Databricks Apache Spark Certified Developer Master Cheat Sheet

https://databricks.com/training/certified-spark-developer

Index

1. GENERAL IMP LINKS

Free online clusters for quick start Spark exercises!

- databricks - free 6GB cluster with preinstall spark and relavent dependencies for notebooks
- zepl - limited resource spark non distributed notebooks
- colab - from google
- Kaggle Kernals (Kaggle kernal > Internet On ; ! Pip install pyspark)

spark on Google colab

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://apache.osuosl.org/spark/spark-2.3.1/spark-2.3.1-bin-hadoop2.7.tgz
# latest spark binary

!tar xf spark-2.3.1-bin-hadoop2.7.tgz
!pip install -q findspark

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.3.1-bin-hadoop2.7"

import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
```

spark on Kaggle Kernals

```
!pip install pyspark

from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
spark
```

References:

https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/
https://www.slideshare.net/cloudera/top-5-mistakes-to-avoid-when-writing-apache-spark-applications
https://pages.databricks.com/rs/094-YMS-629/images/7-steps-for-a-developer-to-learn-apache-spark.pdf
https://docs.databricks.com/spark/latest/gentle-introduction/index.html
http://www.bigdatatrunk.com/developer-certification-for-apache-spark-databricks/

2. POINTS TO CONSIDER

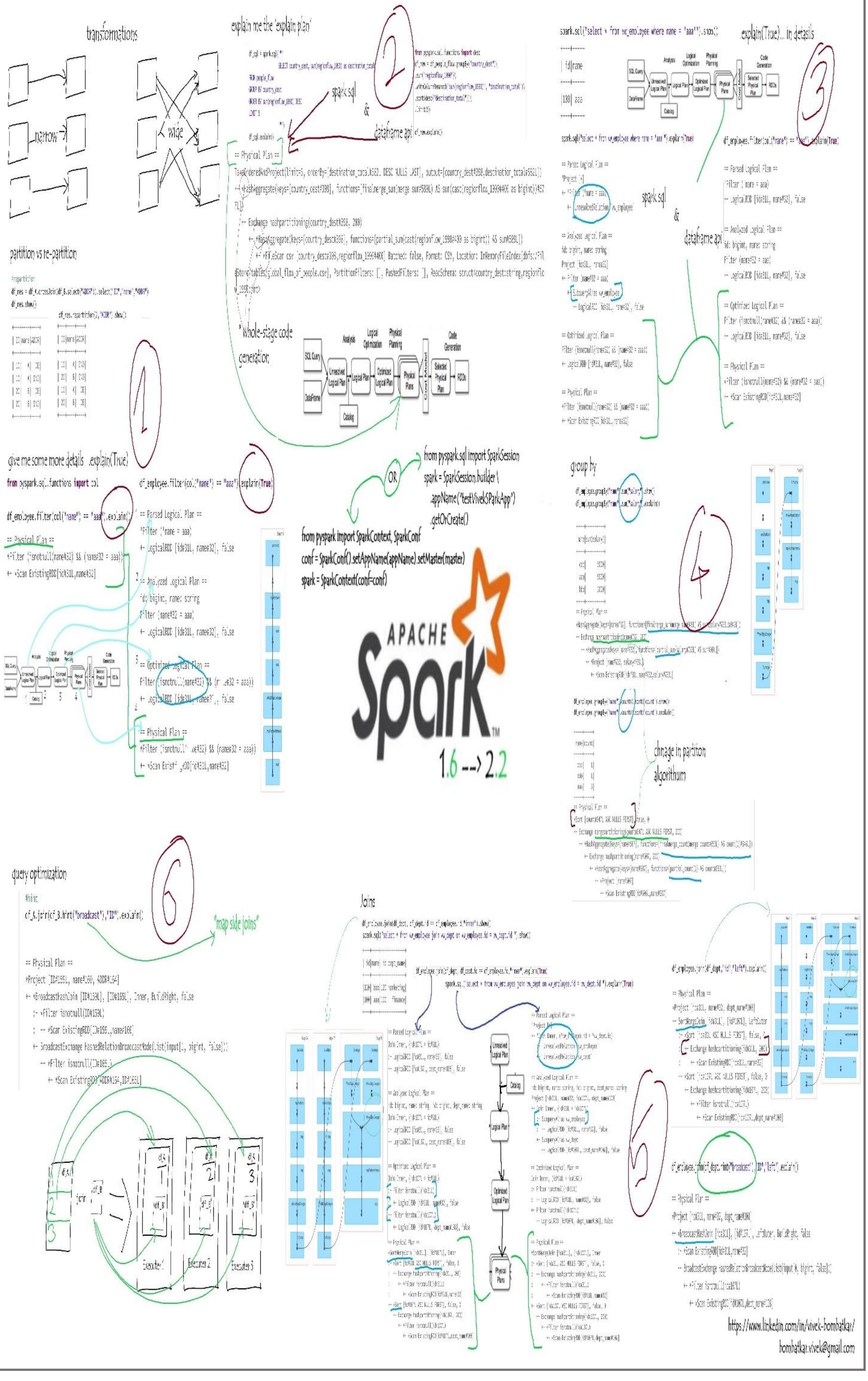
- 40 questions, 90 minutes
- 70% programming Scala, Python and Java, 30% are theory.
- Orielly learning spark : Chapter's 3,4 and 6 for 50% ; Chapters 8,9(IMP) and 10 for 30%
- Programming Languages (Certifications will be offered in Scala or Python)
- Some experience developing Spark apps in production already
- Developers must be able to recognize the code that is more parallel, and less memory constrained. They must know how to apply the best practices to avoid run time issues and performance bottlenecks.

3. COURSE TOPICS

a. Spark Concept

http://spark.apache.org/
https://databricks.gitbooks.io/databricks-spark-reference-applications/content/index.html
https://thachtranerc.wordpress.com/2017/07/10/databricks-developer-certifcation-for-apache-spark-finally-i-made-it/
videos :
https://www.youtube.com/watch?v=7ooZ4S7Ay6Y
https://www.youtube.com/watch?v=tFRPeU5HemU

- a.1 Spark code breakdown to optimizer



transformations

partition vs re-partition

```
#repartition
df_res = df_A.crossJoin(df_B.select("ADDR")).select("ID","name","ADDR")
df_res.show()
```

```
df_res.repartition(2,"ADDR").show()
```

explain me the 'explain plan'

```
df_sql = spark.sql("""
            SELECT country_dest, sum(regionflow_1998) as destination_total
FROM people_flow
GROUP BY country_dest
ORDER BY sum(regionflow_1998) DESC
LIMIT 5
            """)
df_sql.explain()
```

