T'HH:mm:ss.SSSXXX')
- ignoreLeadingWhiteSpace (Default – True)
- ignoreTrailingWhiteSpace (Default – True)
- And more ...

PS - Calculate number of orders in each status and save into 2 files.

```
ordCountByStatus = ord.groupBy("order_status").count()  
ordCountByStatus.coalesce(2).write.save('practice/dump/retail_db/ordCountByStatus',format='csv',sep='#',mode='overwrite',header=True,compression='gzip')
```

Extraction : text file

- To save into a text file, make sure to append all columns into a single column with String type.
- Do not support a header.
- To create a text file from dataframe, it should have only 1 column. So it is preferred to use a rdd for creating text file.

```
ordRDD = ordDF.rdd
```

```
ordRDD.saveAsTextFile('retail_db/orders')
```

Text Params:

- path
- compression
- linesep (default – '\n')

PS: Convert the Order csv file to text file with fixed length of tab and create 10 output files.

```
from pyspark.sql.functions import concat_ws
```

```
ordText = ord.select(concat_ws('\t',ord.order_id,ord.order_date,ord.order_customer_id,ord.order_status).alias('col1'))
```

```
ordText.repartition(10).write.save('practice/dump/retail_db/orderText',format='text')
```

Extraction : parquet File

parquet Params:

- path
- mode (Same as csv)
- partitionBy (Apply partition as per the column. One file per each partition.)
- Compression (Default Compression - spark.sql.parquet.compression.codec)

PS: Convert the Order csv file to parquet file. Create one file for each order status category.

```
ord.write.save('practice/dump/retail_db/orderParquet',format='parquet',mode='overwrite',partitionBy="order_status")
```

Extraction : orc File

orc Params:

- path
- mode (Same as csv)
- partitionBy (Apply partition as per the column. One file per each partition.)
- Compression (Default Compression - spark.sql.orc.compression.codec)

PS: Convert the Order csv file to orc file. Create one file for each order status category.

```
ord.write.save('practice/dump/retail_db/orderOrc',format='orc', partitionBy="order_status")
```

Extraction : json File

json Params:

- path
- mode
- compression (none, bzip2, gzip, lz4,snappy and deflate)
- dateFormat (default – yyyy-MM-dd)
- timestampFormat (default - yyyy-MM-dd'T'HH:mm:ss.SSSXXX)
- lineSep (default – '\n')

PS: Convert the Order csv file to json file. File should be bzip2 compressed and total 1 output file.

```
ord.coalesce(1).write.save('practice/dump/retail_db/orderJson',format='json', compression='bzip2')
```

Extraction : avro File

- Avro Is a third party file format. We need to import its package or jar file while launching pyspark or spark-submit. Spark by default does not support it.
- `pyspark2 --packages org.apache.spark:spark-avro_2.11:2.4.4`

PS: Convert the Order csv file to avro file.

```
ord.write.save('practice/dump/retail_db/orderAvro',format='avro')
```

Extraction: hive Table

insertInto(tableName, overwrite=False):

- Insert data to a Hive Table.
- By default it will append the data. With overwrite=True, it will overwrite the existing data.
- Table should be available, otherwise throws exception.

Ex-

```
ord.write.insertInto('db2.orders')
```

DataFrame: hive Table

saveAsTable(name, format=None, mode=None, partitionBy=None, **options)

- format – File Format
If we do not pass any file format or compression, by default it is snappy compressed and parquet format.
- mode - append, overwrite, error, ignore
Table should not exist if we do not pass a mode.
- partitionBy - Create a partition on the partition column. Number of files created for the hive table is equal to the number of distinct values in that column.
- compression: By default it is snappy compressed.
Pass compression='none' to avoid compression.

Ex-

```
ord.write.saveAsTable(name='db2_orders.order_test', format='orc')
```

Ex-

```
ord.write.saveAsTable(name='db2_orders.order_test', format='orc', mode='overwrite')
```

Ex-

```
ord.write.saveAsTable('db2_orders.orders1',partitionBy='order_status',format='orc',mode='overwrite',compression='none')
```

Extraction : jdbc

- Make sure JDBC jar file is registered using –packages or –jars while launching pyspark or spark-submit
- Typical jdbc files are located at /usr/share/java folder. You may keep it there or copy it to your project lib folder.
pyspark2 --jars <jdbc driver jar file>
- Complete list of properties:
<https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html>

mode: append, overwrite, ignore, error or errorIfExists (default case)

Ex-1 (Write into a Table. Table should not exist.)

```
df.write.format("jdbc") \
.option("url", url) \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("dbtable","new_orders" ) \
.option("user", someUser) \
.option("password", somePassword) \
.save()
```

If Table exist, use mode.

```
df.write.format("jdbc") \
.option("url", url) \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("dbtable","new_orders" ) \
.option("user", someUser) \
.option("password", somePassword) \
.mode("overwrite") \
.save()
```

Extraction : jdbc

createTableColumnTypes : Apply column datatypes instead of defaults.

Ex -2 (Using *createTableColumnTypes*)

```
ord.select('order_id').write.format("jdbc") \
.option("url", url) \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("createTableColumnTypes","order_id char(10)" ) \
.option("dbtable","new_orders" ) \
.option("user", someUser) \
.option("password", somePassword) \
.mode("overwrite") \
.save()
```

Extraction : jdbc

batchsize: The JDBC batch size, which determines how many rows to insert per round trip. This can help performance on JDBC drivers. This option applies only to writing. It defaults to 1000.

fetchsize: reading.

```
ord.select('order_id').write.format("jdbc") \
.option("url", url) \
.option("driver", "oracle.jdbc.driver.OracleDriver") \
.option("createTableColumnTypes","order_id char(10)" ) \
.option("dbtable","new_orders" ) \
.option("user", someUser) \
.option("password", somePassword) \
.option("batchsize",5000) \
.mode("overwrite") \
.save()
```

Extraction : jdbc

queryTimeout: The number of seconds the driver will wait for a Statement object to execute to the given number of seconds. Zero is default and means there is no limit.

```
ord.select('order_id').write.format("jdbc") \  
.option("url", url) \  
.option("driver", "oracle.jdbc.driver.OracleDriver") \  
.option("createTableColumnTypes","order_id char(10)" ) \  
.option("dbtable","new_orders" ) \  
.option("user", someUser) \  
.option("password", somePassword) \  
.option("queryTimeout",1) \  
.mode("overwrite") \  
.save()
```

Performance & Optimization

1. Caching Data in Memory
2. Configuration Options
3. Join Strategy Hints for SQL Queries
4. Coalesce Hint for SQL Queries
5. Adaptive Query Execution

Join Strategies

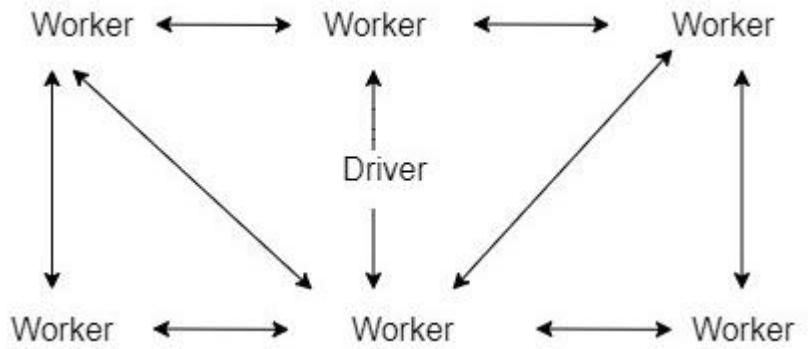
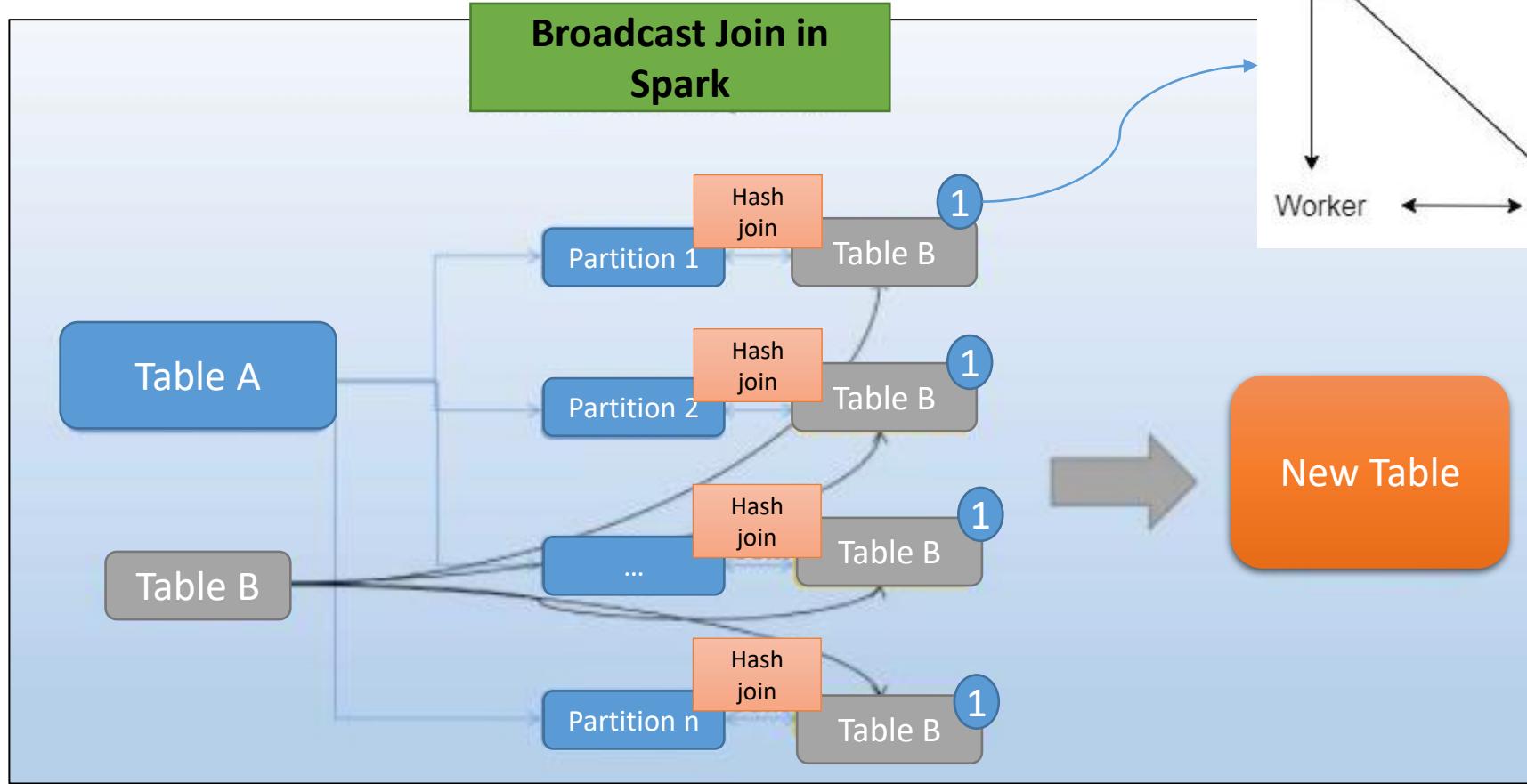
Join Strategies

- Spark has below important join strategies.
 - ✓ Broadcast Join (Hint – BROADCAST)
 - ✓ Shuffle Hash Join (Hint - SHUFFLE_HASH)
 - ✓ Sort Merge Join (Hint - SORT MERGE)
 - ✓ Cartesian Product Join (Hint- BROADCAST)
 - ✓ Broadcast Nested Loop Join(Hint – SHUFFLE_REPLICATE_NL)

