MONASH
University

Week 2 Tutorial

- Week 1 Review
- Accumulators
- SparkSession vs SparkContext
- Data Partitioning
- RDD vs DataFrame
- Searching in RDDs and DataFrames
- Spark SQL

GROUP
OF EIGHT
AUSTRALIA

Week 1 Review

- **VM Setup and Jupyter Notebooks**
- **RDDs**
  - How to create RDDs?
- **Transformation**
  - Map
  - FlatMap
- **Action**
  - Take
  - Collect (take vs collect)
  - Reduce
  - Count
- **Spark UI (port 4040)**

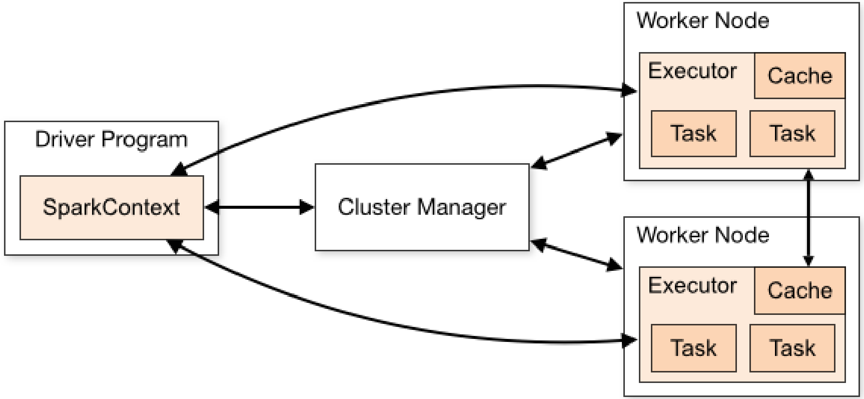
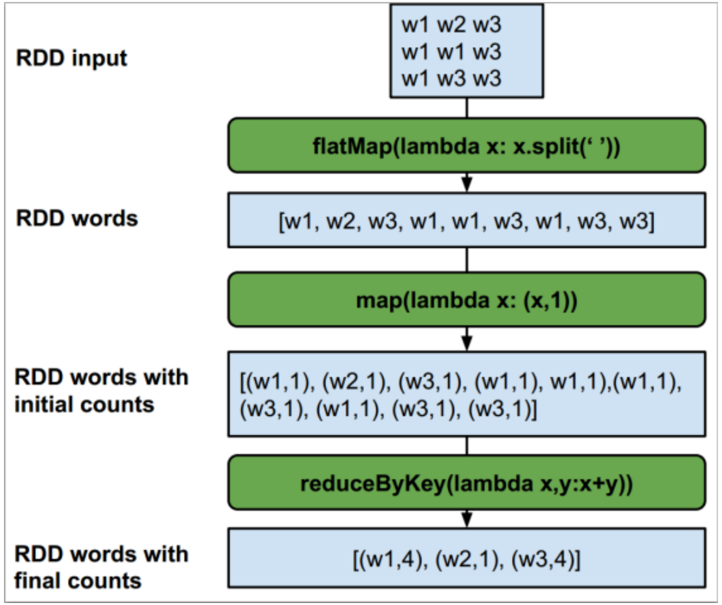


Fig : Src : [https://spark.apache.org/docs/2.3.2/running-on-mesos.html]

Word Count Example Review

```
# step 1: Read the text file twitter.txt
rdd = sc.textFile("twitter.txt")

# step 2: Use a transformation to break the lines to
individual words
words = rdd.flatMap(lambda line: line.split(" "))

# step 3: Use a transformation to convert word to a
key/value pair of (word, 1)
wordCounts = words.map(lambda word: (word, 1))

# step 4: Use a transformation to reduce the value
based on the word
finalrdd = wordCounts.reduceByKey(lambda a,b:a +b)

# step 5: Collect and display the results of the count
finalrdd.collect()
```



RDD input

w1 w2 w3
w1 w1 w3
w1 w3 w3

flatMap(lambda x: x.split(‘ ’))

RDD words

[w1, w2, w3, w1, w1, w3, w1, w3, w3]

map(lambda x: (x,1))

