Philly Area Apache Spark Meetup

By

Nava Nomula

nnomula@Jornaya.com

Nava@LinkedIn

Jornaya

Who am I?

Nava Nomula

Current: Senior Data Engineer @ Jornaya Inc

Past: Teradata Enterprise Data Warehouse (Massive Parallel Processing) background, consultant at Teradata, Comcast, Enterprise Rent-A-Car, Altria, Unilever, Wynn||Encore and few other clients.

Topics for today:

Introduction to Spark

Understanding Spark architecture
Spark Internals & Spark UI

Data Normalization

Understanding Data Normalization

How it helps processing performance

Data Shuffle

What is data shuffle in Spark world

Why and when it happens

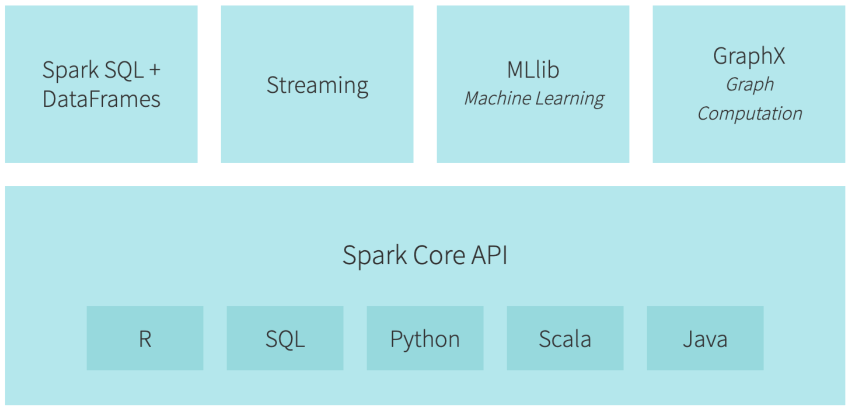
How to manage shuffle

Data Joins

How to join two large datasets
How to avoid the data skew

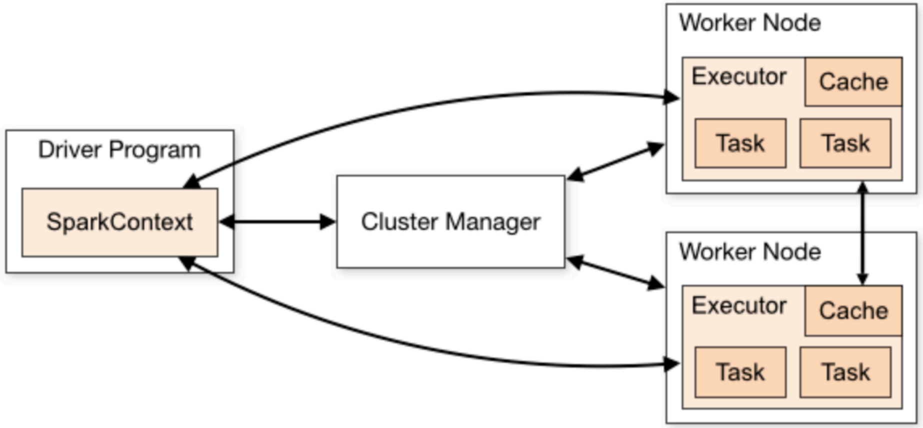
Introduction to Spark

- Apache Spark is an open-source distributed general-purpose cluster-computing framework – Wikipedia
- Apache Spark is a lightning-fast unified analytics engine for big data and machine learning - Data bricks
- Spark eco system