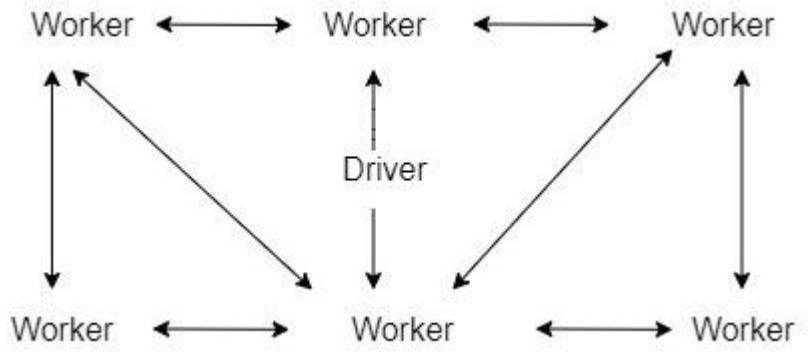
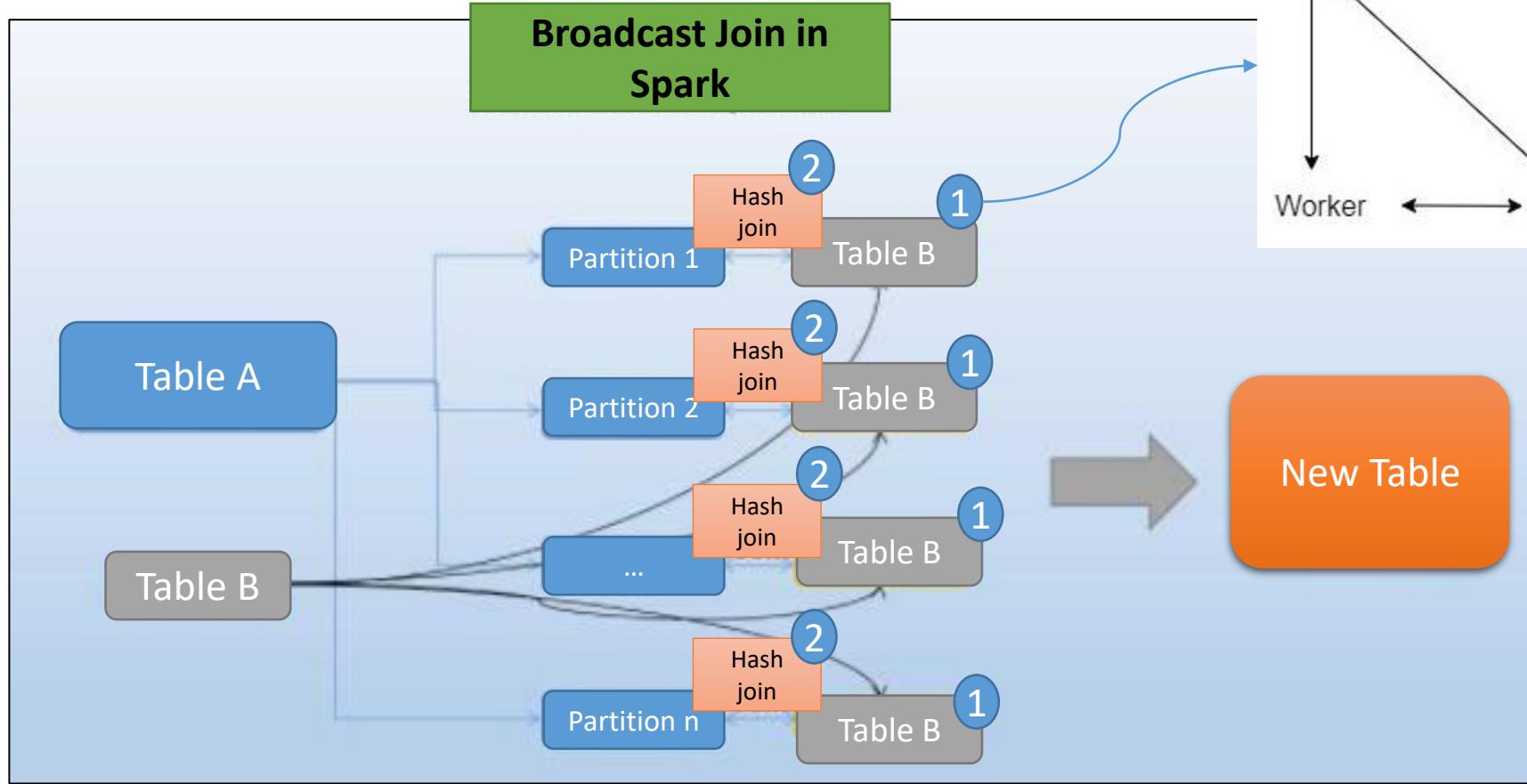
Broadcast Join

Divided into two Steps:

1. Broadcast the smallest dataset to all executors.
2. Perform a hash join

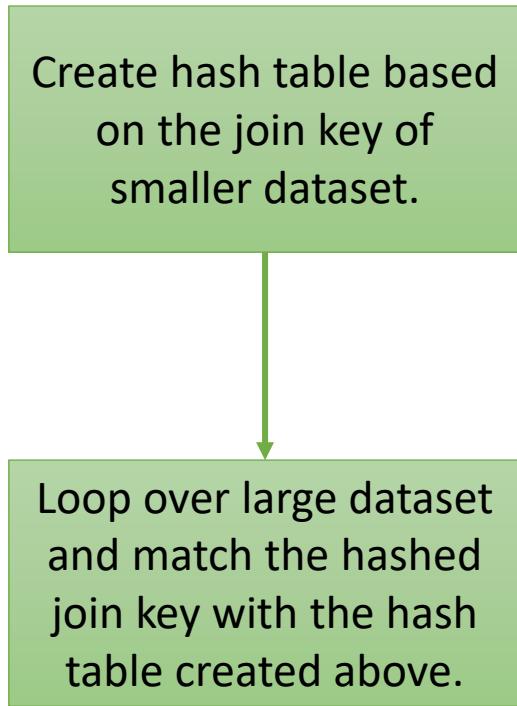


BitTorrent Protocol/
peer to peer protocol.



BitTorrent Protocol/
peer to peer protocol.

Hash Join: Only supported for “=” join.



Broadcast Join:

- One of the most impactful performance Impact optimization techniques we can use.
- It performs a join on two datasets by first broadcasting the smaller one to all Spark executors using BitTorrent peer to peer protocol.
- In this way, each executor has all the information required to perform the join at its location, without needing redistributing data and shuffle.
- Broadcast join can be very efficient for joins between a large table (fact) with relatively small tables (dimensions) that could then be used to perform a **star-schema join**.
- Size of the smaller table should be lesser than: `spark.sql.autoBroadcastJoinThreshold`. Configurable.

Default Size: 10MB

```
int(spark.conf.get("spark.sql.autoBroadcastJoinThreshold"))/1024/1024
```

- Recently Spark has increased the maximum size for the broadcast table from 2GB to 8GB. Thus, it is not possible to broadcast tables which are greater than 8GB.
- Also called map-side join or replicated join.
- Use Hint `BROADCAST` to force Spark Optimizer.

Auto Detection:

- In many cases, Spark can automatically detect whether to use a broadcast join or not, depending on the size of the data. If Spark can detect that one of the joined DataFrames is small (10 MB by default), Spark will automatically broadcast it for us.
- Spark will perform auto-detection when
 - ✓ Constructs a DataFrame from scratch, e.g. spark.range.
 - ✓ It reads from files with schema and/or size information, e.g. Parquet, Avro

Ex – Lets join orders and orderItems files.

```
ordDF = spark.read.load('practice/retail_db/orders',sep=',',format='csv',schema=('order_id int,order_date timestamp,\norder_customer_id int,order_status string'))
```

```
ordItemsDF=spark.read.load('practice/retail_db/order_items',sep=',',format='csv',schema=('order_item_id int,\norder_item_order_id int,order_item_product_id int,quantity tinyint,subtotal float,price float'))
```

```
joined= ordDF.join(ordItemsDF,ordDF.order_id == ordItemsDF.order_item_order_id)
```

```
Joined.explain()
```

Sometimes Not:

- But not always, Spark can do auto detection. For Ex- Local Collection.
- The reason is that Spark will not determine the size of a local collection because it might be big.
- In this case, we can force optimizer to use BROADCAST Hint.

Ex –

```
largeDF = spark.range(1,1000000000)
```

```
data = [(1, 'a'),(2, 'b'),(3, 'c')]
```

```
schema= ['id', 'col2']
```

```
smallDF=spark.createDataFrame(data,schema)
```

```
joined = largeDF.join(smallDF, "id")
```

```
joined.explain()
```

Ex- Use hint.

```
from pyspark.sql.functions import broadcast
```

```
joined = smallDF.hint('broadcast').join(broadcast(largeDF), "id")
```

```
joined.explain()
```

Testing:

- If you want to test the joins without presence of automatic optimization, set `spark.sql.autoBroadcastJoinThreshold` to -1.
- This will disable the automatic broadcast join detection.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

Notes:

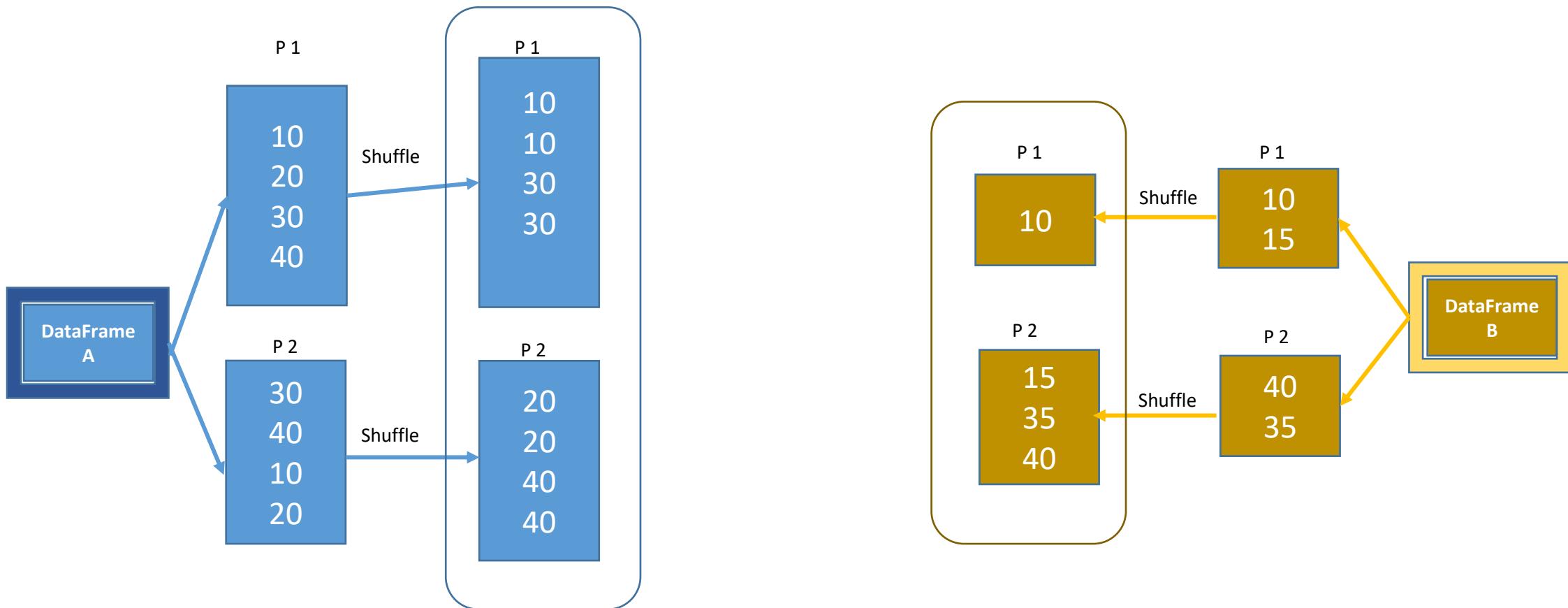
- Table needs to be broadcast less than `spark.sql.autoBroadcastJoinThreshold` the configured value, default 10M (or add a broadcast join the hint).
- Only supported for '=' join.
- Supported for all join types (inner, left, right) except full outer join.
- Faster than any other join strategies.

Shuffle Hash Join

Shuffle Hash Join is divided into two steps:

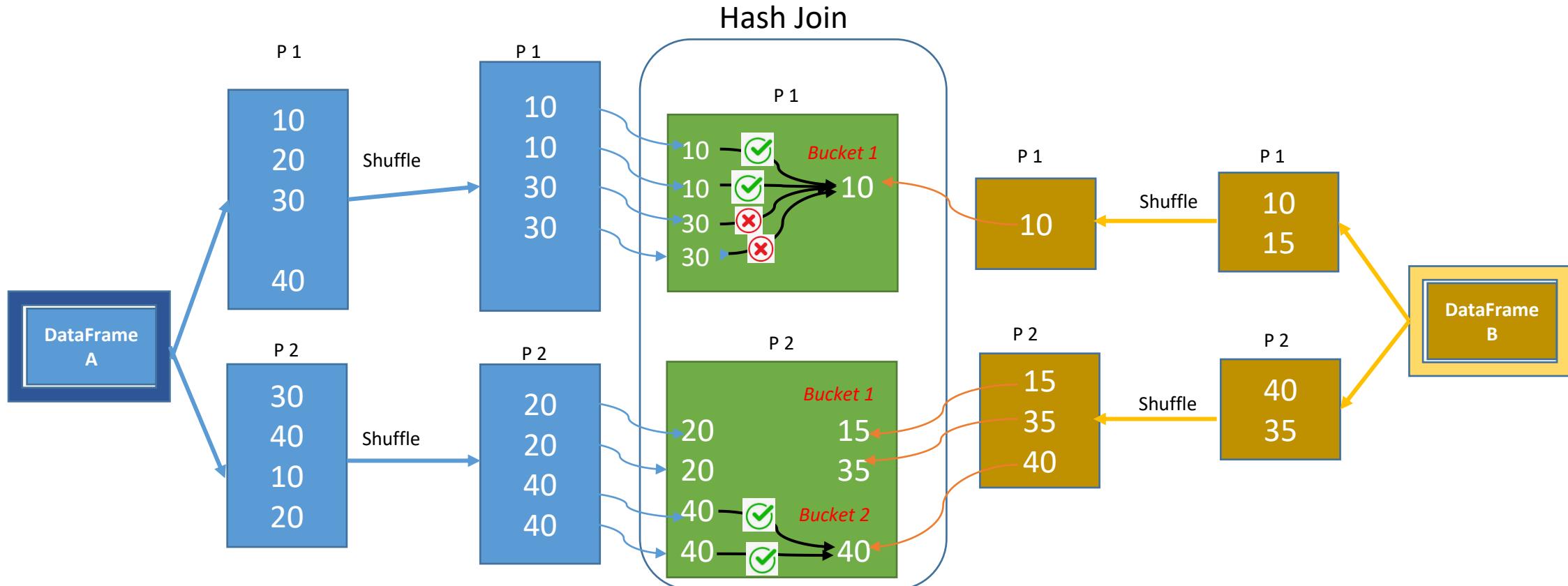
Step#1 : (Shuffle Phase):

- Both datasets are read and shuffled.
- So the same keys from both sides end up in the same partition or task.

