



Spark Optimization

Let's get started..

- ❑ Introductions
- ❑ Agenda
- ❑ Recap of HDFS, YARN and Spark Architecture
- ❑ Scope of Spark Optimization
- ❑ About Lab Environment

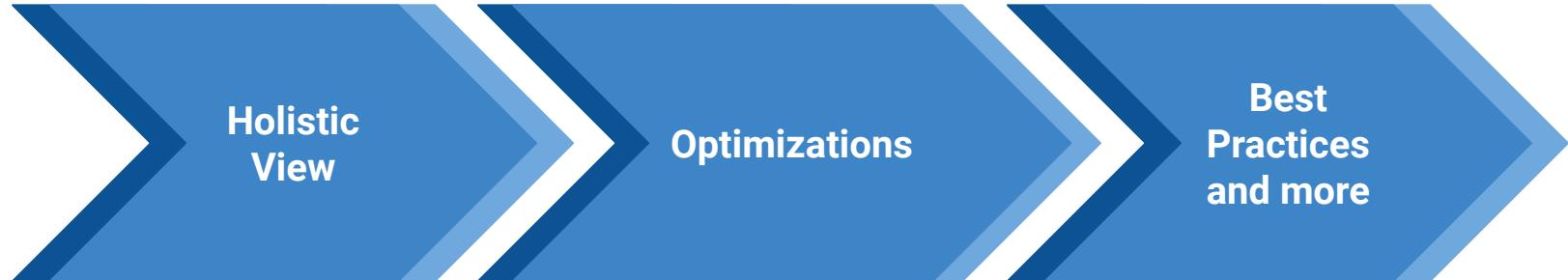
Introductions

- Your Name (How should I call you?)
- Your Role
- Total Years of Experience
- Background in Spark
- What would you like to achieve from this class?

Let's get started..

- Introductions
- Agenda**
- Recap of HDFS, YARN and Spark Architecture
- Scope of Spark Optimization
- About Lab Environment

Agenda



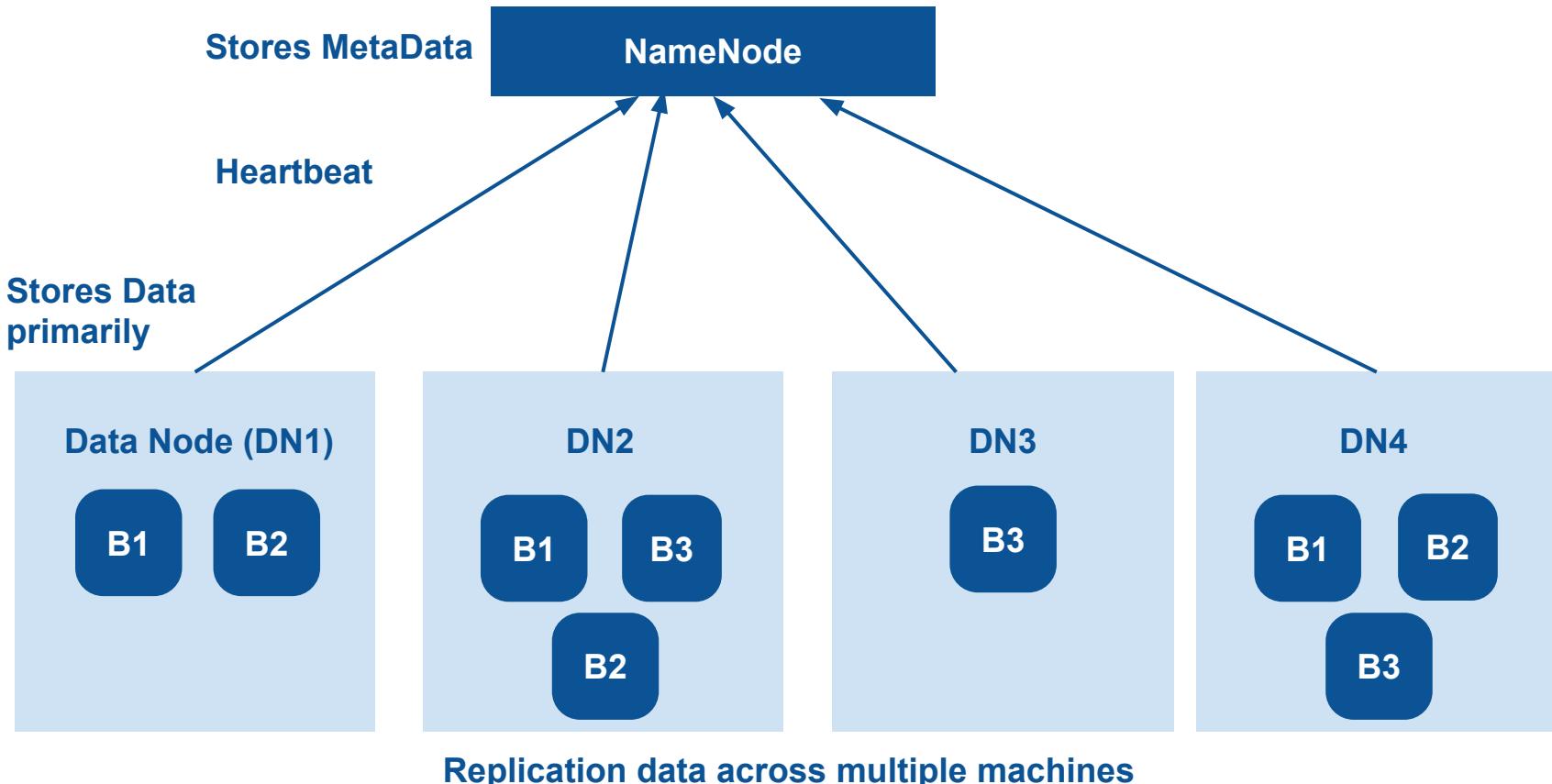
- Scope of Spark Optimization
- Quick Recap of YARN
- Client Mode vs Cluster Mode
- YARN Resource Planning
- Spark Cluster Execution
- Spark Partitions
- Developing High Performance Spark applications
- Debugging/Troubleshooting
- Partitioning
- Caching & Checkpointing
- Spark SQL
- Optimizations
- Joins
- Spark SQL Execution Plans
- Bucketing
- Best Practices

Spark Optimization

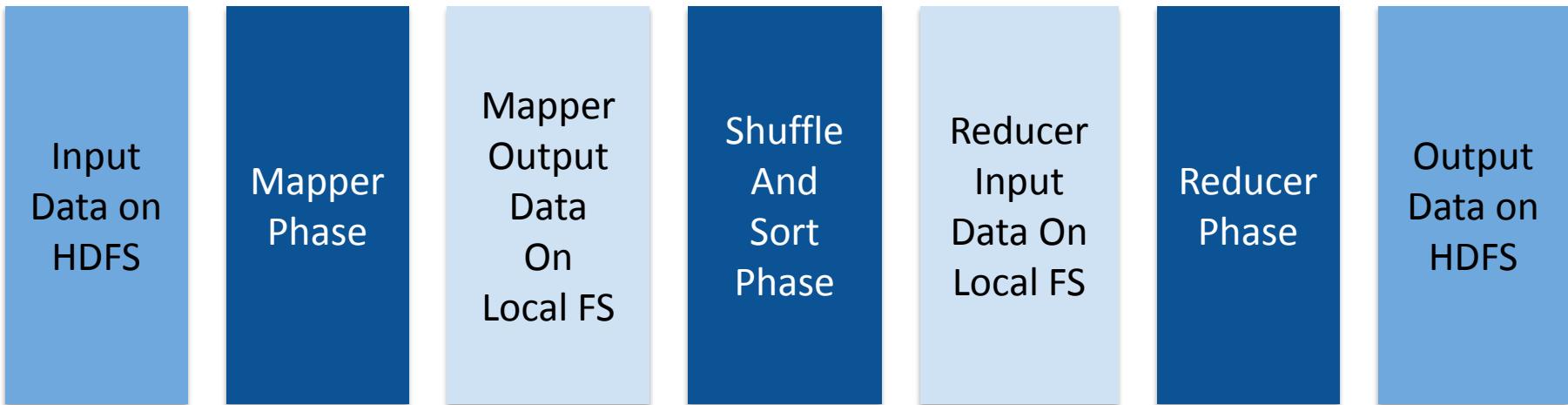
Let's get started..

- Introductions
- Agenda
- Recap of HDFS, YARN and Spark Architecture**
- Scope of Spark Optimization
- About Lab Environment

Physical Architecture of HDFS

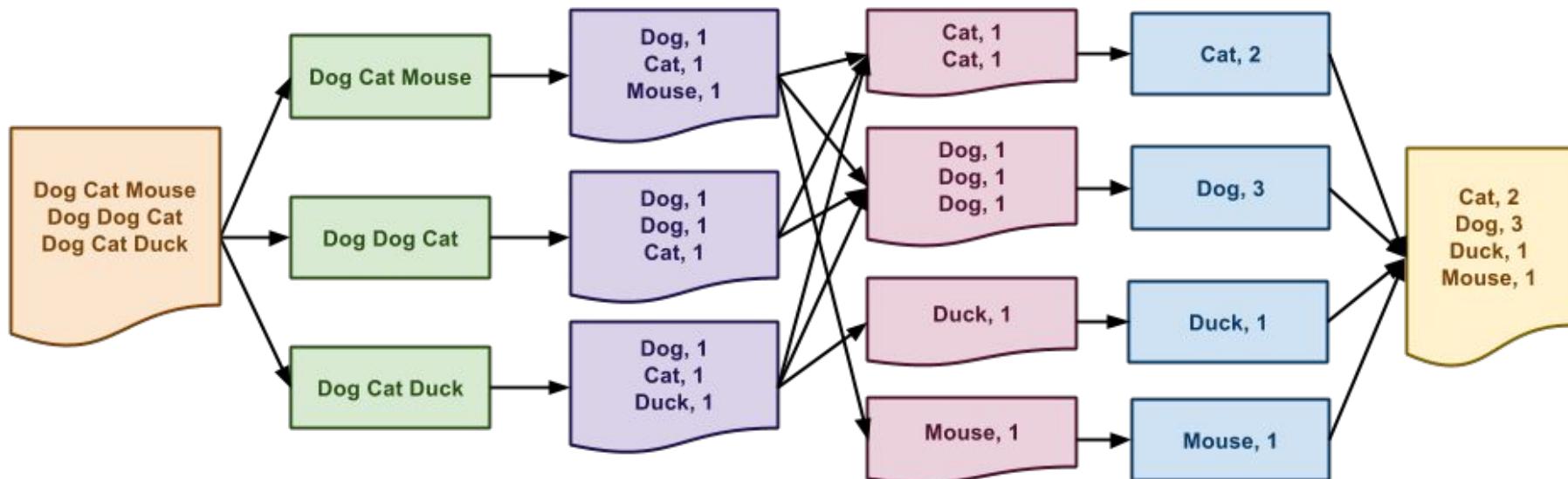


Logical Architecture of MapReduce

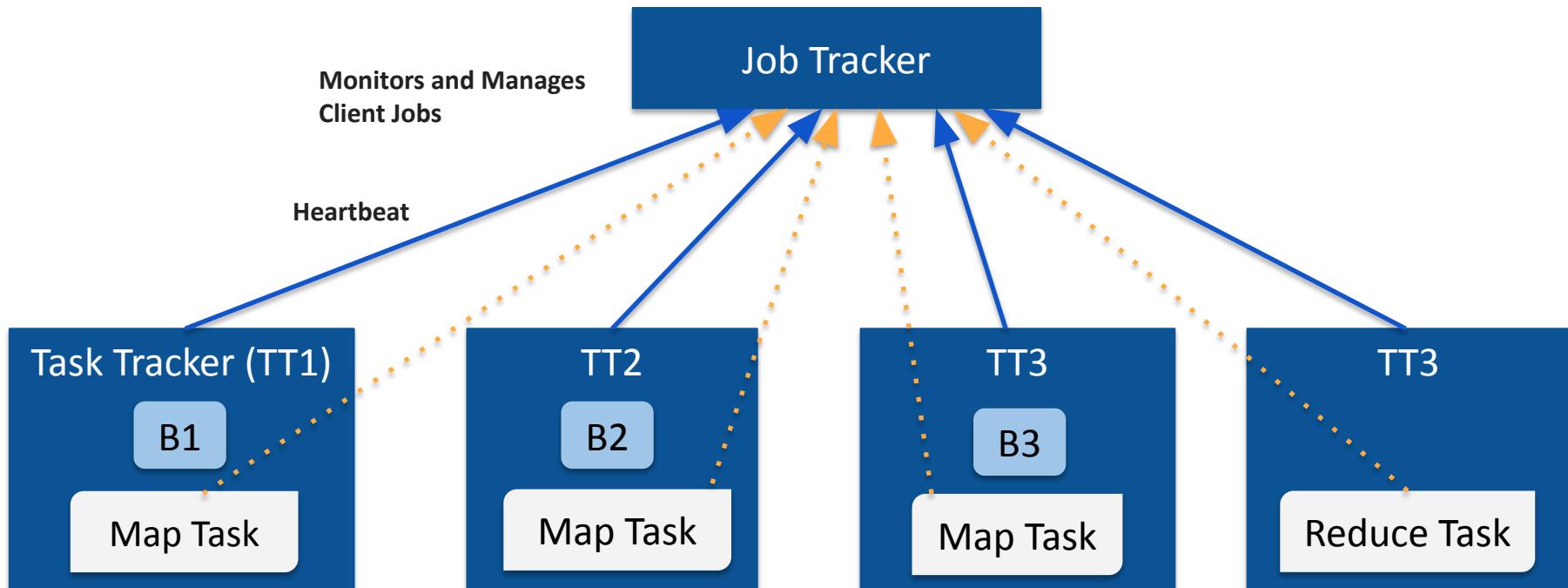


Logical flow of MapReduce

Input Splitting Mapping Shuffling Reducing Merged



Physical Architecture of MapReduce v1



Task Tracker manages Lifecycle of Tasks

Data locality preference is given while execution of Map Tasks

What is YARN?

- YARN stands for “Yet Another Resource Negotiator”
- Responsible for Resource Management of Entire Cluster (Cluster Capacity)
- Acts as Distributed Operating System
- Key Daemons include ResourceManager (RM) and NodeManager (NM)
- Application Master (AM) is just a JVM (not a Daemon)
- AM is terminated once Application finishes
- An application is either a single job or a Directed Acyclic Graph (DAG) of jobs

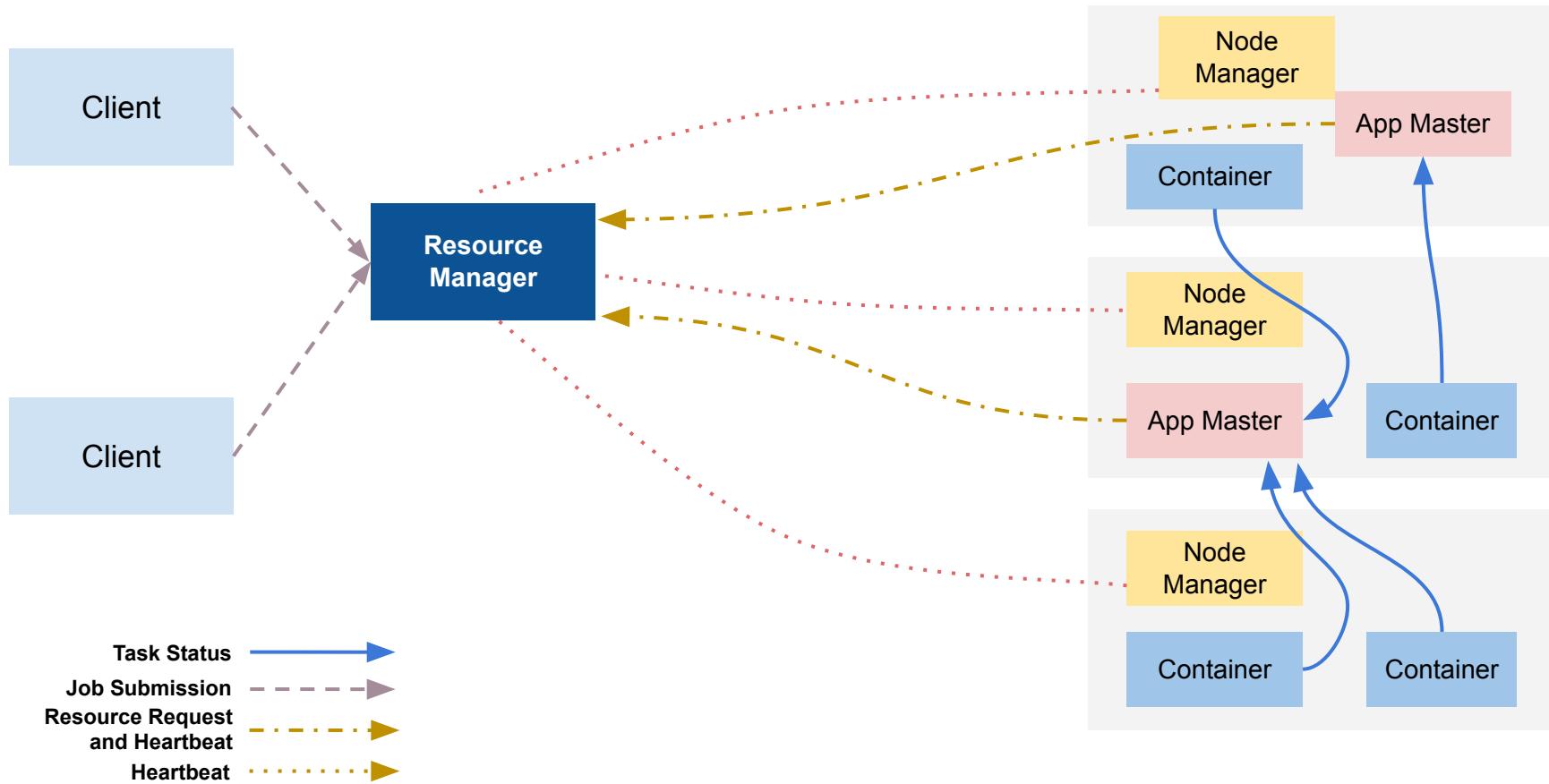


Why YARN?

- Highly Scalable
- Pluggable and Extensible
- Provides Logging (Log File Aggregation), Monitoring, Resource Management etc
- Multiple Types of Applications can execute on YARN e.g. Spark , MR, Tez etc
- YARN Queues enable Administrators to restrict quota to individuals, departments etc
- Centralized Resource Management using User Interface or CLI
- Matured Framework - Already running in many projects in production



YARN Physical Architecture



YARN Daemons

❑ Resource Manager-

- ❑ Responsible for allocation of resources in a cluster
- ❑ Runs on Master Node
- ❑ Monitors Node Managers and Application Masters

❑ Node Manager-

- ❑ Communicates with the Resource Manager
- ❑ Runs on Worker node
- ❑ Per-machine framework daemon who is responsible for containers, monitoring their resource usage and reporting the same to the ResourceManager

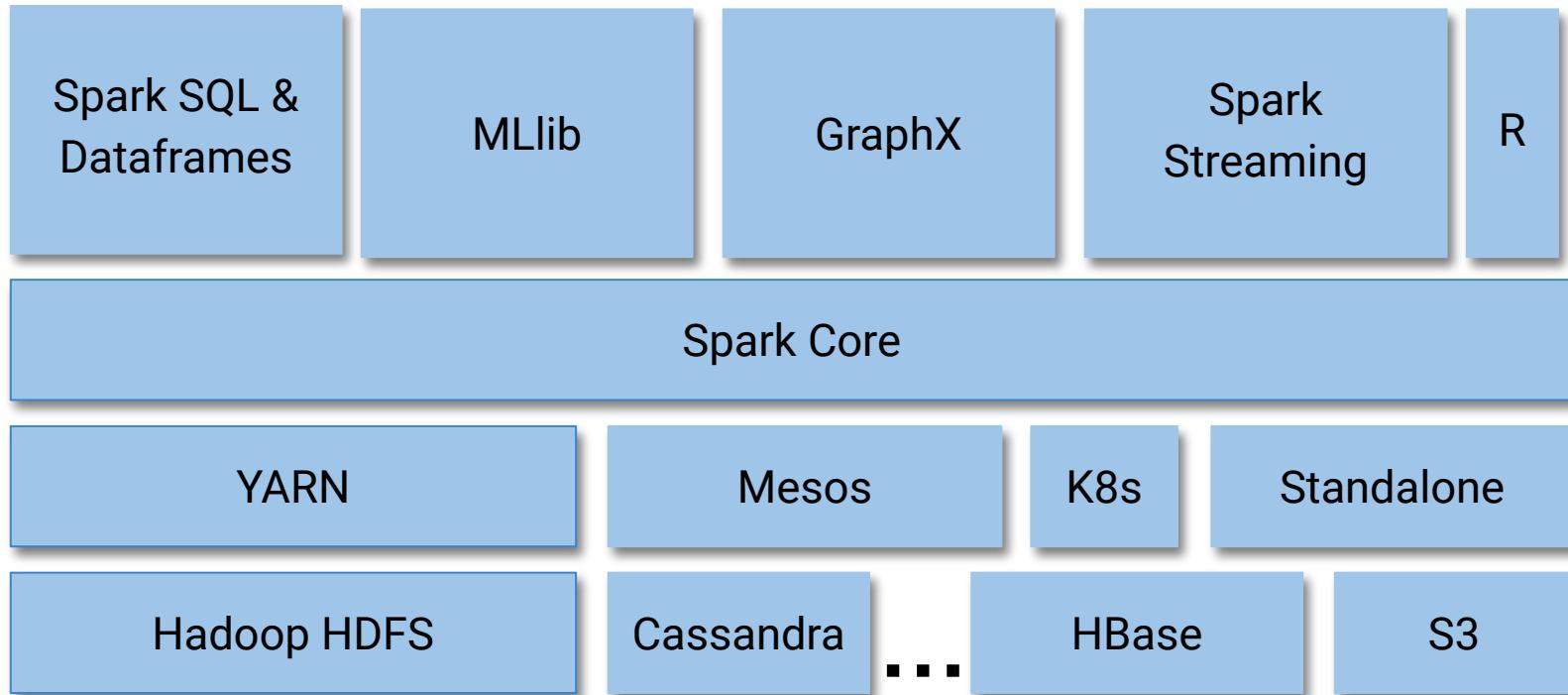
- ❑ **Container-**

- ❑ Created by Resource Manager upon request
 - ❑ Allocates certain amount of resources (CPU, memory) on Worker Node
 - ❑ Applications can run on one or more containers

- ❑ **Application Master-**

- ❑ One Application Master per application
 - ❑ Coordinates the running task scheduled and managed by RM
 - ❑ Runs in a container

Logical Architecture of Spark



Let's get started..

- Introductions
- Agenda
- Recap of HDFS, YARN and Spark Architecture
- Scope of Spark Optimization**
- About Lab Environment

Scope of Spark Optimization

Spark Core & Spark SQL

Distributed OS e.g. YARN

Distributed File System e.g. HDFS

Java

Linux Based OS (ext4 or xfs FileSystem)

Hardware (if hosted on on-prem)

Let's get started..

- Introductions
- Agenda
- Recap of HDFS, YARN and Spark Architecture
- Scope of Spark Optimization
- About Lab Environment**

About Lab Environment

- ❑ Labs are hosted on AWS
- ❑ Lab Machine is a Cloudera Pseudo distributed mode cluster
- ❑ Each Machine is of type having 32 GB RAM and 8 vCores
- ❑ Instructor will be providing an IP to each participant to perform RDP
- ❑ Provide your machine IP so that Support Team can configure your IP in firewall rules to provide you access to the labs

Spark Optimization

Let's get started..