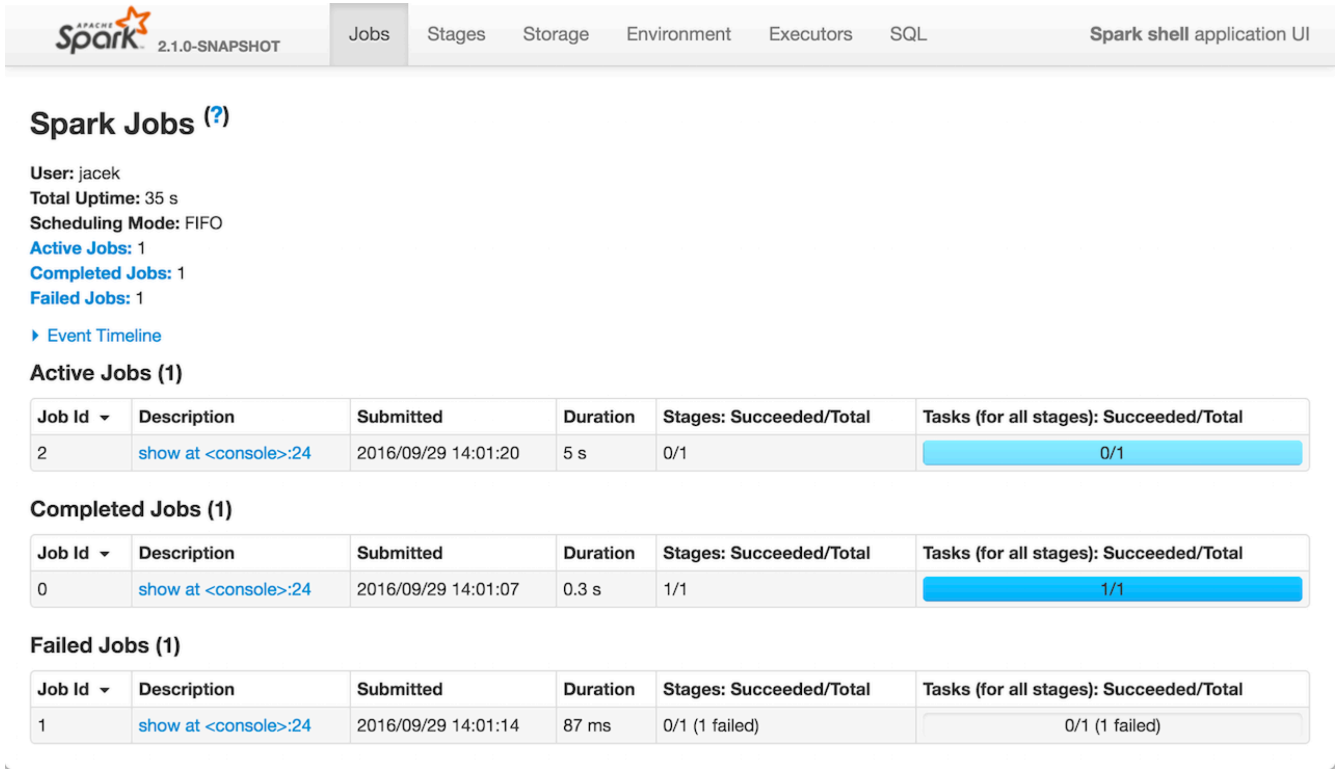
Spark Architecture

- Driver
- Spark Context
- Cluster Manager
- Master Node
- Worker/Core Node



Spark Internals & UI

- Executors - 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
- Core - Number of CPU cores per executor
- Memory - memory configured per executor
- 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
- Task - A unit of work that will be sent to one executor
- RDD - Resilient Distributed Dataset
- DAG - Directed Acyclic Graph
- Partition - a chunk of data in memory (or) disk



Spark 2.1.0-SNAPSHOT | Jobs | Stages | Storage | Environment | Executors | SQL | **Spark shell** application UI

Spark Jobs (?)