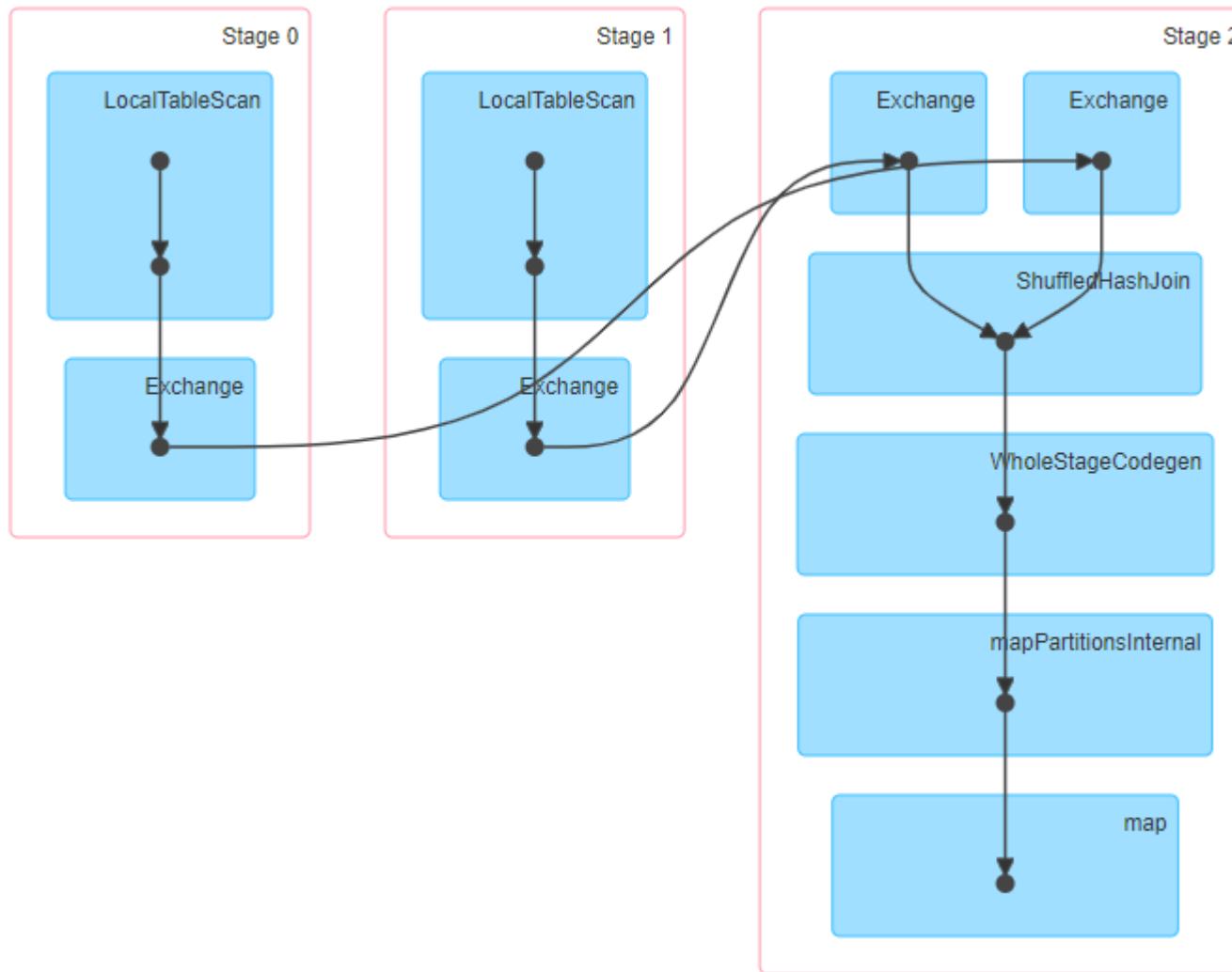


Step#2 : (Hash Join Phase):

- After Shuffle phase, spark picks one side based on statistics(generally the smaller dataset) and it will be hashed into buckets.
- Then hash join is performed.



DAG Visualization:



When does shuffle hash join work:

- Only for equi joins ('=').
- Works well when a dataset can not be broadcasted but one side of the partitioned data after shuffling will be small enough for hash join.

When does shuffle hash join not work:

- Non-equi Joins (<,>,<=,>= etc)
- Does not work with data which are heavily skewed.

Note:

- The join keys don't need to be sortable.
- Supported for all join types except full outer joins.
- Expensive join as both shuffle and hashing are involved. Spark prefers Sort merge join than Shuffle Hash Join. By default `spark.sql.join.preferSortMergeJoin` is True, but configurable.
- Hint: SHUFFLE_HASH

```
spark.conf.set("spark.sql.join.preferSortMergeJoin", "false")
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 2)
```

Ex –

```
df1 = spark.range(1,10000000000)
df2 = spark.range(1,10000000)
```

```
spark.conf.set("spark.sql.join.preferSortMergeJoin", False)
joined = df1.join(df2,"id")
```

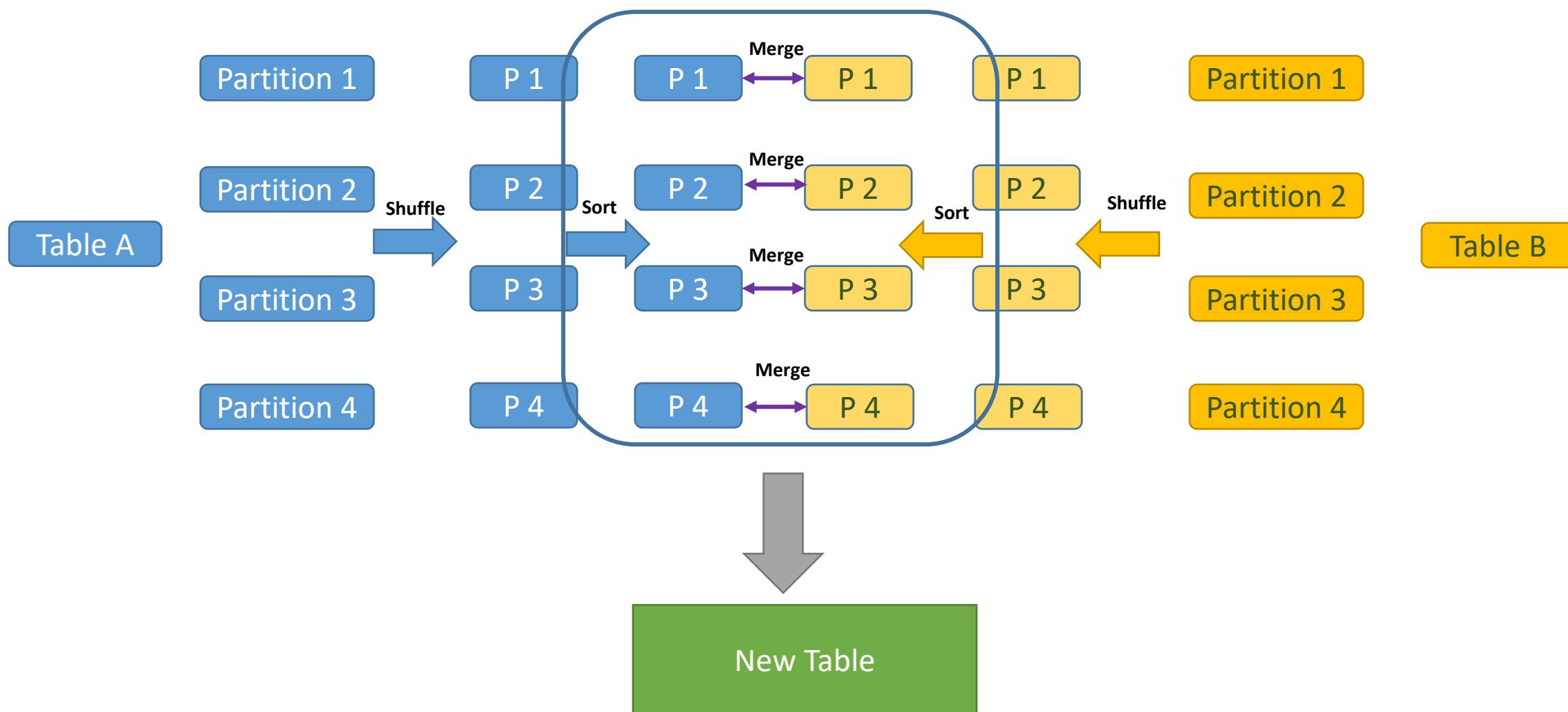
```
joined.explain()
```

```
joined = df1.hint('shuffle_hash').join(df2,"id")
```

Shuffle Sort Merge Join

Shuffle Sort Merge Join:

- From spark 2.3 Merge-Sort join is the default join algorithm in spark. However, this can be turned down by using the internal parameter ‘spark.sql.join.preferSortMergeJoin’ which by default is true.
- Divided into two steps:
 - ✓ The first step is to shuffle and sort the join keys of both the datasets.
 - ✓ Apply the merge algorithm.
- Hint: merge



User Table

| Id | Name |
|----|--------|
| 2 | Json |
| 1 | Kyle |
| 4 | Reid |
| 3 | Robert |

Product Table

| Product Id | User Id |
|------------|---------|
| 101 | 2 |
| 102 | 4 |
| 103 | 4 |
| 104 | 1 |

Sorting on Joining Key

| Id | Name |
|----|--------|
| 1 | Kyle |
| 2 | Json |
| 3 | Robert |
| 4 | Reid |

| Product Id | User Id |
|------------|---------|
| 104 | 1 |
| 101 | 2 |
| 102 | 4 |
| 103 | 4 |

Merge

Notes:

- ✓ Only supported for '=' join.
- ✓ The join keys need to be sortable.
- ✓ Supported for all join types (inner, left, right etc).

Ex –

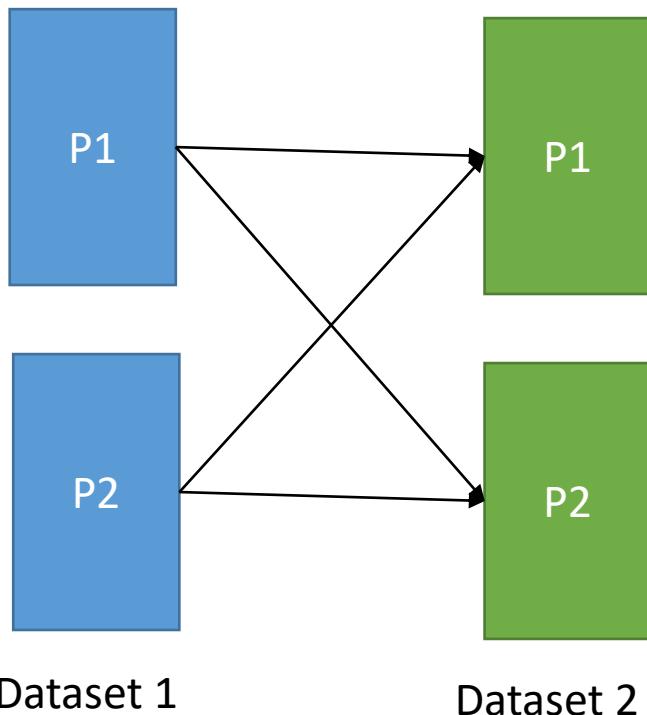
```
df1 = spark.range(1,10000000000)
df2 = spark.range(1,10000000)
joined = df1.join(df2,"id")
joined.explain()

joined = df1.hint('merge').join(df2,"id")
```