```
from pyspark.sql.functions import desc
df_new = df_people_flow.groupBy("country_dest")\
.sum("regionflow_1998")\
.withColumnRenamed("sum(regionflow_1998)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)

df_new.explain()
```

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#582L DESC NULLS LAST], output=[country_dest#398,destination_total#582L])
+- *HashAggregate(keys=[country_dest#398], functions=[finalmerge_sum(merge sum#589L) AS sum(cast(regionflow_1998#400 as bigint))#57
7L])
   +- Exchange hashpartitioning(country_dest#398, 200)
      +- *HashAggregate(keys=[country_dest#398], functions=[partial_sum(cast(regionflow_1998#400 as bigint)) AS sum#589L])
         +- *FileScan csv [country_dest#398,regionflow_1998#400] Batched: false, Format: CSV, Location: InMemoryFileIndex[dbfs:/Fil
eStore/tables/global_flow_of_people.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<country_dest:string,regionflo
w_1998:int>
```

* whole-stage code generation

```
spark.sql("select * from vw_employee where name = 'aaa'").show()
+---+----
| id|name
+---+----
|130| aaa
+---+----
```

.explain(True)... in detasils

```
spark.sql("select * from vw_employee where name = 'aaa'").explain(True)

== Parsed Logical Plan ==
'Project [*]
+- 'Filter ('name = aaa)
   +- 'UnresolvedRelation `vw_employee`

== Analyzed Logical Plan ==
id: bigint, name: string
Project [id#31L, name#32]
+- Filter (name#32 = aaa)
   +- SubqueryAlias vw_employee
      +- LogicalRDD [id#31L, name#32], false

== Optimized Logical Plan ==
Filter (isnotnull(name#32) && (name#32 = aaa))
+- LogicalRDD [id#31L, name#32], false

== Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L,name#32]
```

```
df_employee.filter(col("name") == "aaa").explain(True)

== Parsed Logical Plan ==
'Filter ('name = aaa)
+- LogicalRDD [id#31L, name#32], false

== Analyzed Logical Plan ==
id: bigint, name: string
Filter (name#32 = aaa)
+- LogicalRDD [id#31L, name#32], false

== Optimized Logical Plan ==
Filter (isnotnull(name#32) && (name#32 = aaa))
+- LogicalRDD [id#31L, name#32], false

== Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L,name#32]
```

give me some more details .explain(True)

```
from pyspark.sql.functions import col

df_employee.filter(col("name") == "aaa").explain()

== Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L,name#32]
```

```
df_employee.filter(col("name") == "aaa").explain(True)

1 == Parsed Logical Plan ==
'Filter ('name = aaa)
+- LogicalRDD [id#31L, name#32], false

2 == Analyzed Logical Plan ==
id: bigint, name: string
Filter (name#32 = aaa)
+- LogicalRDD [id#31L, name#32], false

3 == Optimized Logical Plan ==
Filter (isnotnull(name#32) && (name#32 = aaa))
+- LogicalRDD [id#31L, name#32], false

4 == Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L,name#32]
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("testVivekSParkApp")
    .getOrCreate()
```

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName(appName).setMaster(master)
spark = SparkContext(conf=conf)
```

group by

```
df_employee.groupBy("name").sum("salary").show()
df_employee.groupBy("name").sum("salary").explain()

+----+-----------+
|name|sum(salary)|
+----+-----------+
| ccc|       5000|
| aaa|       5000|
| bbb|       2000|
+----+-----------+

== Physical Plan ==
*HashAggregate(keys=[name#732], functions=[finalmerge_sum(merge sum#845L) AS sum(salary#733L)#840L])
+- Exchange hashpartitioning(name#732, 200)
   +- *HashAggregate(keys=[name#732], functions=[partial_sum(salary#733L) AS sum#845L])
      +- *Project [name#732, salary#733L]
         +- *Scan ExistingRDD[id#731L,name#732,salary#733L]
```

```
df_employee.groupBy("name").count().sort("count").show()
df_employee.groupBy("name").count().sort("count").explain()

+----+-----+
|name|count|
+----+-----+
| ccc|    1|
| bbb|    1|
| aaa|    3|
+----+-----+

== Physical Plan ==
*Sort [count#647L ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#647L ASC NULLS FIRST, 200)
   +- *HashAggregate(keys=[name#597], functions=[finalmerge_count(merge count#651L) AS count(1)#646L])
      +- Exchange hashpartitioning(name#597, 200)
         +- *HashAggregate(keys=[name#597], functions=[partial_count(1) AS count#651L])
            +- *Project [name#597]
               +- *Scan ExistingRDD[id#596L,name#597]
```

chnage in partiton algorithum

query optimization

```
#hint
df_A.join(df_B.hint("broadcast"),"ID").explain()

== Physical Plan ==
*Project [ID#159L, name#160, ADDR#164]
+- *BroadcastHashJoin [ID#159L], [ID#165L], Inner, BuildRight, false
   :- *Filter isnotnull(ID#159L)
   :  +- *Scan ExistingRDD[ID#159L,name#160]
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[1, bigint, false]))
      +- *Filter isnotnull(ID#165L)
         +- *Scan ExistingRDD[ADDR#164,ID#165L]
```

"map side joins"

Joins

```
df_employee.join(df_dept, df_dept.id == df_employee.id,"inner").show()
spark.sql("select * from vw_employee join vw_dept on vw_employee.id = vw_dept.id ").show()

+---+----+---+----------+
| id|name| id| dept_name|
+---+----+---+----------+
|120| bbb|120| marketing|
|100| aaa|100|   finance|
+---+----+---+----------+
```

```
df_employee.join(df_dept, df_dept.id == df_employee.id,"inner").explain(True)

== Parsed Logical Plan ==
Join Inner, (id#107L = id#31L)
:- LogicalRDD [id#31L, name#32], false
+- LogicalRDD [id#107L, dept_name#108], false

== Analyzed Logical Plan ==
id: bigint, name: string, id: bigint, dept_name: string
Join Inner, (id#107L = id#31L)
:- LogicalRDD [id#31L, name#32], false
+- LogicalRDD [id#107L, dept_name#108], false

== Optimized Logical Plan ==
Join Inner, (id#107L = id#31L)
:- Filter isnotnull(id#31L)
:  +- LogicalRDD [id#31L, name#32], false
+- Filter isnotnull(id#107L)
   +- LogicalRDD [id#107L, dept_name#108], false