User: jacek
Total Uptime: 35 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1
Failed Jobs: 1

▸ Event Timeline

Active Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:01:20	5 s	0/1	0/1

Completed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:01:07	0.3 s	1/1	1/1

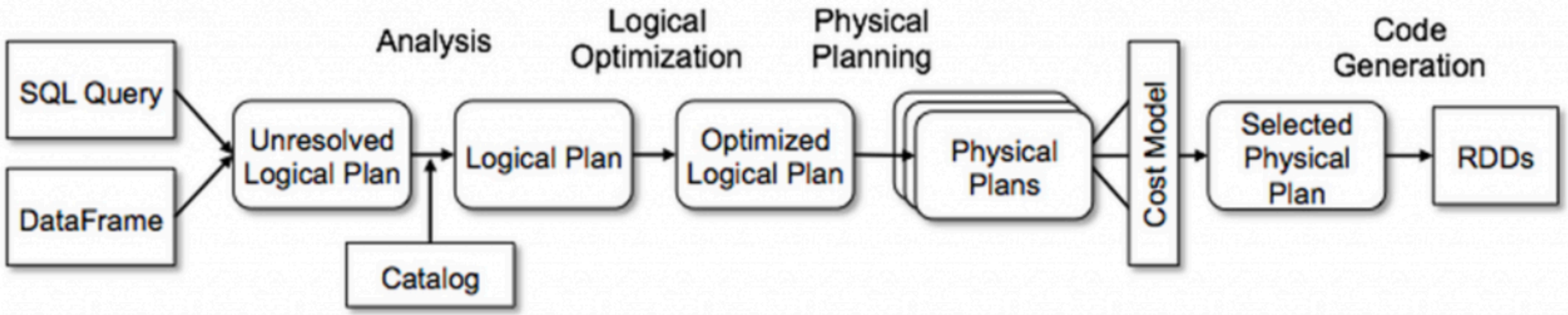
Failed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)

Spark UI

Spark Optimizer

- Unresolved logical plan
- Resolved logical plan
- Physical plan



Spark Optimizer Plans

```
>>> df1.join(df2, df1.key1 == df2.key2).explain(True)
```

== Parsed Logical Plan ==

```
Join Inner, (key1#573L = key2#577L)
:- SubqueryAlias `default`.`unbucketed`
:  +- Relation[key1#573L,class1#574] parquet
+- SubqueryAlias `default`.`bucketed`
   +- Relation[key2#577L,class2#578] parquet
```

== Optimized Logical Plan ==

```
Join Inner, (key1#573L = key2#577L)
:- Filter isnotnull(key1#573L)
:  +- Relation[key1#573L,class1#574] parquet
+- Filter isnotnull(key2#577L)
   +- Relation[key2#577L,class2#578] parquet
```

== Analyzed Logical Plan ==

```
key1: bigint, class1: int, key2: bigint, class2: int
Join Inner, (key1#573L = key2#577L)
:- SubqueryAlias `default`.`unbucketed`
:  +- Relation[key1#573L,class1#574] parquet
+- SubqueryAlias `default`.`bucketed`
   +- Relation[key2#577L,class2#578] parquet
```

== Physical Plan ==

```
*(5) SortMergeJoin [key1#573L], [key2#577L], Inner
:- *(2) Sort [key1#573L ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(key1#573L, 800)
:     +- *(1) Project [key1#573L, class1#574]
:        +- *(1) Filter isnotnull(key1#573L)
:           +- *(1) FileScan parquet default.unbucketed[key1#573L,class1#574]
Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://ip-172-31-16-101.ec2.internal:8020/user/spark/warehouse/unbucketed], PartitionFilters: [], PushedFilters: [IsNotNull(key1)], ReadSchema: struct<key1:bigint,class1:int>
+- *(4) Sort [key2#577L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(key2#577L, 800)
      +- *(3) Project [key2#577L, class2#578]
         +- *(3) Filter isnotnull(key2#577L)
            +- *(3) FileScan parquet default.bucketed[key2#577L,class2#578]
Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://ip-172-31-16-101.ec2.internal:8020/user/spark/warehouse/bucketed], PartitionFilters: [], PushedFilters: [IsNotNull(key2)], ReadSchema: struct<key2:bigint,class2:int>, SelectedBucketsCount: 100 out of 100
```

Spark Memory Management

How many executors?