Cartesian Product Join

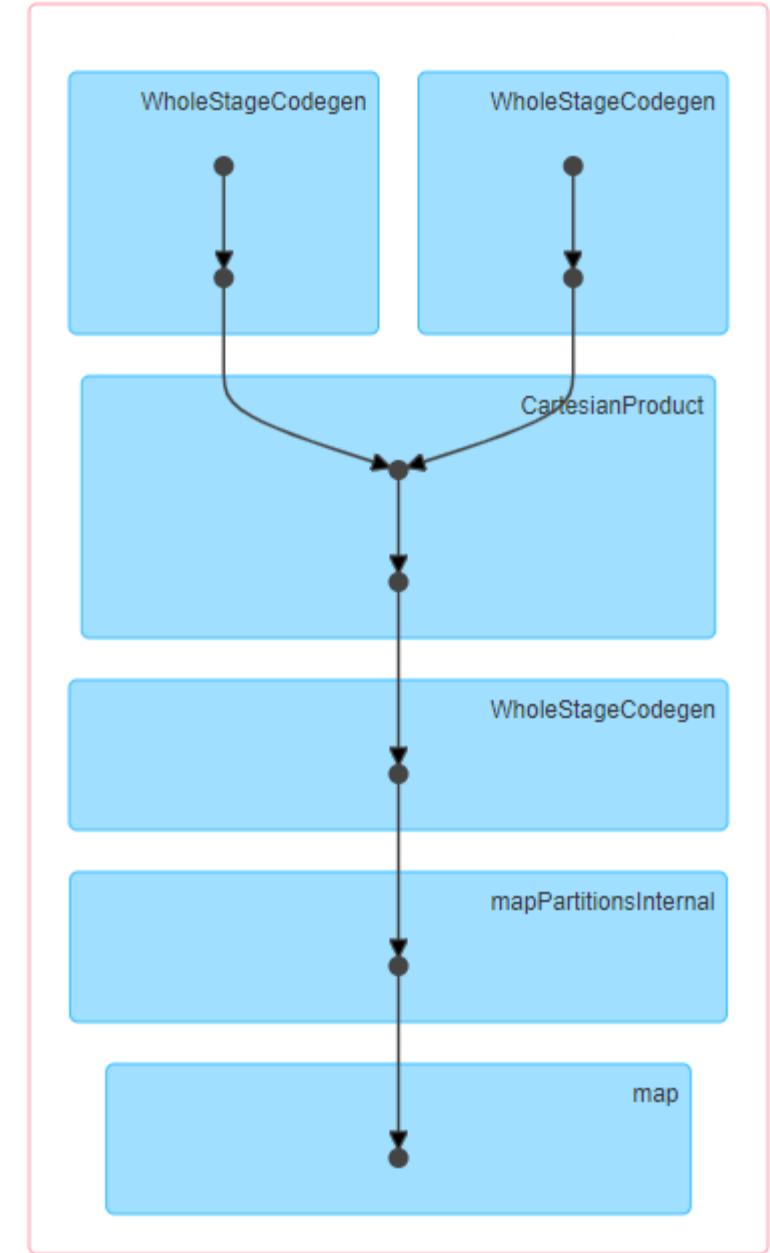
Cartesian Product Join:

- Steps:
 - ✓ Both datasets are read.
 - ✓ All partitions from one of the dataset are sent to all partitions in the other dataset. So shuffle.
 - ✓ Once partitions from both dataset are available on one side, a nested loop join is performed.
 - ✓ If there are M records in first dataset and N records in the second dataset, nested loop is performed on $M * N$ records.
- Very expensive and high possibility of OOM errors. Lots of shuffle.
- Supports both “=” and non-equi joins ($\leq, <, \geq, >$)
- Supports all join types.
- Hint : `shuffle_replicate_nl`



```
for rec in table1:  
    for rec in table2:  
        on table1.join_key = table2.join_key
```

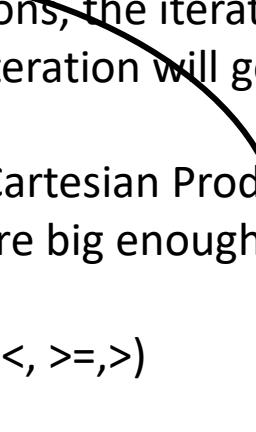
DAG Visualization: The join is executed all in 1 stage.



Broadcast Nested Loop Join

Broadcast Nested Loop Join:

- 2 Steps:
 - ✓ Broadcast: Smallest dataset is broadcasted to all executors processing the bigger dataset.
 - ✓ Nested Loop Join Phase: Every Record from one dataset is attempted to join with every record from another dataset in a nested loop.
- Since this join is used for non-equi conditions, the iteration can not stop as soon as a match is encountered like in Sort Merge Join. The iteration will go through the entire dataset.
- No sort in this join.
- Very slow, but no Shuffle. Preferred than Cartesian Product Join.
- This join will not work when either sides are big enough for broadcasting and you could see Out Of Memory exceptions.
- Supports both “=” and non-equi joins (\leq , $<$, \geq , $>$)
- Supports all join types.



```
for rec in table1:  
    for rec in table2:  
        on table1.join_key = table2.join_key
```

```
spark.conf.set("spark.sql.crossJoin.enabled",True)
df1 = spark.range(1,1000)
df2 = spark.range(1,100)
joined = df1.join(df2)
joined.explain()
```

| | Cartesian Join | Broadcast Hash Join | Sort merge Join | Shuffle Hash Join | Broadcast Nested Loop Join |
|---------------------------|----------------|---------------------|-----------------|-------------------|----------------------------|
| Inner-Join | ✓ | ✓ | ✓ | ✓ | ✓ |
| Left-Join | ✗ | ✓ | ✓ | ✓ | ✓ |
| Right-Join | ✗ | ✓ | ✓ | ✓ | ✓ |
| Cross-Join | ✗ | ✓ | ✓ | ✓ | ✓ |
| Left-Semi_Join | ✗ | ✓ | ✓ | ✓ | ✓ |
| Left-Anti-Semi-Join | ✗ | ✓ | ✓ | ✓ | ✓ |
| Outer-Join | ✗ | ✗ | ✓ | ✓ | ✓ |
| Support Equi Join | ✓ | ✓ | ✓ | ✓ | ✓ |
| Support Non Equi Join | ✓ | ✗ | ✗ | ✗ | ✓ |
| Require Sortable Join key | ✗ | ✗ | ✓ | ✗ | ✗ |

How Spark Prioritize Join Strategies:

When More than one hint is specified and hints are applicable:

- If its an '=' join:
 - ✓ Broadcast hint : Pick broadcast hash join
 - ✓ Merge hint : Pick sort-merge join
 - ✓ Shuffle_Hash hint : Pick shuffle hash join
 - ✓ Shuffle_replicate_nl hint : Pick Cartesian Product join.
- If its not '=' join:
 - ✓ Broadcast hint: Pick broadcast nested loop join.
 - ✓ Shuffle_replicate_nl : Pick Cartesian product if join type is inner like.

When no hints are specified or hints are not applicable

- If its an '=' join:
 - ✓ Pick broadcast hash join if one side is small enough to broadcast, and the join type is supported.
 - ✓ Pick shuffle hash join if one side is small enough to build the local hash map, and is much smaller than the other side, and spark.sql.join.preferSortMergeJoin is false.
 - ✓ Pick sort-merge join if join keys are sortable.
 - ✓ Pick Cartesian product if join type is inner .
 - ✓ Pick broadcast nested loop join as the final solution. It may OOM but there is no other choice.
- If its not '=' join:
 - ✓ Pick broadcast nested loop join if one side is small enough to broadcast.
 - ✓ Pick cartesian product if join type is inner like.
 - ✓ Pick broadcast nested loop join as the final solution. It may OOM but we don't have any other choice.

```
>>> spark.conf.set("spark.sql.crossJoin.enabled",True)
>>> df1 = spark.range(1,1000)
>>> df2 = spark.range(1,100)
>>> joined = df1.join(df2)
>>> joined.explain()
== Physical Plan ==
BroadcastNestedLoopJoin BuildRight, Inner
:- *(1) Range (1, 1000, step=1, splits=2)
+- BroadcastExchange IdentityBroadcastMode
  +- *(2) Range (1, 100, step=1, splits=2)
```

Driver Configurations

Driver Options:

- When we apply collect(), take() operations on datasets, it requires the data to be moved to Driver. If we do so on huge dataset, it can crash the driver process with Out Of Memory errors(OOM).
- If you observe, we perform most of the computational work of a Spark Job in the Executors and so we rarely required to do any performance tuning for the driver.
- However sometimes, the job may fail if they collect too much data to the driver.
- Setting a proper limit can protect the driver from out of memory errors.

Spark-submit Options:

--driver-memory : Memory for driver (e.g. 1000M, 2G) (Default: 1024M)

- Driver Memory is the amount of memory to use for driver process, i.e. the process running the main() function of the application and where SparkContext is instantiated.

--driver-cores :

- Number of cores used by the driver, only in cluster mode (Default: 1).
- Generally not required unless you want to perform some local computations in parallel.

Configuration Properties:

spark.driver.memory : Default 1024

spark.driver.cores : Default 1

spark.driver.maxResultSize:

- Limit of each Spark action (e.g. collect) in bytes.
- Should be at least 1M, or 0 for unlimited. Jobs will be aborted if the total size is above this limit.
- Having a high limit may cause out-of-memory errors in driver (depends on spark.driver.memory and memory overhead of objects in JVM).
- Setting a proper limit can protect the driver from out-of-memory errors.

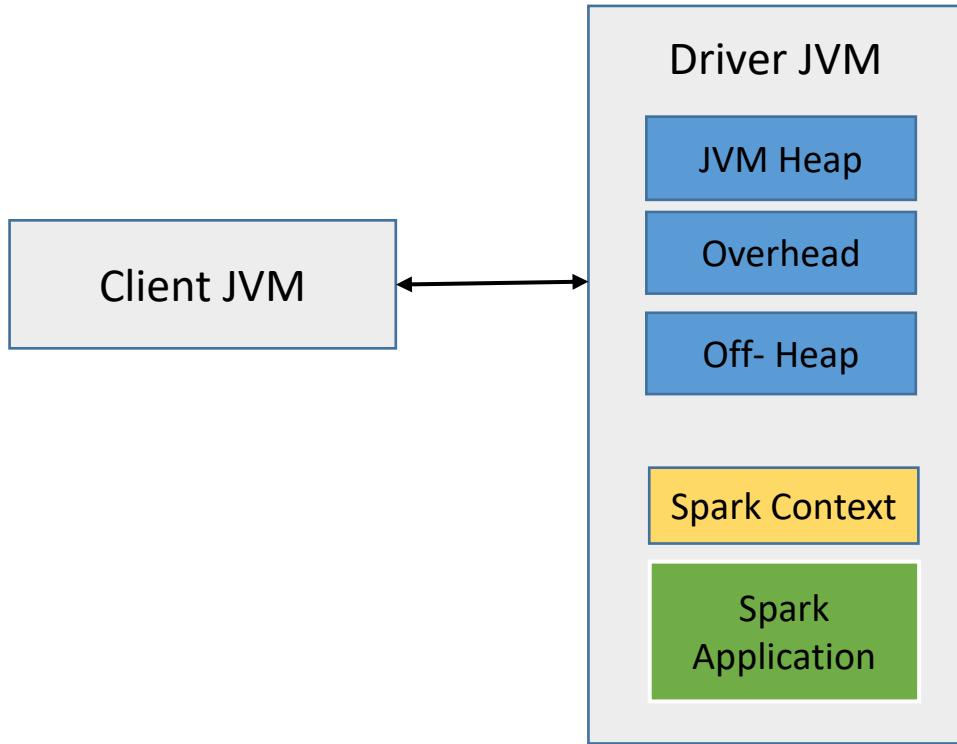
spark.driver.memoryOverhead: Default driverMemory * 0.10, with minimum of 384

- Amount of overhead (non-heap) memory to be allocated per driver process 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%).

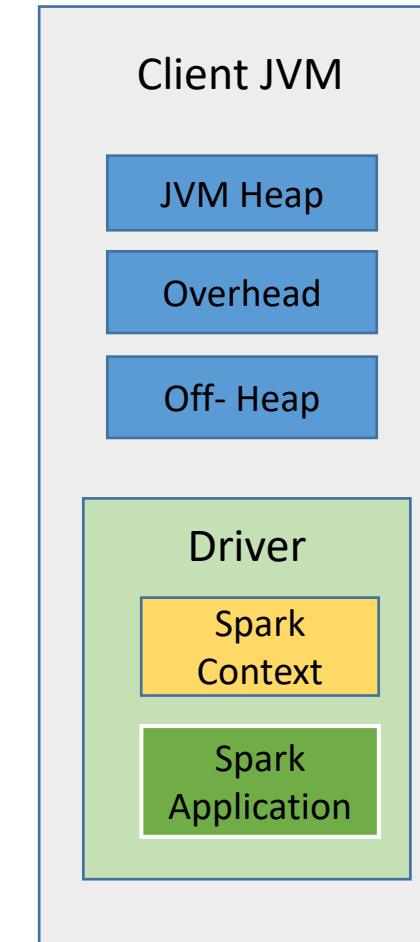
All Properties:

<https://spark.apache.org/docs/latest/configuration.html>

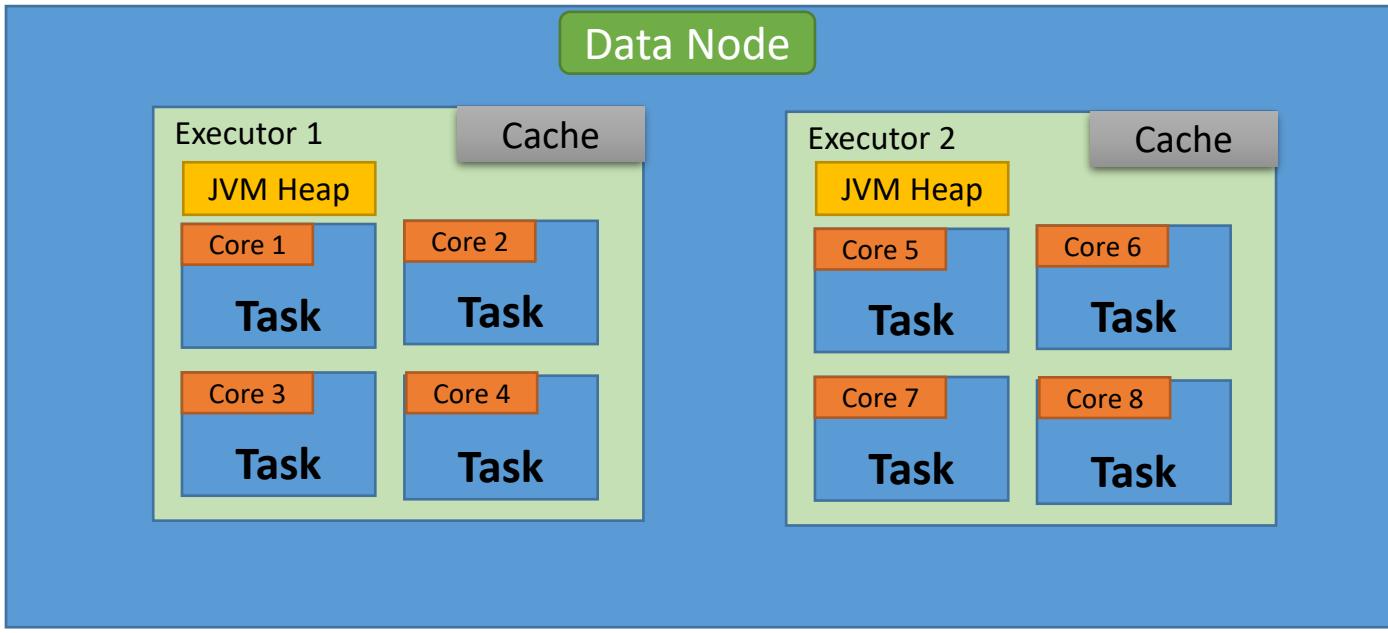
Driver in Cluster deploy mode:



Driver in Client deploy mode:



Executors Configurations



- **Executors and Cores:**

- ✓ Executors are created in worker/data nodes and they are in charge of running tasks in a given spark job.
- ✓ Each executor comprises a JVM. They are launched at the beginning of a spark application and run the entire lifetime of the spark job.
- ✓ After they run the assigned task, they send the results to the driver.
- ✓ They also provide in-memory storage for RDDs that are cached by user programs.
- ✓ Each worker node can have one or multiple cores.
- ✓ To run the tasks in parallel, we can launch executors with multiple cores.

Below three options play a very important role in spark performance as they control the amount of CPU & memory of our spark application.

Lets understand how to configure them.

--executor-memory: Memory per executor (e.g. 1000M, 2G) (Default: 1G).

--num-executors: Number of executors to launch (Default: 2).

--executor-cores: Number of cores per executor. (Default: 1 in YARN mode).

Below conf application properties are related to Executors.

spark.executor.memory : Default (1G)

spark.executor.cores: Default 2.

Spark.executor.memoryOverhead : 10% or 384MB (Whichever is higher)

- 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%).
- When we plan the performance tuning we need to consider this as well.

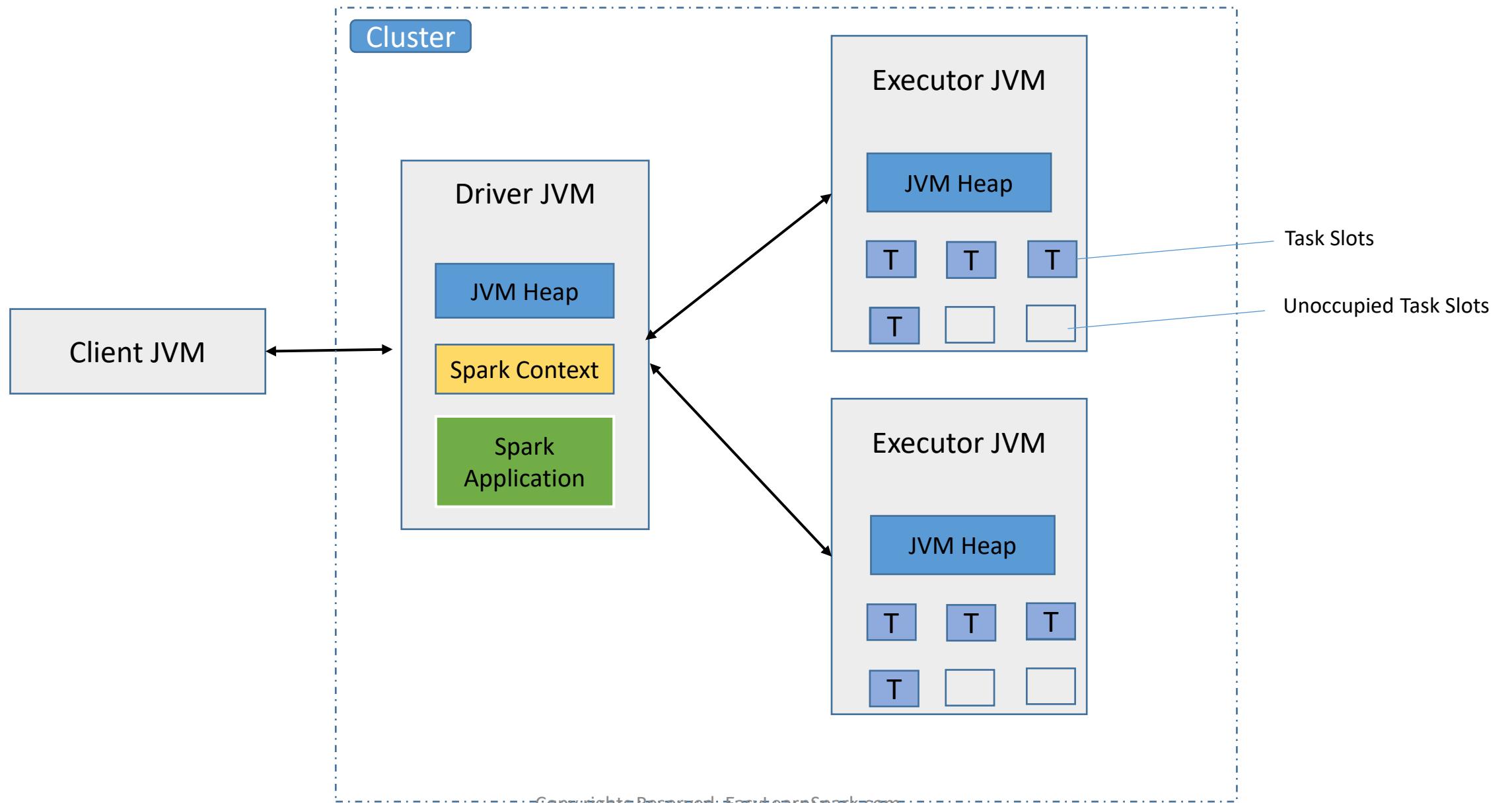
spark.executor.heartbeatInterval: Default 10s

- Interval between each executor's heartbeats to the driver. Heartbeats let the driver know that the executor is still alive.

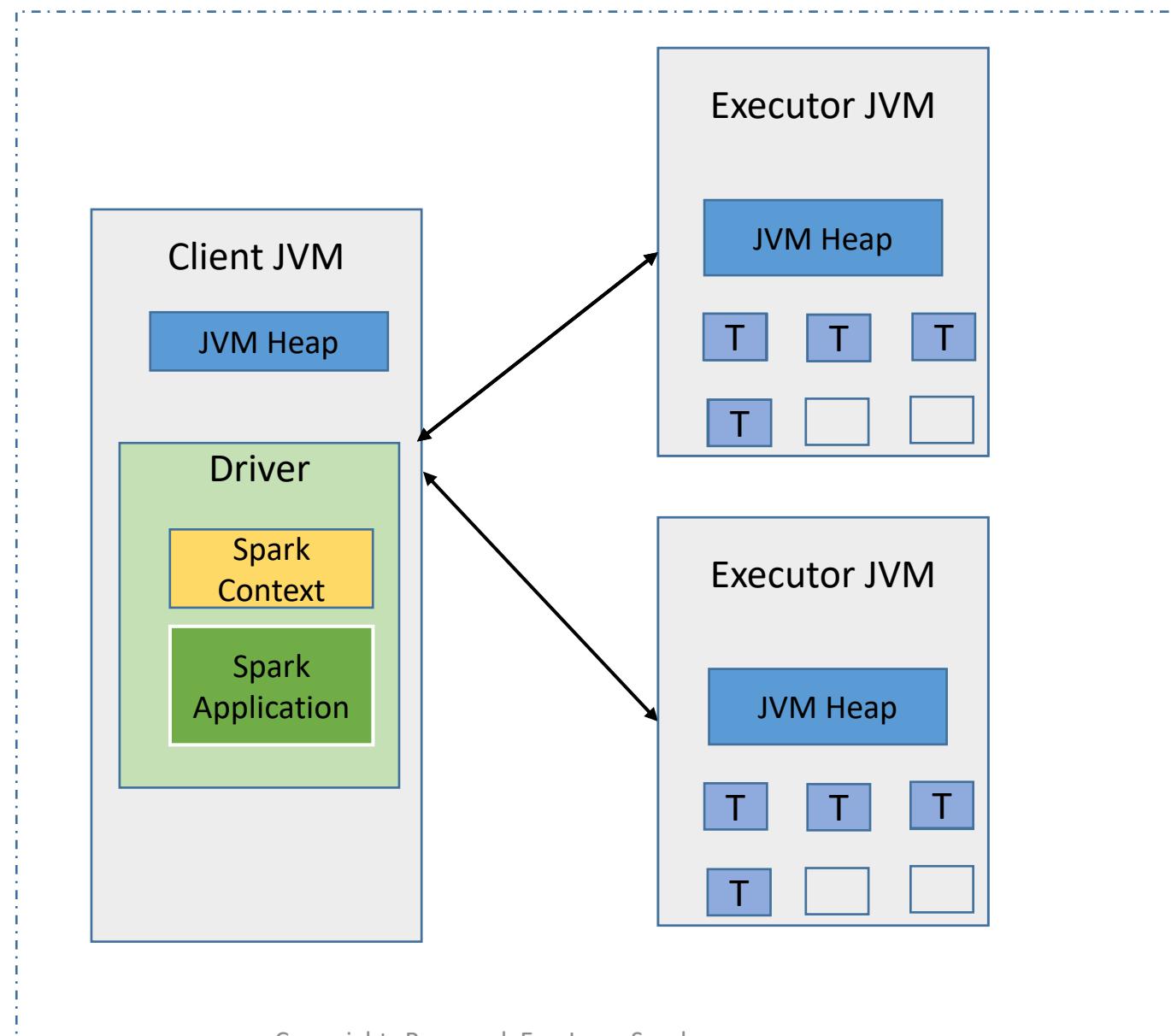
List of all Configuration properties:

<https://spark.apache.org/docs/2.4.0/configuration.html>

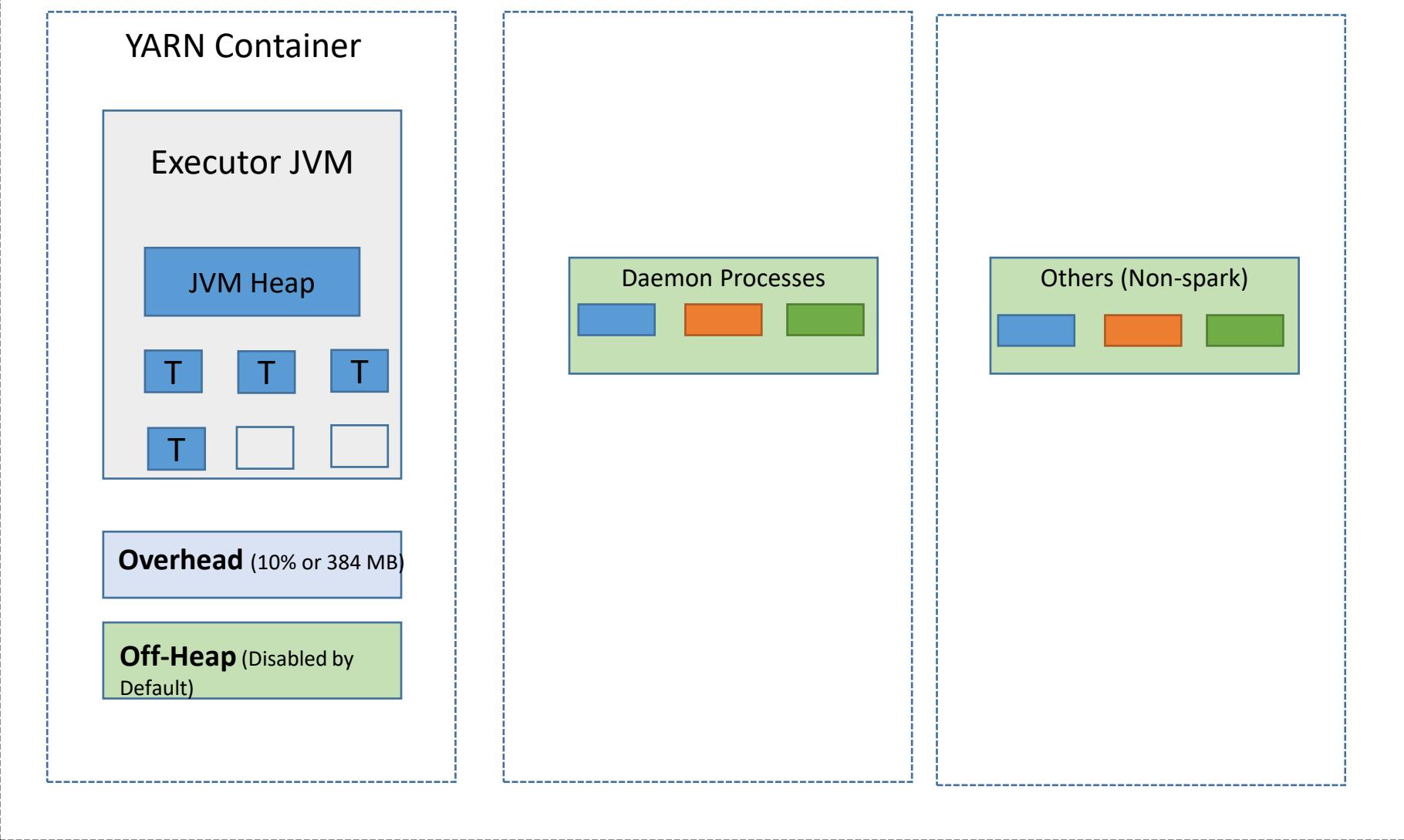
Spark Runtime Components in Cluster deploy mode:



Spark Runtime Components in Client deploy mode:



Node Manager



Lets Assume :-

Cluster Configuration:

10 Nodes

16 cores per each Node.