== Physical Plan ==
*SortMergeJoin [id#31L], [id#107L], Inner
:- *Sort [id#31L ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(id#31L, 200)
:     +- *Filter isnotnull(id#31L)
:        +- *Scan ExistingRDD[id#31L,name#32]
+- *Sort [id#107L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(id#107L, 200)
      +- *Filter isnotnull(id#107L)
         +- *Scan ExistingRDD[id#107L,dept_name#108]
```

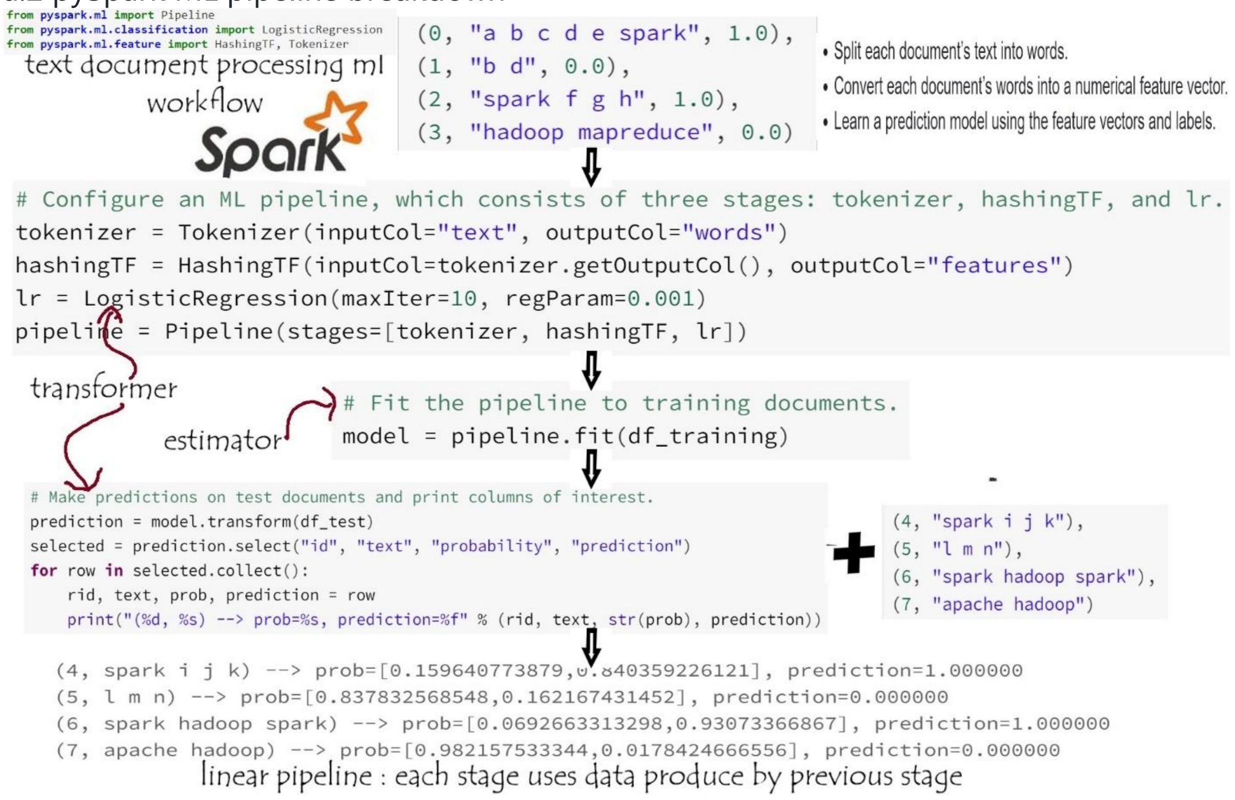
```
spark.sql("select * from vw_employee join vw_dept on vw_employee.id = vw_dept.id ").explain(True)

== Parsed Logical Plan ==
'Project [*]
+- 'Join Inner, ('vw_employee.id = 'vw_dept.id)
   :- 'UnresolvedRelation `vw_employee`
   +- 'UnresolvedRelation `vw_dept`

== Analyzed Logical Plan ==
id: bigint, name: string, id: bigint, dept_name: string
Project [id#31L, name#32, id#107L, dept_name#108]
+- Join Inner, (id#31L = id#107L)
   :- SubqueryAlias vw_employee
   :  +- LogicalRDD [id#31L, name#32], false
   +- SubqueryAlias vw_dept
      +- LogicalRDD [id#107L, dept_name#108], false

== Optimized Logical Plan ==
Join Inner, (id#31L = id#107L)
:- Filter isnotnull(id#31L)
:  +- LogicalRDD [id#31L, name#32], false
+- Filter isnotnull(id#107L)
   +- LogicalRDD [id#107L, dept_name#108], false

== Physical Plan ==
*SortMergeJoin [id#31L], [id#107L], Inner
:- *Sort [id#31L ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(id#31L, 200)
:     +- *Filter isnotnull(id#31L)
:        +- *Scan ExistingRDD[id#31L,name#32]
+- *Sort [id#107L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(id#107L, 200)
      +- *Filter isnotnull(id#107L)
         +- *Scan ExistingRDD[id#107L,dept_name#108]
```

```
df_employee.join(df_dept,"id","left").explain()

== Physical Plan ==
*Project [id#31L, name#32, dept_name#108]
+- SortMergeJoin [id#31L], [id#107L], LeftOuter
   :- *Sort [id#31L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(id#31L, 200)
   :     +- *Scan ExistingRDD[id#31L,name#32]
   +- *Sort [id#107L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(id#107L, 200)
         +- *Filter isnotnull(id#107L)
            +- *Scan ExistingRDD[id#107L,dept_name#108]
```

```
df_employee.join(df_dept.hint("broadcast"),"ID","left").explain()

== Physical Plan ==
*Project [id#31L, name#32, dept_name#108]
+- *BroadcastHashJoin [id#31L], [id#107L], LeftOuter, BuildRight, false
   :- *Scan ExistingRDD[id#31L,name#32]
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
      +- *Filter isnotnull(id#107L)
         +- *Scan ExistingRDD[id#107L,dept_name#108]
```

https://www.linkedin.com/in/vivek-bombatkar/
bombatkar.vivek@gmail.com

- a.2 pySpark ML pipeline breakdown


text document processing ml
workflow
Spark
transformer
estimator
linear pipeline : each stage uses data produce by previous stage

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer
```

```
(0, "a b c d e spark", 1.0),
(1, "b d", 0.0),
(2, "spark f g h", 1.0),
(3, "hadoop mapreduce", 0.0)
```

- Split each document's text into words.
- Convert each document's words into a numerical feature vector.
- Learn a prediction model using the feature vectors and labels.

```
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

```
# Fit the pipeline to training documents.
model = pipeline.fit(df_training)
```

```
# Make predictions on test documents and print columns of interest.
prediction = model.transform(df_test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))
```

+

```
(4, "spark i j k"),
(5, "l m n"),
(6, "spark hadoop spark"),
(7, "apache hadoop")
```

```
(4, spark i j k) --> prob=[0.159640773879,0.840359226121], prediction=1.000000
(5, l m n) --> prob=[0.837832568548,0.162167431452], prediction=0.000000
(6, spark hadoop spark) --> prob=[0.0692663313298,0.93073366867], prediction=1.000000
(7, apache hadoop) --> prob=[0.982157533344,0.0178424666556], prediction=0.000000
```

- a.3 Action[1] --> Job[1] --> Stages[n] --> Tasks[n]
    - new ***job*** is created on ***actions***
    - new ***stages*** will be create if there is ***data shuffle*** in job. I.e. dependency on output of first stage
    - new ***tasks*** will be created based on ***number of partitions*** in RDD in cluster.

```
rdd1 = sc.textFile("f1") #transformation - stage 1
rdd2 = sc.textFile("f2")  #transformation - stage 2
rdd3 = rdd1.join(rdd2) #transformation + shuffle - stage 3
rdd4 = rdd3.mapPartition() #transformation - stage 3
rdd5 = rdd4.filter() #transformation - stage 3
rdd5.collect() #actions - stage 3
```

- a.4 Spark Standalone Mode
    - In addition to running on the Mesos or YARN cluster managers, Spark also provides a simple standalone deploy mode.