- Introductions
- Agenda
- Recap of HDFS, YARN and Spark Architecture
- Scope of Spark Optimization
- About Lab Environment



Holistic View

Agenda:

- ❑ Spark Physical Execution
- ❑ Clustering with Spark

Chapter Outline

- ❑ Partitions
- ❑ Executors
- ❑ Key Execution Modes
- ❑ Parallel Operations
- ❑ Tasks and Stages
- ❑ Narrow vs Wide Dependency
- ❑ Spark UI
- ❑ Executor Memory Architecture
- ❑ Key Properties
- ❑ Spark on YARN Detailed Architecture
- ❑ Resource Planning
- ❑ Discussion on Garbage Collection

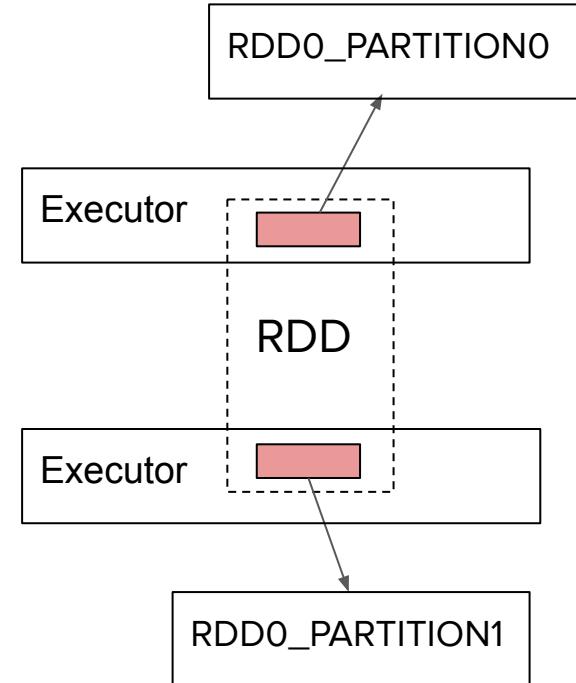
Spark Physical Execution

Partitions

- ❑ Partition is for tweaking degree of parallelism in Spark
- ❑ Suggestion not an instruction to Spark framework
- ❑ **sc.textFile({path to file/directory},n)**
 - ❑ First argument - path to file/directory,
 - ❑ Second argument - number of partitions

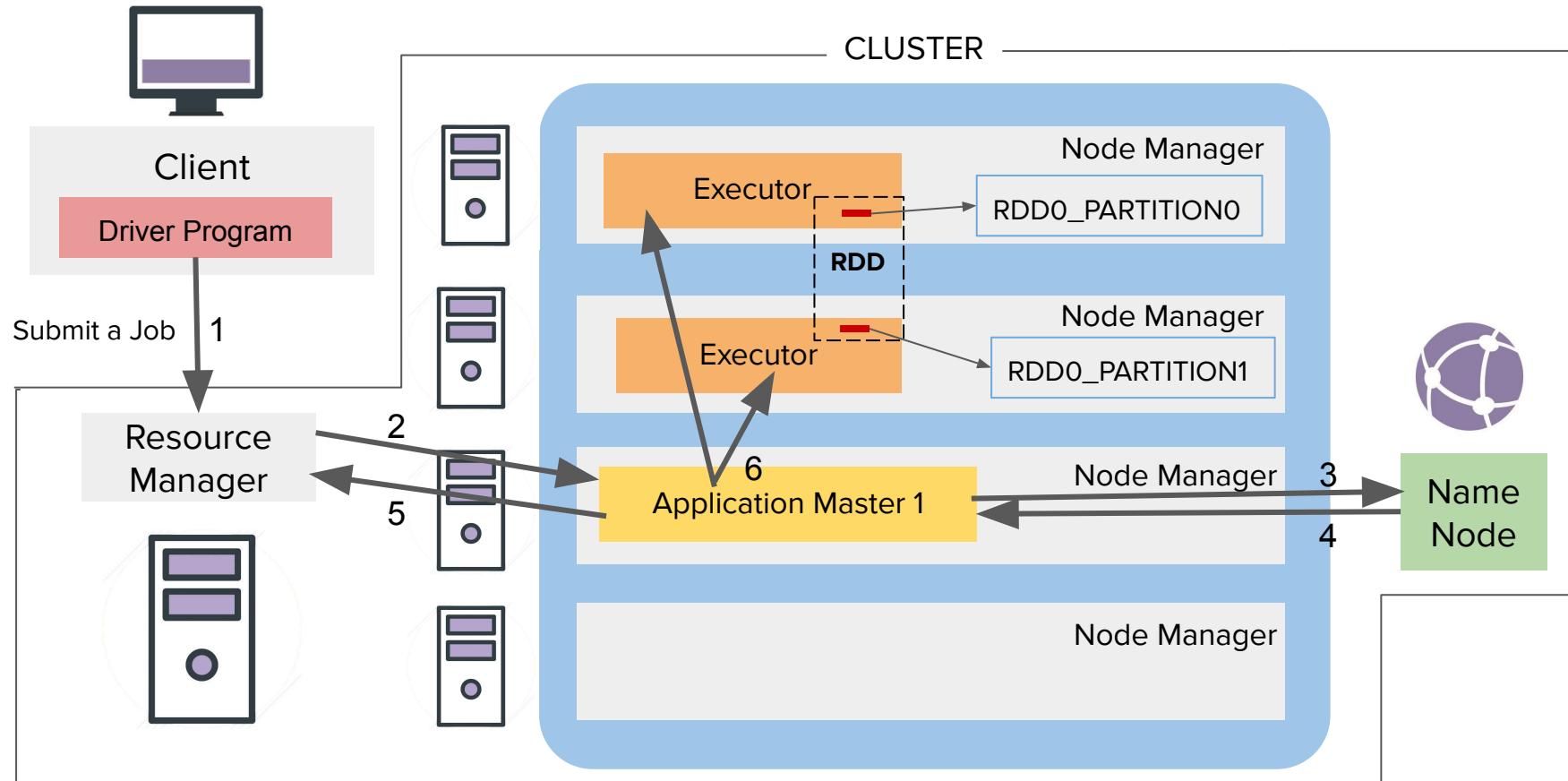
```
rdd0=sc.textFile({file/directory},2)
```

Number of partitions



Spark Physical Execution

Partitions



Things to keep in mind while determining Partition

- ❑ Apache Spark can only run a single concurrent task for every partition of an RDD, up to the number of cores in your cluster (and probably 2-3x times that)
- ❑ For choosing a “good” number of partitions, you generally want at least as many as the number of executors for parallelism
- ❑ You can check this value by calling `sc.defaultParallelism`

Partitions

- ❑ **General Rules:** When Number of Partitions > 100 & < 10000

1

Lower Bound

2 x Number of Cores in Cluster

For e.g.:

Core Cluster= 100 Core

=> Partitions= $2 \times 100 = 200$ Partitions

2

Upper Bound

Each Task shouldn't take less

Than 100ms to execute

=> Partitioned Data is too small

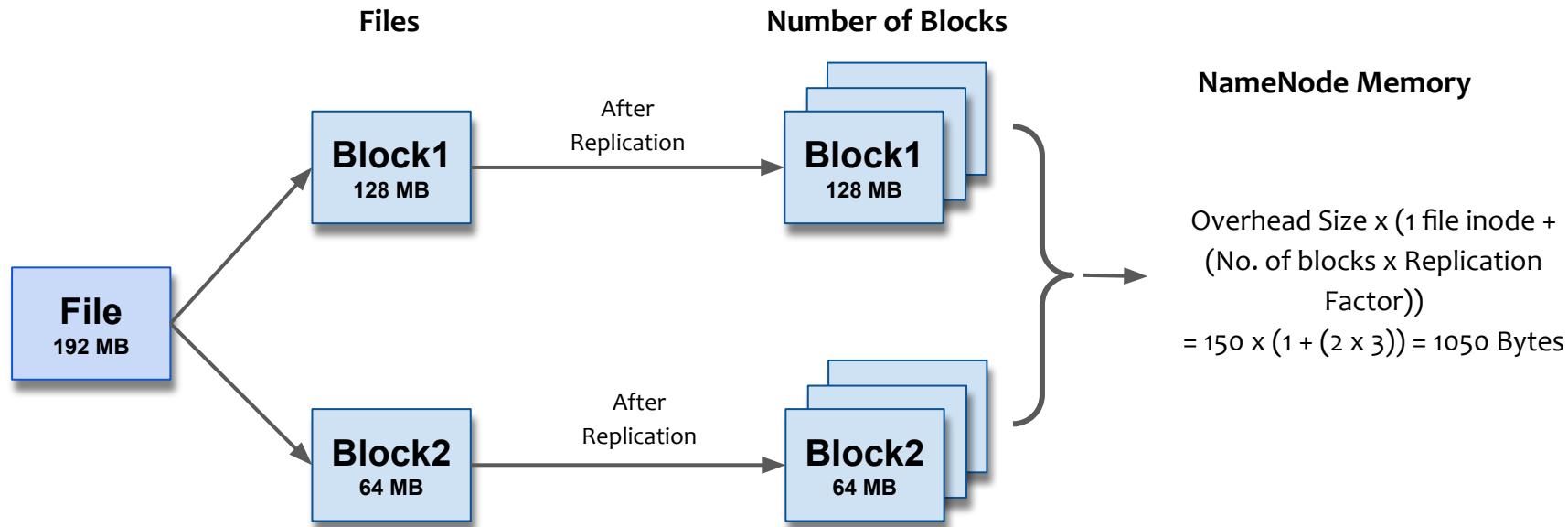
Small Partitions Problem

- ❑ Small Files are a common challenge in the Apache Spark.
- ❑ Small File problem leads to inefficient
 - ❑ Namenode memory utilization
 - ❑ block scanning throughput degradation
 - ❑ application layer performance

Partitions

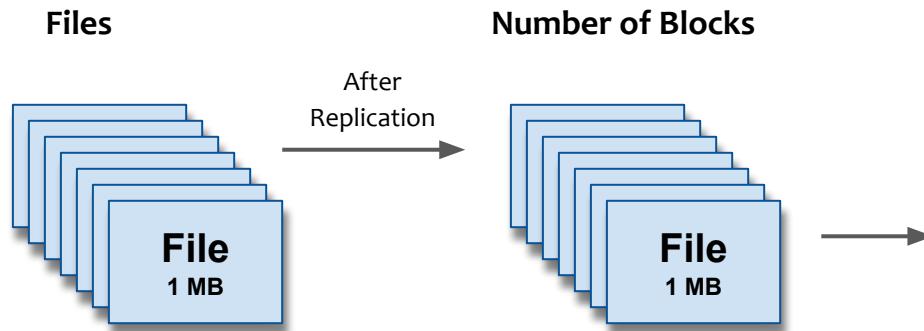
The two scenarios illustrate the small files issue in HDFS:

Scenario 1 (1 large file of 192 MB):



Partitions

Scenario 2 (192 small files, 1MiB each):



Name Node Memory

Overhead Size x (1 file inode + (No. of blocks x
Replication Factor))
 $= 150 \times (192 + (192 \times 3)) = 115200 \text{ Bytes}$

Partitions

- ❑ Scenario 2 has 192 1 MB files. These files are then replicated across the cluster. The total memory required by the Namenode to store the metadata of these
- ❑ $\text{files} = 150 \times (192 + (192 \times 3)) = 115200 \text{ Bytes}$
- ❑ Hence, we can see that we require more than 100x memory on the Namenode heap to store the multiple small files as opposed to one big 192MB file

Partitions

Diagnosing & Handling Post Filtering Issues (Skewness)

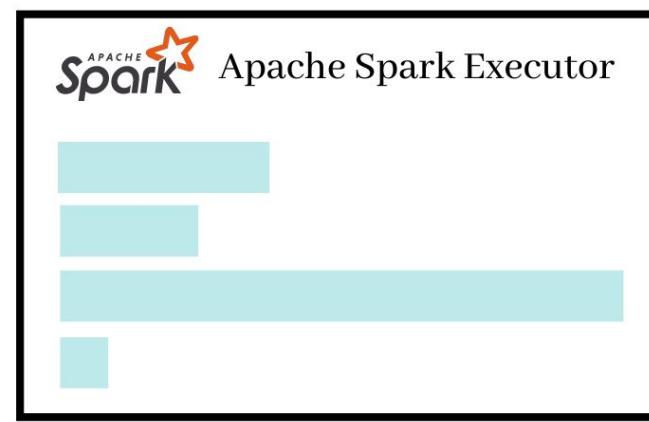
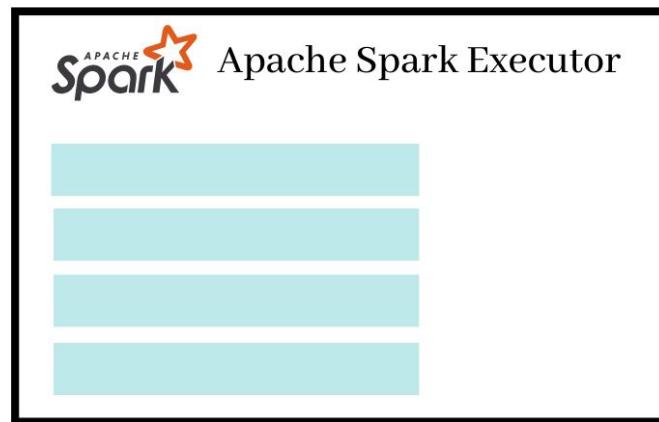
In Apache Spark, data skewness is caused by transformations that change data partitioning like join, groupBy, and orderBy

For example:

Joining on a key that is not evenly distributed across the cluster, causing some partitions to be very large and not allowing Spark to process data in parallel

Partitions

Even Distribution vs Distribution with Skew



Partitions

repartition()

- The **repartition** method can be **used** to either increase or decrease the number of partitions in a DataFrame
- **Repartition** is a full Shuffle operation, whole data is taken out from existing partitions and equally distributed into newly formed partitions

coalesce()

- The **coalesce** method reduces the number of partitions in a DataFrame
- **Coalesce** avoids full shuffle, instead of creating new partitions, it shuffles the data using Hash Partitioner (Default), and adjusts into existing partitions

Partitions

❑ repartition() vs coalesce()

```
val rdd1 = spark.sparkContext.parallelize(Range(0,25), 6)
println("parallelize : "+rdd1.partitions.size)

val rdd2 = rdd1.repartition(4)
println("Repartition size : "+rdd2.partitions.size)
rdd2.saveAsTextFile("/tmp/re-partition")
```

```
Partition 1 : 1 6 10 15 19
Partition 2 : 2 3 7 11 16
Partition 3 : 4 8 12 13 17
Partition 4 : 0 5 9 14 18
```

```
val rdd3 = rdd1.coalesce(4)
println("Repartition size : "+rdd3.partitions.size)
rdd3.saveAsTextFile("/tmp/coalesce")
```

```
Partition 1 : 0 1 2
Partition 2 : 3 4 5 6 7 8 9
Partition 4 : 10 11 12
Partition 5 : 13 14 15 16 17 18 19
```

Partitions

- ❑ RDD operations work on each element of an RDD
- ❑ Few operations work on each Partition
 - ❑ **mapPartitions**- Similar to map, but runs separately on each partition (block) of the RDD
 - ❑ **mapPartitionsWithIndex**- Similar to mapPartitions, but also provides function with an integer value representing the index of the partition
 - ❑ **foreachPartition**- call a function for each partition

Partitions

Example:

Python

```
def printLine(iter):  
    print iter.next()  
  
data= ...  
data.foreachPartition(lambda x : printLine(x))
```

Important Notes about Partitions

❑ Partitions during Reading data

- ❑ Size of files/data can be controlled in the Spark job input

By assuming source has sufficient partitions

`spark.sql.files.maxPartitionBytes - 128 MB (default)`

❑ Partitions during Shuffling data

- ❑ Determines how much degree of parallelism required in the subsequent stage (stage to be discussed soon)

Eg: `spark.sql.shuffle.partitions`

❑ Partitions during Output data

- ❑ Size, composition and Number of output files can be controlled

Eg: Coalesce(n) to shrink

`df.write.option("maxRecordsPerFile",N)`

Partitions

- Partitions
- Executors**
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

Executors

- A JVM process launched for an application on a worker node. It executes program and keeps data in memory. Each application has its own executors

The screenshot shows a web browser window titled "PySparkShell - Executor..." with the URL "localhost:4040/executors/". The browser's address bar also displays "localhost:4040/executors/". The page header includes the Spark logo and navigation links: Jobs, Slaves, Storage, Environment, and Executors (which is highlighted with a purple box). Below the header, the title "Executors (1)" is displayed. A summary section shows "Memory: 0.0 B Used (267.3 MB Total)" and "Disk: 0.0 B Used". The main content is a table with the following data:

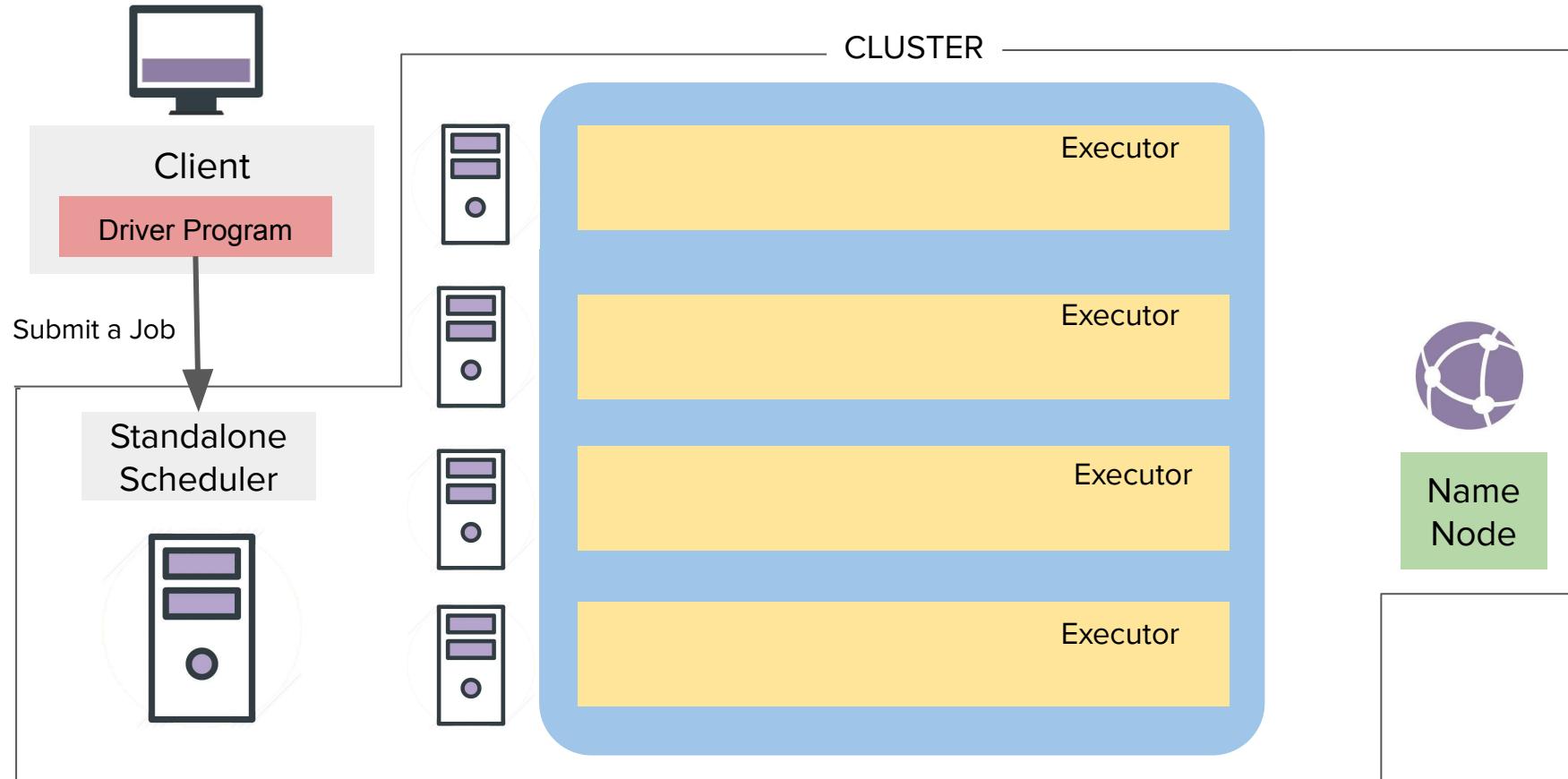
Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Thread Dump
<driver>	localhost:6906	0	0.0 B / 267.3 MB	0.0 B	8	2	10	12	1.2 s	10.0 KB	0.0 B	0.0 B	Thread Dump

Chapter Outline

- Partitions
- Executors
- Key Execution Modes**
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