64GB RAM per Node

Cluster Configuration:

- 10 Nodes
- 16 cores per each Node.
- 64GB RAM per Node

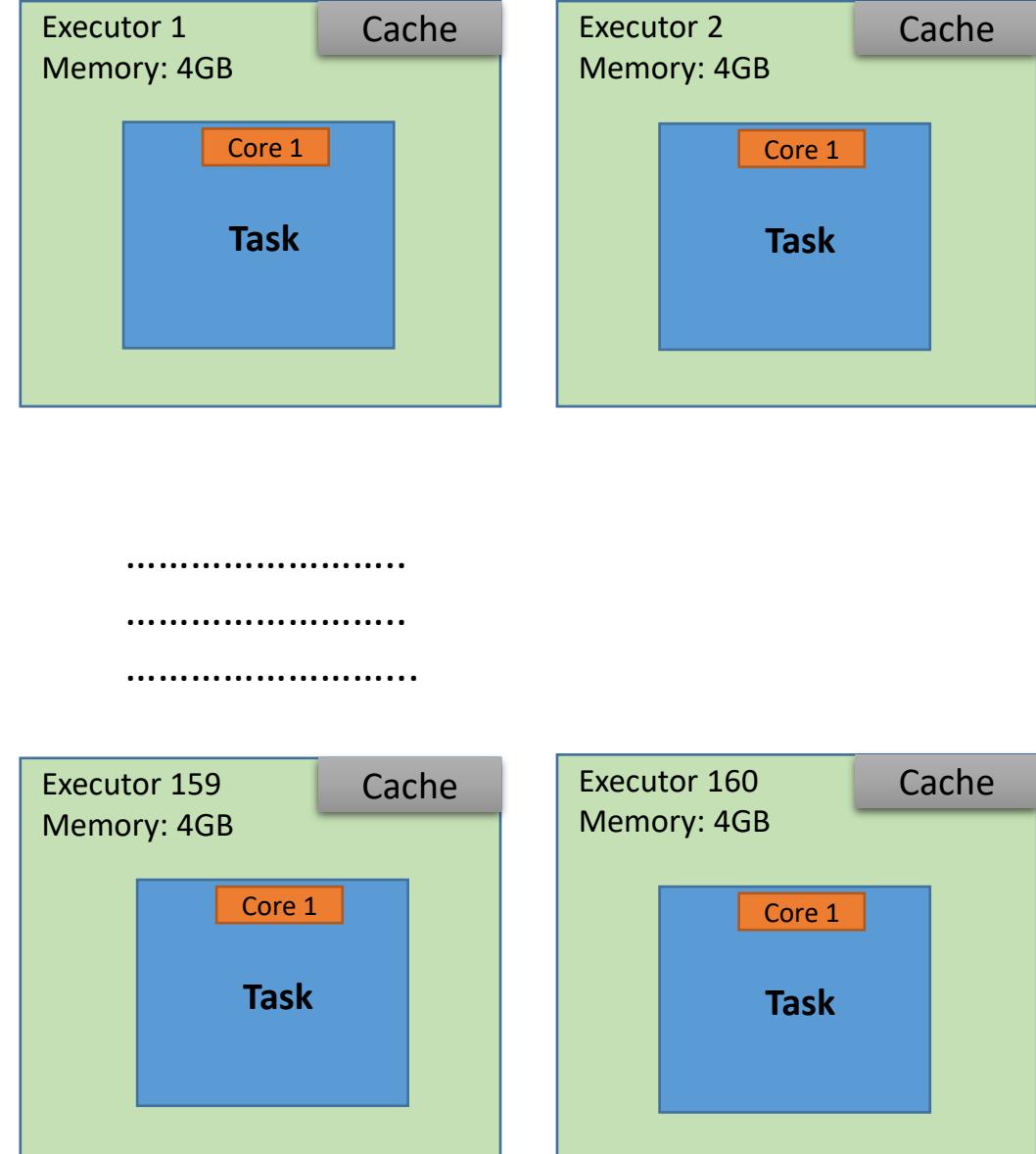
Spark Job was run with one core per executor.

- Total number of cores in cluster = $16 * 10 = 160$
- So number of executors = 160
- Number of executors per Node = $160/10 = 16$
- Memory per Executor = $64/16 = 4GB$

Problems:

- With only one executor per core, we will not be able to take advantage of running multiple tasks in the same JVM.
- There is ~10% overhead for each JVM. Due to 160 executors 160 JVM processes would be created and so lot of unnecessary overheads.
- Shared variables (Broadcast, Accumulator) will be copied 160 times.
- Not leaving enough memory for YARN Daemon processes and Application Manager.
- Not enough memory for executors.

NOT GOOD



Cluster Configuration:

10 Nodes

16 cores per each Node.

64GB RAM per Node

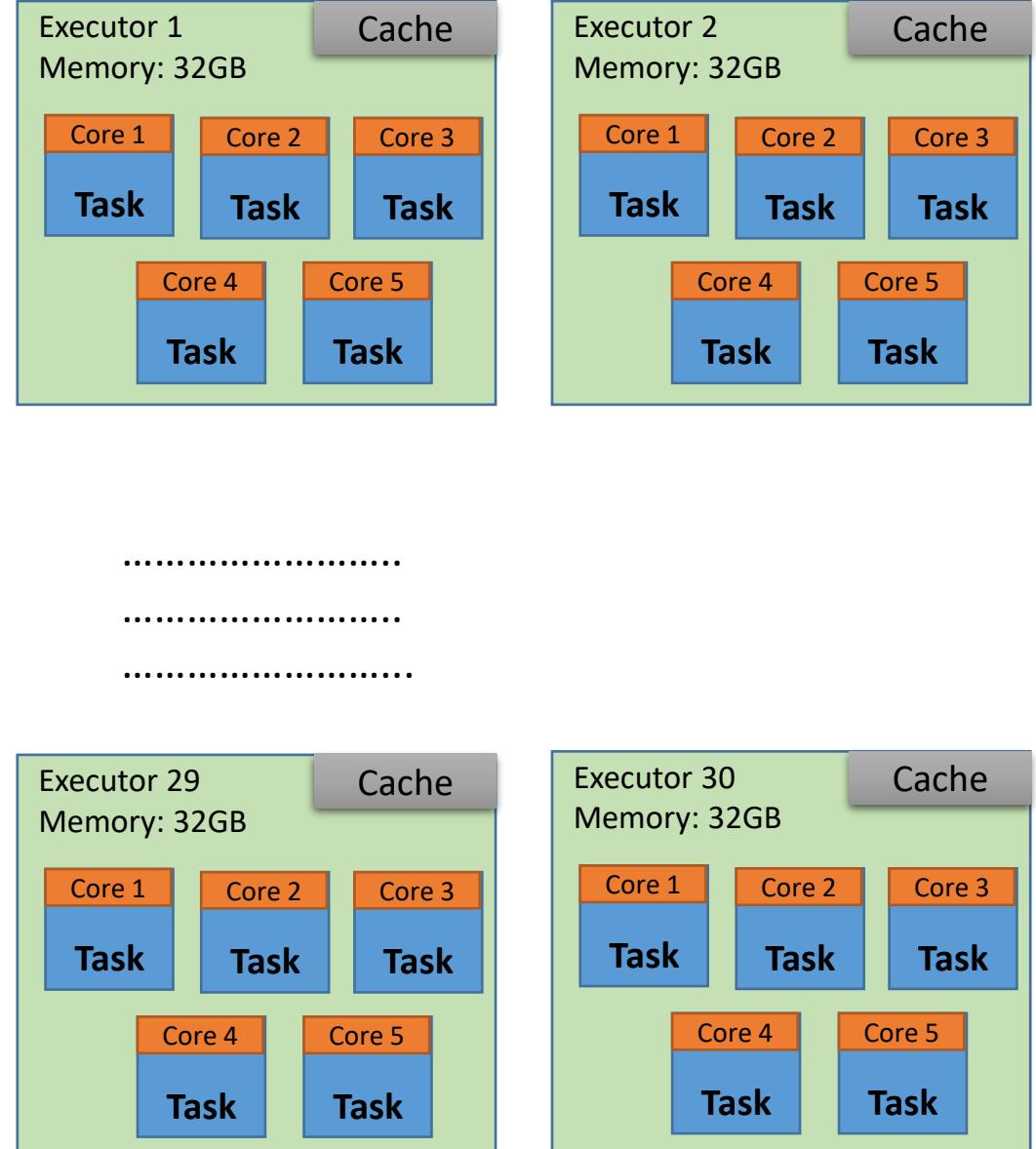
Spark Job was run with 5core per executor.(--executors-cores = 5)

- Let's Leave 1 core for YARN Daemons.
So total available cores per node for processing job = $16 - 1 = 15$
So total number of cores in cluster = $15 * 10 = 150$
- So Number of available Executors = $150 / 5 = 30$
- Let's leave 1 executor for Yarn Application Master.
So total number of available executors = $30 - 1 = 29$
- Number of Executors per each Node = $30 / 10 = 3$
- Memory per Executor: $64 \text{ GB} / 3 = 21.3 \text{ GB}$
Heap overhead around 10% (2.13GB)
So actual Memory per executor = 19.7GB

With this approach, we have achieved the parallelism and best throughputs.

**GOOD
BALANCED
RECOMMENDED**

Approach ...



Let's see how we can pass these options in spark-submit:

```
spark2-submit \
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key<=<value> \
--driver-memory 2G \
--executor-memory 64G \
--executor-cores 5 \
--num-executors 10 \
--jars <comma separated dependencies> \
--packages <package name> \
--py-files \
<application> <application args>
```

Parallelism Configurations

As the shuffle operations re-partitions the data, we can use below two configurations to control the number of partitions shuffle creates.