RDD words with initial counts

[(w1,1), (w2,1), (w3,1), (w1,1), w1,1),(w1,1), (w3,1), (w1,1), (w3,1), (w3,1)]

reduceByKey(lambda x,y:x+y))

RDD words with final counts

[(w1,4), (w2,1), (w3,4)]

Fig : [Source]

Accumulators

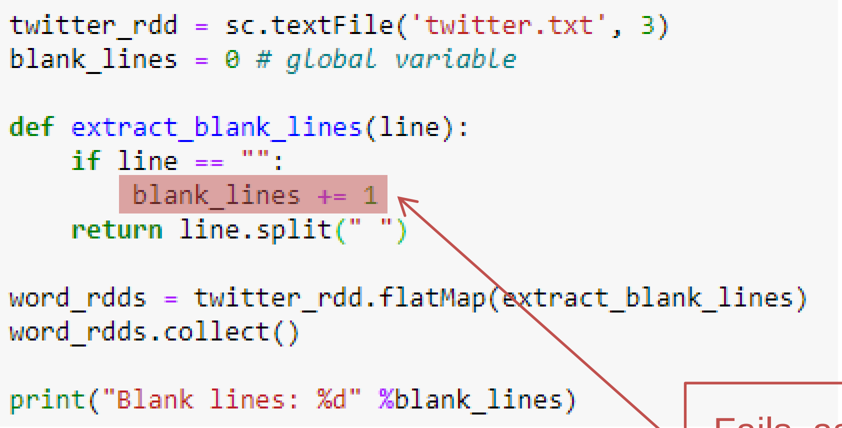
Accumulators

- Accumulators provides a simple syntax for aggregating values from worker nodes back to the driver program.
- They are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
- They can be used to implement counters (as in MapReduce) or sums.

Broadcast Variables

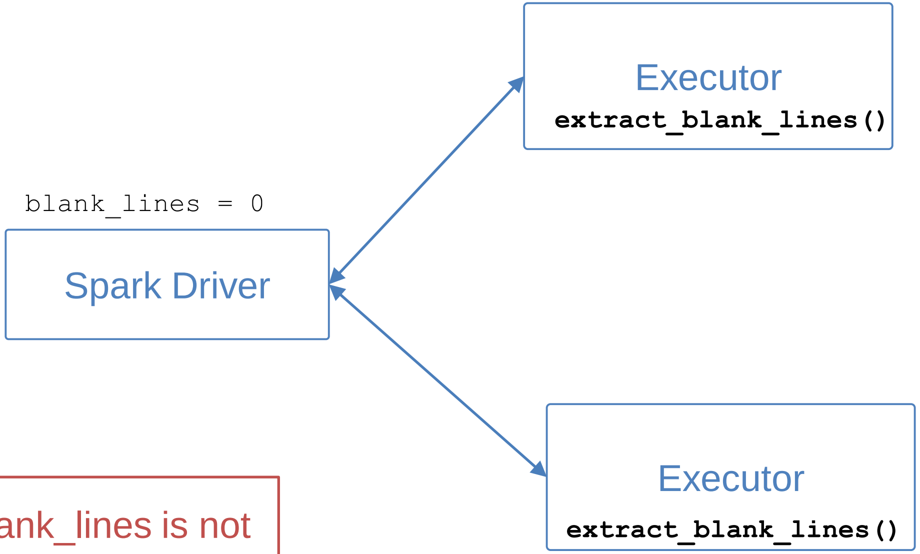
- Broadcast variables allow the program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations.
- Spark automatically sends all variables referenced in your closures to the worker nodes.