```
./bin/spark-shell --master spark://IP:PORT
# URL of the master
```

- a.5 supervise flag to spark-submit

  - In standalone cluster mode supports restarting your application automatically if it exited with non-zero exit code.
    `spark-submit --supervise ...`

- a.6 Dynamic Allocation
  - https://spark.apache.org/docs/latest/configuration.html#dynamic-allocation
  - scales the ***number of executors*** registered with this application up and down based on the workload.
  - `spark.dynamicAllocation.enabled`

- a.7 Speculative execution
  - `spark.speculation`
  - If set to "true", if one or more tasks are running slowly in a stage, they will be re-launched.

- a.8 locality wait
  - `spark.locality.wait`
  - How long to wait to launch a data-local task before giving up and launching it on a less-local node.
  - The same wait will be used to step through multiple locality levels (process-local, node-local, rack-local and then any).
  - It is also possible to customize the waiting time for each level by setting spark.locality.wait.node, etc.
  - You should increase this setting if your tasks are long and see poor locality, but the default usually works well.

a.10 Performance Tunning

- http://spark.apache.org/docs/latest/tuning.html
- when tuning a Spark application – most importantly, data serialization and memory tuning, CPU, network bandwidth, memory
- Data Serialization:
  - Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation.
  - Java serialization (default)
  - Kryo serialization: SparkConf and calling conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer").
- Memory Tuning:

- the amount of memory used by your objects (you may want your entire dataset to fit in memory),
- the cost of accessing those objects
- the overhead of garbage collection (if you have high turnover in terms of objects).

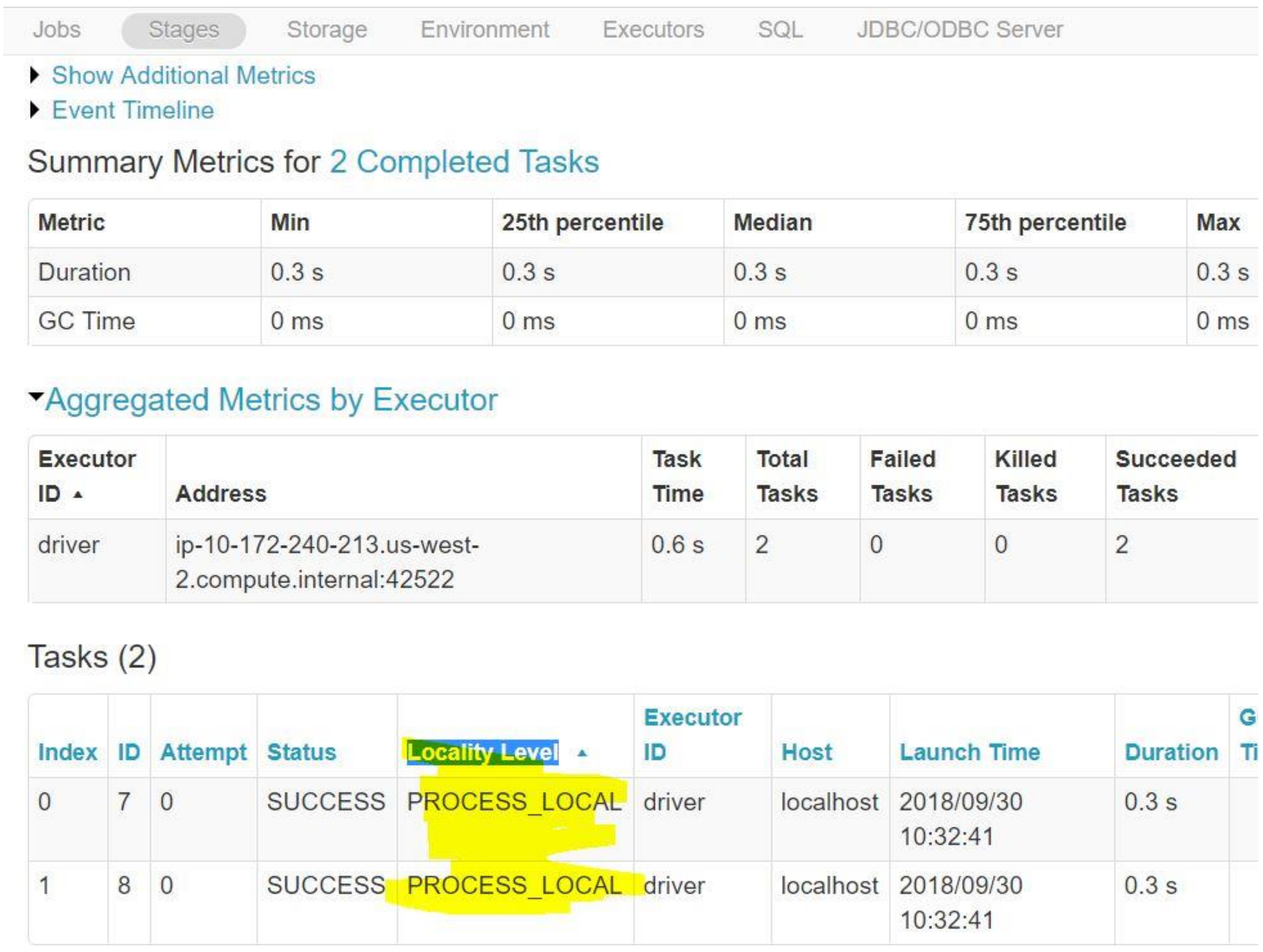
- Memory Management Overview :
  - two categories: execution and storage.
  - ***Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations,
  - ***Storage memory refers to that used for caching and propagating internal data across the cluster.***
  - When no execution memory is used, storage can acquire all the available memory and vice versa.
  - `spark.memory.fraction`
  - `spark.memory.storageFraction`
- ***How Determining Memory Consumption***
- create an RDD, put it into cache, and look at the "Storage" page in the web UI
- SizeEstimator's estimate - consumption of a particular object
- With cache(), you use only the default storage level MEMORY_ONLY. With persist(), you can specify which storage level you want.
  - MEMORY_ONLY
  - MEMORY_ONLY_SER
  - MEMORY_AND_DISK
  - MEMORY_AND_DISK_SER
  - DISK_ONLY
- Tuning Data Structures
  - avoid the Java features that add overhead, such as pointer-based data structures and wrapper objects.
  - prefer arrays of objects, and ***primitive types***, instead of the standard Java or Scala collection classes
  - Avoid nested structures with a lot of small objects and pointers when possible.
  - Consider using numeric IDs or ***enumeration objects*** instead of strings for keys.

- Serialized RDD Storage
    - When your objects are still too large to efficiently store despite this tuning, a much simpler way to reduce memory usage is to store them in serialized formt
    - Downside is performance hit, as it add overhead of deserialization every time
- Garbage Collection Tuning
- ***Level of Parallelism***
    - Spark automatically sets the number of
    - "map" tasks to run on each file according to its size (though you can control it through optional parameters to SparkContext.textFile, etc),
    - and for distributed "reduce" operations, it uses the largest parent RDD's number of partitions.
    - `spark.default.parallelism`
    - recommend ***2-3 tasks per CPU core*** in your cluster.
    - You can safely increase the level of parallelism to more than the number of cores in your clusters.
- ***Memory Usage of Reduce Tasks***
- Spark's shuffle operations (sortByKey, groupByKey, reduceByKey, join, etc) build a ***hash table*** within each task to perform the grouping, which can often be large.
- The simplest fix here is to increase the level of parallelism, so that each task's input set is smaller
- ***Broadcasting Large Variables***
    - in general tasks larger than about 20 KB are probably worth optimizing.
- ***Data Locality***

https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-data-locality.html

- If data and the code that operates on it are together then computation tends to be fast
- Typically it is faster to ship serialized code from place to place than a chunk of data because code size is much smaller than data. - Spark builds its scheduling around this general principle of data locality.

- Spark prefers to schedule all tasks at the best locality level, but this is not always possible.
- In situations where there is no unprocessed data on any idle executor, Spark switches to lower locality levels.
- There are two options:
  - a) wait until a busy CPU frees up to start a task on data on the same server, or
  - b) immediately start a new task in a farther away place that requires moving data there.
- What Spark typically does is wait a bit in the hopes that a busy CPU frees up.
- Once that timeout expires, it starts moving the data from far away to the free CPU.
- You should increase these settings if your tasks are long and see poor locality, but the default usually works well.
- The best means of checking whether a task ran locally is to inspect a given stage in the Spark UI.
- In the Stages tab of spark UI ***Locality Level*** column displays which locality a given task ran with.
- Locality Level : PROCESS_LOCAL, NODE_LOCAL, RACK_LOCAL, or ANY



Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.3 s	0.3 s	0.3 s	0.3 s	0.3 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks
driver	ip-10-172-240-213.us-west-2.compute.internal:42522	0.6 s	2	0	0	2

Tasks (2)

Index	ID	Attempt	Status	Locality Level ▲	Executor ID	Host	Launch Time	Duration	G Ti
0	7	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/09/30 10:32:41	0.3 s	
1	8	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/09/30 10:32:41	0.3 s	

Kryo serialization

https://spark.apache.org/docs/latest/tuning.html#data-serialization

- ***For most programs, switching to Kryo serialization and persisting data in serialized form will solve most common performance issues

a.11 Job Scheduling

- http://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application
-

a.12 Spark Security

- http://spark.apache.org/docs/latest/security.html

a.13 Hardware Provisioning

- http://spark.apache.org/docs/latest/hardware-provisioning.html

a.14 Shuffles

- http://hydronitrogen.com/apache-spark-shuffles-explained-in-depth.html
-

a.15 Partitioning