- Assign based on your system capacity
- Dynamic allocation is preferred (it decides based on load)

How many cores?

- Assign based on how much parallelism your application need

How much memory?

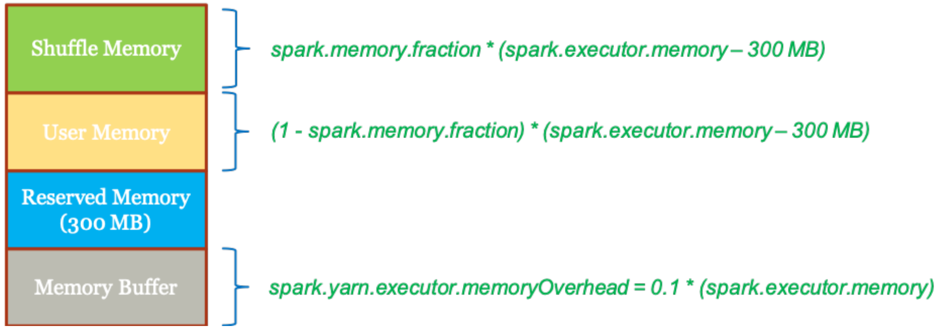
- Assign based on size of biggest partition after shuffle

Spark configuration example

```
{
 "Classification": "spark-defaults",
 "Properties": {
  "spark.sql.shuffle.partitions": "800",
  "spark.executor.instances": ”10",
  "spark.executor.cores": ”12",
  "spark.executor.memory": "128G"
 }
}
```

Spark Memory Allocation

- Shuffle Memory – It is an in-memory available for executor (default is ~60-70 % of total). If this is full then the spilling takes place to disk.
  - Execution – for transformations
  - Storage – for cache
- User Memory – default ~40% (can be controlled by changing the memory fraction)
- Example:
  - Request → ("spark.executor.cores": ”12", "spark.executor.memory": "128G”)
  - Get → ~82GB per executor; ~7GB per core



How to configure the right capacity?

Begin at looking the source data involved

What is the size of data to scan and process

How many partitions it got stored as source

How many shuffle operations your application perform

- Aggregations, Joins, repartitions, partition by

Are you enforcing any parallelism (setting default partitions)

How many partitions your application expecting to write the output

Look at your system capacity

How many nodes? (worker nodes)

What is the node configuration?

- Cores and memory

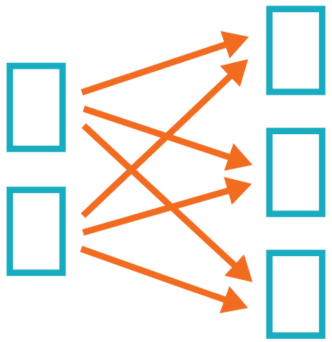
How to configure the right capacity?

EC2 Capacity (r5.24xlarge)	Spark Configs	Allocated (after provision)
2 x : 96 v Core, 768G	Executors : Dynamic Cores : 12 Memory : 128G	depends on load (10 max) 12 81.9G
2 x : 96 v Core, 768G	Cores : 6 Memory : 128G	10 12 81.9G
2 x : 96 v Core, 768G	Executors : 10 Cores : 12 Memory : 128G	10 12 81.9G

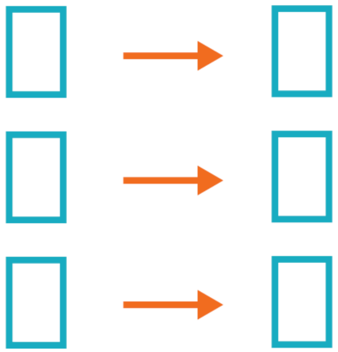
Data Shuffle

- Transformations – always lazy
  - Narrow transformations – always map and part of single stage
  - Wide transformation – initiate the shuffle, these are expensive
- Actions
  - Collect, take, save, write etc.,