Accumulators

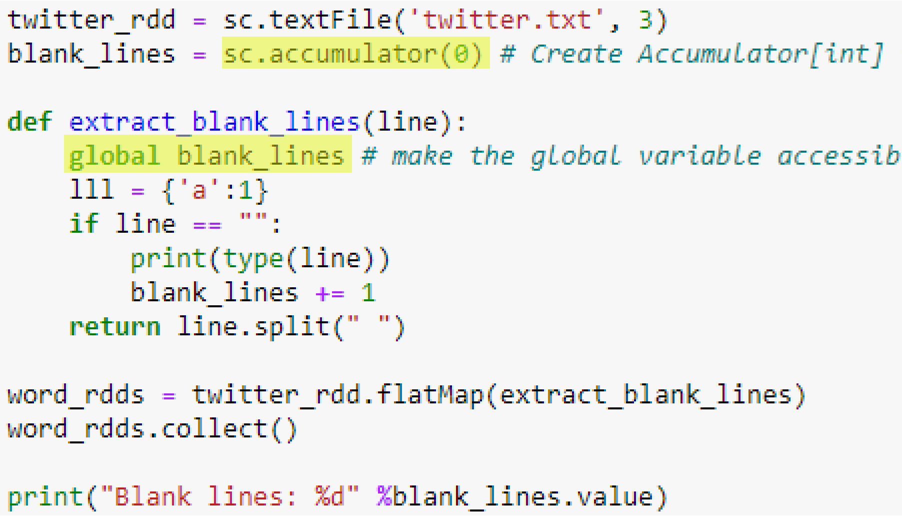


```
twitter_rdd = sc.textFile('twitter.txt', 3)
blank_lines = 0 # global variable

def extract_blank_lines(line):
    if line == "":
        blank_lines += 1
    return line.split(" ")

word_rdds = twitter_rdd.flatMap(extract_blank_lines)
word_rdds.collect()

print("Blank lines: %d" %blank_lines)
```

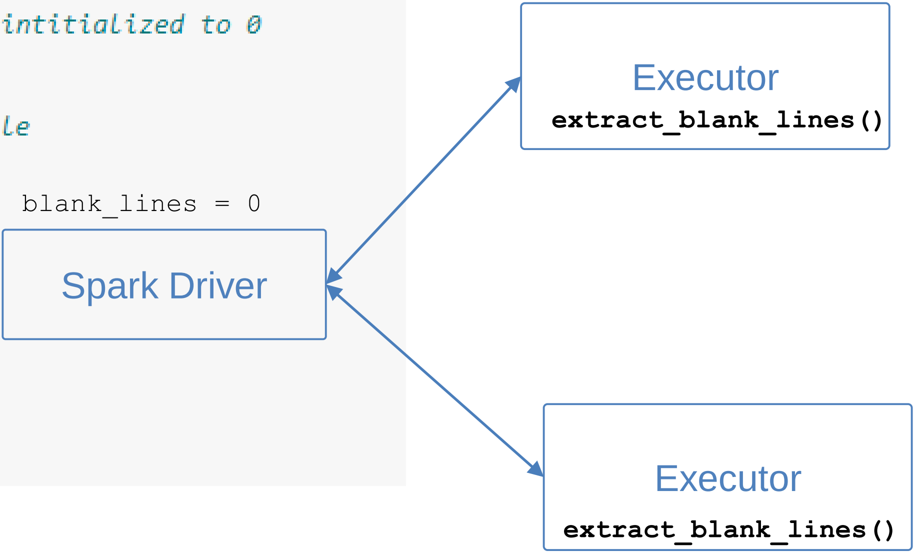
Fails, as blank_lines is not accessible in the executors