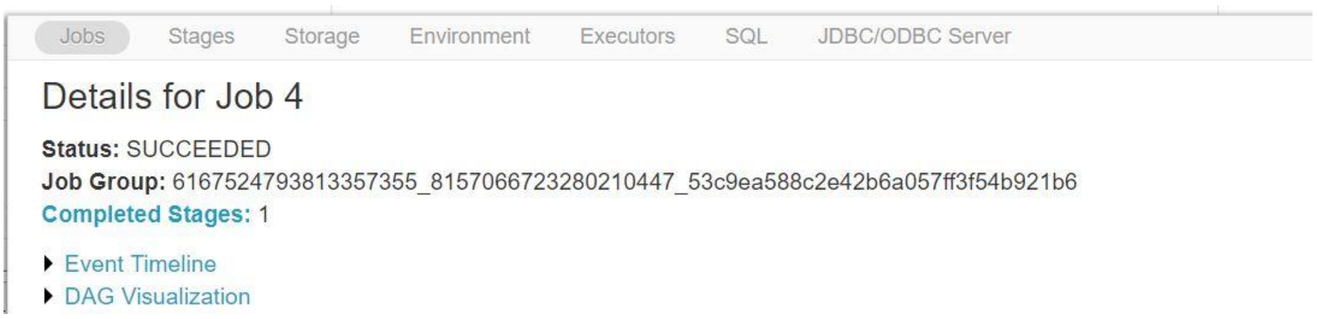
- https://medium.com/parrot-prediction/partitioning-in-apache-spark-8134ad840b0
- https://techmagie.wordpress.com/2015/12/19/understanding-spark-partitioning/
- https://www.talend.com/blog/2018/03/05/intro-apache-spark-partitioning-need-know/
    - Every node in a Spark cluster contains one or more partitions.
    - too few (causing less concurrency, data skewing & improper resource utilization)
    - too many (causing task scheduling to take more time than actual execution time)
    - By default, it is set to the total number of cores on all the executor nodes.
    - Partitions in Spark do not span multiple machines.
    - Tuples in the same partition are guaranteed to be on the same machine.
    - Spark assigns one task per partition and each worker can process one task at a time.

b. WEB UI / Spark UI

spark web ui
https://www.cloudera.com/documentation/enterprise/5-9-x/topics/operation_spark_applications.html

- A job can be in a running, succeeded, failed or unknown state.
- `JOBS --> STAGES --> TASKS`

Below tabs from spark UI



Details for Job 4

Status: SUCCEEDED
Job Group: 6167524793813357355_8157066723280210447_53c9ea588c2e42b6a057ff3f54b921b6
Completed Stages: 1

- Event Timeline
- DAG Visualization

- i. ***JOBS tab*** : The Jobs tab consists of two pages, i.e. All Jobs and Details for Job pages.

- ii. ***STAGES tab***:
  - Stages tab in web UI shows the current state of 'all stages of all jobs' in a Spark application (i.e. a SparkContext)
  - two optional pages for the tasks and statistics for a stage (when a stage is selected) and pool details (when the application works in FAIR scheduling mode).
  - ***Summary Metrics*** :
    - for Completed Tasks in Stage : The summary metrics table shows the metrics for the tasks in a given stage that have already finished with SUCCESS status and metrics available.
    - The table consists of the following columns: Metric, Min, 25th percentile, Median, 75th percentile, Max.

- iii. ***STORAGE tab*** :
  - When created, StorageTab creates the following pages and attaches them immediately: A. StoragePage B.RDDPage
  - All Stages Page: shows the task details for a stage given its id and attempt id.
  - Stagev Details page / The Fair Scheduler Pool Details page : shows information about a Schedulable pool and is only available when a Spark application uses the FAIR scheduling mode (which is controlled by spark.scheduler.mode setting).

- iv. ***ENVIRONMENT tab***: Shows various details like total tasks, Input, Shuffle read & write, etc

-

v. ***EXECUTORS tab*** : list all executors used

  - Input - total data processed or read by the application from hadoop or spark storage
  - Storage Memory - tatal memory used or available

-

vi. ***SQL tab***: SQL tab in web UI shows SQLMetrics per physical operator in a structured query physical plan.

  - By default, it displays all SQL query executions.
  - However, after a query has been selected, the SQL tab displays the details for the structured query execution

c. RDD + DataFrame + DataSets + SparkSQL

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

- Internally, each RDD is characterized by 5 main properties:
  - a. A list of partitions
  - b. A function for computing each split
  - c. A list of dependencies on other RDDs
  - d. Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - e. Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
- Types of RDD
- type based on how RDDs made
- HadoopRDD, FilterRDD, MapRDD, ShuffleRDD, S3RDD , etc

d. Streaming
https://spark.apache.org/docs/latest/streaming-programming-guide.html

e. SparkMLLib

https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-mllib/spark-mllib.html

f. GraphLib

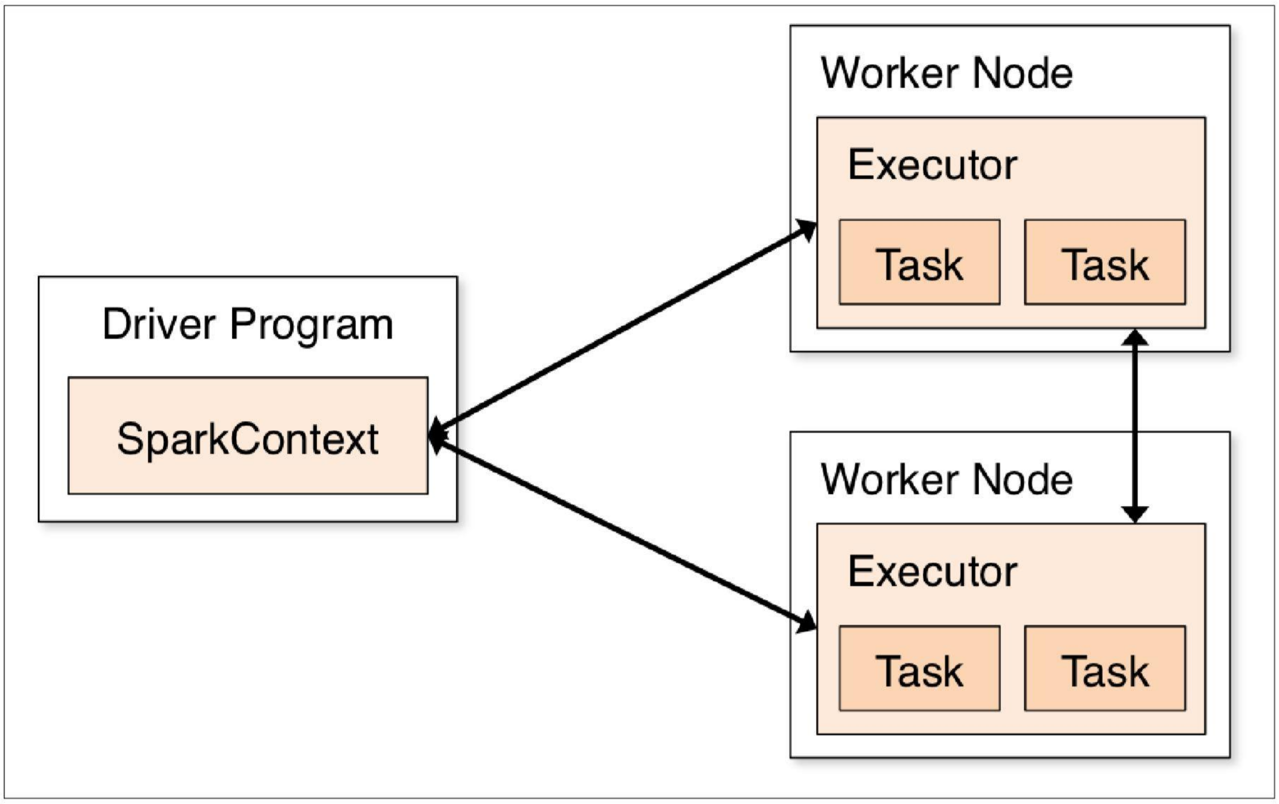
https://spark.apache.org/docs/latest/graphx-programming-guide.html

4. NOTES FROM THE BOOKS / GUIDES.

4.1 Learning Spark: Lightning-Fast Big Data

Introduction to Data Analysis with Spark

- cluster computing platform
- Spark application consists of a driver program that launches various parallel operations on a cluster.
- driver programs typically manage a number of nodes called executors
- ***SparkContext*** represents a connection to a computing cluster.



Programming with RDDs

- Resilient Distributed Dataset (RDD)
- an immutable distributed collection of objects
- It split into multiple partitions, which may be computed on different nodes of the cluster
- ***Transformations*** construct a new RDD from a previous one.
- ***Actions***, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system
- lazy evaluation - Spark only computes them in a lazy fashion
- to reuse an RDD in multiple actions, you can ask Spark to persist it using RDD.persist().
- three options for passing functions into Spark - lambda. top level function or locally define functions
- reduce / fold

Working with Key-Value Pairs

- RDDs containing key-value pairs. These RDDs are called Pair RDDs.
- Transformations one pair rdd : reduceByKey / foldByKey, combineByKey, countByValue, groupByKey, mapValues, flatMapValues, keys, values, sortByKey

- Transformations on two pair rdd : substractByKey, join, rightOuterJoin, leftOuterJoin, cogroup
- Actions : collectAsMap(), lookup()
- Most operator accept a second parameter giving the number of partitions to use when creating the grouped or aggregated RDD
- repartitioning your data is a fairly expensive operation
- Partitioning will not be helpful in all applications — for example, if a given RDD is only scanned once, there is no point in partitioning it in advance. It is only useful when a dataset is reused multiple times in key-oriented operations such as joins.
- ***partitionBy***
- HashPartitioner

Loading and Saving Your Data

- Comprassion optison : gzip, lzo, bzip2, zlib, Snappy

Advanced Spark Programming

- ***accumulators*** to aggregate information.
- One of the most common uses of accumulators is to count events that occur during job execution for debugging purposes.
- Note that tasks on worker nodes cannot access the accumulator's value — from the point of view of these tasks, accumulators are write-only variables.
- ***speculative execution*** Spark can preemptivley launch a "speculative" copy of the task on another node, and take its result if that finishes.
- accumulators updated in actions vs in transformations
- broadcast variables to efficiently distribute large values. allow the program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations.
- ***PrePartition operations***: mapPartition, foreachPartition, mapPartitionWithIndex

Running on a Cluster

- When running in cluster mode, Spark utilizes a master-slave architecture with one central coordinator and many distributed workers.
- The central coordinator is called the driver.