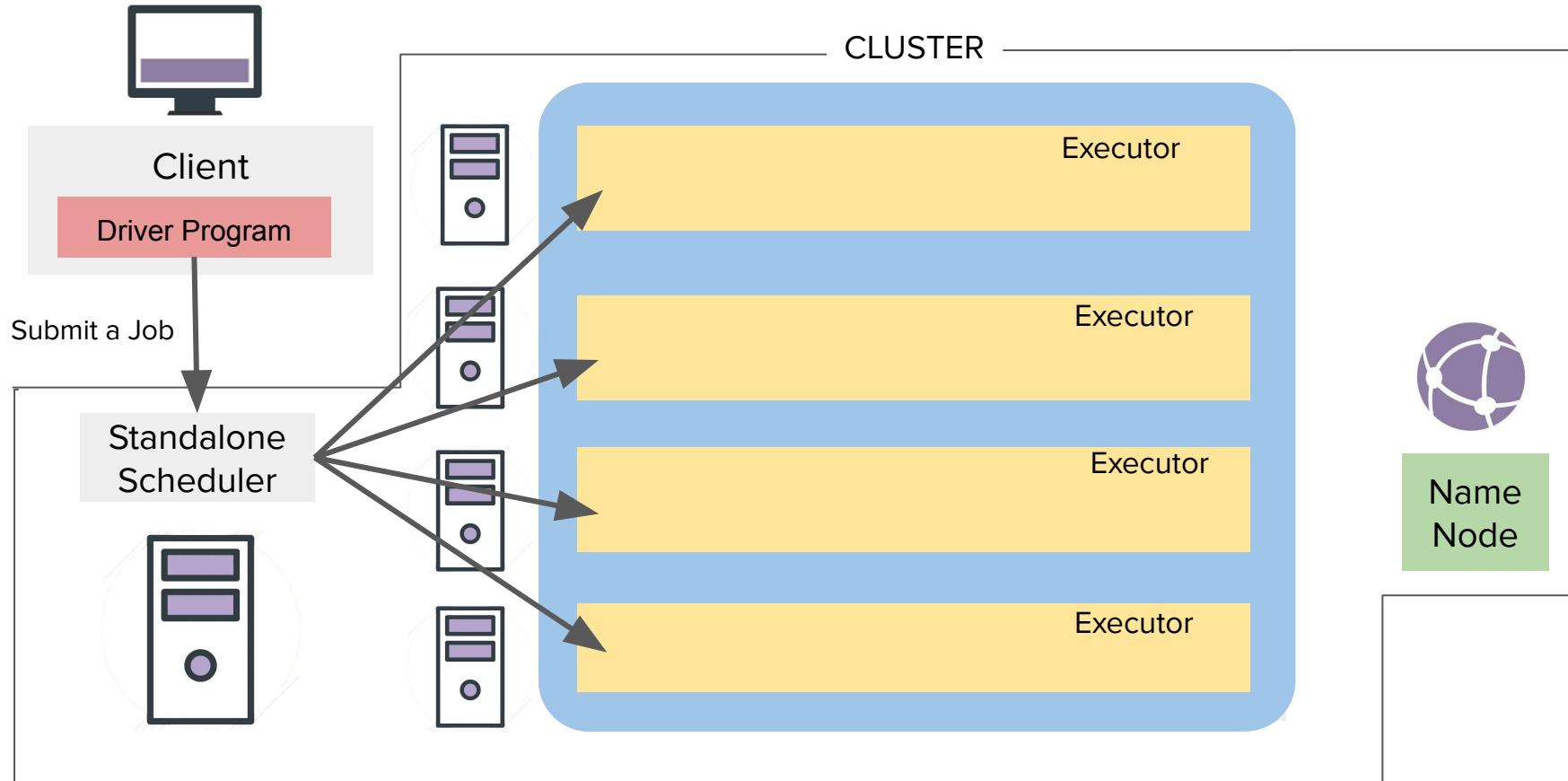
Spark Physical Execution

Standalone Mode



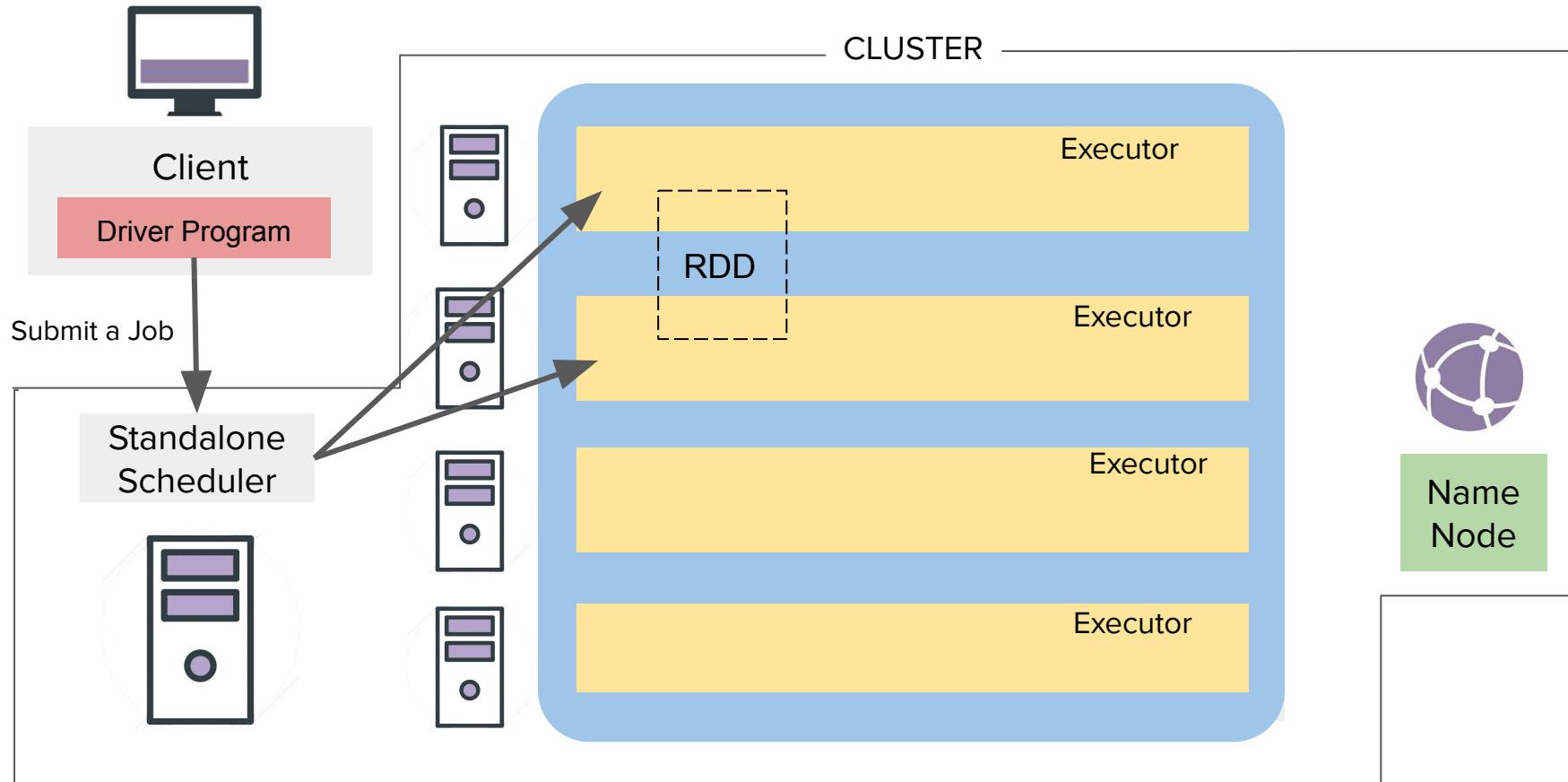
Spark Physical Execution

Standalone Mode



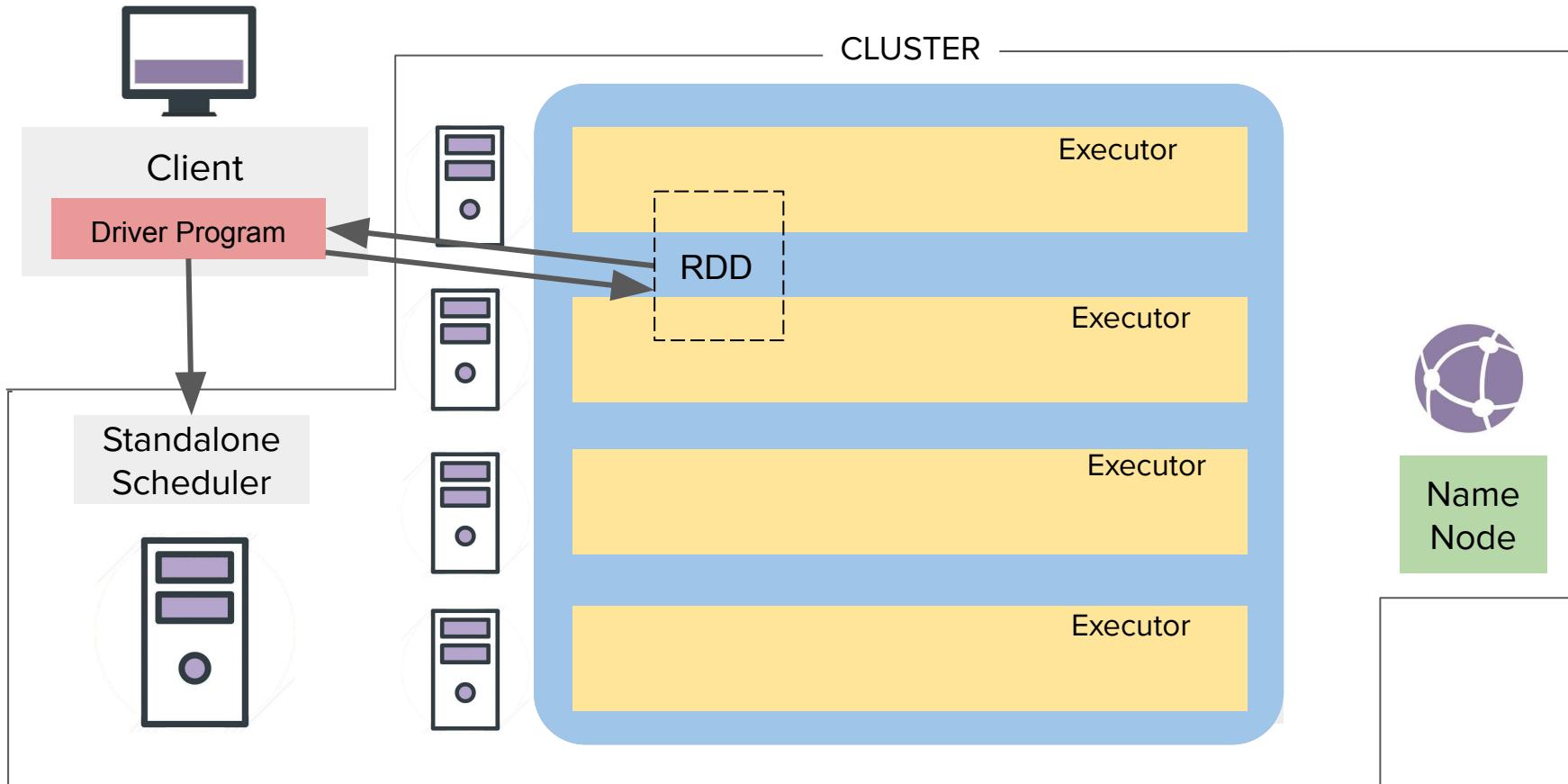
Spark Physical Execution

Standalone Mode



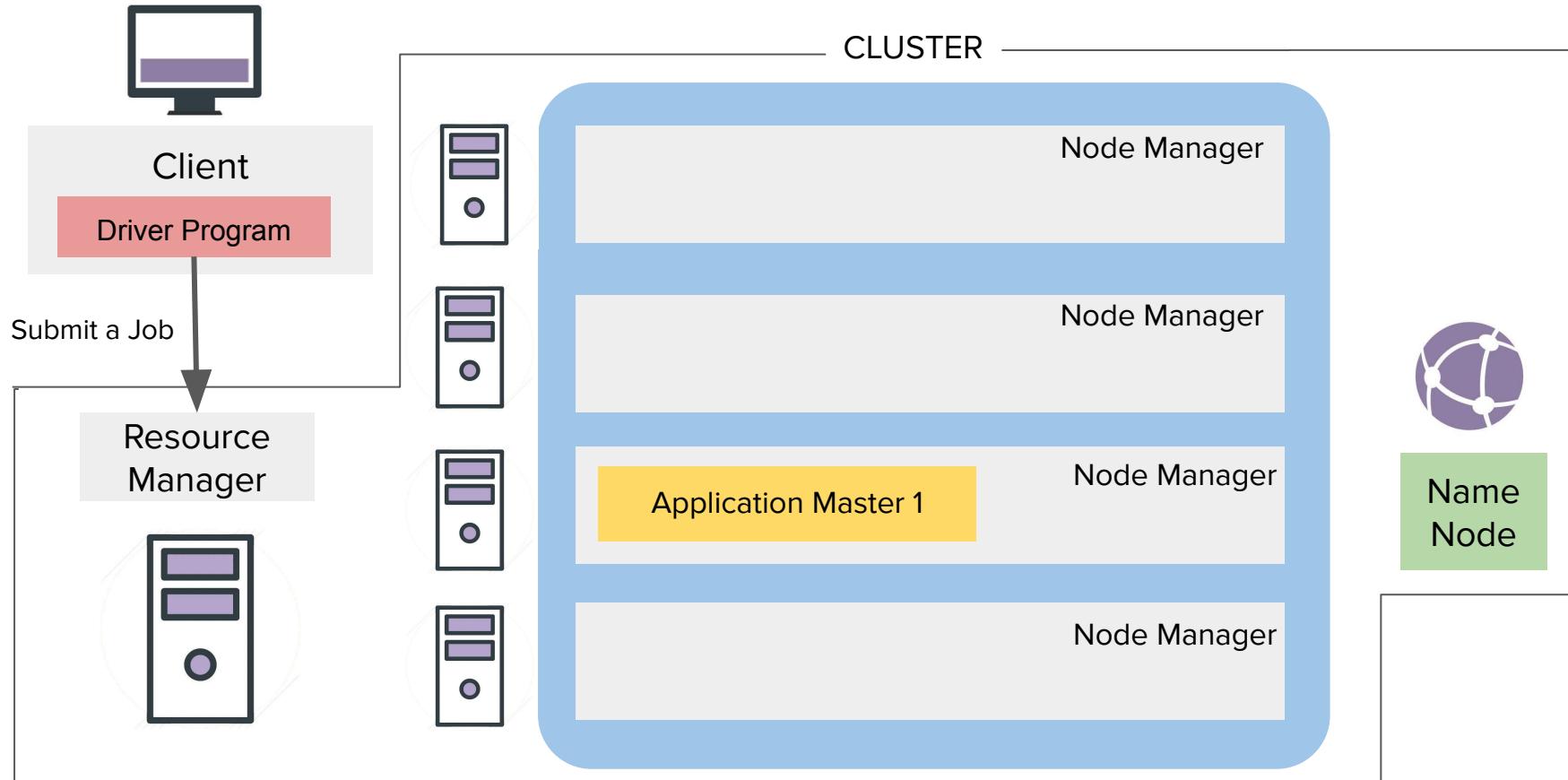
Spark Physical Execution

Standalone Mode



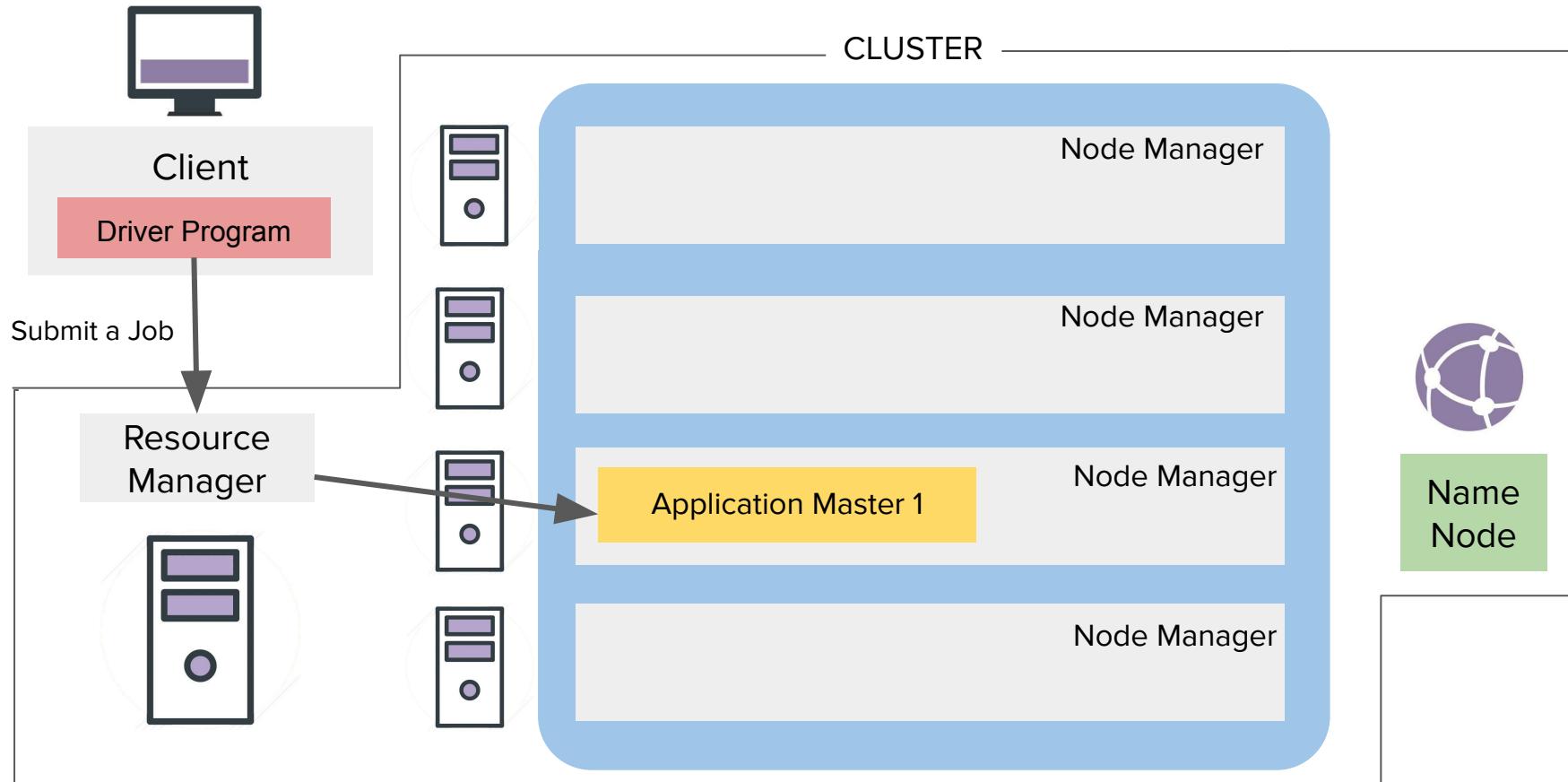
Spark Physical Execution

YARN-Client Mode



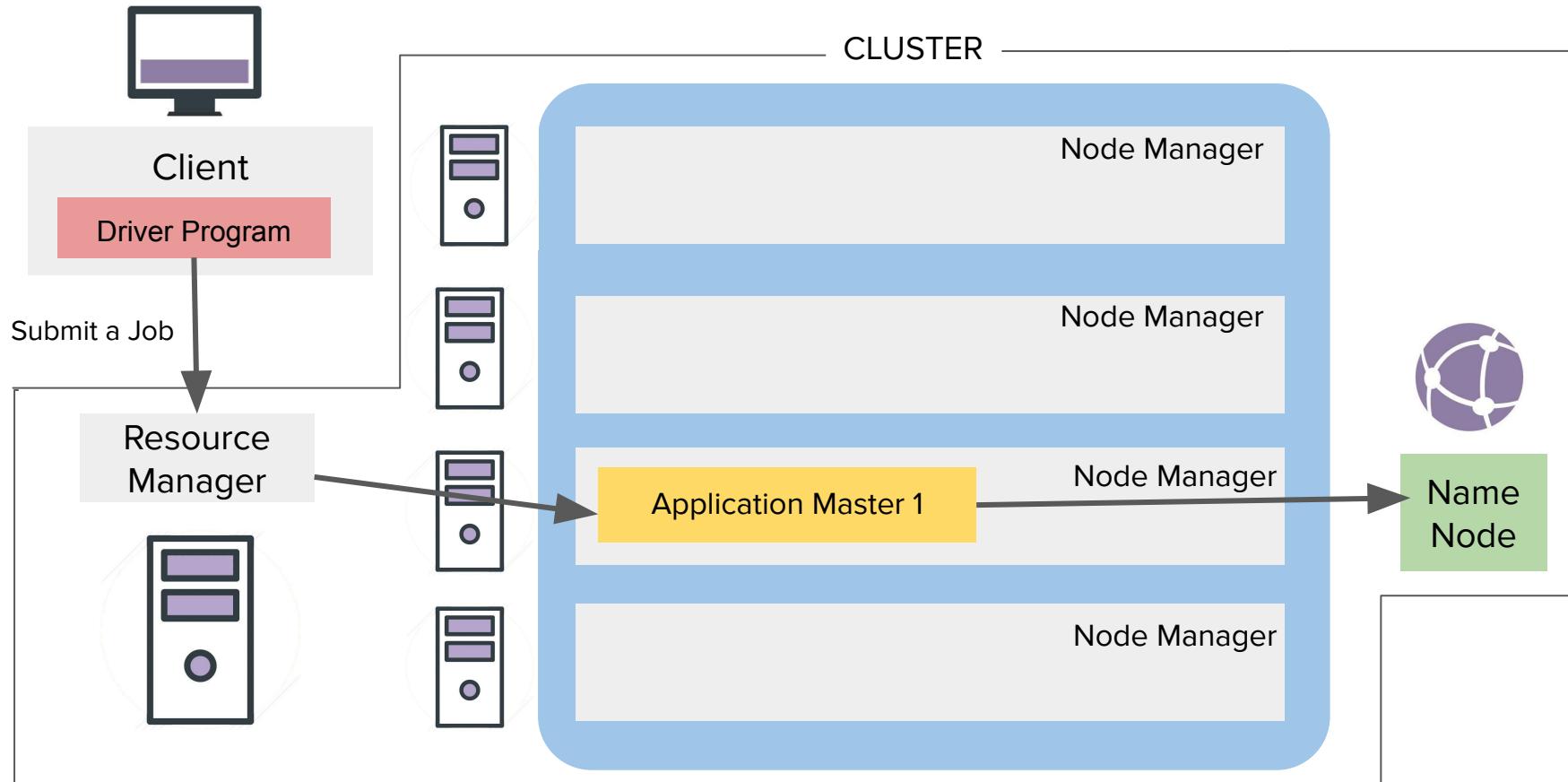
Spark Physical Execution

YARN-Client Mode



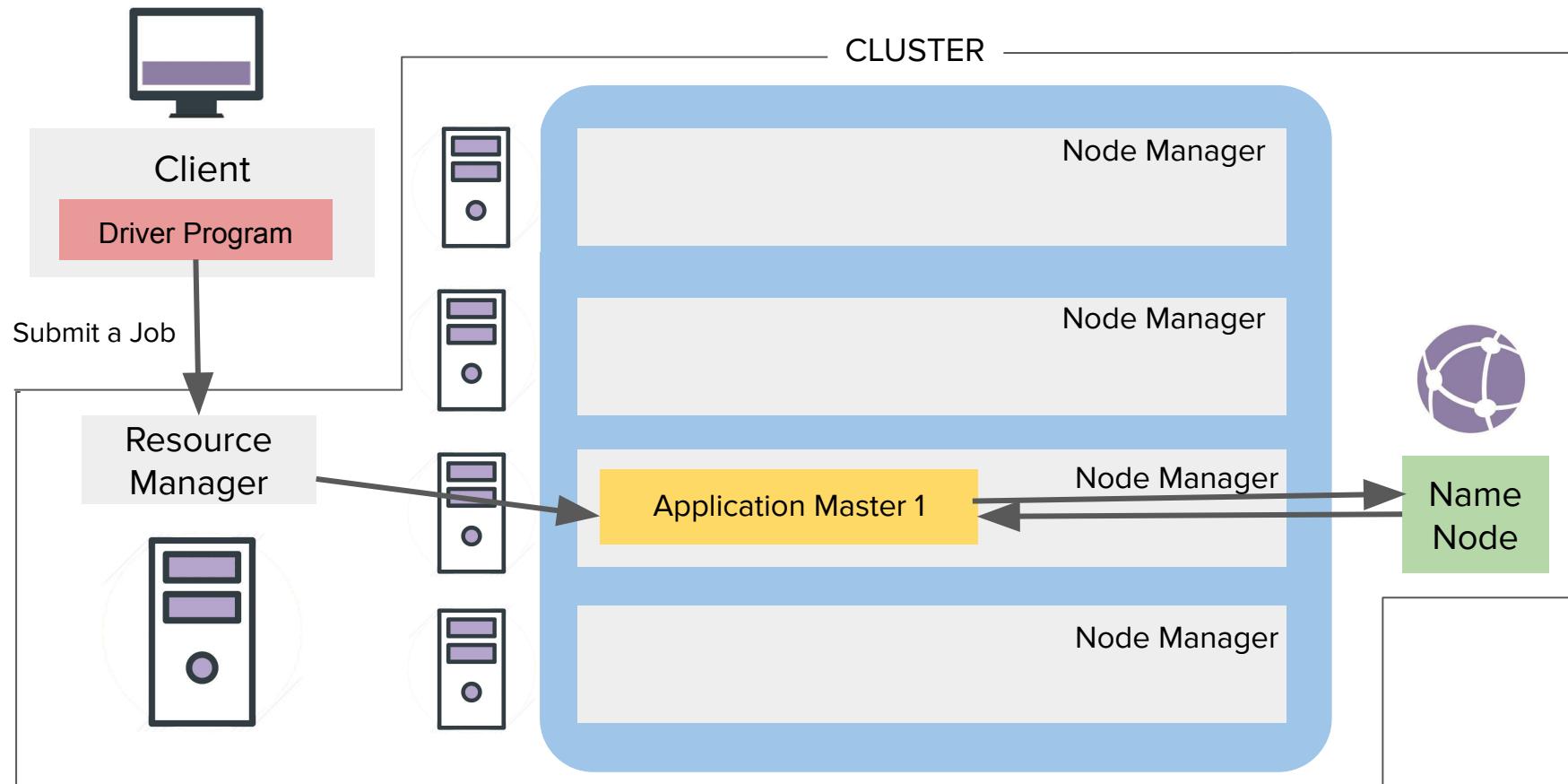
Spark Physical Execution

YARN-Client Mode



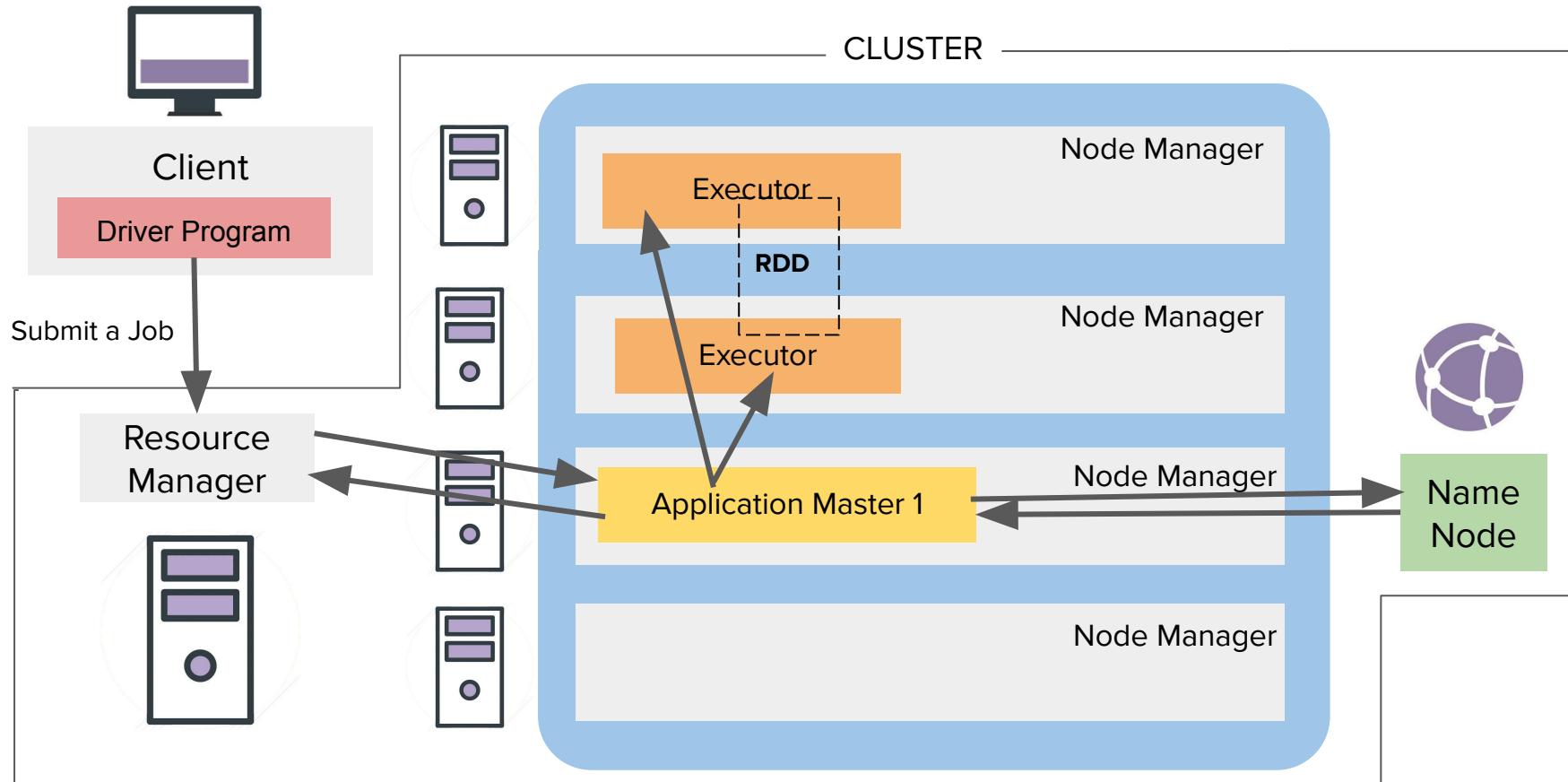
Spark Physical Execution

YARN-Client Mode



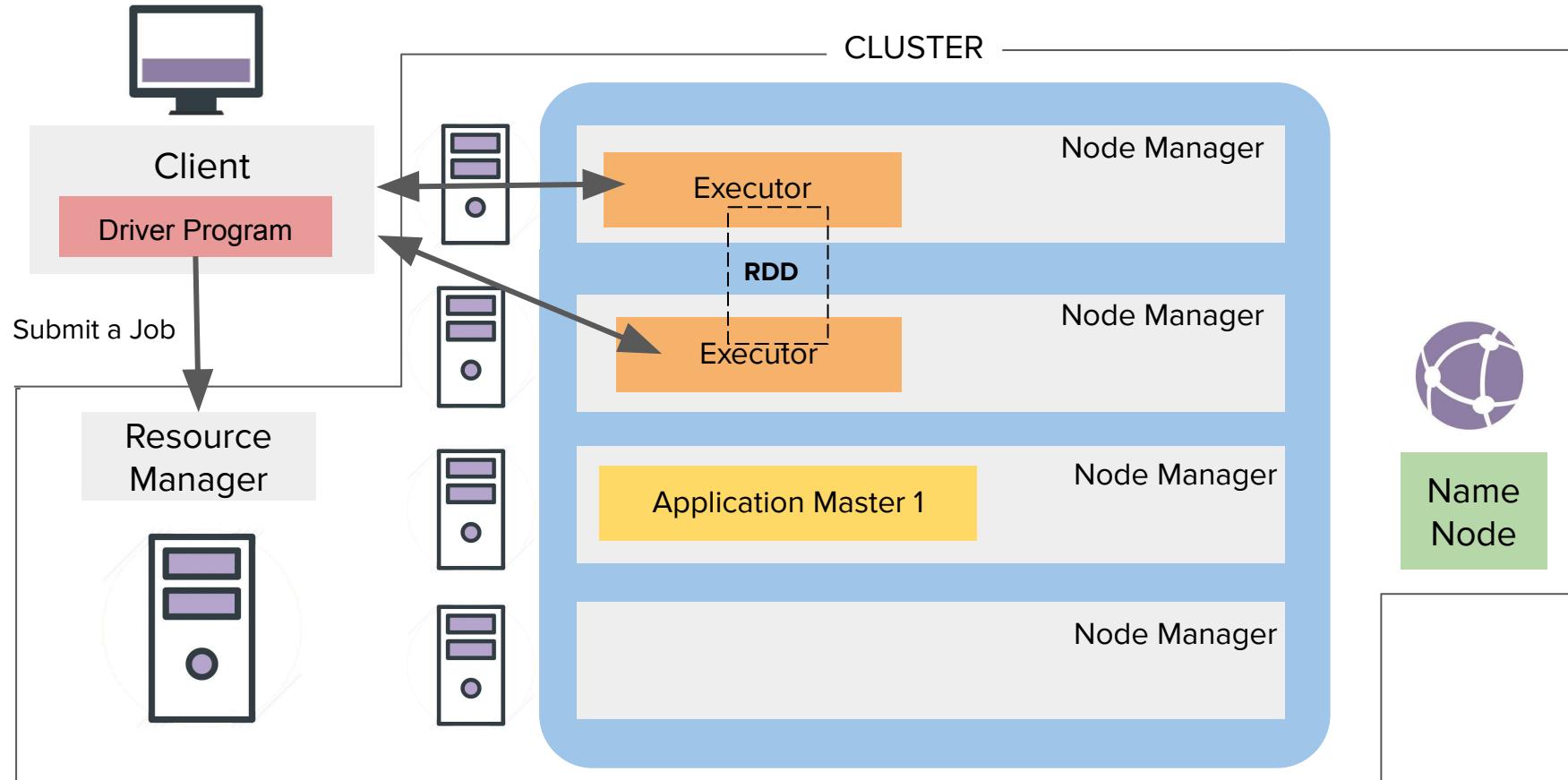
Spark Physical Execution

YARN-Client Mode



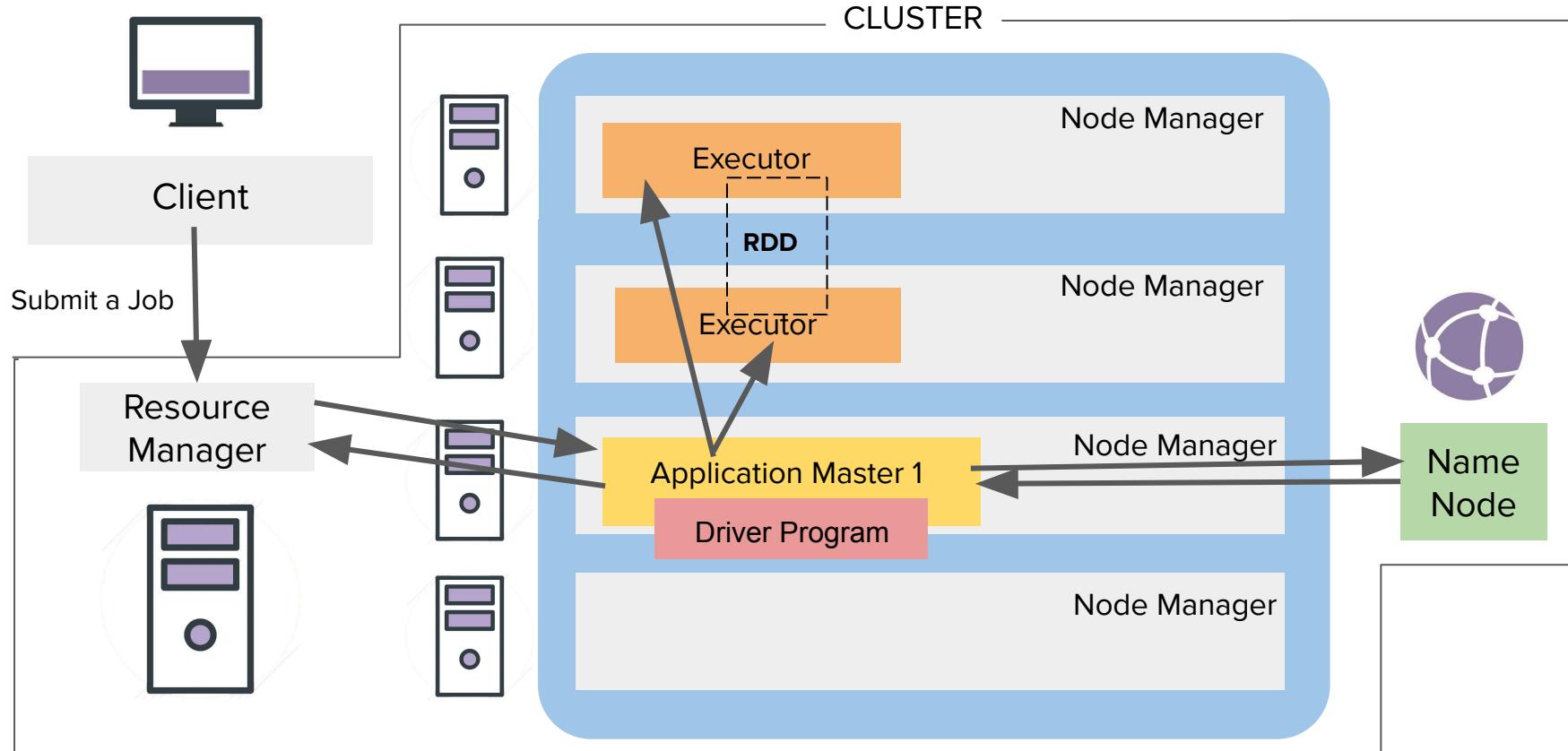
Spark Physical Execution

YARN-Client Mode



Spark Physical Execution

YARN-Cluster Mode



Chapter Outline

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations**
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