- spark.default.parallelism
- spark.sql.shuffle.partitions

spark.default.parallelism :

- Only applicable to RDD.
- Default value set to the number of all cores on all nodes in a cluster.
- RDD wider transformations like reduceByKey(), groupByKey(), join() triggers the data shuffling. Prior to using these operations, use the below code to set the desired partitions for shuffle operations. Change the value accordingly.

```
spark.conf.set("spark.default.parallelism",150)  
spark.conf.get("spark.default.parallelism")
```

Ex –

```
pyspark2 --master yarn --conf spark.default.parallelism=150  
rdd = sc.parallelize(range(1000))  
rdd.getNumPartitions()  
150
```

spark.sql.shuffle.partitions:

- Only applicable to DataFrames.
- Default to 200.
- For DataFrame, wider transformations like groupBy(), join() triggers the data shuffling hence the result of these transformations results in partition size same as the value set in spark.sql.shuffle.partitions.

```
spark.conf.get("spark.sql.shuffle.partitions")
```

Ex –

```
data=([('Ram',1),('Raj',2),('Ram',1),('Joann',4),('Raj',2),('Robert',5),('Reid',6),('Sam',7))
```

```
df = spark.createDataFrame(data=data,schema=('name','id'))
```

```
df.rdd.getNumPartitions()
```

```
2
```

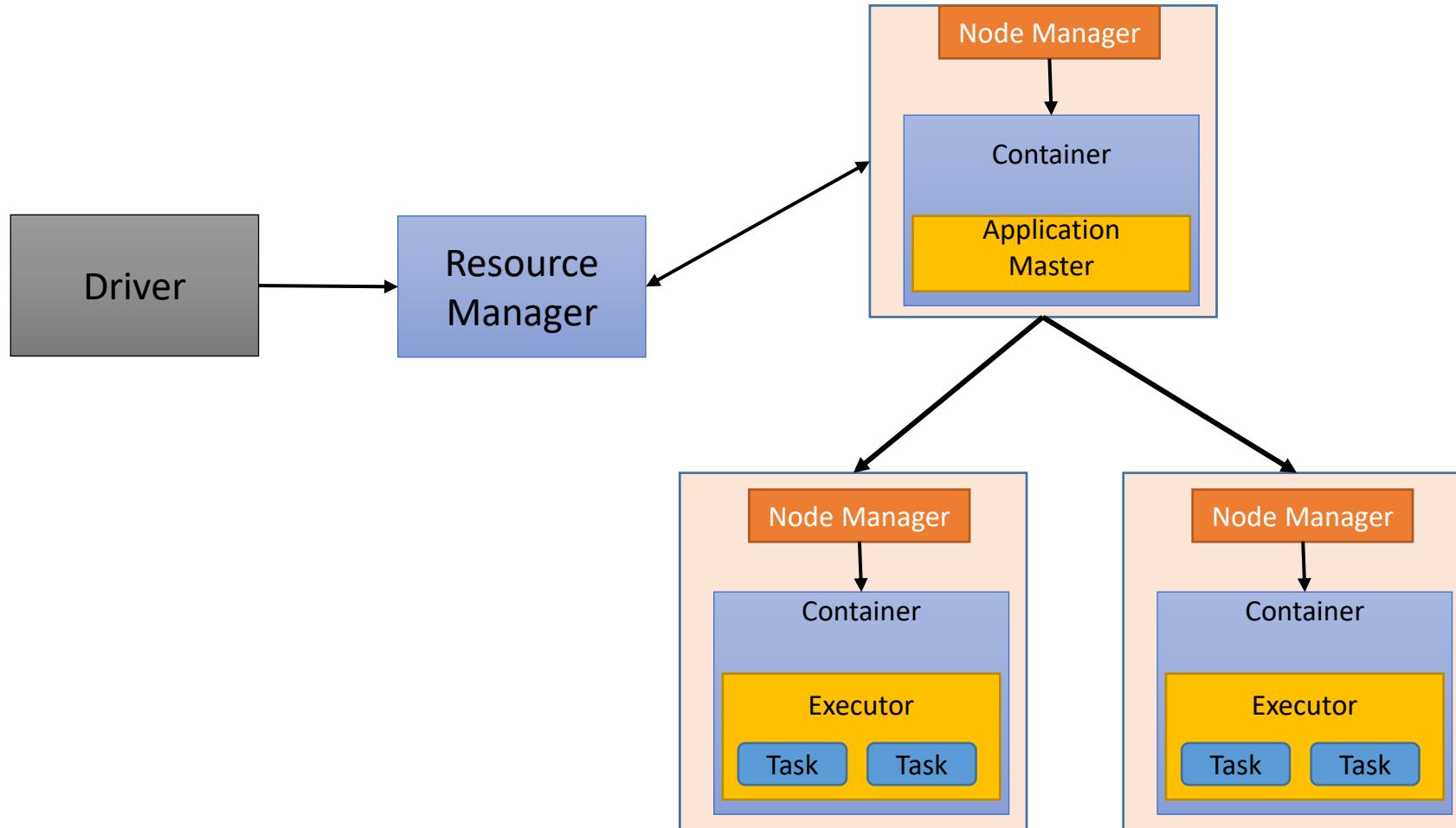
```
df1 = df.groupBy("name").count()
```

```
df1.rdd.getNumPartitions()
```

```
200
```

Memory Management

YARN Cluster Manager

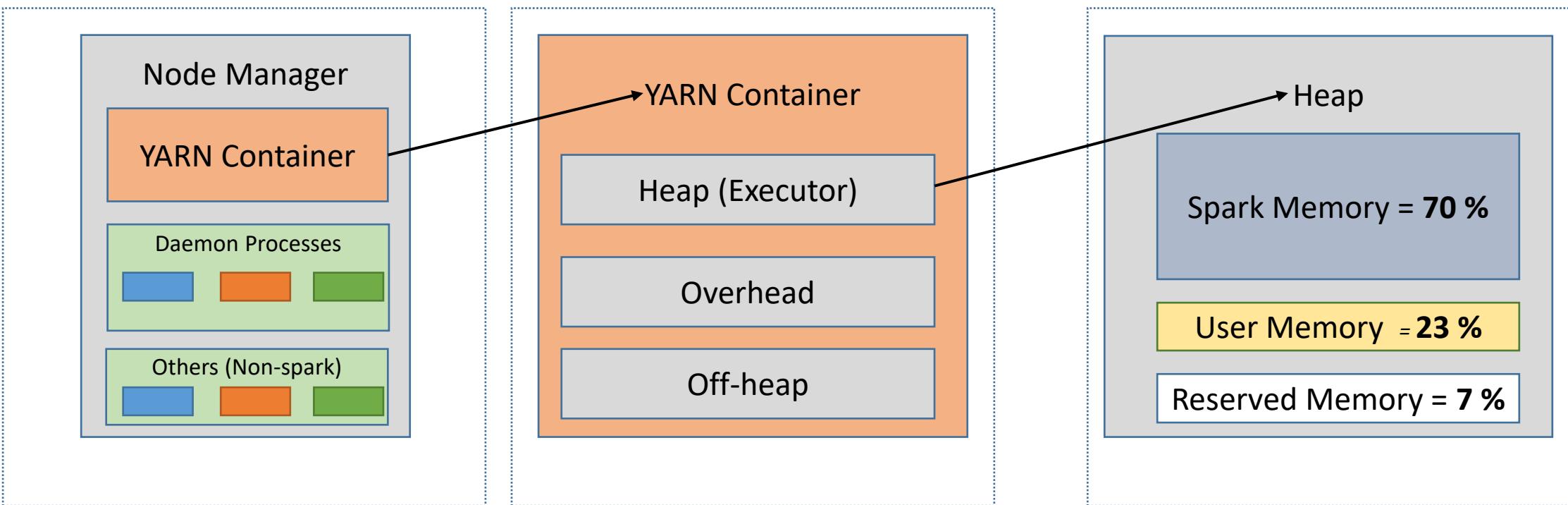


YARN as Spark Cluster Manager:

- Yet Another Resource Negotiator for Hadoop 2.x.
- Cluster Management – Used for resource allocation and Scheduling.
- It has 3 major components –
 - ✓ Resource Manager
 - ✓ Node Manager
 - ✓ Application Master

Flow:

1. Client submit the Spark application. Driver instantiates SparkContext.
2. Driver talks to the cluster manager(YARN) and negotiates resources.
3. The YARN resource manager search for a Node Manger which will, in turn, launch an ApplicationMaster for the specific job in a container.
4. The ApplicationMaster registers itself with the resource Manager.
5. The ApplicationMaster negotiates containers for executors from the ResourceManager. Can request for more resources from RM.
6. The ApplicationMaster notifies the Node Managers to launch the containers and executors. Executor then executes the tasks.
7. Driver communicates with executors to coordinate the processing of tasks of an application.
8. Once the tasks are complete, ApplicationMaster un-registers with the Resource Manager.



70%, 23% and 7% are when we allocate 4GB to Executor(Heap) and use the default Options.

Spark 1.6

yarn-site.xml

Node Manager(YARN)

Executor Container(YARN)

Heap
`Spark.executor.memory`

Execution Memory
 $Usable\ Memory * spark.memory.fraction * (1 - spark.memory.storageFraction)$

Storage Memory
 $Usable\ Memory * spark.memory.fraction * spark.memory.storageFraction$

User Memory ($Usable\ Memory * (1 - spark.memory.fraction)$)

Reserved Memory ($reserved_system_memory_bytes = 300MB$)

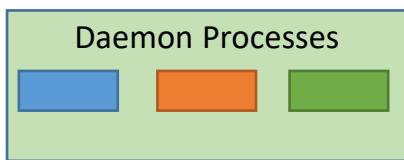
Overhead
`spark.executor.memoryOverhead`
10% or a minimum 384MB

Off-Heap
`spark.memory.offHeap.size`
It is disabled by default.

No Hard Boundary

Usable Memory

Executor Memory



- Off-heap Memory:
 - ✓ We had covered this in the RDD Persistence.
 - ✓ Off-Heap Memory is a segment of memory lies outside the JVM, but is used by JVM for certain use-cases. Off-Heap memory can be used by Spark explicitly as well to store serialized data-frames and RDDs.
 - ✓ Spark may use off-heap memory for data-intensive applications.
 - ✓ User can also persist data at Off-heap memory using persist method.
 - ✓ Off-heap storage is not managed by JVM's GC mechanism and so must be explicitly handled by the application.
 - ✓ It is disabled by default.
`spark.memory.offHeap.enabled` (Default False)
`spark.memory.offHeap.size` → Can be set after enabling it.
- Overhead Memory:
 - ✓ Default - 10% of executor memory with a minimum of 384MB.
 - ✓ Can be set by property:
`spark.executor.memoryOverhead`
 - ✓ It basically covers expenses like VM overheads, interned strings, other native overheads, etc.

- Executor Container is a JVM process. It allocates memory to 3 sections – Heap Memory, Off-Heap Memory and Overhead memory.

Heap Memory:

- ✓ All objects in heap memory are bound by GC.
- ✓ 3 Regions – Reserved Memory, User Memory and Spark Memory(Unified Execution/Storage Memory).
- ✓ Reserved Memory: Reserved to store internal objects. Hardcoded to 300MB.
- ✓ User Memory:
 - ❖ Stores all the user defined data structures, any UDFs created by the user etc.
 - ❖ Not managed by spark.
 - ❖ Formula - $\text{Usable Memory} * (1 - \text{spark.memory.fraction})$
 $\text{Ex} - 3796 * (1 - 0.75) = 950 \text{ MB}$
- ✓ Spark Memory: Managed By Spark.

Storage Memory:

- ❖ Used for storing all the cached data, shared variables.
- ❖ Any persist() operation with Memory storage level.
- ❖ Spark deletes old data and insert new data using LRU mechanism.
- ❖ Data might store in disk once removed from cache or recomputed if MEMORY_ONLY is set..

Formula - $\text{Usable Memory} * \text{spark.memory.fraction} * \text{spark.memory.storageFraction}$

$$\text{Ex} - 3796 * 0.75 * 0.50 = 1423 \text{ MB}$$

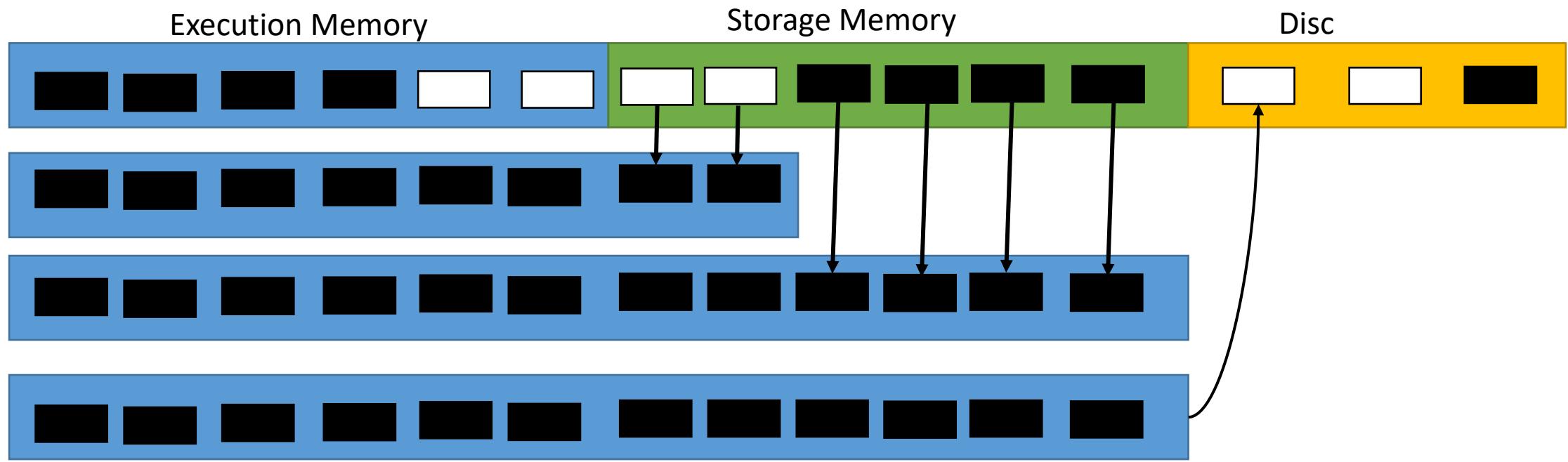
Execution Memory:

- ❖ Used by Spark for objects created during execution of a task.
- ❖ When you perform an T and A, all the intermediate results are stored here.
- ❖ Ex – it is used to store hash table for hash aggregation step.
- ❖ Supports spilling on disk if not enough memory is available.