Accumulator



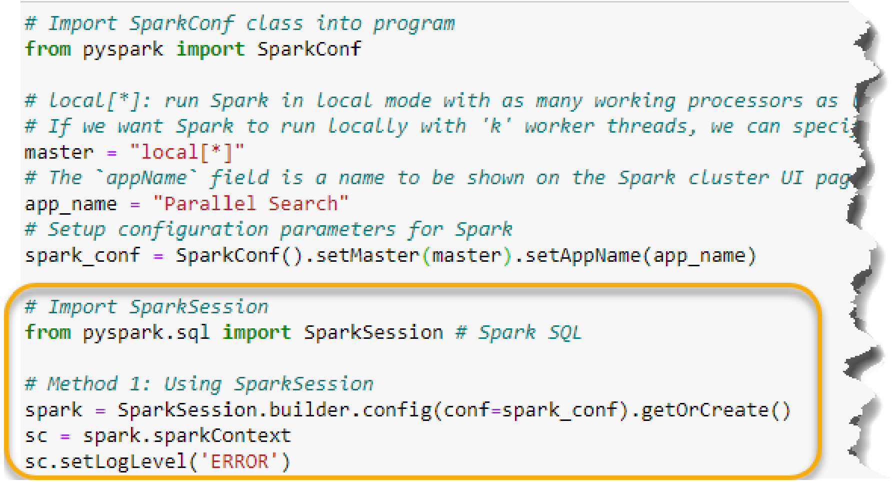
```
twitter_rdd = sc.textFile('twitter.txt', 3)
blank_lines = sc.accumulator(0) # Create Accumulator[int] intitialized to 0

def extract_blank_lines(line):
    global blank_lines # make the global variable accessible
    lll = {'a':1}
    if line == "":
        print(type(line))
        blank_lines += 1
    return line.split(" ")

word_rdds = twitter_rdd.flatMap(extract_blank_lines)
word_rdds.collect()

print("Blank lines: %d" %blank_lines.value)
```



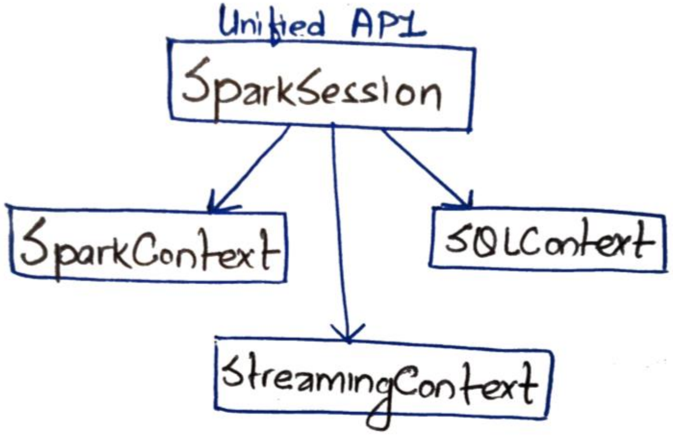
Introducing SparkSession

SparkContext vs SparkSession

- Unified entry point of Spark application from Spark 2.0



```
# Import SparkConf class into program
from pyspark import SparkConf

# local[*]: run Spark in local mode with as many working processors as
# If we want Spark to run locally with 'k' worker threads, we can speci
master = "local[*]"
# The `appName` field is a name to be shown on the Spark cluster UI pag
app_name = "Parallel Search"
# Setup configuration parameters for Spark
spark_conf = SparkConf().setMaster(master).setAppName(app_name)

# Import SparkSession
from pyspark.sql import SparkSession # Spark SQL

# Method 1: Using SparkSession
spark = SparkSession.builder.config(conf=spark_conf).getOrCreate()
sc = spark.sparkContext
sc.setLogLevel('ERROR')
```