Parallel Operations

- ❑ RDD operations are executed parallelly
 - ❑ Various operations like map, flatmap, filter preserve partition
 - ❑ Operations like groupByKey, reduceByKey that do sorting and shuffling perform repartitioning

Name	Age	Country
Mark	27	US
Simon	32	Netherlands
Joshua	21	US
Paul	45	US
Andrew	31	Netherlands

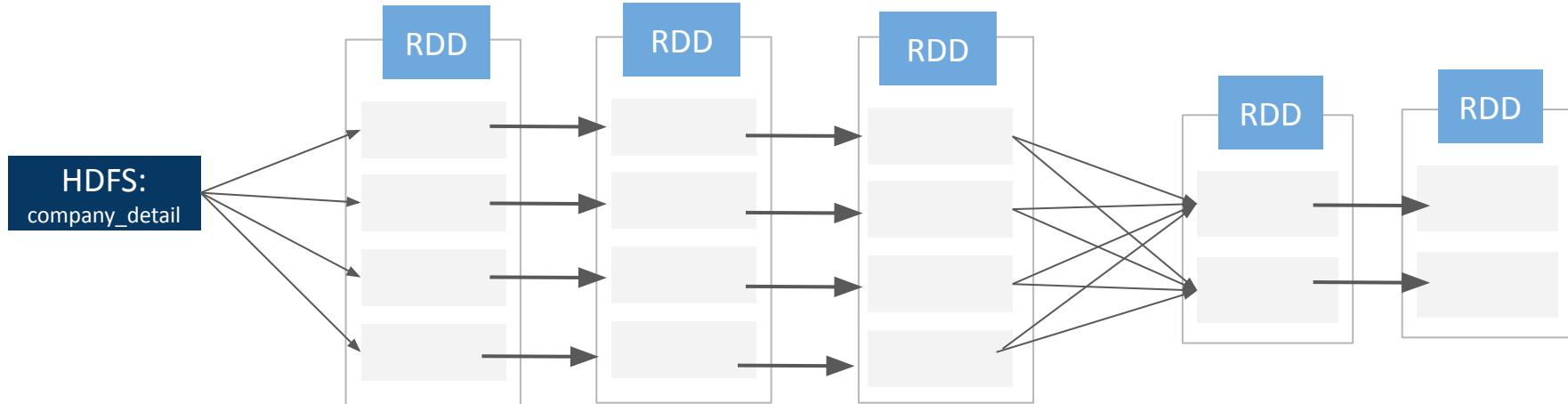
company_detail.txt

Parallel Operations

- Example : `groupByKey()` operation

```
>>>data= sc.textFile("company_detail.txt")
>>>data_split=data.map(lambda line: line.split(','))
>>>data_fields=data_split.map(lambda fields: (fields[2],fields[1]))
>>>data_group=data_fields.groupByKey(2)
>>>data_group.mapValues(lambda x: sum(x)/len(x))
```

Python

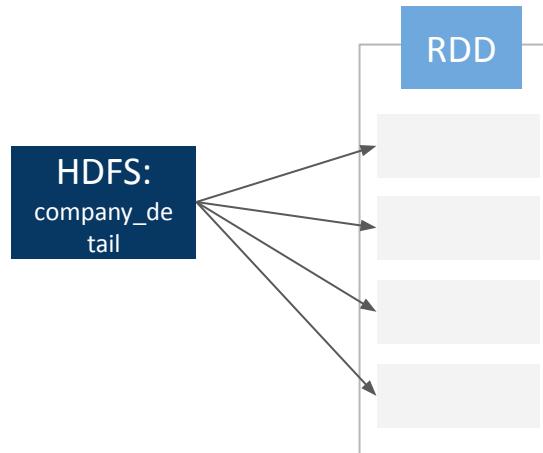


Parallel Operations

- Example : `groupByKey()` operation

```
>>>data= sc.textFile("company_detail.txt")
```

Python

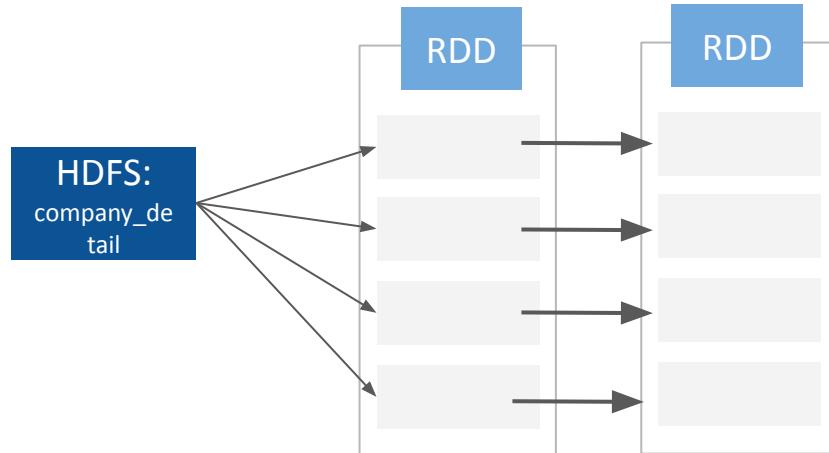


Parallel Operations

- Example : `groupByKey()` operation

```
>>>data= sc.textFile("company_detail.txt")
>>>data_split=data.map(lambda line: line.split(' '))
```

Python

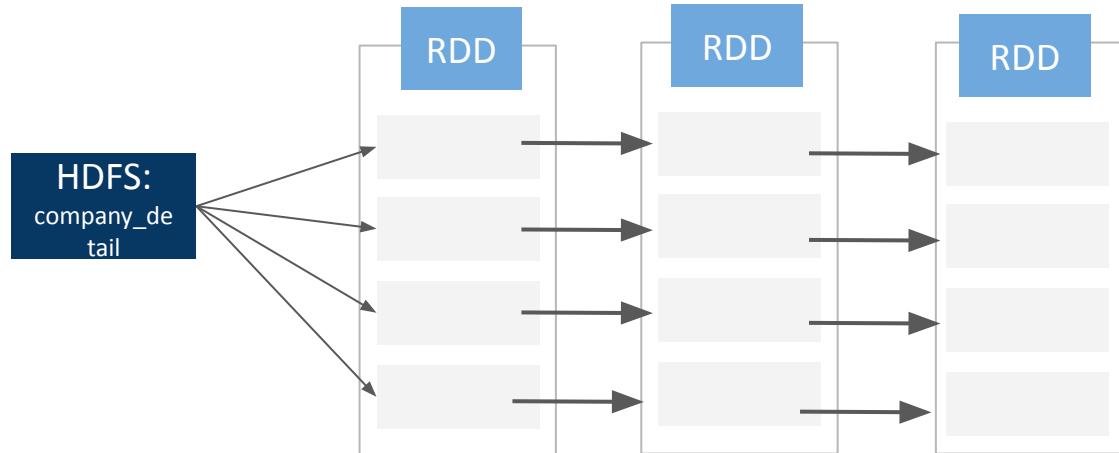


Parallel Operations

- Example : `groupByKey()` operation

```
>>>data= sc.textFile("company_detail.txt")
>>>data_split=data.map(lambda line: line.split(' '))
>>>data_fields=data_split.map(lambda fields: (fields[2],fields[1]))
```

Python

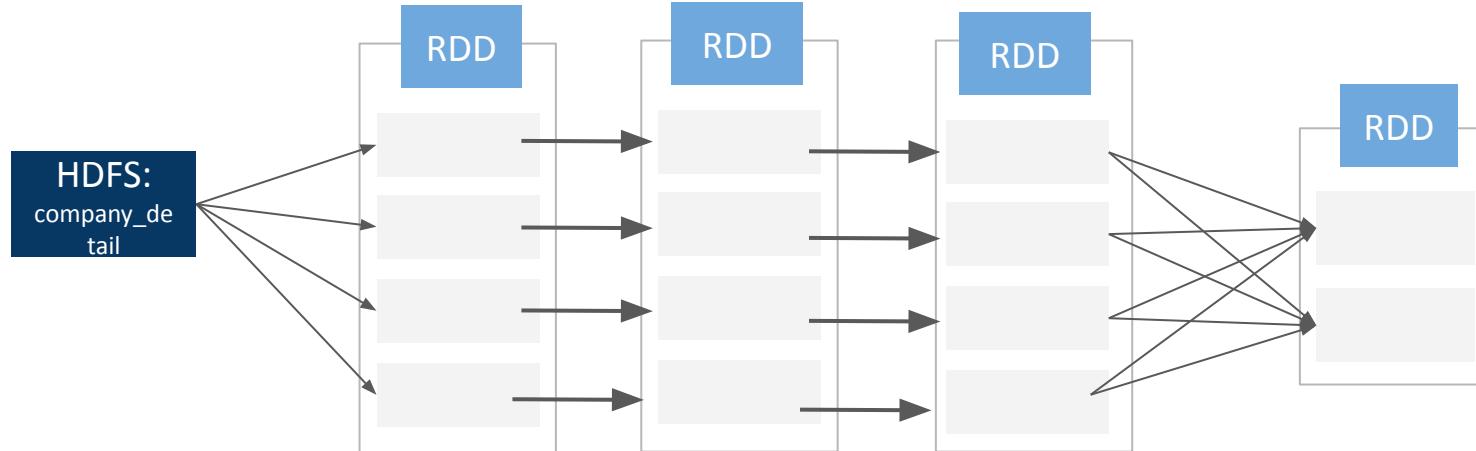


Parallel Operations

- Example : `groupByKey()` operation

```
>>>data= sc.textFile("company_detail.txt")
>>>data_split=data.map(lambda line: line.split(', '))
>>>data_fields=data_split.map(lambda fields: (fields[2],fields[1]))
>>>data_group=data_fields.groupByKey(2)
```

Python

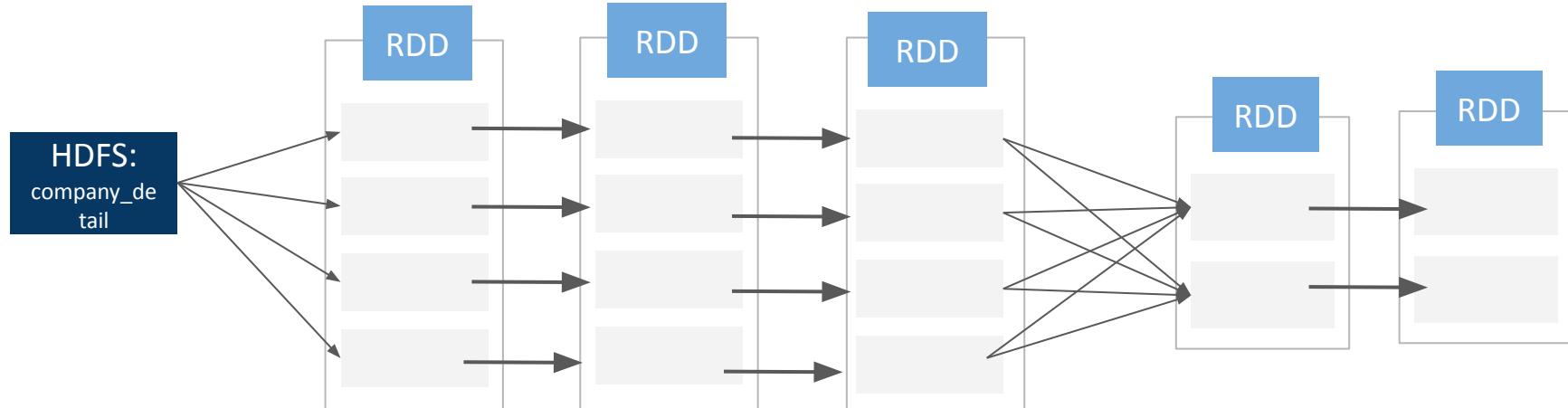


Parallel Operations

- Example : `groupByKey()` operation

```
>>>data= sc.textFile("company_detail.txt")
>>>data_split=data.map(lambda line: line.split(','))
>>>data_fields=data_split.map(lambda fields: (fields[2],fields[1]))
>>>data_group=data_fields.groupByKey(2)
>>>data_group.mapValues(lambda x: sum(x)/len(x)))
```

Python

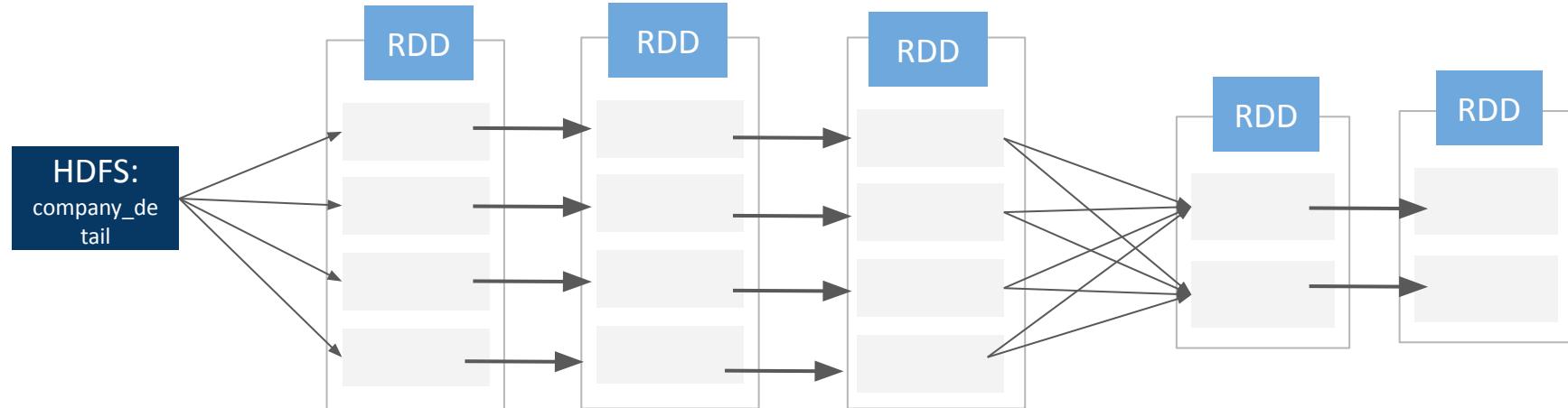


Parallel Operations

- Example : groupByKey() operation

```
>>>val data= sc.textFile("company_detail.txt")
>>> val data_split=data.map( line=> line.split(', '))
>>>val data_fields=data_split.map( fields=> (fields(2),fields(1)))
>>>val data_group=data_fields.groupByKey(2)
>>>data_group.mapValues( x=> x.sum/x.size.toDouble))
```

Scala



Chapter Outline

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages**
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

Tasks and Stages

- ❑ A stage is a set of parallel tasks, one per partition of an RDD, that compute partial results of a function executed as part of a Spark job
- ❑ A task can only belong to one stage and operate on a single partition

Tasks and Stages

```
>>>data= sc.textFile("company_detail.txt") \
    .map(lambda line: line.split(' , ')) \
    .map(lambda fields: (fields[2],fields[1])) \
    .groupByKey([2]) \
    .mapValues(lambda x: sum(x)/len(x))
```

Python

```
>>>print data.toDebugString()
```

```
(2) PythonRDD[15] at RDD at PythonRDD.scala:42 []
| MapPartitionsRDD[14] at mapPartitions at PythonRDD.scala:338 []
| ShuffledRDD[13] at partitionBy at NativeMethodAccessorImpl.java:-2 []
+- (4) PairwiseRDD[12] at groupByKey at <ipython-input-6-066445f5a562>:1 []
| PythonRDD[11] at groupByKey at <ipython-input-6-066445f5a562>:1 []
| company_detail.txt MapPartitionsRDD[9] at textFile at
NativeMethodAccessorImpl.java:-2 []
| company_detail.txt HadoopRDD[8] at textFile at NativeMethodAccessorImpl.java:-2 []
```

Stage 2

Stage 1

Tasks and Stages

```
>>>val data= sc.textFile("company_detail.txt") \  
    .map( line=> line.split(',') ) \  
    .map( fields=> (fields[2],fields[1]) ) \  
    .groupByKey(2) \  
    .mapValues( x=> x.sum/x.size.toDouble))
```

Scala

```
>>>print data.toDebugString()
```

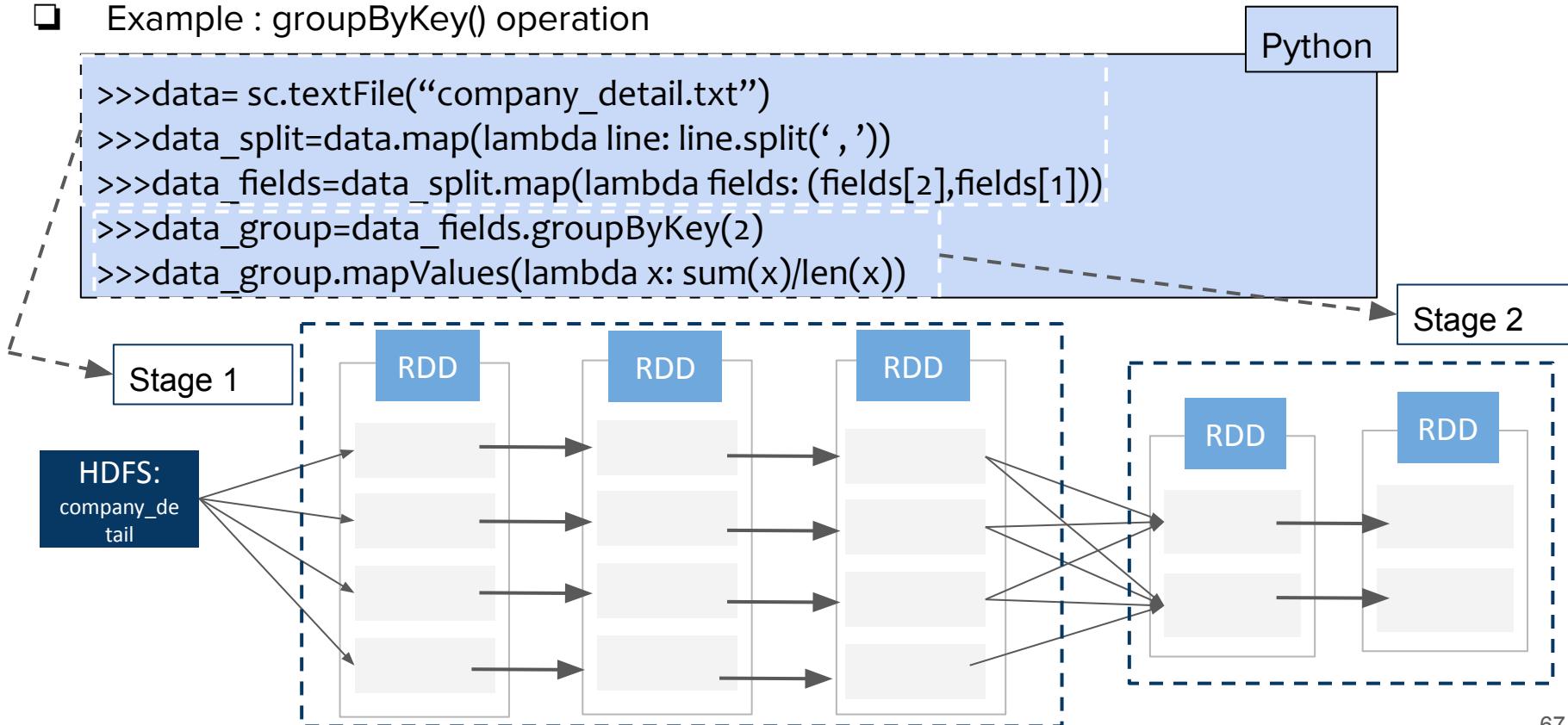
Stage 2

Stage 1

Tasks and Stages

- Example : groupByKey() operation

```
>>>data= sc.textFile("company_detail.txt")
>>>data_split=data.map(lambda line: line.split(', '))
>>>data_fields=data_split.map(lambda fields: (fields[2],fields[1]))
>>>data_group=data_fields.groupByKey(2)
>>>data_group.mapValues(lambda x: sum(x)/len(x))
```

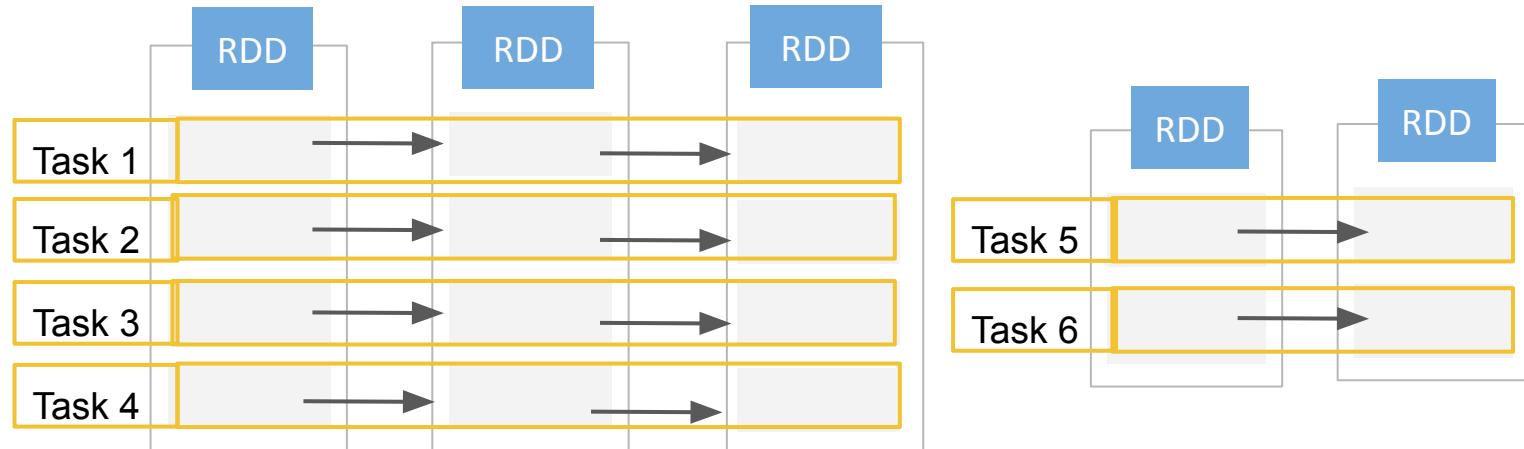


Tasks and Stages

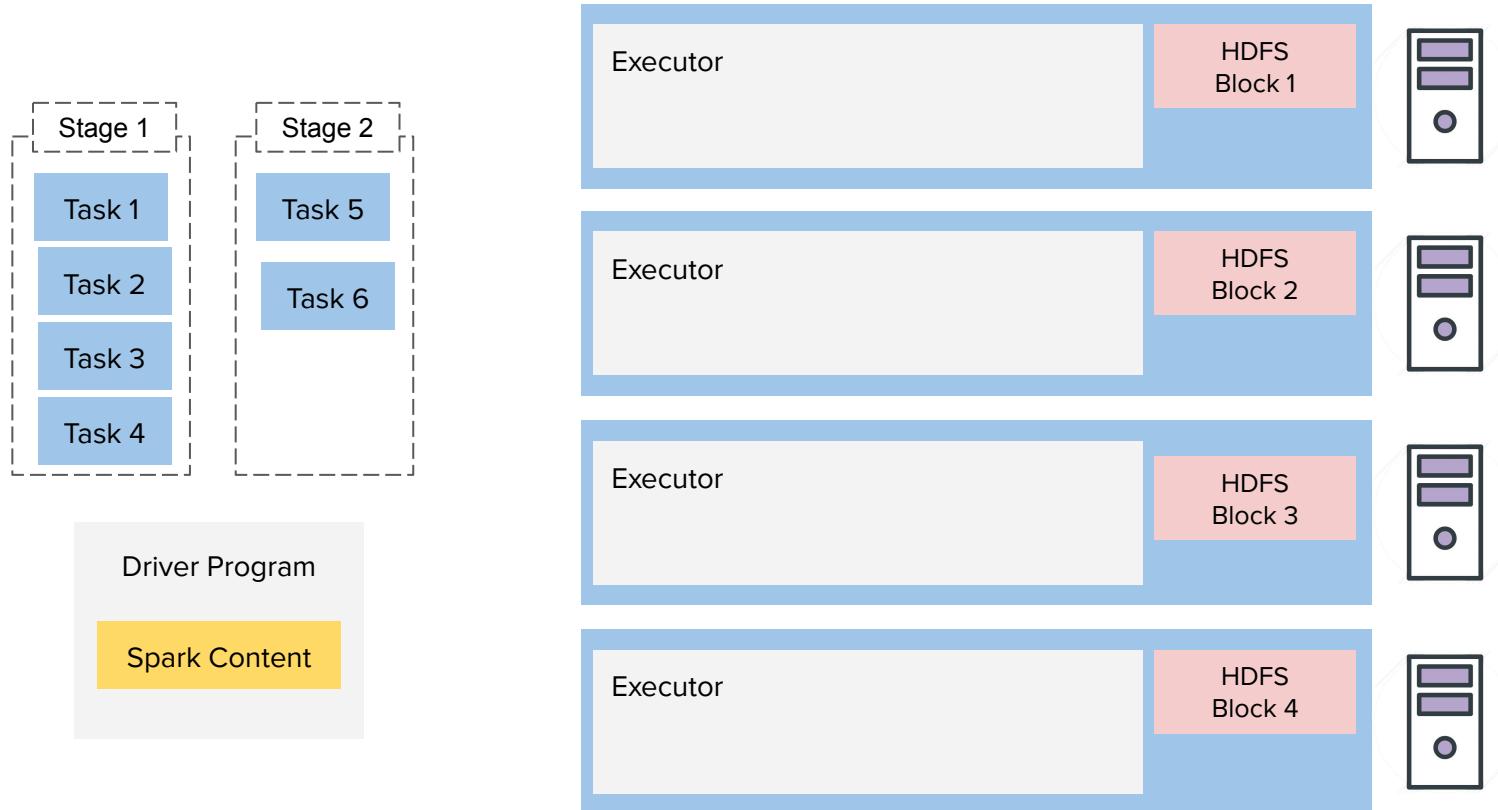
- Example : `groupByKey()` operation

```
>>>data= sc.textFile("company_detail.txt")
>>>data_split=data.map(lambda line: line.split(' '))
>>>data_fields=data_split.map(lambda fields: (fields[2],fields[1]))
>>>data_group=data_fields.groupByKey(2)
>>>data_group.mapValues(lambda x: sum(x)/len(x)))
```