- The driver communicates with potentially larger number of distributed workers called executors.

- The driver runs in its own Java process and each executor is a Java process.
- A driver and its executors are together termed a Spark application.
- A Spark application is launched on a set of machines using an external service called a cluster manager.
- Driver program main duties :
    - a. compiling user program into task
    - b. scheduling task on executor
- Executor
    - a. running the tasks
    - b. in-memory storage for rdd
- Sparks Dirver & Executor VS YARNs Master & Worker
    - For instance Apache YARN runs a master daemon (called the Resource Manager) and several worker daemons called (Node Managers).
    - Spark will run both drivers and executors on YARN worker nodes.
- spark2-submit options types :
    - The first is the location of the cluster manager along with an amount of resources you'd like to request for your job (as shown above).
    - The second is information about the runtime dependencies of your application, such as libraries or files you want to be present on all worker machines.

4.2 High Performance Spark - Holden Karau and Rachel Warren

Spark Model of Parallel Computing: RDDs

- driver (or master node) perform operations on data in parallel.
- Spark represents large datasets as RDDs, immutable distributed collections of objects,
- which are stored in the executors or (slave nodes).
- The objects that comprise RDDs are called partitions
- Partitions may be (but do not need to be) computed on different nodes of a distributed system.
- Spark can keep an RDD loaded in memory on the executor nodes throughout the life of a Spark application for faster access
- RDDs are immutable, so transforming an RDD returns a new RDD rather than the existing one.
- Actions trigger the scheduler, which builds a directed acyclic graph (called the DAG), based on the dependencies between RDD transformations.

- Then, using this series of steps called the execution plan, the scheduler computes the missing partitions for each stage until it computes the whole RDD.

In Memory Storage and Memory Management

- Spark offers three options for memory management:
    - in memory deserialized data - higher performace but consume high memory
    - in memory as serialized data - slower performance but low disk space
    - on disk - slower and nothing in memory, can be more fault tolarent for long string transformations
- The persist() function in the RDD class lets the user control how the RDD is stored.
- By default, persist() stores an RDD as deserialized objects in memory.

five main properties to represent an RDD internally.

- partitions()
- iterator(p, parentIters)
- dependencies()
- partitioner()
- preferredLocations(p)

Resource Allocation Across Applications

- static allocation
- dynamic allocation

The Anatomy of a Spark Job

spark application -> jobs -> stages -> tasks

- ***jobs***
    - highest element of Spark's execution hierarchy.
    - Each Spark job corresponds to one action
- ***stages***
    - As mentioned above, a job is defined by calling an action.
    - The action may include several transformations, which breakdown of jobs into stages.

  - Several transformations with narrow dependencies can be grouped into one stage.
  - It is possible to executed stages in parallel if they are used to compute different RDDs
  - wide transformations needed to compute one RDD have to be computed in sequence
  - one stage can be computed without moving data across the partitions.
  - Within one stage, the tasks are the units of work done for each partition of the data.
- ***tasks***
  - A stage consists of tasks.
  - The task is the smallest unit in the execution hierarchy
  - each can represent one local computation.
  - One task cannot be executed on more than one executor.
  - However, each executor has a dynamically allocated number of slots for running tasks
  - The number of tasks per stage corresponds to the number of partitions in the output RDD of that stage.

Spark SQL's column operators are defined on the column class, so a filter containing the expression 0 >= df.col("friends") will not compile since Scala will use the >= defined on 0. Instead you would write df.col("friend") <= 0 or convert 0 to a column literal with lit

- ***Transformations*** : types

  - filters
  - sql standard functions
  - 'when' - for if then else
  - Specialized DataFrame Transformations for Missing & Noisy Data
  - Beyond Row-by-Row Transformations
  - Aggregates and groupBy - agg API
  - windowing
  - sorting - orderBy
  - Multi DataFrame Transformations

- Tungsten

  - Tungsten is a new Spark SQL component that provides more efficient Spark operations by working directly at the byte level.

  - Tungsten includes specialized in-memory data structures tuned for the type of operations required by Spark
  - improved code generation, and a specialized wire protocol.

- Query Optimizer

  - Catalyst is the Spark SQL query optimizer,
  - which is used to take the query plan and transform it into an execution plan that Spark can run.
  - Much as our transformations on RDDs build up a DAG, Spark SQL builds up a tree representing our query plan, called a logical plan.
  - Spark is able to apply a number of optimizations on the logical plan
  - also choose between multiple physical plans for the same logical plan using a cost-based mode.

Joins (SQL & Core)

- In order to join data, Spark needs the data that is to be joined to live on the same partition.
- The default implementation of join in Spark is a shuffled hash join.
- Shuffel could be avoided if
  - a. ***Both RDDs have a known partitioner.***
  - b. map side join -One of the datasets is small enough to fit in memory, in which case we can do a broadcast hash join
- ***Left semi joins***
  - are the only kind of join which only has values from the left table.
  - A left semi join is the same as filtering the left table for only rows with keys present in the right table.
  - `df1.join(df2, df1("name") === df2("name"), "leftsemi")`
- ***Broadcast Hash Joins***
  - df1.join(broadcast(df2), "key")
  - Spark also automatically uses the spark.sql.conf.autoBroadcastJoinThreshold to determine if a table should be broadcast.

4.5 "Programming Guides" from http://spark.apache.org/docs/latest/

Passing Functions to Spark

- There are three recommended ways to do this:
  - Lambda expressions. Lambdas do not support multi-statement functions or statements that do not return a value.)
  - Local defs inside the function calling into Spark, for longer code.
  - Top-level functions in a module.
  - method in a class instance (as opposed to a singleton object), this requires sending the object that contains that class along with the method.
  -

Lading any external files to spark dataframe : spark.read.load / spark.read