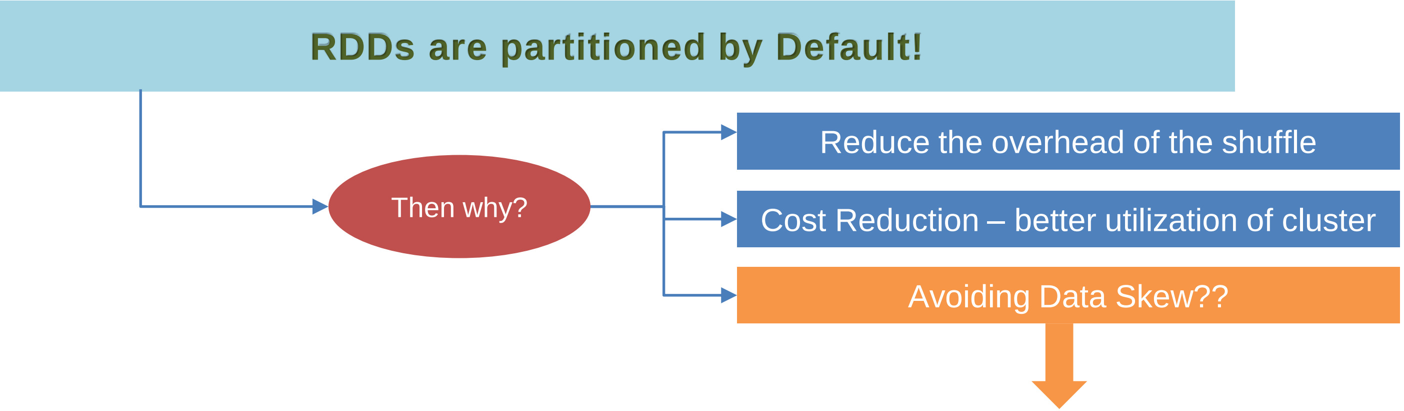
Data Partitioning

Data Partitioning Strategies :

1. **Round-robin partitioning** : distribute evenly among processors
2. **Range data partitioning** : partition based on given range
3. **Hash data partitioning** : partition based on a particular attribute using a hash function

DEMO : Partitioning in a RDD!

Data Partitioning in Spark



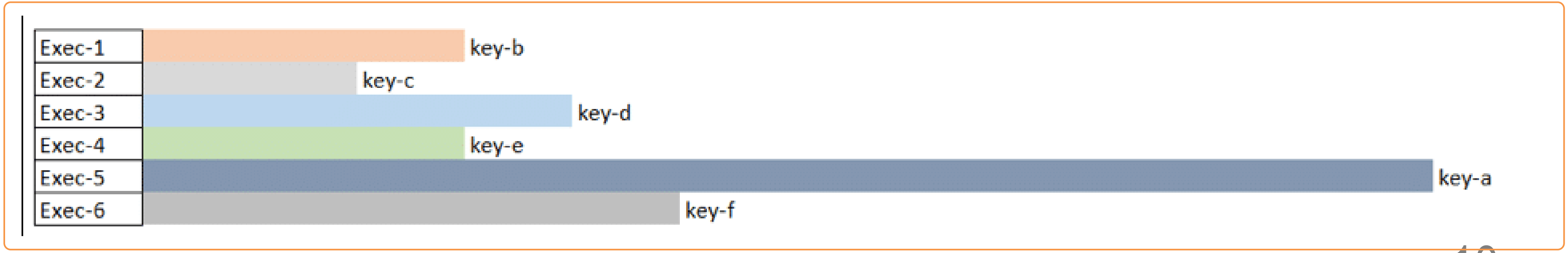
RDDs are partitioned by Default!

Then why?

Reduce the overhead of the shuffle

Cost Reduction – better utilization of cluster

Avoiding Data Skew??



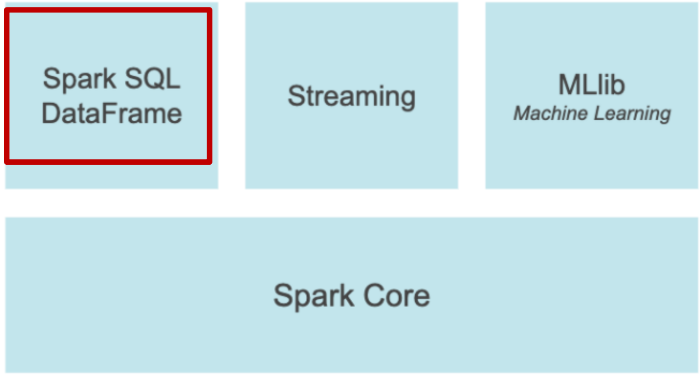
https://www.talend.com/blog/2018/03/05/intro-apache-spark-partitioning-need-know/

Parallel Search in RDD

- Searching in RDDs using Multiple Conditions
- Finding max/min values of an attribute in RDDs

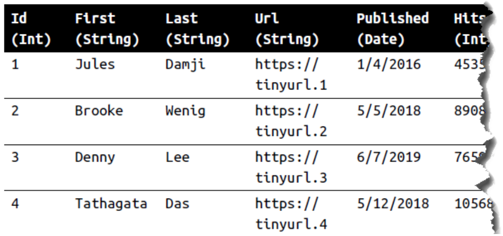
Spark SQL

- ❑ A Spark module for **structured data processing.**
- ❑ It provides a programming abstraction called DataFrame and act as distributed SQL query engine.
- ❑ Unlike the basic Spark RDD API, Spark SQL provide Spark with more information about the structure of both the data and the computation being performed (used for optimization).