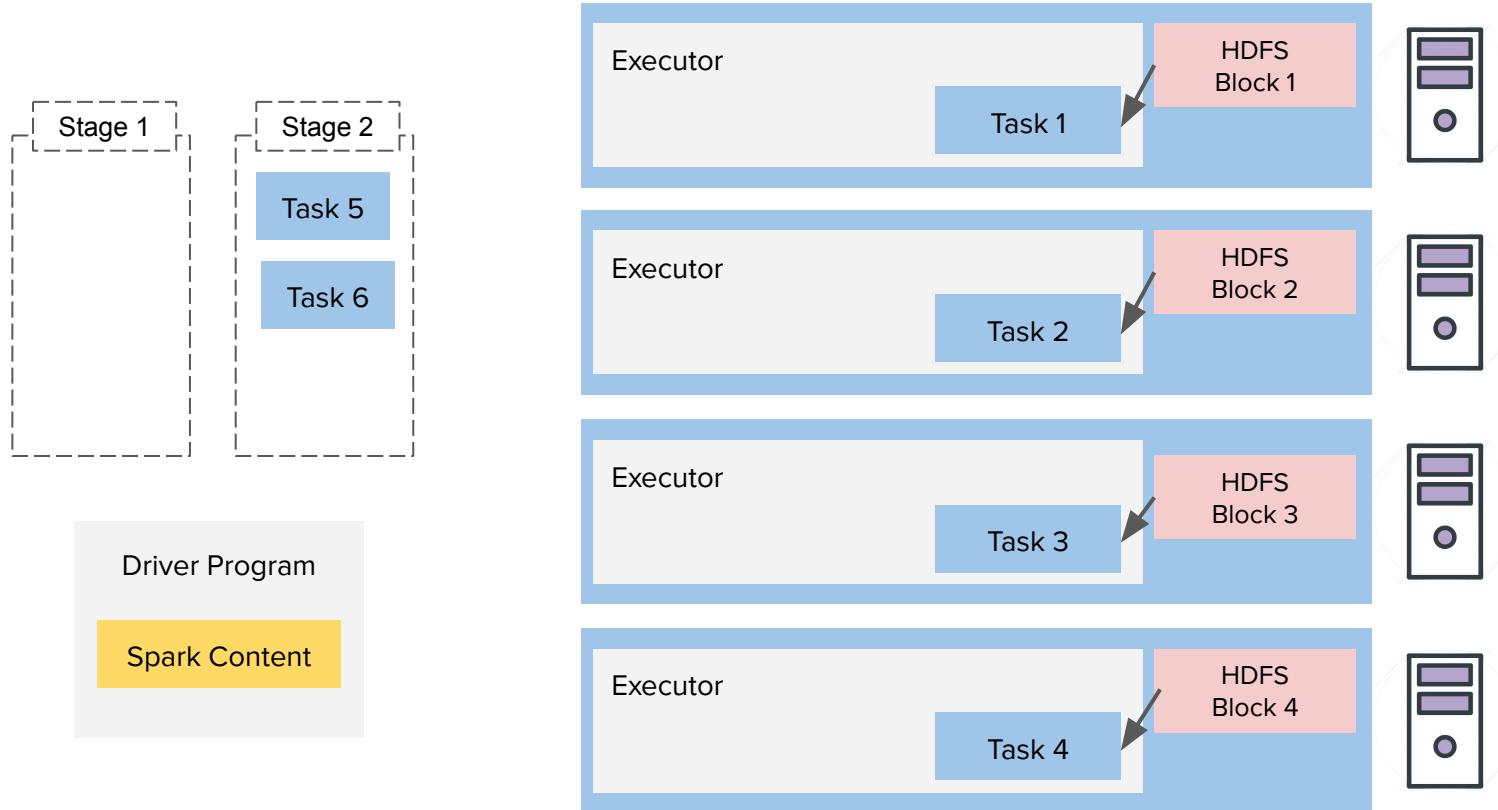
Python



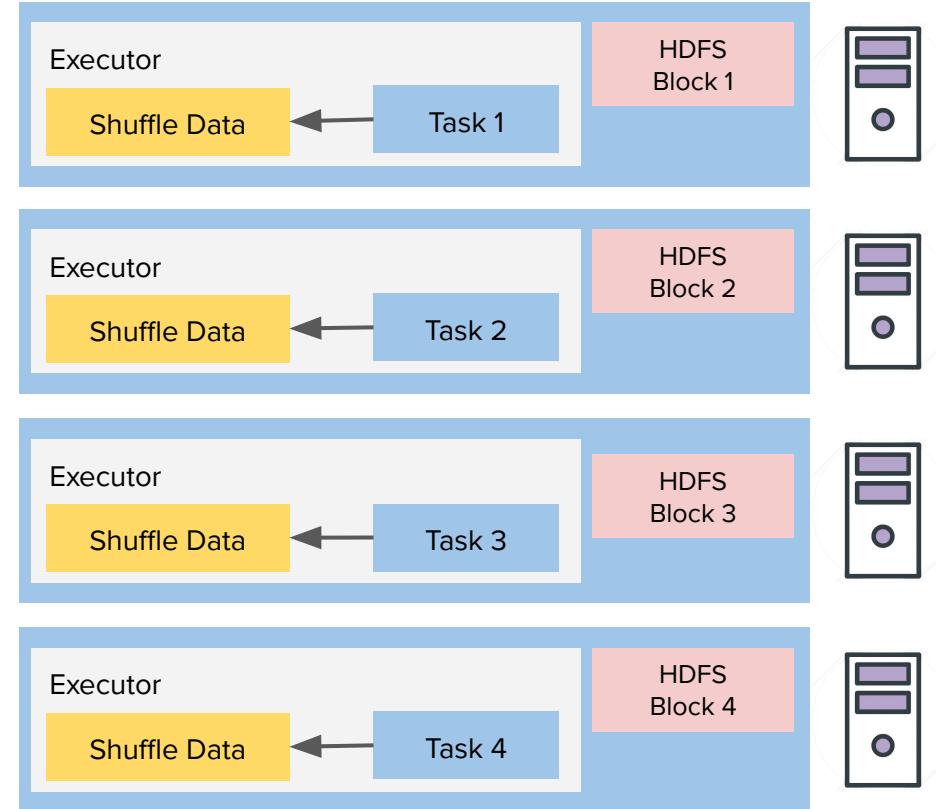
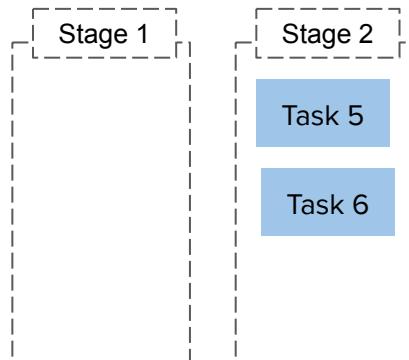
Tasks and Stages



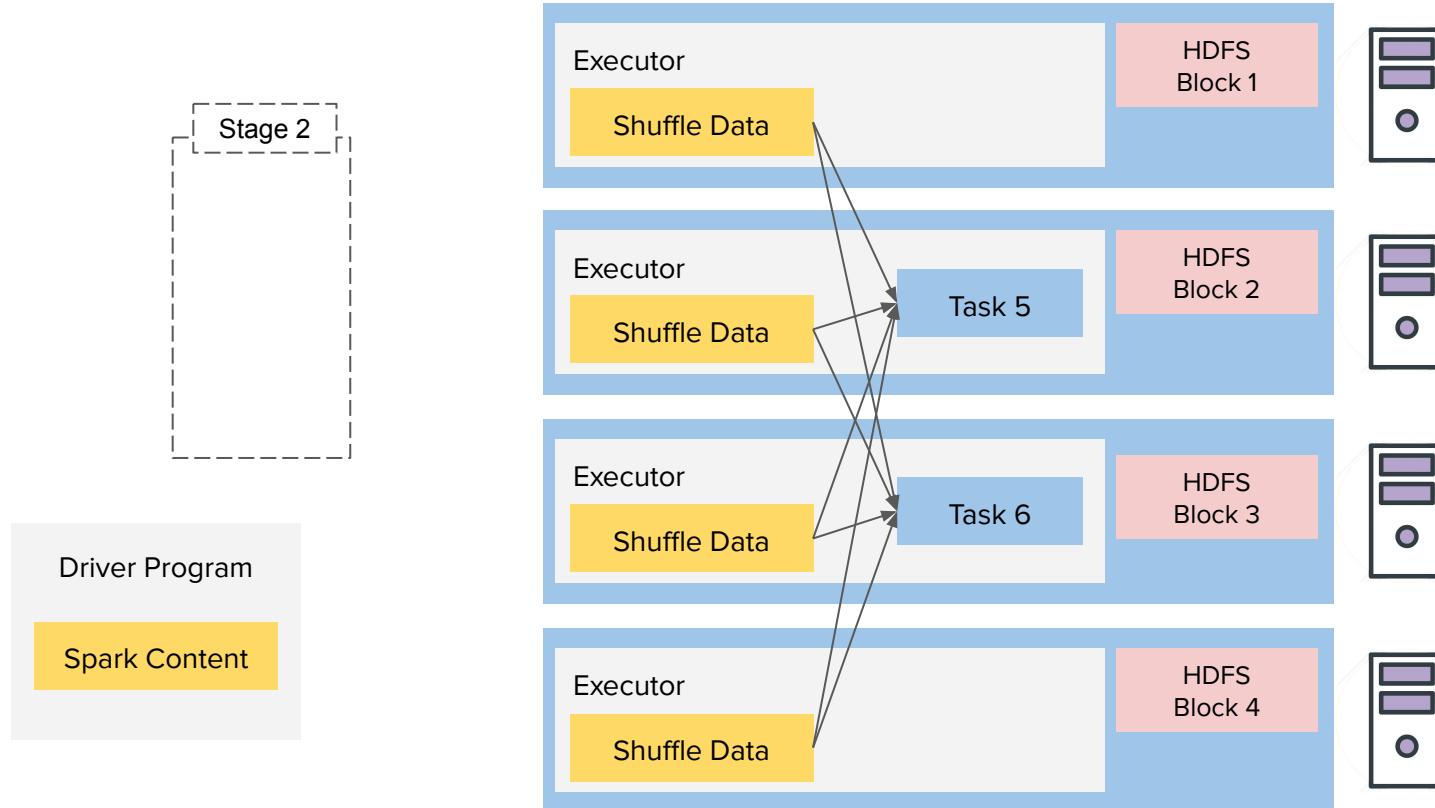
Tasks and Stages



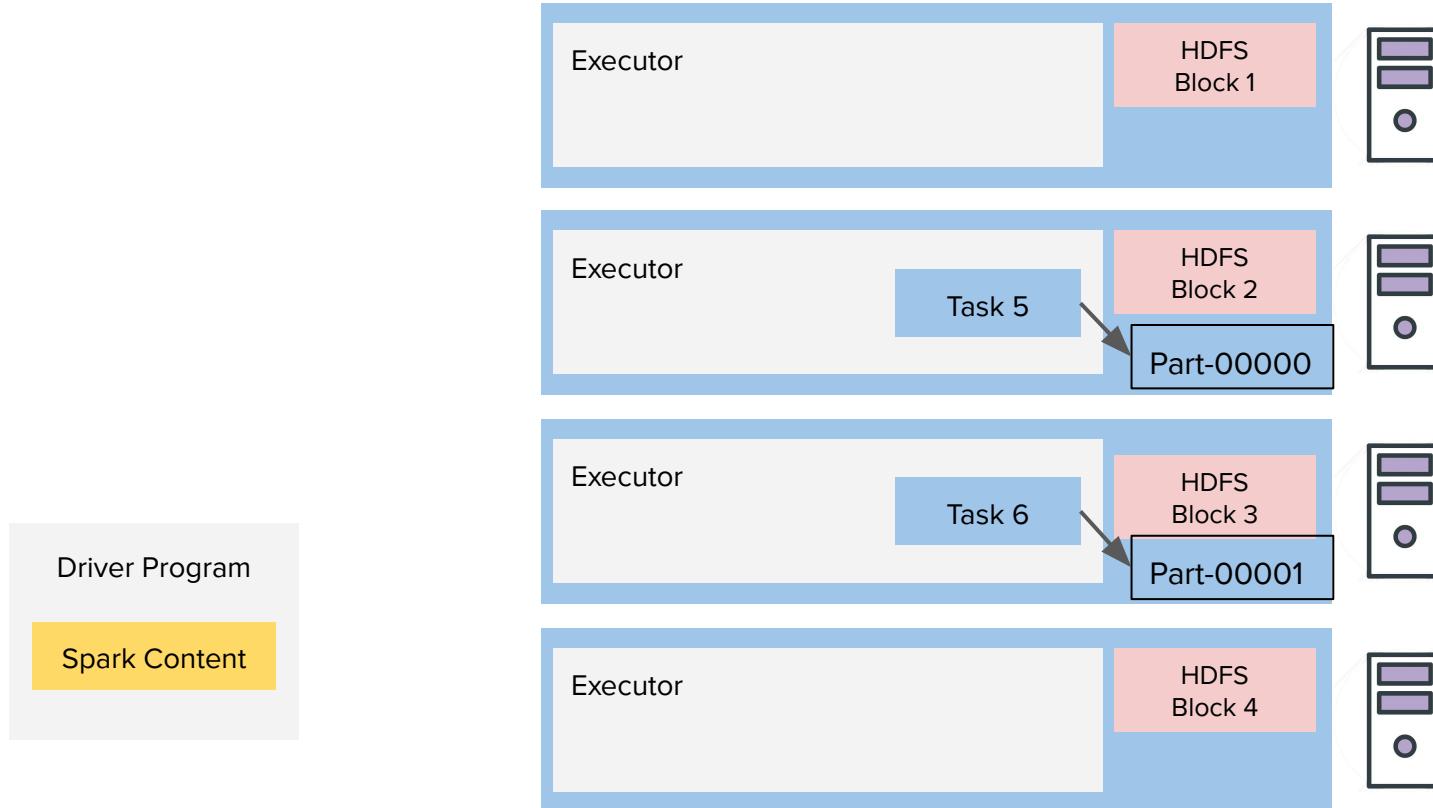
Tasks and Stages



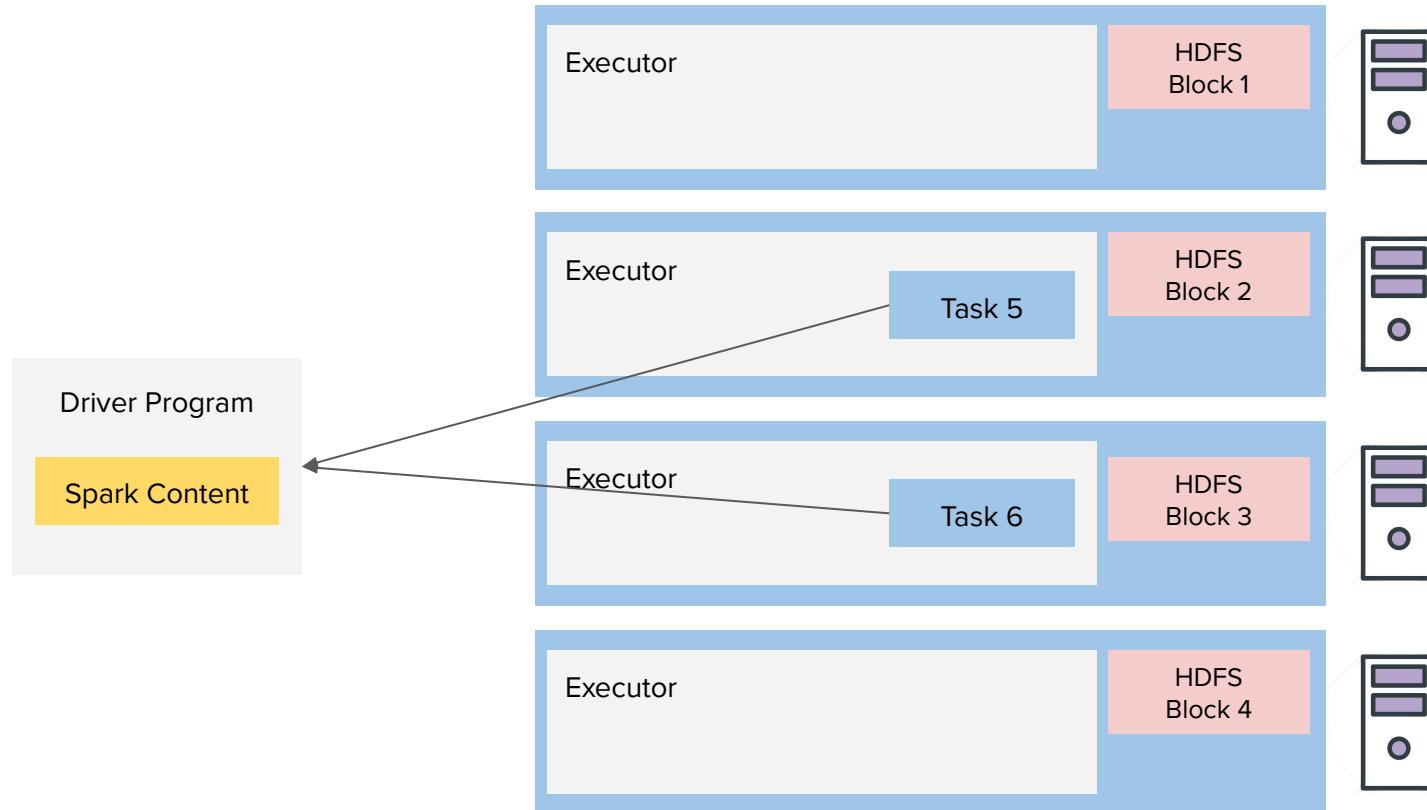
Tasks and Stages



Tasks and Stages

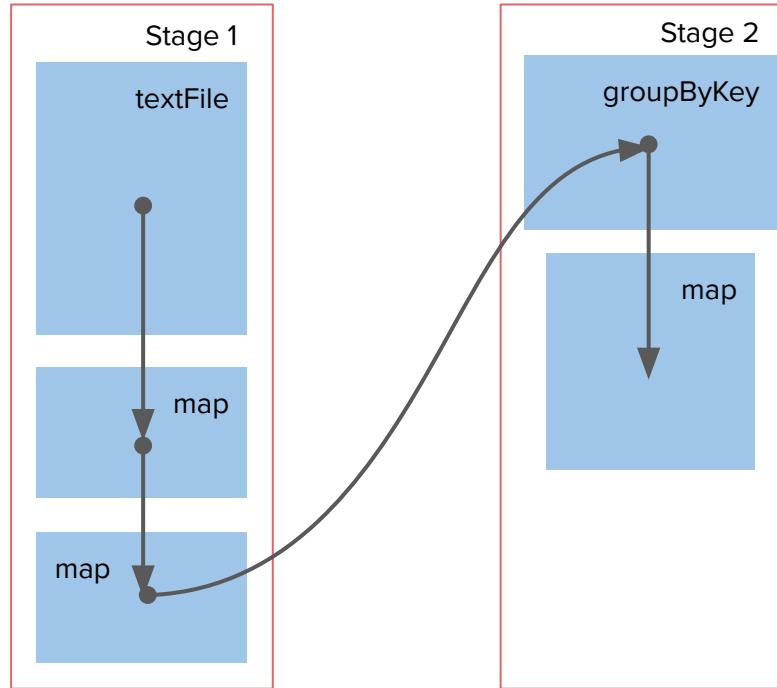


Tasks and Stages



Tasks and Stages

Job Details



Chapter Outline

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency**
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

Narrow Dependencies vs Wide Dependencies

Narrow dependencies

When each partition at the parent RDD is used by at most one partition of the child RDD, then we have a narrow dependency. Computations of transformations with this kind of dependency are rather fast as they do not require any data shuffling over the cluster network. In addition, optimizations such as *pipelining* are also possible

Example:

map, filter and union transformations

Narrow Dependencies vs Wide Dependencies

Wide dependencies

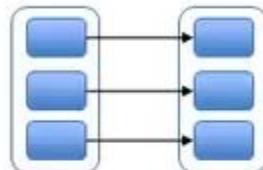
When each partition of the parent RDD may be depended on by multiple child partitions (wide dependency), then the computation speed might be significantly affected as we might need to shuffle data around different nodes when creating new partitions

Example:

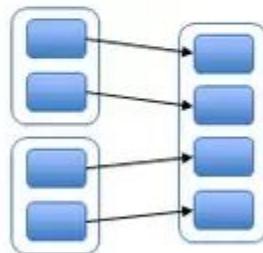
groupByKey operations and join operations whose inputs are not co-partitioned

Narrow Dependencies vs Wide Dependencies

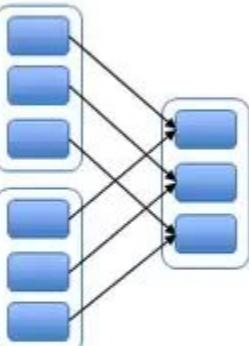
“Narrow” deps:



map, filter

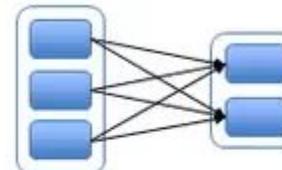


union

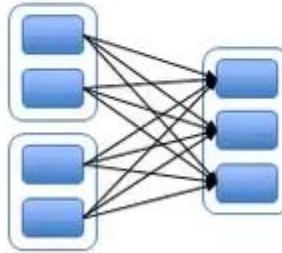


join with
inputs co-
partitioned

“Wide” (shuffle) deps:



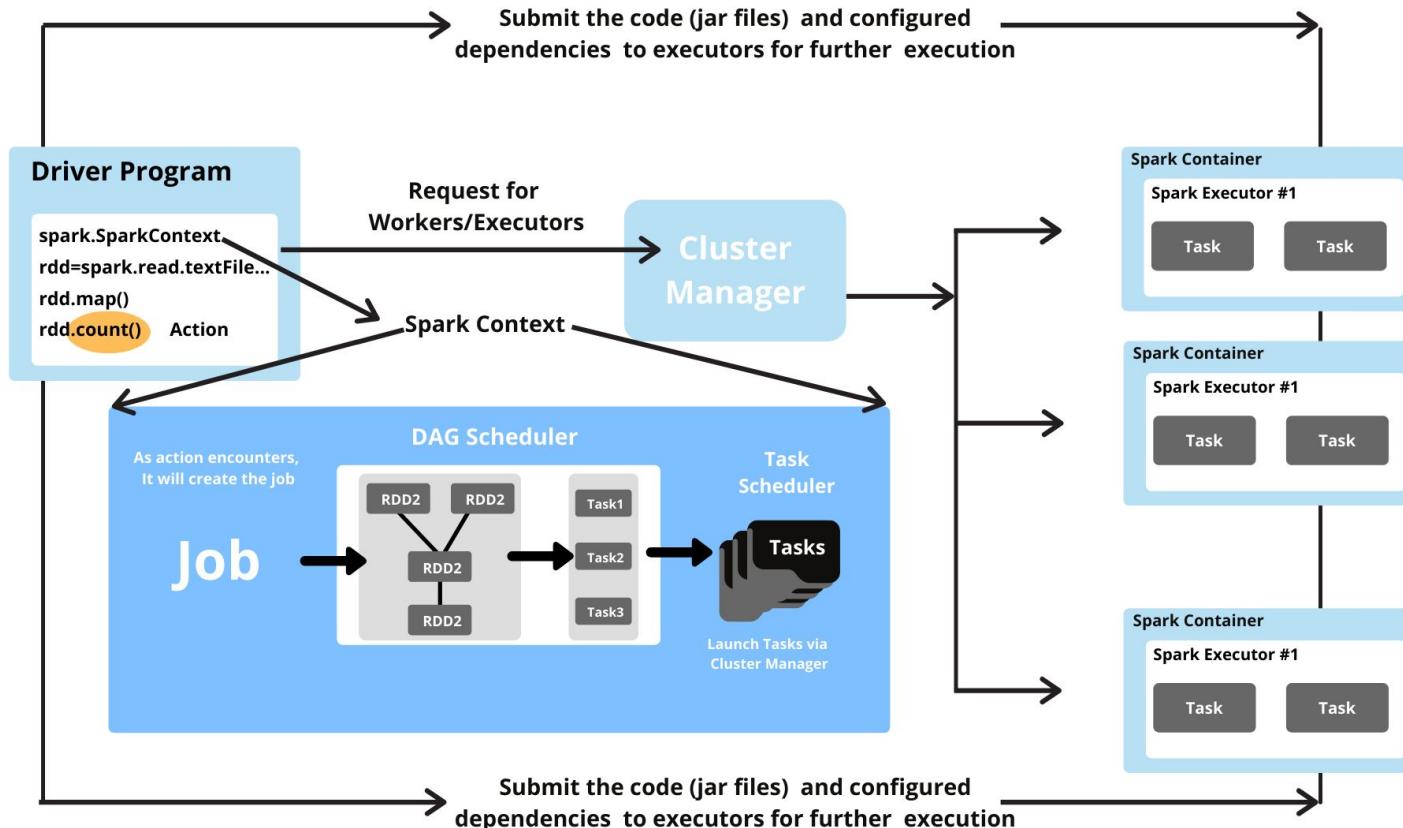
groupByKey



join with inputs not
co-partitioned

Spark Physical Execution

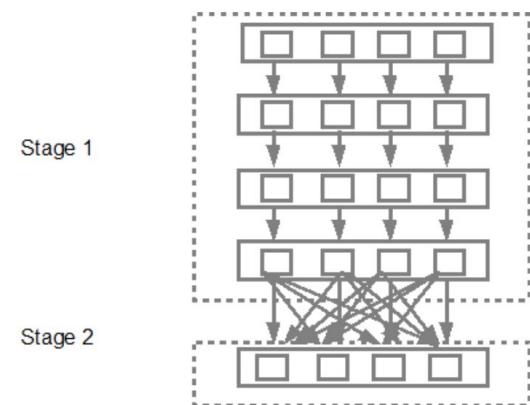
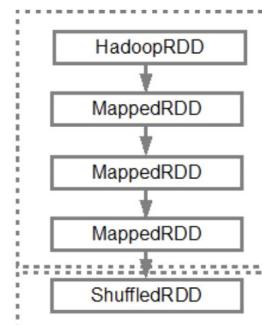
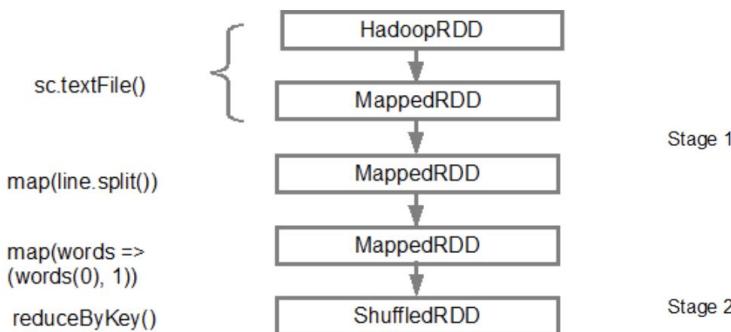
Spark Physical Architecture



Spark Core Plan

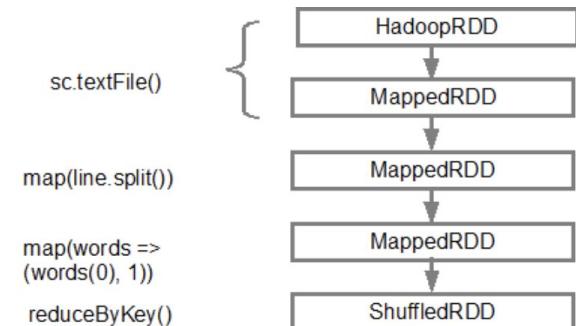
```
val input = sc.textFile("log.txt")
val splitedLines = input.map(line => line.split(" "))
    .map(words => (words(0), 1))
    .reduceByKey{(a,b) => a + b}
```

```
(2) ShuffledRDD[6] at reduceByKey at <console>:25 []
+-(2) MapPartitionsRDD[5] at map at <console>:24 []
| MapPartitionsRDD[4] at map at <console>:23 []
| log.txt MapPartitionsRDD[1] at textFile at <console>:21 []
| log.txt HadoopRDD[0] at textFile at <console>:21 []
```

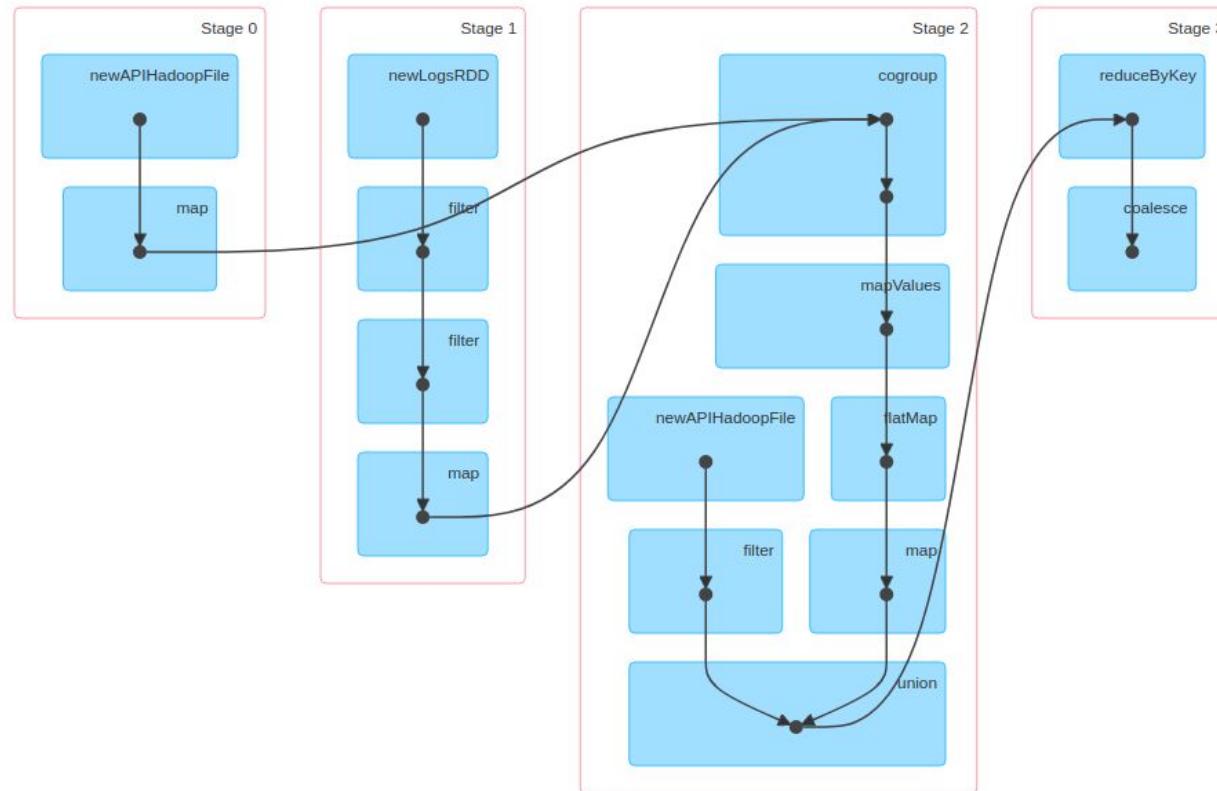


Spark Core Plan (DAG)

- ❑ Once an Action is called on RDD, Spark creates the DAG and submits it to the DAG scheduler
- ❑ The DAG scheduler divides operators into stages of tasks
- ❑ The DAG scheduler pipelines operators together e.g. Many map operators can be scheduled in a single stage
- ❑ The final result of a DAG scheduler is a set of stages



Spark Core Plan



Spark Core Plan (Task Scheduler)

- ❑ The Stages are passed on to the Task Scheduler
- ❑ The task scheduler launches tasks via cluster manager (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies of the stages
- ❑ The Worker executes the tasks on the Slave

Spark Optimization

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI**
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

Spark UI

Job Tab

Spark Jobs (?)

User: pablo

Total Uptime: 9.1 min

Scheduling Mode: FIFO

Active Jobs: 1

Completed Jobs: 7

Event Timeline

Enable zooming

Executors

 Added

 Removed

Executor driver added

Executor 0 added

Jobs

 Succeeded

 Failed

 Running

17:42
Sat 10 August

17:43

17:44

17:45

17:46

17:47

17:48

17:49

17:50

17:

coun

Job Tab: Job Details

▼ Active Jobs (1)

Page: 1

1 Pages. Jump to . Show items in a page.

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	count at <console>:26 count at <console>:26 (kill)	17:50:13	17 s	0/2	0/5 (4 running)

Page: 1

1 Pages. Jump to . Show items in a page.

▼ Completed Jobs (7)

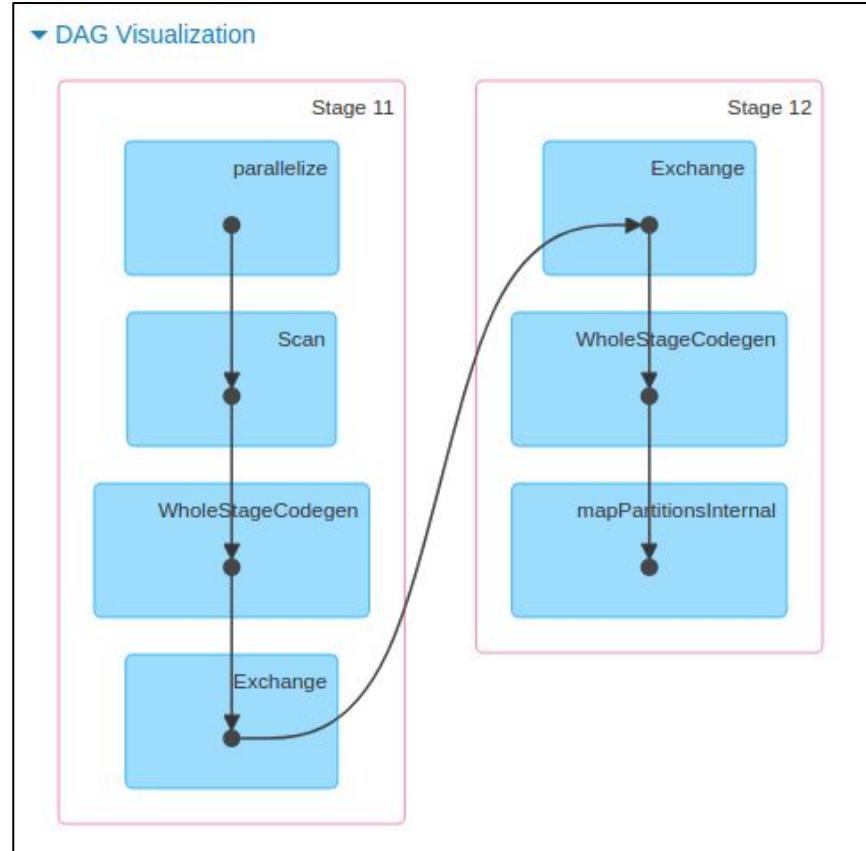
Page: 1

1 Pages. Jump to . Show items in a page.

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
6	show at <console>:26 show at <console>:26	17:49:30	0.4 s	1/1	1/1
5	show at <console>:28 show at <console>:28	17:48:32	0.8 s	3/3	9/9
4	show at <console>:28 show at <console>:28	17:47:40	2 s	3/3	9/9

Spark UI

Job Tab: DAG



Spark UI

Job Tab: Stage Information

Completed Stages (2)

Page:

1 Pages. Jump to . Show items in a page.

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
12	count at <console>:26 +details	17:50:44	56 ms	1/1			236.0 B	
11	count at <console>:26 +details	17:50:13	31 s	4/4				236.0 B

Page:

1 Pages. Jump to . Show items in a page.

Stage Tab

Stages for All Jobs

Active Stages: 1

Pending Stages: 1

Completed Stages: 13

▼ Fair Scheduler Pools (3)

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
production	2	1	1	3	FAIR
test	3	2	0	0	FIFO
default	0	1	0	0	FIFO

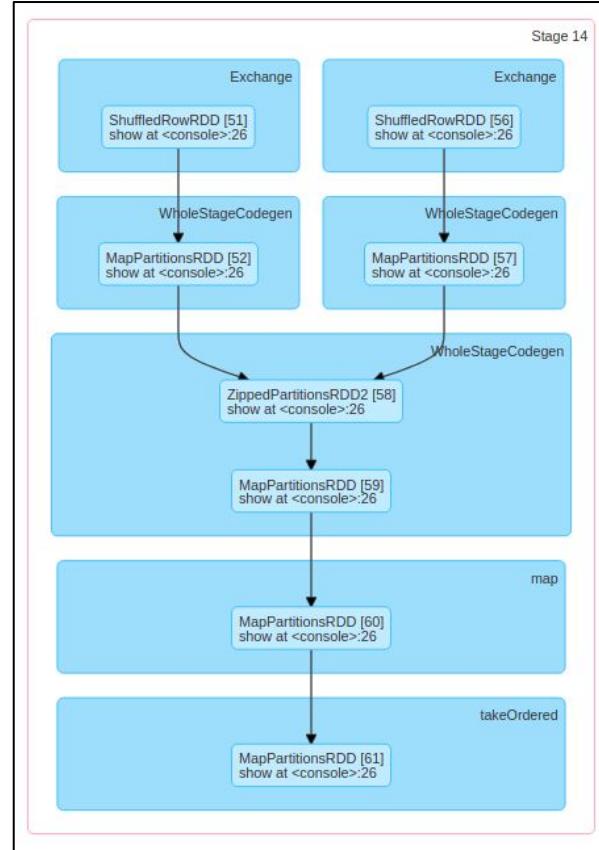
Spark UI

Stage Tab

Active Stages (1)										
Page: 1 1 Pages. Jump to <input type="text"/> . Show <input type="text"/> items in a page. <input type="button" value="Go"/>										
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	
13	production	show at <console>:26	+details (kill) 2019/08/27 18:25:33	0.2 s	0/4 (4 running)					
Pending Stages (1)										
Page: 1 1 Pages. Jump to <input type="text"/> . Show <input type="text"/> items in a page. <input type="button" value="Go"/>										
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	
14	default	show at <console>:26	+details Unknown	Unknown	0/200					
Completed Stages (13)										
Page: 1 1 Pages. Jump to <input type="text"/> . Show <input type="text"/> items in a page. <input type="button" value="Go"/>										
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	
12	production	show at <console>:26	+details 18:25:33	0.3 s	4/4				1205.5 KiB	
11	production	show at <console>:28	+details 18:24:36	0.3 s	1/1				12.6 KiB	
10	production	show at <console>:28	+details 18:24:35	0.3 s	4/4				1205.5 KiB	
9	production	show at <console>:28	+details 18:24:35	0.3 s	4/4				1205.5 KiB	
8	default	foreach at <console>:27	+details 18:01:04	93 ms	4/4					
7	default	foreach at <console>:27	+details 17:59:31	0.5 s	4/4					
6	production	show at <console>:28	+details 17:56:10	64 ms	1/1				12.6 KiB	
5	production	show at <console>:28	+details 17:56:10	0.2 s	4/4				1205.5 KiB	

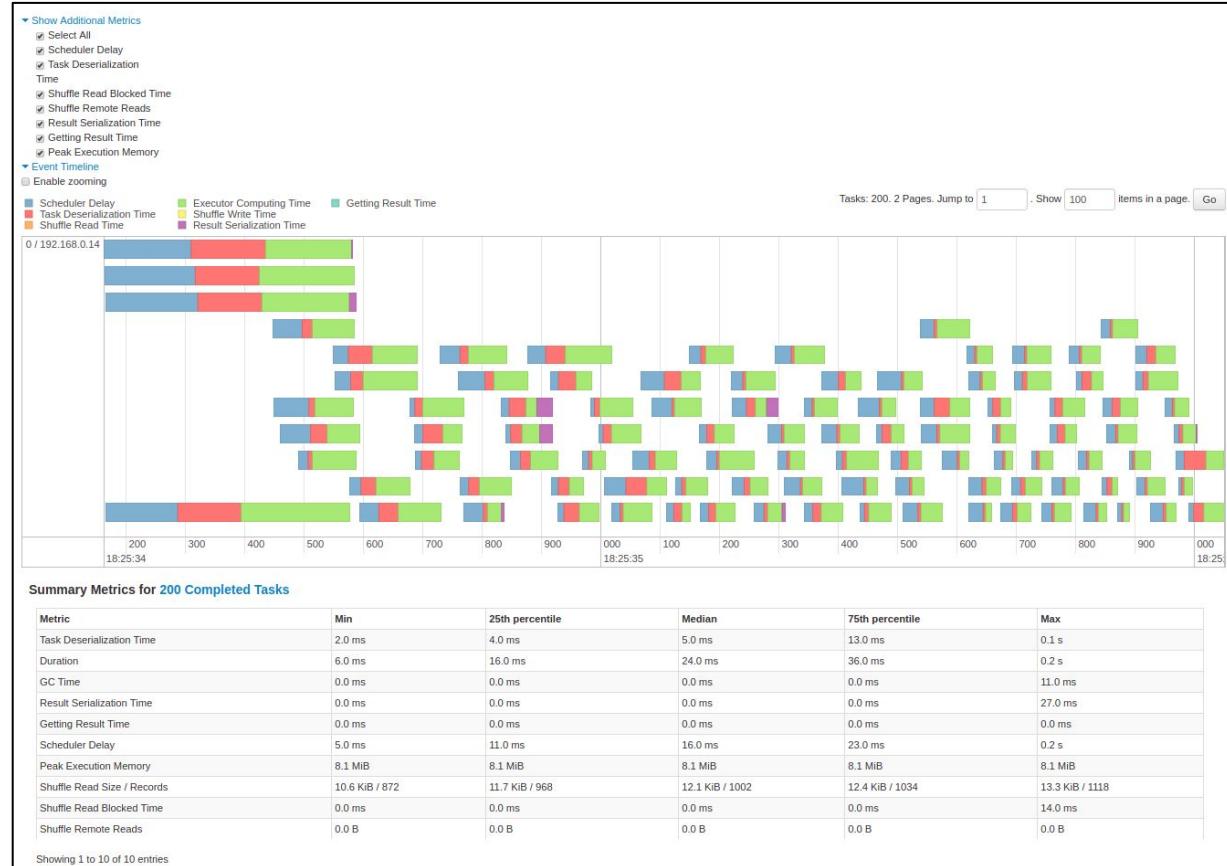
Spark UI

Stage Tab: Details



Spark UI

Stage Tab: Details



Spark UI

Storage Tab

Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
1	rdd	Memory Serialized 1x Replicated	5	100%	236.0 B	0.0 B
4	LocalTableScan [count#7, name#8]	Disk Serialized 1x Replicated	3	100%	0.0 B	2.1 KiB

RDD Storage Info for rdd

Storage Level: Memory Serialized 1x Replicated

Cached Partitions: 5

Total Partitions: 5

Memory Size: 236.0 B

Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
10.12.221.7:52707	236.0 B (366.3 MiB Remaining)	0.0 B (0.0 B Remaining)	0.0 B

5 Partitions

Page: 1	Storage Level	Size in Memory	Size on Disk	Executors	1 Pages. Jump to <input type="text" value="1"/> . Show <input type="text" value="100"/> items in a page. <input type="button" value="Go"/>
rdd_1_0	Memory Serialized 1x Replicated	40.0 B	0.0 B	'	
rdd_1_1	Memory Serialized 1x Replicated	40.0 B	0.0 B	'	
rdd_1_2	Memory Serialized 1x Replicated	40.0 B	0.0 B	'	
rdd_1_3	Memory Serialized 1x Replicated	56.0 B	0.0 B	'	
rdd_1_4	Memory Serialized 1x Replicated	60.0 B	0.0 B	'	

Spark UI

Environment Tab

Environment

Runtime Information

Name	Value
Java Home	/Library/Java/JavaVirtualMachines/jdk1.8.0_221.jdk/Contents/Home/jre
Java Version	1.8.0_221 (Oracle Corporation)
Scala Version	version 2.12.8

Spark Properties

Name	Value
spark.app.id	local-1565684968905
spark.app.name	Spark shell
spark.driver.host	10.12.221.7
spark.driver.port	58229
spark.executor.id	driver
spark.home	/Users/zrf/Dev/OpenSource/spark
spark.jars	
spark.master	local[*]
spark.repl.class.outputDir	/private/var/folders/vt/5vsxj6rz6syxf4ssyklm0000gn/T/spark-bae57fcf-54f9-48b5-8e71-cfb15c126e64/repl-e2551dc1-3350-47db-a5eb-b0148ef86325
spark.repl.class.uri	spark://10.12.221.7:58229/classes
spark.scheduler.mode	FIFO
spark.sql.catalogImplementation	in-memory
spark.submit.deployMode	client
spark.submit.pyFiles	
spark.ui.showConsoleProgress	true

Hadoop Properties

System Properties

Classpath Entries

Environment Tab

Hadoop Properties

Name	Value
dfs.ha.fencing.ssh.connect-timeout	30000
file.blocksize	67108864
file.bytes-per-checksum	512
file.client-write-packet-size	65536
file.replication	1
file.stream-buffer-size	4096
fs.AbstractFileSystem.file.impl	org.apache.hadoop.fs.local.LocalFs
fs.AbstractFileSystem.ftp.impl	org.apache.hadoop.fs.ftp.FtpFs
fs.AbstractFileSystem.har.impl	org.apache.hadoop.fs.HarFs
fs.AbstractFileSystem.hdfs.impl	org.apache.hadoop.fs.Hdfs
fs.AbstractFileSystem.viewfs.impl	org.apache.hadoop.fs.viewfs.ViewFs

Spark UI

Environment Tab

System Properties

Name	Value
SPARK_SUBMIT	true
awt.toolkit	sun.lwawt.macosx.LWCToolkit
file.encoding	UTF-8
file.encoding.pkg	sun.io
file.separator	/
gopherProxySet	false
java.awt.graphicsenv	sun.awt.CGraphicsEnvironment
java.awt.printerjob	sun.lwawt.macosx.CPrinterJob
java.class.version	52.0

Spark UI

Environment Tab

Classpath Entries

Resource	Source
/Users/zrf/Dev/OpenSource/spark/assembly/target/scala-2.12/jars/RoaringBitmap-0.7.45.jar	System Classpath
/Users/zrf/Dev/OpenSource/spark/assembly/target/scala-2.12/jars/activation-1.1.1.jar	System Classpath
/Users/zrf/Dev/OpenSource/spark/assembly/target/scala-2.12/jars/aircompressor-0.10.jar	System Classpath
/Users/zrf/Dev/OpenSource/spark/assembly/target/scala-2.12/jars/antlr4-runtime-4.7.1.jar	System Classpath
/Users/zrf/Dev/OpenSource/spark/assembly/target/scala-2.12/jars/aopalliance-1.0.jar	System Classpath

Spark Physical Execution

Spark UI

Executor Tab

Executors

Show Additional Metrics

- Select All
- On Heap Memory
- Off Heap Memory

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(3)	0	5.9 KiB / 1.1 GiB	0.0 B	2	0	0	5	5	4 s (0.2 s)	0.0 B	0.0 B	0.0 B	0
Total(3)	0	5.9 KiB / 1.1 GiB	0.0 B	2	0	0	5	5	4 s (0.2 s)	0.0 B	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0

Executors

Show 20 entries

Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
1	10.12.221.27:55834	Active	0	2 KiB / 366.3 MiB	0.0 B	1	0	0	3	3	2 s (0.1 s)	0.0 B	0.0 B	0.0 B	stdout	Thread Dump
0	10.12.221.27:55835	Active	0	2 KiB / 366.3 MiB	0.0 B	1	0	0	2	2 s (94.0 ms)	0.0 B	0.0 B	0.0 B	stdout	Thread Dump	
driver	10.12.221.27:55827	Active	0	2 KiB / 366.3 MiB	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump	

Showing 1 to 3 of 3 entries

Previous 1 Next

Spark UI

Executor Tab

[Back to Master](#)

Showing 4944 Bytes: 0 - 4944 of 4944

Top of Log

```
Spark Executor Command: "/Library/Java/JavaVirtualMachines/jdk1.8.0_221.jdk/Contents/Home/bin/java" "-cp" "/Users/zrf/Dev/OpenSource/spark/conf:/Users/zrf/Dev/OpenSource/spark/assembly/target/scala-2.12/jars/*"-Xmx1024M" "-Dspark.driver.port=56124" "org.apache.spark.executor.CoarseGrainedExecutorBackend" "--driver-url" "spark://CoarseGrainedScheduler@10.12.221.27:56124" "--executor-id" "1" "--hostname" "10.12.221.27" "--cores" "1" "--app-id" "app-20190827153447-0000" "--worker-url" "spark://Worker@10.12.221.27:56127"
=====
```

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties

```
19/08/27 15:34:50 INFO CoarseGrainedExecutorBackend: Started daemon with process name: 3867@ZRFs-MAC.local
19/08/27 15:34:50 INFO SignalUtils: Registered signal handler for TERM
19/08/27 15:34:50 INFO SignalUtils: Registered signal handler for HUP
19/08/27 15:34:50 INFO SignalUtils: Registered signal handler for INT
19/08/27 15:34:52 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
19/08/27 15:34:52 INFO SecurityManager: Changing view acls to: zrf
19/08/27 15:34:52 INFO SecurityManager: Changing modify acls to: zrf
19/08/27 15:34:52 INFO SecurityManager: Changing view acls groups to:
19/08/27 15:34:52 INFO SecurityManager: Changing modify acls groups to:
19/08/27 15:34:52 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(zrf); groups with view permissions: Set(); users with modify permissions: Set(zrf); groups with modify permissions: Set()
19/08/27 15:34:53 INFO TransportClientFactory: Successfully created connection to /10.12.221.27:56124 after 228 ms (0 ms spent in bootstraps)
19/08/27 15:34:54 INFO SecurityManager: Changing view acls to: zrf
19/08/27 15:34:54 INFO SecurityManager: Changing modify acls to: zrf
19/08/27 15:34:54 INFO SecurityManager: Changing view acls groups to:
19/08/27 15:34:54 INFO SecurityManager: Changing modify acls groups to:
19/08/27 15:34:54 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(zrf); groups with view permissions: Set(); users with modify permissions: Set(zrf); groups with modify permissions: Set()
```