Wide Transformations (shuffles)
1 to N



Narrow Transformations
1 to 1



Controlling shuffle

- Shuffles can be defaulted at spark application level
- Shuffles can be managed within spark application by
  - Repartition() – to increase even partition (write in-memory)
  - Coalesce() – to decrease (write in-memory)
  - Partition By() – partition by element (write to disk)
  - Bucket By() - partition by element (write to disk)

```
df.rdd.getNumPartitions()

dd = df.rdd.mapPartitionsWithIndex(lambda x,it: [(x,sum(1 for _ in it))]).collect()

min(dd,key=lambda item:item[1])
max(dd,key=lambda item:item[1])
```

Data Joins

- **Join operations**
  - SortMergeJoins (Standard) - Both sides are large
  - Broadcast Joins (Fastest) - One side is small
  - Skew Joins (Salting)
    - Add Column to each side with random int between 0 and spark.sql.shuffle.partitions – 1 to both sides
    - Add join clause to include join on generated column above
    - Drop temp columns from result
  - Skewed Aggregates

```
df.groupBy(“city”, “state”).agg(<f(x)>).orderBy(col.desc)

val saltVal = random(0, spark.conf.get(org...shuffle.partitions) - 1)

df.withColumn(“salt”, lit(saltVal)) .groupBy(“city”, “state”, “salt”)
.agg(<f(x)>)
.drop(“salt”) .orderBy(col.desc)
```

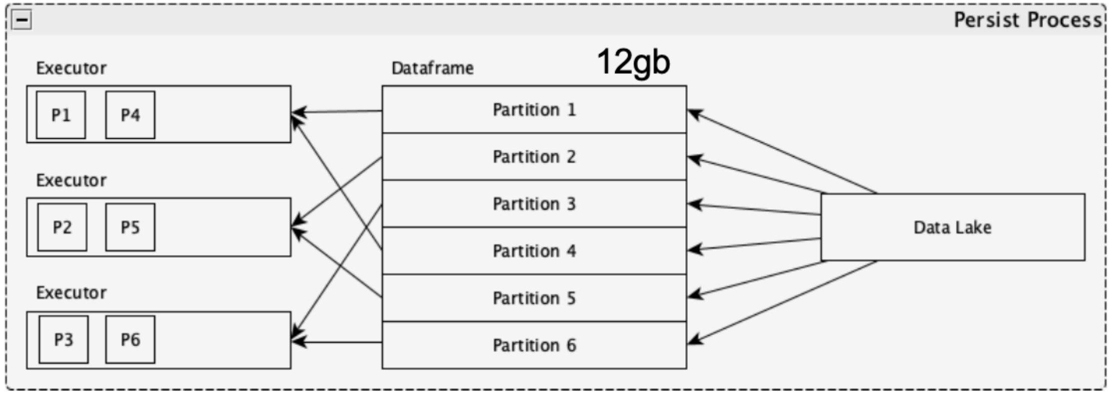
Data Shuffle examples

```
spark.range(1, 100000001, 1, 100)\
    .select(col('id').alias('key1'), (rand(101) * 100).cast(IntegerType()).alias('class1'))\
    .write.format("parquet")\
    .saveAsTable("unbucketed")

spark.range(1, 100000001, 1, 100)\
    .select(col('id').alias('key2'), (rand(101) * 100).cast(IntegerType()).alias('class2'))\
    .write.format("parquet")\
    .bucketBy(100, "key2").sortBy("class2")\
    .saveAsTable("bucketed")

df1 = spark.table("unbucketed")
df2 = spark.table("bucketed")
df3 = spark.table("bucketed")
```