Two uses of Spark SQL:

- ❑ **DataFrames APIs:**
  - Dataframe is distributed collection of data organized into *named columns*
- ❑ **To execute SQL queries**



Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int
1	Jules	Damji	https:// tinyurl.1	1/4/2016	4535
2	Brooke	Wenig	https:// tinyurl.2	5/5/2018	8908
3	Denny	Lee	https:// tinyurl.3	6/7/2019	765
4	Tathagata	Das	https:// tinyurl.4	5/12/2018	10568

Spark Core

- ▪ underlying execution engine that all other functionalities build on it
- ▪ Working with an RDD

https://spark.apache.org/docs/latest/sql-programming-guide.html

https://spark.apache.org/docs/latest/api/python/index.html

Partitioning with DataFrames

repartition()

repartitionByRange()

Round-robin partitioning :

```
df_round = df.repartition(5)
```

Range data partitioning :

```
df_range = df.repartitionByRange(5,"balance")
```

Hash data partitioning :

```
column_hash = "education"
df_hash = df.repartition(column_hash)
```

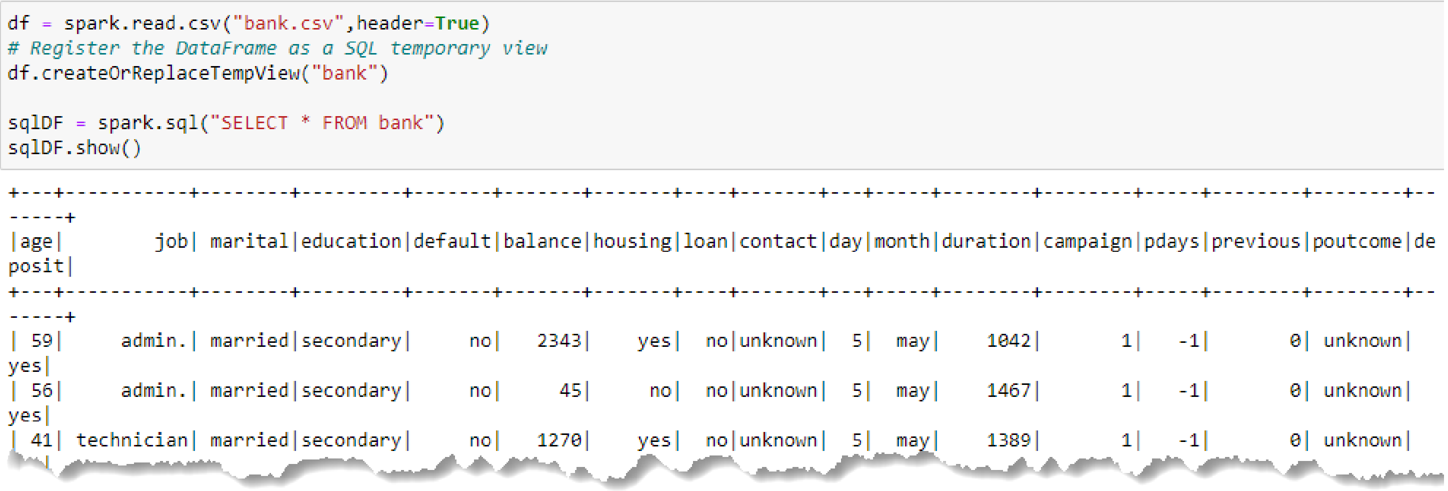
Searching in Dataframe

- Filter() - Perform search
- Where() - Alias to Filter()
- Select() - Select particular columns to show
- Show() - Select rows to show