Spark UI

Executor Tab

Thread dump for executor 0

Updated at 2019/08/27 17:06:52

[Collapse All](#)

Search:

Thread ID	Thread Name	Thread State	Thread Locks
32	Executor task launch worker for task 4	TIMED_WAITING	
<pre>sun.misc.Unsafe.park(Native Method) java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:215) java.util.concurrent.SynchronousQueue\$TransferStack.awaitFill(SynchronousQueue.java:460) java.util.concurrent.SynchronousQueue\$TransferStack.transfer(SynchronousQueue.java:362) java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:941) java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1073) java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1134) java.util.concurrent.ThreadPoolExecutor\$Worker.run(ThreadPoolExecutor.java:624) java.lang.Thread.run(Thread.java:748)</pre>			
38	block-manager-slave-async-thread-pool-0	TIMED_WAITING	
<pre>sun.misc.Unsafe.park(Native Method) java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:215) java.util.concurrent.locks.AbstractQueuedSynchronizer\$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:2078) java.util.concurrent.LinkedBlockingQueue.poll(LinkedBlockingQueue.java:467) java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1073) java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1134) java.util.concurrent.ThreadPoolExecutor\$Worker.run(ThreadPoolExecutor.java:624) java.lang.Thread.run(Thread.java:748)</pre>			
39	block-manager-slave-async-thread-pool-1	TIMED_WAITING	

SQL Tab

SQL

Completed Queries: 3

▼ Completed Queries (3)

Page:

1 Pages. Jump to . Show items in a page.

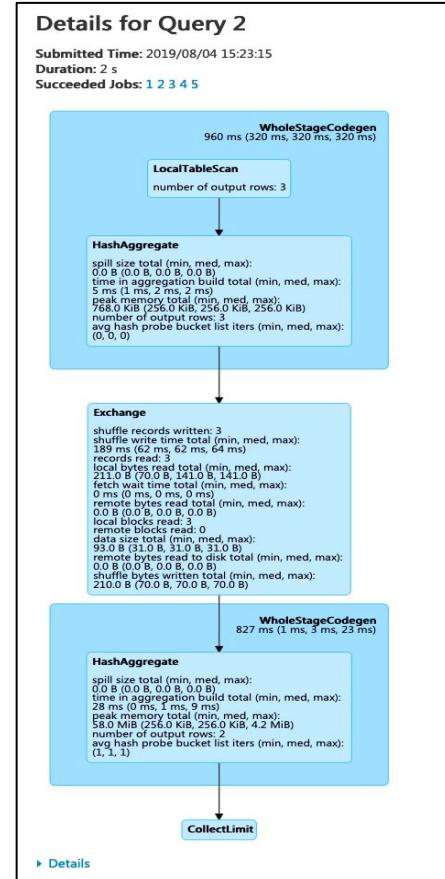
ID	Description	Submitted	Duration	Job IDs
2	show at <console>:24	2019/08/04 15:23:15 +details	2 s	[1][2][3][4][5]
1	createGlobalTempView at <console>:26	2019/08/04 15:23:09 +details	0.3 s	
0	count at <console>:26	2019/08/04 15:23:01 +details	2 s	[0]

Page:

1 Pages. Jump to . Show items in a page.

Spark UI

SQL Tab



Streaming Tab: JDBC/ODBC Server tab

JDBC/ODBC Server

Started at: 2019/09/05 18:01:08

Time since start: 4 hours 25 minutes 24 seconds

2 session(s) are online, running 0 SQL statement(s)

Session Statistics (2)

User	IP	Session ID	Start Time	Finish Time	Duration	Total Execute
anonymous	127.0.0.1	e311723a-95ae-42b2-90ae-b659f0277468	22:26:13		19 seconds 628 ms	0
planga82	127.0.0.1	9d6fc5d-696c-404d-be03-0e5a5e8f6805	18:01:17		4 hours 25 minutes 15 seconds	8

Streaming Tab: JDBC/ODBC Server tab

SQL Statistics (4)

User	JobID	GroupID	Start Time	Finish Time	Close Time	Execution Time	Duration	Statement	State	Detail
anonymous		f165de80-146f-4b23-93e6-c1d44e03512d	08:31:34	08:31:34	08:31:34	245 ms	250 ms	select not_fount_udf(1) from employee	CLOSED	Undefined function: 'not_fount_udf'. This function is neither a registered temporary function nor a permanent function registered in the database 'default'; line 1 pos 7
anonymous	[2] [3]	057d5b20-ce43-4e3d-9f45-375bc1e7635a	08:30:12	08:30:27	08:30:29	14 seconds 608 ms	16 seconds 463 ms	select name, count(*) from employee group by name order by name	CLOSED	= Parsed Logical Plan = + details
anonymous	[1]	8c520411-2c97-44be-9d68-94d159557c04	08:29:22	08:29:38	08:29:38	15 seconds 862 ms	16 seconds 403 ms	select * from employee	CLOSED	= Parsed Logical Plan = + details
anonymous	[0]	dafad4b8-5a10-453b-b189-fc6755298093	08:28:49	08:29:07	08:29:07	17 seconds 106 ms	17 seconds 521 ms	insert into table employee values ("eid1","Belen","1000","NY")	CLOSED	= Parsed Logical Plan = + details

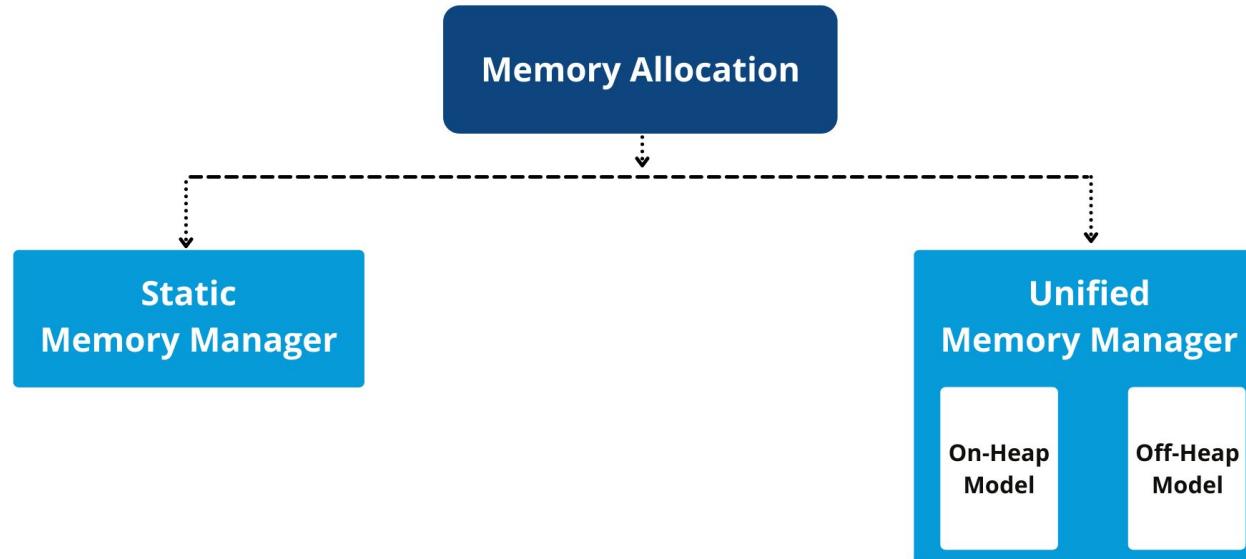
Spark Optimization

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture**
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

Executor Memory Architecture

Two types of Spark Memory Management -

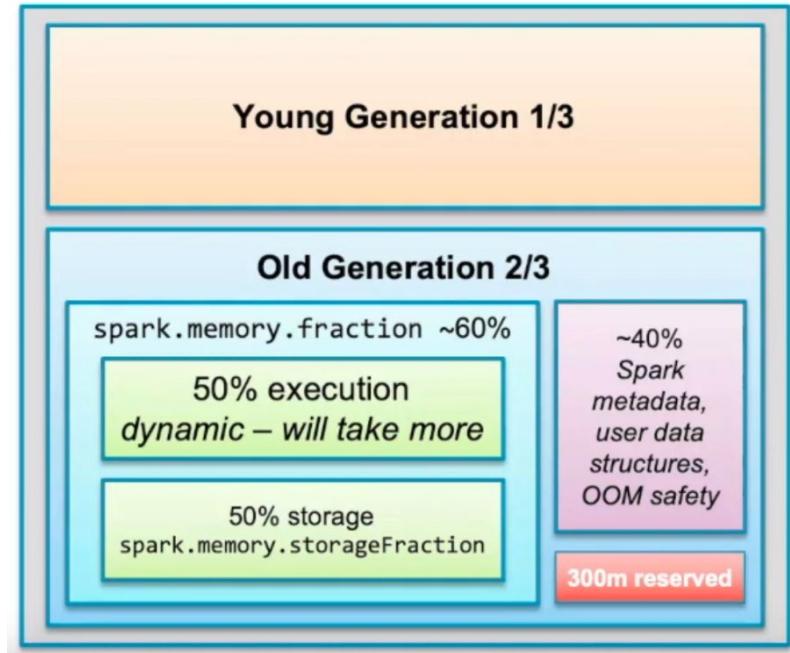
- (i) Static Memory Manager (Before Spark 1.6)
- (ii) Unified Memory Manager (After Spark 1.6) - Focus of our discussion



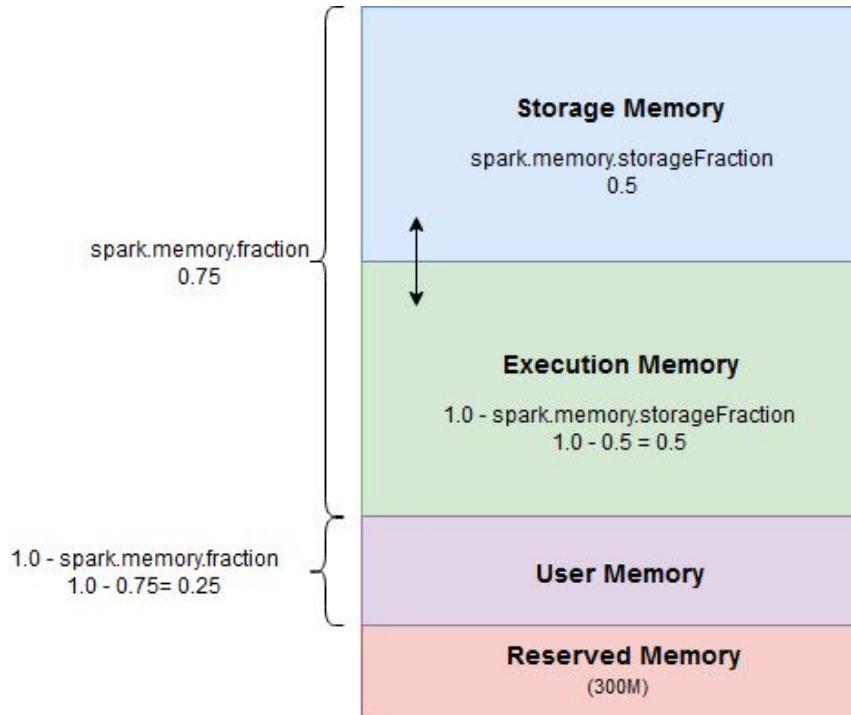
Spark Memory Management:

[SPARK-1000: Consolidate Storage and execution memory management](#)

- ❑ NewRatio controls Young/Old proportion
- ❑ **spark.memory.fraction** sets storage and execution space to ~60% tenures space



Unified Memory Manager: On-Heap Model



Executor Memory Architecture

Spark Memory: It is the memory pool managed by Apache Spark. If you want to calculate the size then you can do by,

(“Java Heap” - “Reserved Memory”) * spark.memory.fraction

The whole pool is divided into 2 parts:

- ❑ **Storage Memory:** It is used for both storing Apache Spark Cached data and for temporary space serialized data. Here, all the “broadcast” variables can also be stored
- ❑ **Execution Memory:** It is used for storing the objects required during the execution of Spark tasks. Such as, it store shuffled immediate buffer in the Map side in memory

Executor Memory Architecture

Storage Memory pool can borrow some space from Execution Memory pool only if there is some space in Execution Memory pool available

“Spark Memory” * spark.memory.storageFraction = (“Java Heap” - “Reserved Memory”) * spark.memory.fraction * spark.memory.storageFraction

By default,

$$(\text{“Java Heap”} - 300\text{MB}) * 0.75 * 0.5 = (\text{“Java Heap”} - 300\text{MB}) * 0.375$$

For **4GB heap**, it would result in **1423.5MB** of RAM in initial Storage Memory Region

Unified Memory Manager: Off-Heap Model

Off-heap refers to objects that are managed by the OS(Operating System) but stored outside the process heap in native memory

Accessing this data is a bit slower than accessing the on-heap storage but still faster than reading/writing from a disk

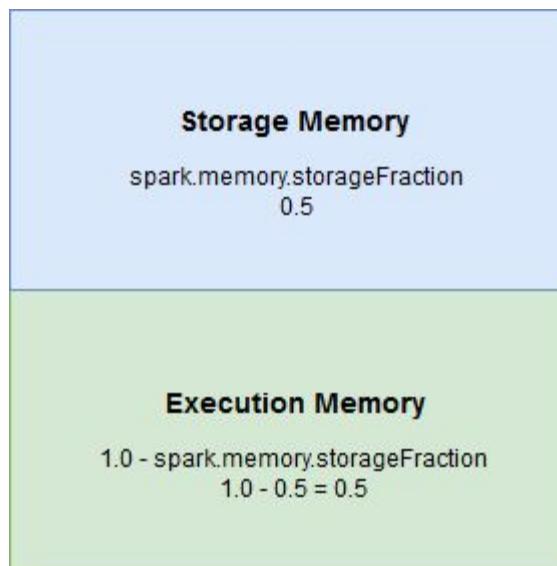
Off-heap memory usage is available for execution and storage regions

Unified Memory Manager: Off-Heap Model

- ❑ Spark 1.6 began to introduce **Off-heap memory** ([SPARK-11389](#)).
- ❑ By default, **Off-heap** memory is **disabled**
- ❑ But we can enable it by the ***spark.memory.offHeap.enabled*** parameter
- ❑ We can also set the memory size by ***spark.memory.offHeap.size*** parameter

Unified Memory Manager: Off-Heap Model

Compared to the On-heap memory, the model of the Off-heap memory is relatively simple, including only Storage memory and Execution memory, and its distribution is shown in the following picture:



Executor Memory Architecture

Summary - Bigger Picture

Storage Memory of Executor=

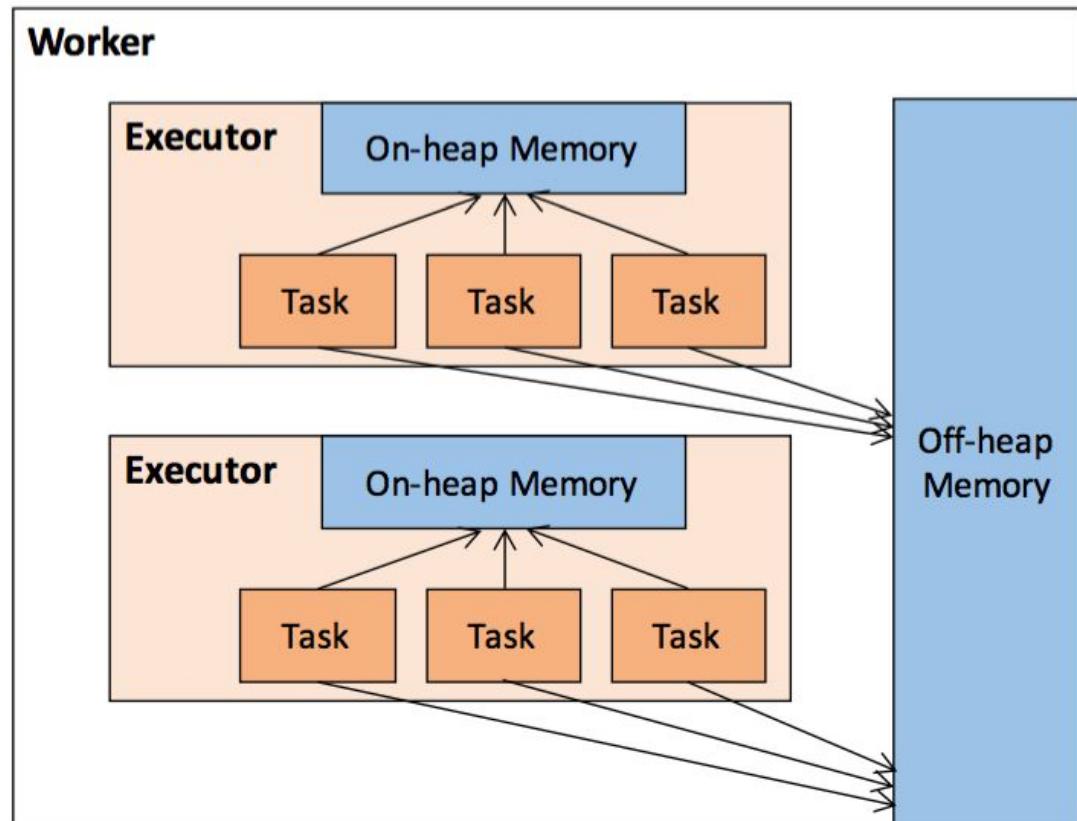
Storage Memory On-Heap +

Storage Memory Off-Heap

Execution Memory of Executor=

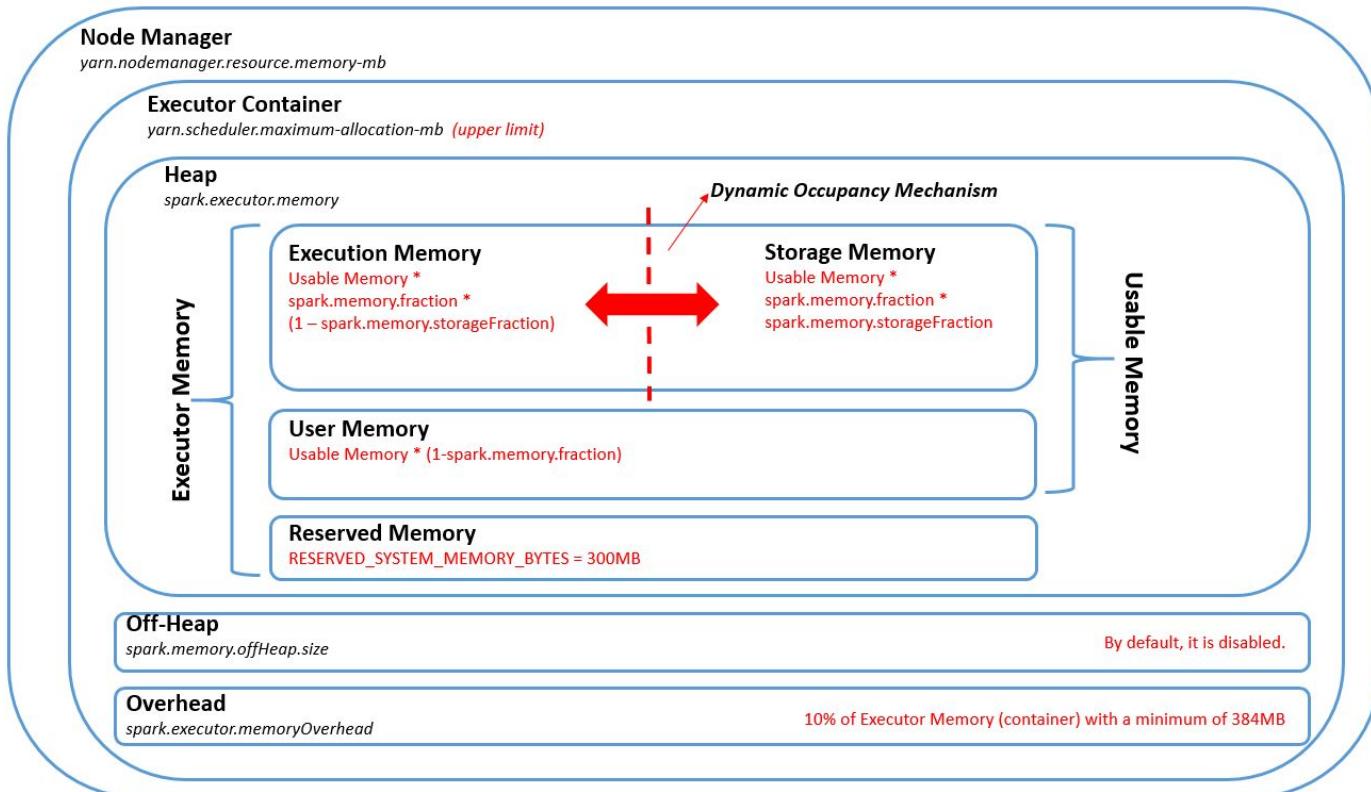
Execution Memory On-Heap +

Execution Memory Off-Heap



Executor Memory Architecture

Summary - Bigger Picture



Spark Optimization

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties**
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

Key Properties

Properties	Default Value	Description
spark.app.name	(none)	The name of your application. This will appear in the UI and in log data
spark.driver.cores	1	Number of cores to use for the driver process, only in cluster mode
spark.driver.maxResultSize	1g	Limit of total size of serialized results of all partitions for each Spark action (e.g. collect) in bytes. Should be at least 1M, or 0 for unlimited
spark.driver.memory	1g	Amount of memory to use for the driver process, i.e. where SparkContext is initialized, in the same format as JVM memory strings with a size unit suffix ("k", "m", "g" or "t") (e.g. 512m, 2g)

Key Properties

Properties	Default Value	Description
spark.driver.resource.{resourceName}.amount	0	Amount of a particular resource type to use on the driver. If this is used, you must also specify the spark.driver.resource
spark.driver.extraClassPath	(none)	Extra classpath entries to prepend to the classpath of the driver
spark.driver.extraJavaOptions	(none)	A string of extra JVM options to pass to the driver. This is intended to be set by users. For instance, GC settings or other logging
spark.driver.extraLibraryPath	(none)	Set a special library path to use when launching the driver JVM

Key Properties

Properties	Default Value	Description
spark.driver.userClassPathFirst	false	(Experimental) Whether to give user-added jars precedence over Spark's own jars when loading classes in the driver
spark.executor.memory	512MB	Amount of memory to use per executor process
spark.executor.instances	2	The number of executors to be run
spark.driver.memory	1024MB	Amount of memory to use for driver process

Key Properties

Properties	Default Value	Description
spark.executor.extraClassPath	(none)	Extra classpath entries to prepend to the classpath of executors. This exists primarily for backwards-compatibility with older versions of Spark
spark.executor.extraLibraryPath	(none)	Set a special library path to use when launching executor JVM's
spark.executor.logs.rolling.maxRetainedFiles	(none)	Sets the number of latest rolling log files that are going to be retained by the system. Older log files will be deleted. Disabled by default
spark.executor.logs.rolling.enableCompression	false	Enable executor log compression. If it is enabled, the rolled executor logs will be compressed. Disabled by default

Key Properties

Properties	Default Value	Description
spark.executor.logs.rolling.maxSize	(none)	Set the max size of the file in bytes by which the executor logs will be rolled over. Rolling is disabled by default
spark.executorEnv.[EnvironmentVariableName]	(none)	Add the environment variable specified by EnvironmentVariableName to the Executor process

For more Properties visit: <https://spark.apache.org/docs/latest/configuration.html>

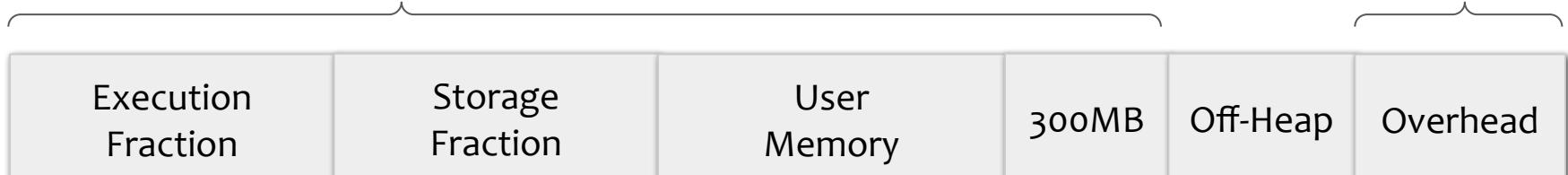
Summary of Executor Properties

--executor-memory

--conf spark.executor.memory

--conf spark.mesos.executor.memoryOverhead

--conf spark.yarn.executor.memoryOverhead



--conf spark.memory.fraction



--conf spark.memory.storageFraction

--conf spark.memory.offHeap.size



Summary of Driver Properties

--driver-memory

--conf spark.driver.memory

--conf spark.mesos.executor.memoryOverhead

--conf spark.yarn.executor.memoryOverhead

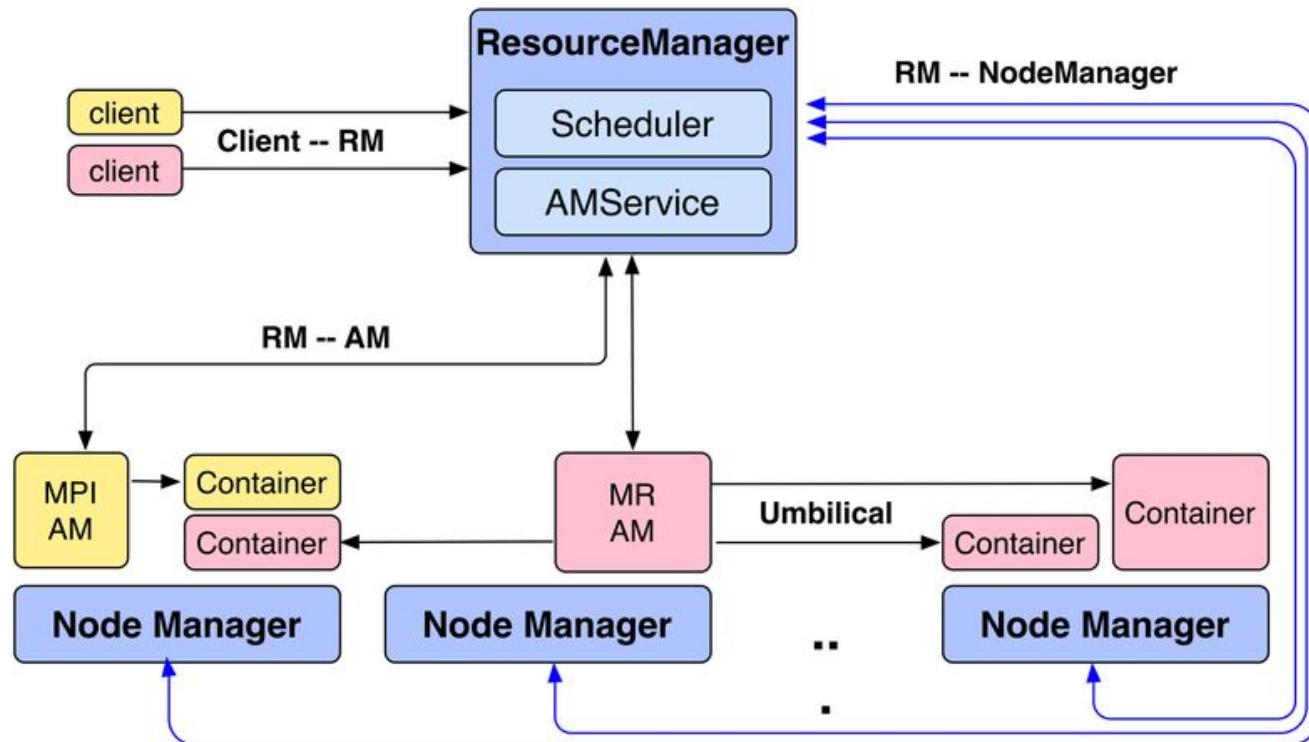


--conf spark.driver.maxResultSize=1G

Spark Optimization

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture**
- Resource Planning
- Discussion on Garbage Collection

Spark on YARN Detailed Architecture



Spark on YARN Detailed Architecture

YARN Container Memory:

- ❑ Spark Executor Memory + Overhead =>

spark.executor.memory + spark.yarn.executor.memoryOverhead

where

spark.yarn.executor.memoryOverhead = Max(384MB, 7% of spark.executor.memory)

Spark Optimization

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning**
- Discussion on Garbage Collection

Resource Planning: Cluster resource requirements

Consider a 10 node cluster with following configs and analyze different possibilities of executors-core-memory distribution:

Cluster Config:

- 10 Nodes
- 16 cores per Node
- 64GB RAM per Node
- 3 cores in each Node will be used by DataNode, NodeManager, OS
- 16 GB in each Node will be used by OS
- 1 GB in each Node will be used by DataNode
- 1 GB in each Node will be used by NodeManager

First Approach: Tiny executors [One Executor per core]:

- ❑ ‘--num-executors’ = ‘In this approach, we’ll assign one executor per core’
 - = ‘total-cores-in-cluster’
 - = ‘num-available-cores-per-node * total-nodes-in-cluster’
 - = $(16 - 3) \times 10 = 130$
- ❑ ‘--executor-cores’ = 1 (one executor per core)
- ❑ ‘--executor-memory’ = ‘amount of memory per executor’
 - = ‘mem-per-node/num-executors-per-node’
 - = $46\text{GB}/13 = 3.5 \text{ GB} (\text{approx})$

Second Approach: Fat executors (One Executor per node):

- ❑ ‘--num-executors’ = ‘In this approach, we’ll assign one executor per core’
= ‘total-available-cores-in-cluster’
= 13
- ❑ ‘--executor-cores’ = ‘one executor per node means all the available cores of the node’
= ‘total-available-cores-in-a-node’
= 13
- ❑ ‘--executor-memory’ = ‘amount of memory per executor’
= ‘mem-per-node/num-executors-per-node’
= $46\text{GB}/1 = 46\text{GB}$

Third Approach: Balance between Fat & Tiny:

- ❑ Let's assign 4 to 5 cores per executor=>
- ❑ --executor-core = 4 (for good HDFS throughput)
- ❑ So, Total available of cores in cluster= $13 \times 10 = 130$
- ❑ Total executors in the cluster = (total cores/num_cores_per_executor) = $130/4 = 32$ approx

So, recommended config is: 32 executors, 14GB memory each and 4 cores each!!