```
df_json = spark.read.load("FILE_LOCATION.json",format="json)
df_csv = spark.read.load("FILE_LOCATION.csv", format="csv", sep=",", inferSchema =
"true", header = "true")
df_parquet = spark.read.parquet("FILE_LOCATION.parquet")
df_jdbc = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .load()
```

Writing data to external : sdf.write.save & write.option("path":"DIR_LOCATION").saveAsTable("tble1")

- .saveAsTable("tble1") : For file-based data source, e.g. text, parquet, json, etc. you can specify a custom table path via the path option. When the table is dropped, the custom table path will not be removed and the table data is still there.

```
sdf.write.parquet("DIR_LOCATION")
sdf.write.save(FILE_LOCATION.parquet)
```

- ***partitionBy*** creates a directory structure as described in the Partition Discovery section. columns with ***high cardinality***.
- ***bucketBy*** distributes data across a fixed number of buckets and can be used when a number of unique values is unbounded.

```
df.write
    .partitionBy("favorite_color")
    .bucketBy(42, "name")
    .saveAsTable("people_partitioned_bucketed")
```

Schema Merging

- Like ProtocolBuffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed.
- In this way, users may end up with multiple Parquet files with different but mutually compatible schemas.
- The Parquet data source is now able to automatically detect this case and merge schemas of all these files.

```
spark.read.option("mergeSchema", "true").parquet("FOLDER_LOCATION")
```

Parquet Files

- Parquet is a columnar format that is supported by many other data processing systems.
- Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data.
- When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

HIVE vs Parquet

- Hive is case insensitive, while Parquet is not
- Hive considers all columns nullable, while nullability in Parquet is significant

Best way to load data from URL to spark - Pandas

```
#Example to load csv
import pandas as pd
sdf =
spark.createDataFrame(pd.read_csv("https://raw.githubusercontent.com/fivethirtyeig
ht/data/master/airline-safety/airline-safety.csv"))
```

Pandas in spark

- Scalar Pandas UDFs are used for vectorizing scalar operations.
- They can be used with functions such as select and withColumn
- toPandas() will convert the Spark DataFrame into a Pandas DataFrame, which is of course in memory.