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#

Spark SQL

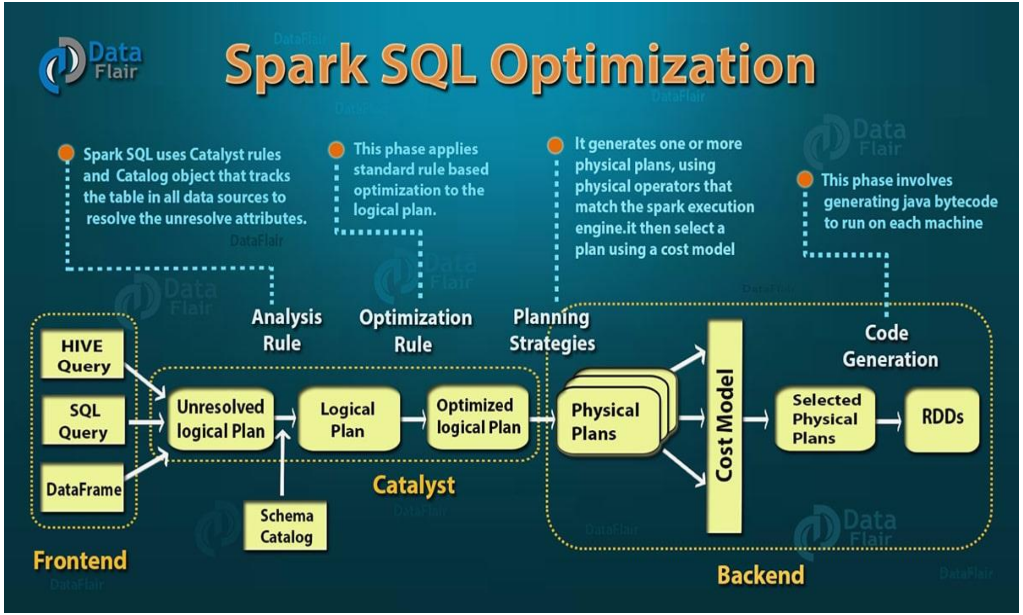
- To execute SQL queries.
- For further reading link
- Temporary views in Spark SQL



```
df = spark.read.csv("bank.csv",header=True)
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("bank")

sqlDF = spark.sql("SELECT * FROM bank")
sqlDF.show()
```

```
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+--
-----+
|age|        job| marital|education|default|balance|housing|loan|contact|day|month|duration|campaign|pdays|previous|poutcome|de
posit|
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+--
-----+
| 59|     admin.| married|secondary|     no|   2343|    yes|  no|unknown|  5|  may|    1042|       1|   -1|       0| unknown|
yes|
| 56|     admin.| married|secondary|     no|     45|     no|  no|unknown|  5|  may|    1467|       1|   -1|       0| unknown|
yes|
| 41| technician| married|secondary|     no|   1270|    yes|  no|unknown|  5|  may|    1389|       1|   -1|       0| unknown|
```

Logical plan & Physical plan


Data Flair
Spark SQL Optimization
Spark SQL uses Catalyst rules and Catalog object that tracks the table in all data sources to resolve the unresolve attributes.
This phase applies standard rule based optimization to the logical plan.
It generates one or more physical plans, using physical operators that match the spark execution engine.it then select a plan using a cost model
This phase involves generating java bytecode to run on each machine
Analysis Rule
Optimization Rule
Planning Strategies
Code Generation
HIVE Query
SQL Query
DataFrame
Unresolved logical Plan
Logical Plan
Optimized logical Plan
Physical Plans
Cost Model
Selected Physical Plans
RDDs
Schema Catalog
Catalyst
Frontend
Backend

https://data-flair.training/blogs/spark-sql-optimization/

- ❑ Execution plan - set of operations executed to translate a query language statement (SQL, Spark SQL, Dataframe operations etc.) to a set of optimized logical and physical operations.