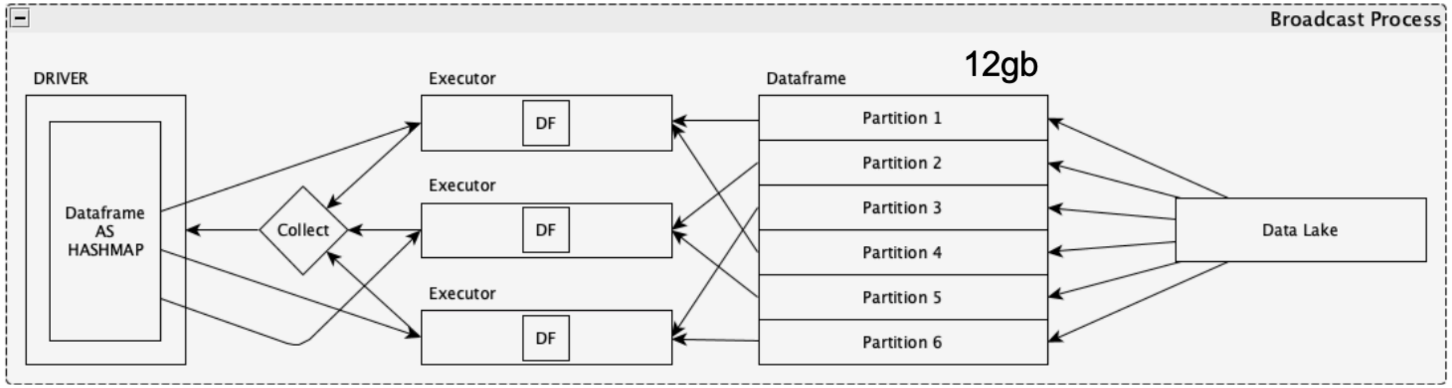
Data Shuffle examples

JOIN OPERATION	DATA SHUFFLE UNDER THE HOOD	PERFORMANCE
df1.join(df2, df1.key1 == df2.key2).explain()	df1 & df2 both shuffled across into (default) partitions by key	Very expensive & Very Slow great chance of OOM
df1.repartition('key1').join(df2, df1.key1 == df2.key2).explain()	df1 shuffled across into (default) partitions by key	Expensive & Slow chance of OOM
df1.repartition(100, 'key1').join(df2, df1.key1 == df2.key2).explain()	df1 shuffled across into 100 partitions by key	Least expensive & Slow
df3.join(df2, df3.key2 == df2.key2).explain()	No shuffling	Cheap and Fastest

Persistence Vs. Broadcast



Attempt to send compute to the data



Data availability guaranteed -> each executor has entire dataset

Expensive Operations

- Repartition
  – Use Coalesce or Shuffle Partition Count
- Count – Do you really need it?
- DistinctCount
  – use approxCountDistinct()
- If distinct is required, put it in the right place

  – Use dropDuplicates
  – dropDuplicates BEFORE the join
  – dropDuplicates BEFORE the groupBy

Other recommendations

- Utilize Lazy Loading (Data Skipping)
- Maximize Your Hardware
- Right Size Spark Partitions
- Balance
- Optimized Joins
- Minimize Data Movement
- Minimize Repetition

Data Normalization

- It’s a practice to avoid the redundancy
- Minimal storage without missing information
- Easy to scan less data
- Less capacity required to process
- Example:

Student	Assessment	Grade	Date
Dave	English	B	11/01/2019
Dave	English	B	11/02/2019
Dave	English	B	11/03/2019
Dave	English	B	11/04/2019
Dave	English	A	11/05/2019
Dave	English	A	11/06/2019

Student	Assessment	Grade	Start Date	End Date
Dave	English	B	11/01/2019	11/04/2019
Dave	English	A	11/05/2019	11/06/2019

DEMO

- Data Normalization
- Data Shuffle
- Data Joins

Questions?