Formula - $\text{Usable Memory} * \text{spark.memory.fraction} * (1 - \text{spark.memory.storageFraction})$

$$\text{Ex} - 3796 * 0.75 * (1-0.5) = 1423 \text{ MB}$$

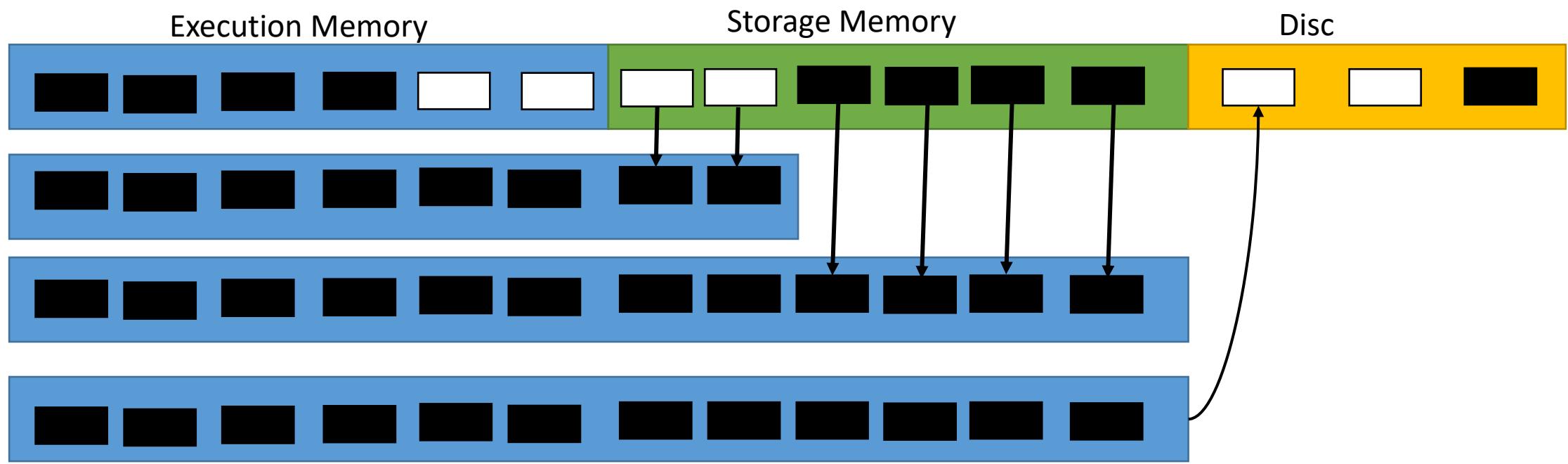
Dynamic Occupancy Mechanism :



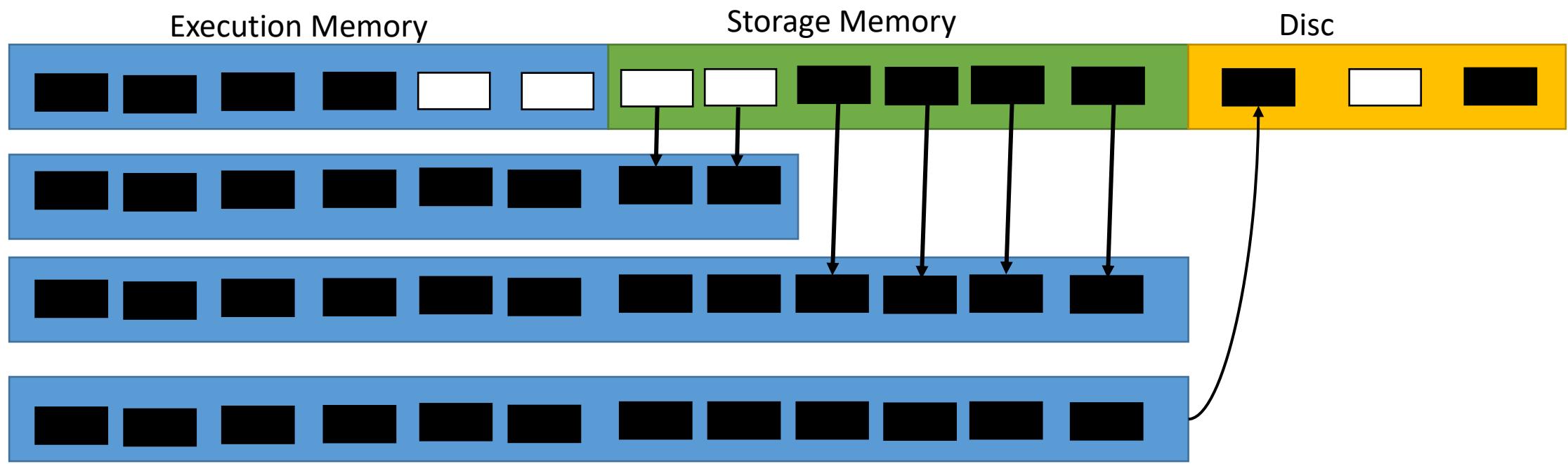
Important Notes for Execution Memory:

- Execution can not be break.
- Execution memory can borrow space from Storage memory if blocks are not used in Storage memory.
- If blocks from Execution memory is used by Storage memory and Execution needs more memory, it can forcefully evict the excess blocks occupied by Storage Memory. Blocks from storage memory will be written to disk or recomputed (of persistence level is MEMOY_ONLY).
- Write into disk if still more memory is required.

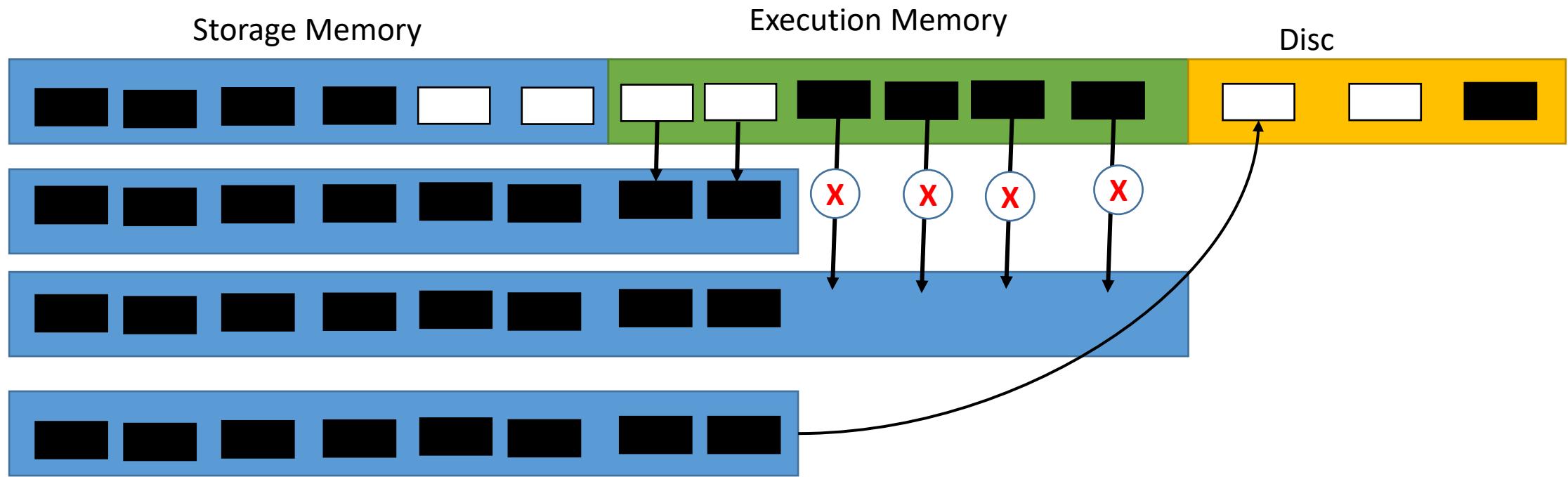
Dynamic Occupancy Mechanism :



Dynamic Occupancy Mechanism :



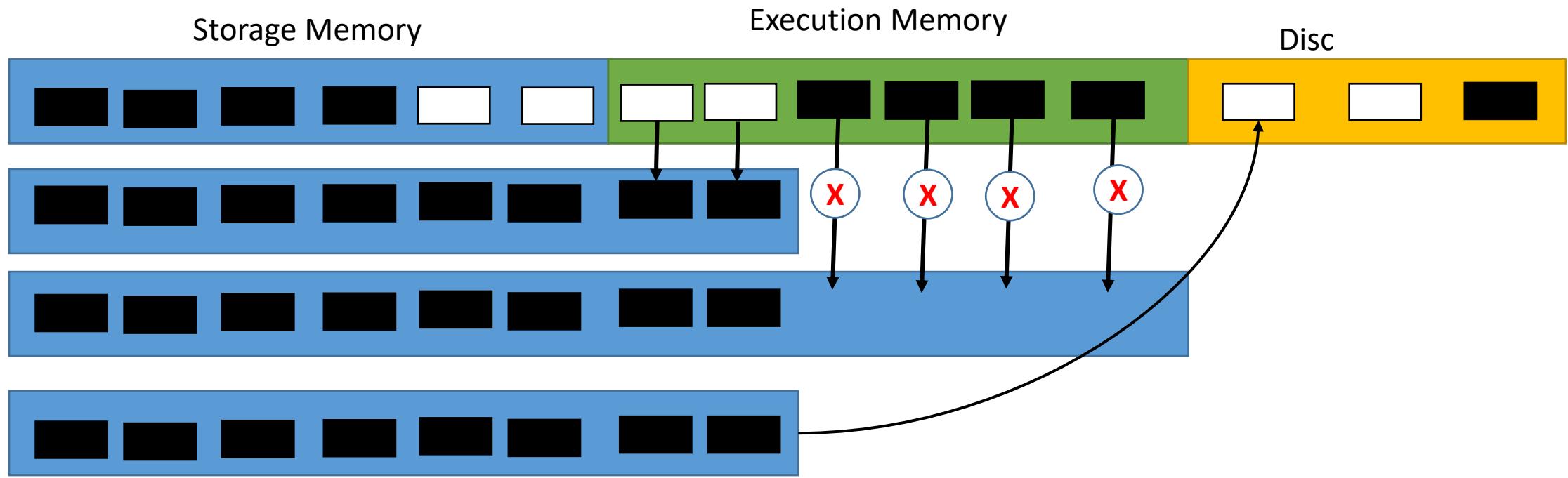
Dynamic Occupancy Mechanism :



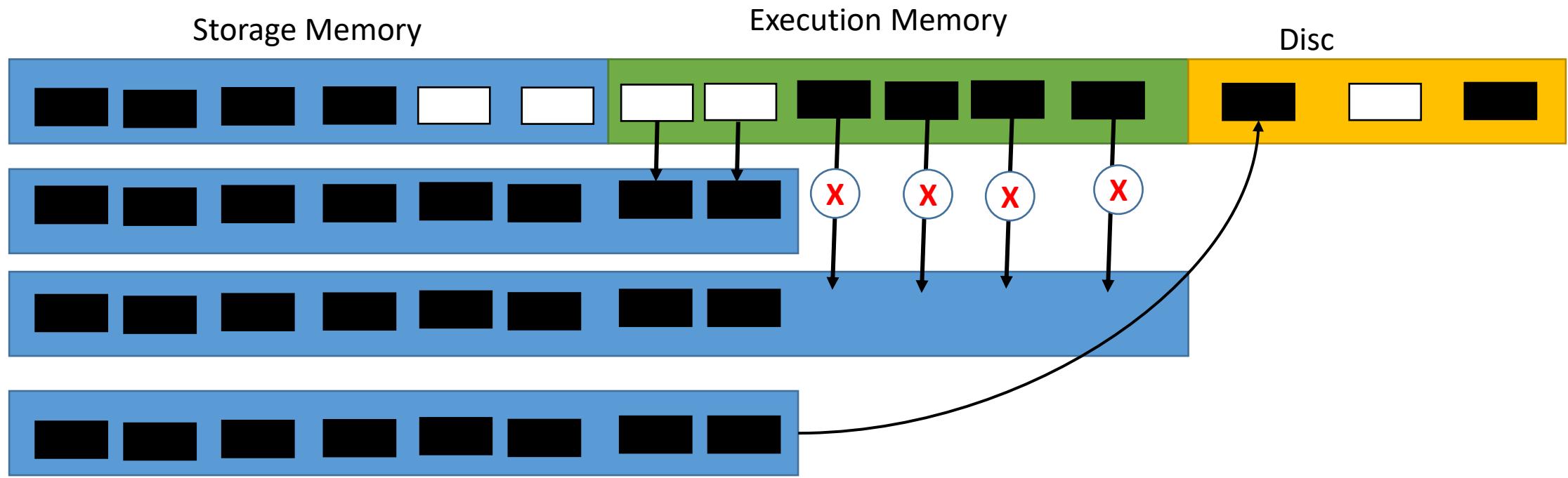
Important Notes for Storage Memory:

- *Storage memory can borrow space from execution memory only if blocks are not used in Execution memory.*
- *Storage memory can not forcefully evict the excess blocks occupied by Execution Memory. It will wait till spark releases the excess blocks stored by Execution memory and then occupies them.*

Dynamic Occupancy Mechanism :



Dynamic Occupancy Mechanism :



Ex - For understanding purpose.

Take 4GB Memory Allocation for executor and leave the default configurations.

Executor Memory(EM) = 4GB

Reserved Memory(RM) =300MB

Usable Memory = EM-RM= 4GB – 300 MB = 4096MB – 300MB = 3796 MB

`spark.memory.fraction` (Fraction of heap space used for execution and storage) = 0.75

`Spark.memory.storageFraction`(Amount of storage memory immune to eviction) = 0.50