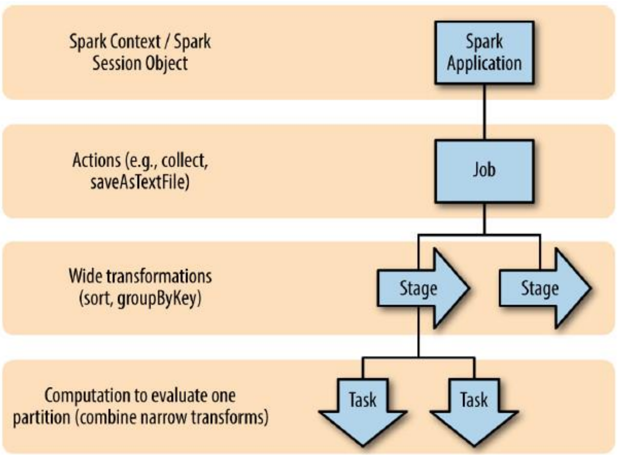
- ❑ Logical plan - Abstract of all data transformation steps (specified by the user) that need to be executed. No details about the Driver(Master Node) or Executor (Worker Node)

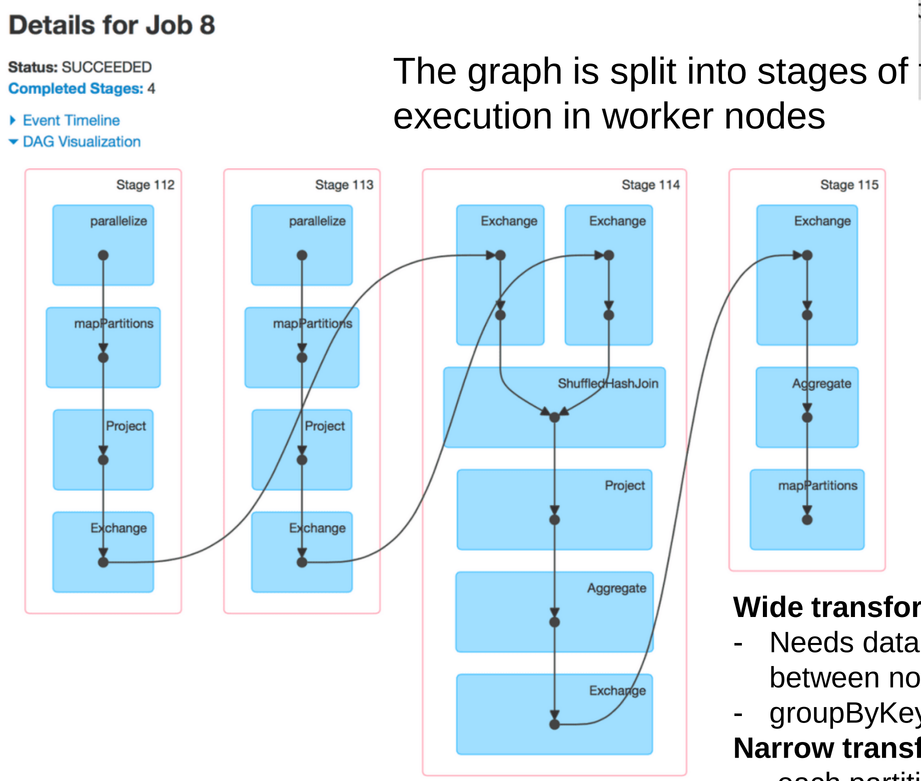
- ❑ Physical plan - Contains more specific description of how execution should happen (e.g., specific RDD to create, specific choice of algorithms for join or agg., how data partitioned/shuffled)

- ❑ Once the finest Physical Plan is selected, executable code (DAG of RDDs) for the query is created

Directed Acyclic Graph (DAG)

DAG is a graph denoting a sequence of operations (transformations & actions) that are performed on the targeted RDDs


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task



Details for Job 8

Status: SUCCEEDED

Completed Stages: 4

▸ Event Timeline

▾ DAG Visualization

The graph is split into stages of tasks for execution in worker nodes

Wide transformation

- Needs data shuffling between nodes
- groupByKey(),join()…

Narrow transformation

- each partition of output RDD depends only on a **single partition** of input RDD.
- map(), filter()…

https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html
https://www.linkedin.com/pulse/demystifying-spark-jobs-stages-data-shuffling-shahzad-aslam/

Thank You!

See you next week.