```
def multi_fun(a, b):
  return a * b

x = pd.Series([1,2,3,4])
multi = pandas_udf(multi_fun,returnType=LongType())
sdf= spark.createDataFrame(pd.DataFrame(x, columns=["x"]))
sdf.select(multi(col("x"),col("x"))).show()
```

Grouped Map on Pandas df : Split-apply-combine

- Grouped map Pandas UDFs are used with groupBy().apply() which implements the "split-apply-combine" pattern.
- Split-apply-combine consists of three steps:
    - Split the data into groups by using DataFrame.groupBy.
    - Apply a function on each group. The input data contains all the rows and columns for each group.
    - Combine the results into a new DataFrame.

```
from pyspark.sql.functions import  pandas_udf, PandasUDFType

sdf_grp = spark.createDataFrame([(1,10),(2,10),(3,30)],("id","v"))

@pandas_udf("id integer, v double", PandasUDFType.GROUPED_MAP)
def fun_1(pdf):
  v = pdf.v
  return pdf.assign(v = v - v.mean())

sdf_grp.groupBy("id").apply(fun_1).show()
```

Arrow : JVM to Python data xfer

- Apache Arrow is an in-memory columnar data format.
- that is used in Spark to efficiently transfer data between JVM and Python processes
- good with Pandas/NumPy data.
- PyArrow - pip install pyspark[sql]
- ***'spark.sql.execution.arrow.enabled' to 'true'

NaN

- There is specially handling for not-a-number (NaN)
- when dealing with float or double types that does not exactly match standard floating point semantics.

5. SPARKSESSION & PYSPARK.SQL.FUNCTIONS f

http://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html

- functions could be passed to API to perform operations
- like aggregate functions used with 'agg' API

```
from pyspark.sql import functions f
```

lit()

Creates a Column of literal value

```
df.withColumn("col1", f.when("col2") == 0, f.lit("Y")).otherwise(f.lit("N")))
```

monotonically_increasing_id()

A column that generates monotonically increasing 64-bit integers. monotonically increasing and unique, but not consecutive.

```
df.withColumn("new_id", f.monotonically_increasing_id())
```

SparkSession.table()

Returns the specified table as a DataFrame.

expr()

Parses the expression string into the column that it represents

```
col_condn = f.exppr("if(col is null, 1,0)")
df.withColumn("col1",col_condn)
```

JOIN

http://www.learnbymarketing.com/1100/pyspark-joins-by-example/
https://spark.apache.org/docs/2.3.0/api/python/pyspark.sql.html
https://spark.apache.org/docs/2.3.0/api/python/_modules/pyspark/sql/dataframe.html#DataFrame.join

```
df_res = df_one.join(df_two,df_one.col1 == df_two.col1,"left")
df_res = df_one.join(other=df_two,on=["col1"],how="left")
df_res = df_one.alias("a").join(df_two.alias("b"),col("a.col1") ==
col("b.col1"),"left")
```

- (inner, outer, left_outer, right_outer, leftsemi)
- Join takes three parameters: DataFrame on the right side of the join, Which fields are being joined on, and what type of join

- An ***inner join*** is the default join type used
- default `inner`. Must be one of: `inner`, `cross`, `outer`, `full`, `full_outer`, `left`, `left_outer`, `right`, `right_outer`, `left_semi`, and `left_anti`
- 'leftsemi' if you care only for the left columns and just want to pull in the records that match in both table A and table B, y
-

distinct()

https://stackoverflow.com/questions/30959955/how-does-distinct-function-work-in-spark

- shuffle data accross partition

dataFrame.checkpoint

https://dzone.com/articles/what-are-spark-checkpoints-on-dataframes
https://stackoverflow.com/questions/35127720/what-is-the-difference-between-spark-checkpoint-and-persist-to-a-disk

- Returns a checkpointed version of this dataset.
- Checkpointing can be used to ***truncate the logical plan of this dataset***, which is especially useful in iterative algorithms where the plan may grow exponentially.
- It will be saved to files inside the checkpoint directory set with SparkContext#setCheckpointDir.
- types : Eager Checkpoint & Non-Eager Checkpoint
- eager – Whether to checkpoint this DataFrame immediately

```
df.checkpoint
```

pyspark.sql.window

https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html

```
window_condn = window \
               .partitionby(df.col_date) \
               .rangeBetween(min_val, max_val) \
                .rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
\
                .OrderBy(df.col_date)

df_new = df.withColumn('col1',f.sum('col2')).over(window_condn)
```

df.pivote

https://databricks.com/blog/2016/02/09/reshaping-data-with-pivot-in-apache-spark.html

- Pivots a column of the current DataFrame and perform the specified aggregation.
- There are two versions of pivot function: one that requires the caller to specify the list of distinct values to pivot on, and one that does not.
- The latter is more concise but less efficient, because Spark needs to first compute the list of distinct values internally.

f.explode(col)

Returns a new row for each element in the given array or map.

```
new_df = df.select(f.explode(df.col1))
```

df.groupBy(*cols)

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

```
df.groupBy(df.col1).count().collect()
```

df.agg(*expr)

Aggregate on the entire DataFrame without groups (shorthand for df.groupBy.agg()).

```
df.groupBy(df.col1).agg(a.max(df.col1))
```

- get non groupby cols in df

```
pyspark.sql.functions import collect_set
df.groupBy(df.col1).agg(a.collect_set(df.col2))

col1  collect_set(col2)
1     [2,3]
5     [7,9]
```

pyspark.sql.types.ArrayType

```
from pyspark.sql.functions import udf
@udf(returnType=ArrayType(StringType()))
def func_test_udf(a):
   return []

 new_df = df.withColumn("new_col", func_test_udf(df.col1))

An error occurred while calling
None.org.apache.spark.sql.execution.python.UserDefinedPythonFunction. Trace:
class org.apache.spark.sql.types.ArrayType]) does not exist
```

df.filter()

```
df_new = df.filter(col("col1").isNotNull & col("col1") > 100)
```

f.first(col)

Aggregate function: returns the first value in a group.

df.distinct()

```
df.select("abc").distinct()
```

add / remove cols

```
sdf.withColumn("new_col",lit(1))

sdf.drop("new_col)
```

eqNullSafe

- include NULL values in the join

```
sdf_1.join(sdf_2,sdf_1.col_1.eqNullSafe(sdf_2.col_1))
```

RDD , partition , tasks etc..

https://qubole.zendesk.com/hc/en-us/articles/217111026-Reference-Relationship-between-Partitions-Tasks-Cores

- `# of Spark RDD / Data Frame Partitions = Result of Partitioning Logic for Spark Function`
- For the first task this is driven by the number of files in the source: `# of Tasks required for Stage = # of Source Partitions`
- For the subsequent tasks this is driven by the number of partitions from the prior stages: `# of Tasks required for Stage = # of Spark RDD / Data Frame Partitions`