Resource Planning: Cluster resource requirements

- ❑ Leaving 1 executor for ApplicationManager => --num-executors = 29
- ❑ Number of executors per node = $30/10 = 3$
- ❑ Memory per executor = $64GB/3 = 21GB$
- ❑ Counting off heap overhead = 7% of 21GB = 3GB. So, actual --executor-memory = $21-3 = 18GB$

So, recommended config is: 29 executors, 18GB memory each and 5 cores each!!

Resource Planning: Dynamic Resource Allocation

- ❑ Explicit application-wide allocation of executors can have its downsides
- ❑ There are some situations in which there is no need to have a specific number of executors for the duration of the whole computation but would instead want some scaling
- ❑ There can be not enough resources available on the cluster at a given time but would like to run our computation regardless, there may be processing a transformation that requires much less resources and would not like to hog more than that is required, etc
- ❑ This is where dynamic resource allocation comes in

Resource Planning: Dynamic Resource Allocation

Dynamic Allocation Use Cases

- ❑ Long-running ETL jobs
- ❑ Interactive Application (Jupyter Notebook)
- ❑ Applications with large shuffles

Resource Planning: Dynamic Resource Allocation

- ❑ Better Resource Utilization
- ❑ Good for multi-tenant environment

Resource Planning: Dynamic Resource Allocation

spark.dynamicAllocation.enabled= false

Whether to use dynamic resource allocation

spark.dynamicAllocation.executorIdleTimeout= 60s

If an executor has been idle for more than this duration, the executor will be removed

spark.dynamicAllocation.cachedExecutorIdleTimeout= infinity

If an executor which has cached data blocks has been idle for more than this duration, the executor will be removed

Resource Planning: Speculative Execution

- ❑ Although we do all the optimizations, there are situations in which we get poor performance of tasks
- ❑ For these situations, there is need to instruct Spark to re-execute tasks automatically after it detects such stragglers
- ❑ To do this, enable the ***spark.speculation*** setting
- ❑ **Speculation** is set to true, then **Spark** will identify the slow running tasks and run the **speculative** tasks on other nodes to complete the job more quickly
- ❑ The tasks that finish first are accepted and the second task gets killed by **Spark**

Configuring Speculative Execution

spark.speculation= false

If set to “true”, performs speculative execution of tasks

spark.speculation.interval= 100ms

spark.speculation.quantile= 0.75

Fraction of tasks which must be complete before speculation is enabled for a particular stage

Tune RPC Server Threads

- ❑ Frequent driver OOM when running many tasks in parallel
- ❑ Huge backlog of RPC requests built on Netty server of the driver
- ❑ Increase RPC server thread to fix OOM

spark.rpc.io.serverThreads= 64

Resource Planning: Data Locality

- ❑ Spark is a data parallel processing framework, which means it will execute tasks as close to where the data lives as possible (i.e. **minimize data transfer**)
- ❑ For optimizing processing tasks, Spark tries to place the execution code as close as possible to the processed data. This is called ***data locality***

Spark describes data locality concept in 5 levels:

- PROCESS_LOCAL** - data and processing are localized on the same JVM
- NODE_LOCAL** - data and processing are in the same node but on different executor
This level is slower than the previous one because it has to move the data between processed
- RACK_LOCAL** - data is located in other node than processing but both nodes are on the same rack. Obviously here the data is moved through the network
- NO_PREF** - means no locality preference
- ANY** - data is elsewhere but not on the same rack

Resource Planning: Data Locality

Spark.locality.wait

Its value is defined in time units (e.g. s for seconds) and indicates how long Spark can wait to acquire the data on given locality

The configuration enhances more fine-grained control with the entries reserved for:

Spark.locality.wait.process (PROCESS_LOCAL)

spark.locality.wait.node (NODE_LOCAL)

spark.locality.wait.rack (RACK_LOCAL)

Resource Planning: Data Locality

org.apache.spark.scheduler.TaskLocation

It is the interface representing the task location. Among available implementations we can distinguish: **ExecutorCacheTaskLocation**, **HostTaskLocation** and **HDFSCacheTaskLocation**

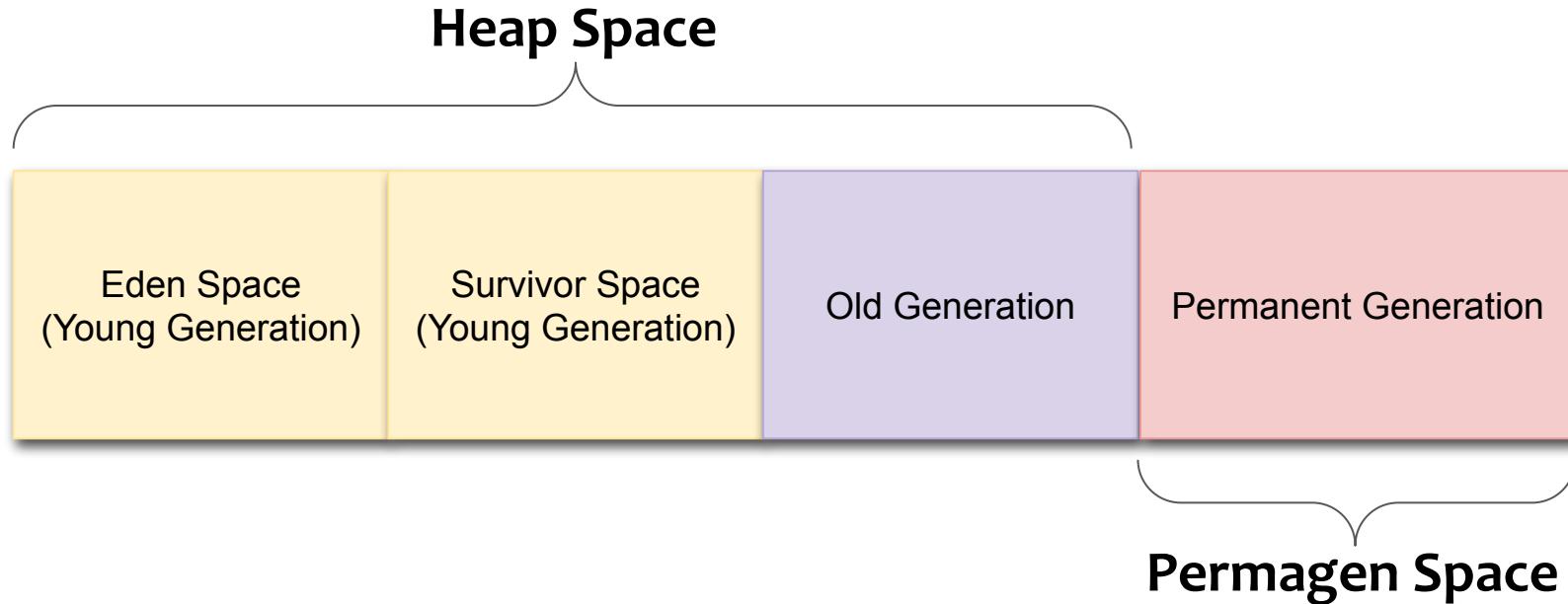
Spark Optimization

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection**

Garbage Collection Overview

- ❑ Garbage Collection happens in Heap Space
- ❑ Heap space is divided into Young and Old generations
- ❑ The young generation consists of an area called Eden along with two smaller survivor spaces
- ❑ Newly created objects are initially allocated in Eden
- ❑ Each time a *minor GC* occurs, the JVM copies live objects in Eden to an empty survivor space and also copies live objects in the other survivor space that is being used to that empty survivor space
- ❑ Objects that have survived some number of minor collections will be copied to the old generation

Java (JVM) Memory Model



Java Heap Space:

Java objects are instantiations of Java Classes. Our JVM has an internal representation of those Java objects and those internal representations are stored in the heap. This java heap memory is divided again into regions, called generations

- ❑ **Young Generation:** It is the place where lived for short period and divided into two spaces:
 - ❑ **Eden Space:** When object created using new keyword memory allocated on this space. Newly created Objects are usually located in this space. When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to the survivor spaces
 - ❑ **Survivor Space:** This is the pool which contains objects which have survived after java garbage collection from Eden Space

Garbage Collection Overview

- ❑ **Tenured Generation:** This pool is basically contain tenured and virtual(reserved) space and will be holding those objects which survived after garbage collection from Young Generation. This memory pool contains objects which survived after multiple collection means object which survived after garbage collection from Survivor space

Java PermGen Space

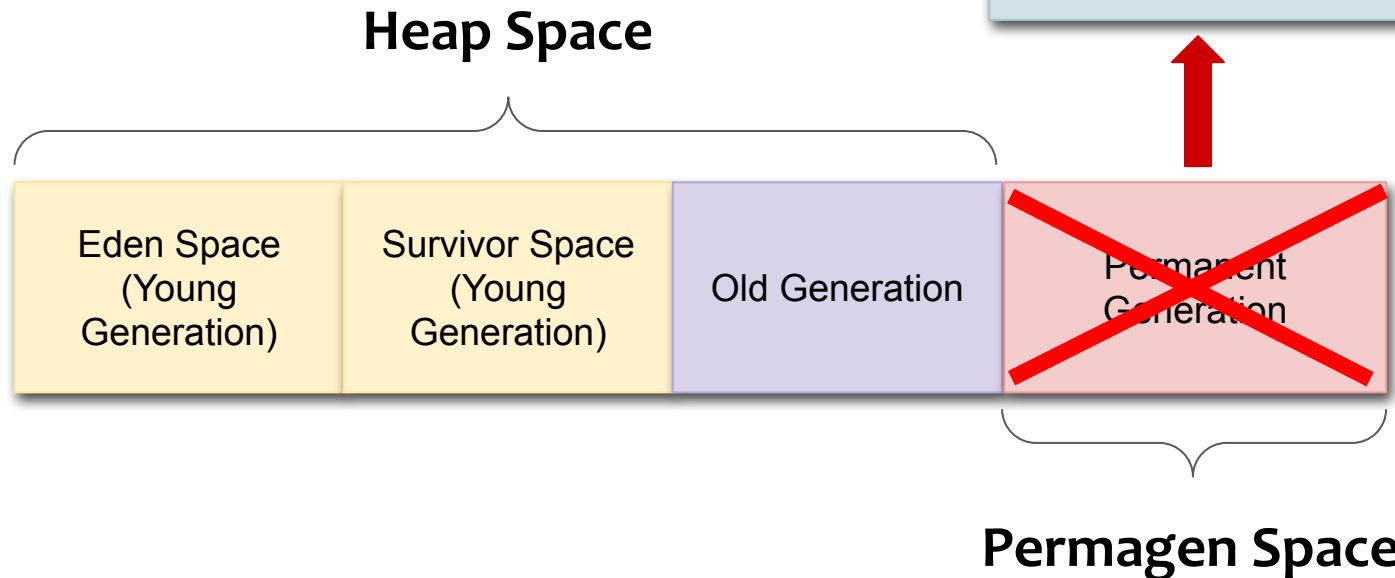
JVM also has an internal representation of the Java Classes and those are stored in the permanent generations. Class definitions are stored here, as are static instances. The PermGen is garbage collected like the other parts of the heap

PermGen contains meta-data of the classes and the objects i.e. pointers into the rest of the heap where the objects are allocated. The PermGen also contains Class-loaders which have to be manually destroyed at the end of their use else they stay in memory and also keep holding references to their objects on the heap. PermGen always has a fixed maximum size

Java Metaspace

With JDK8, the permGen Space has been removed. This metadata is now stored in a native memory are called as “MetaSpace”. This memory is not a contiguous Java Heap memory. It allows for improvements over PermGen space in Garbage Collection, auto tuning, concurrent de-allocation of metadata

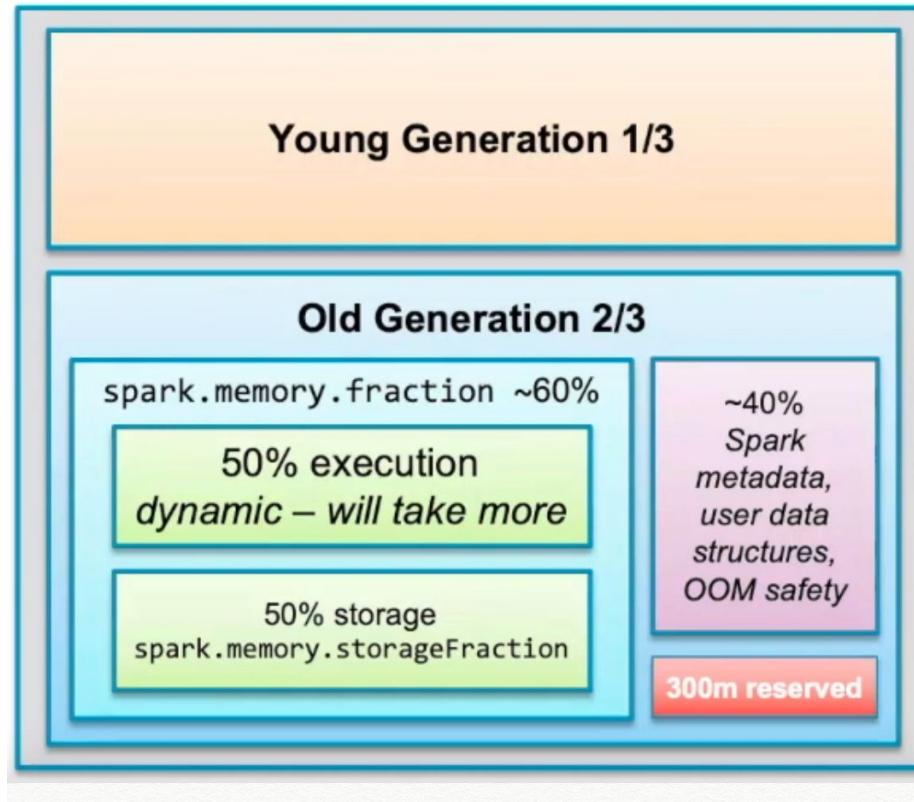
Java (JVM) Memory Model



Garbage Collection Overview

PermGen Space	MetaSpace
PermGen always has a fixed maximum size	MetaSpace by default auto increases its size depending on the underlying OS
Contiguous Java Heap Memory	Native Memory(provided by underlying OS)
Max size can be set using XX:MaxPermSize	Max size can be set suing XX:MetaspaceSize
Comparatively inefficient Garbage collection. Frequent GC pauses and no concurrent deallocation	Comparatively efficient Garbage collection. Deallocation class data concurrently and not during GC pauses

Discussion on Garbage Collection



Tuning Memory Configurations

Garbage Collection Tuning

- ❑ Large contiguous in-memory buffers allocated by Spark's shuffle internals
- ❑ G1GC suffers from fragmentation due to Humongous Allocations, if object size is more than 32 MB (Maximum region size of G1GC)
- ❑ Use parallel GC instead of G1GC

spark.executor.extraJavaOptions= -XX:ParallelGCThreads=4 -XX:+UseParallelGC

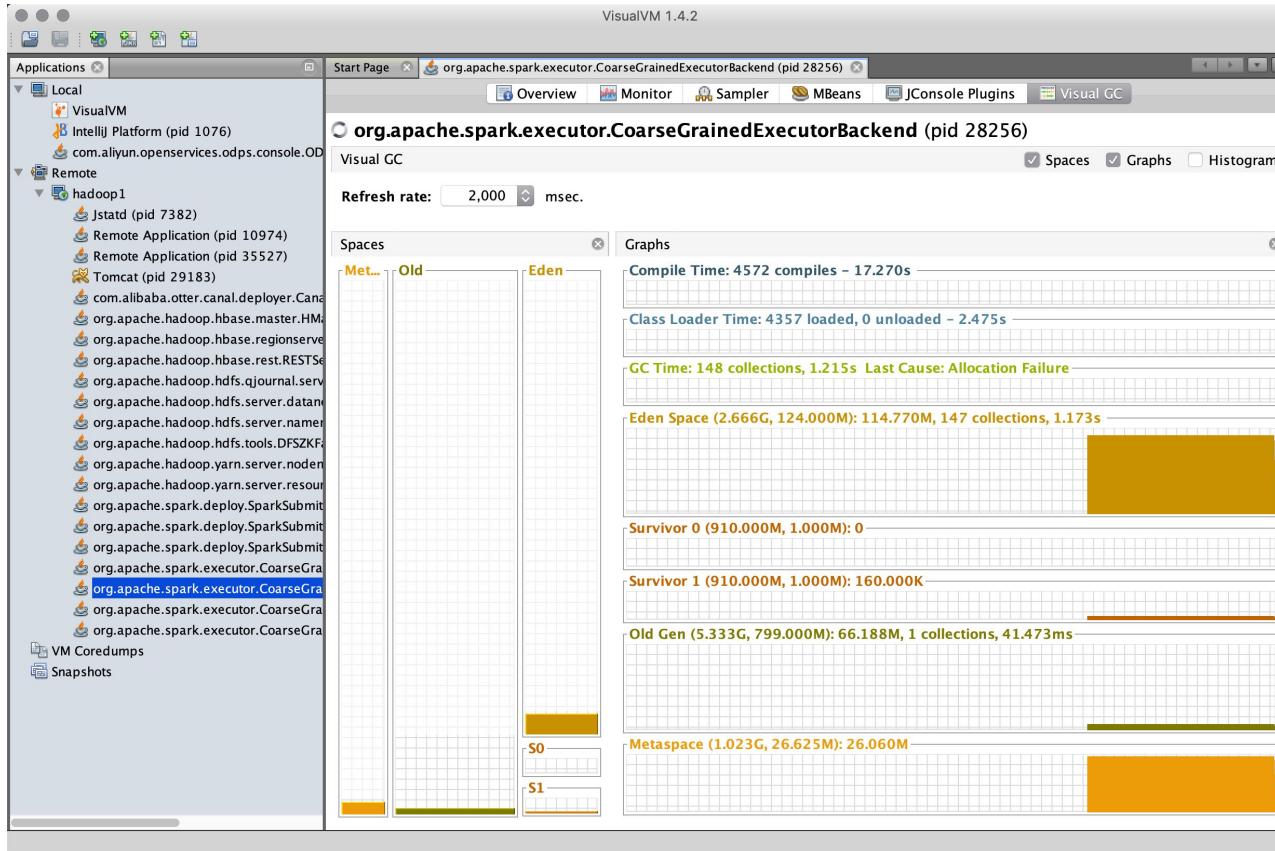
Memory Problems

- ❑ Symptoms:
 - ❑ Inexplicably bad performance
 - ❑ Inexplicable executor/machine failures (can indicate too many shuffle files too)
- ❑ Diagnosis:
 - ❑ Set spark.executor.extraJavaOptions to include
 - ❑ XX:+PrintGCDetails
 - ❑ -XX:+HeapDumpOnOutOfMemoryError
- ❑ Resolution:
 - ❑ Increase spark.executor.memory
 - ❑ Increase number of partitions
 - ❑ Re-evaluate program structure(!)

Field Guide to Spark GC Tuning

- ❑ Lots of minor GC - easy fix
 - ❑ Increase Eden space (high allocation rate)
- ❑ Lots of major GC - need to diagnose the trigger
 - ❑ Triggered by promotion - increase Eden space
 - ❑ Triggered by Old Generation filling up - increase Old Generation space or decrease spark.memory.fraction
- ❑ Full GC before stage completes
 - ❑ Trigger minor GC earlier and more often

Discussion on Garbage Collection



Spark Optimization

- Partitions
- Executors
- Key Execution Modes
- Parallel Operations
- Tasks and Stages
- Narrow vs Wide Dependency
- Spark UI
- Executor Memory Architecture
- Key Properties
- Spark on YARN Detailed Architecture
- Resource Planning
- Discussion on Garbage Collection

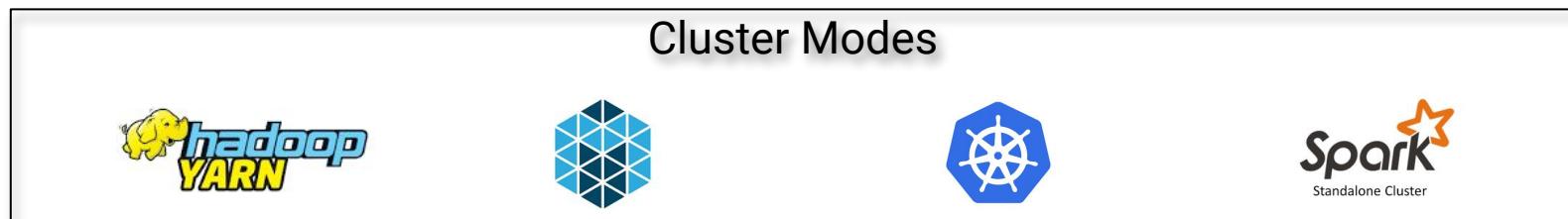
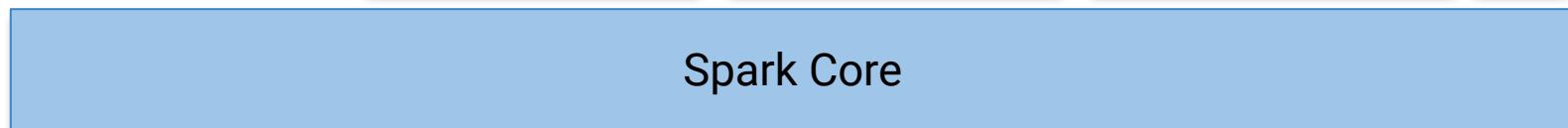
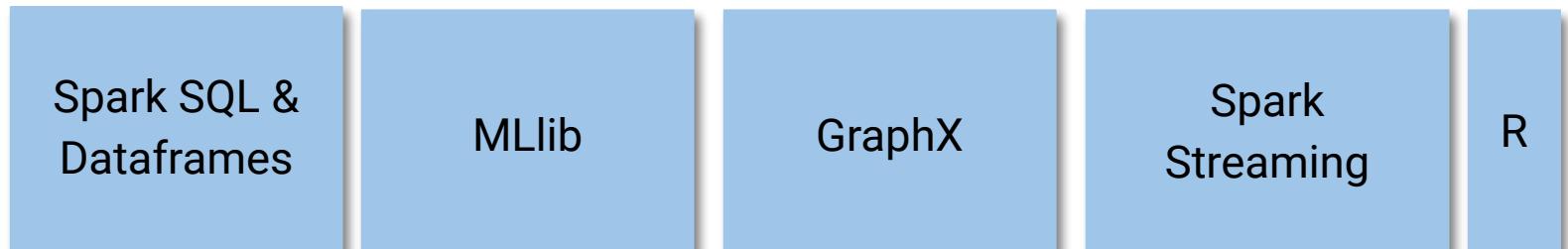
Agenda:

- Spark Physical Execution
- Clustering with Spark

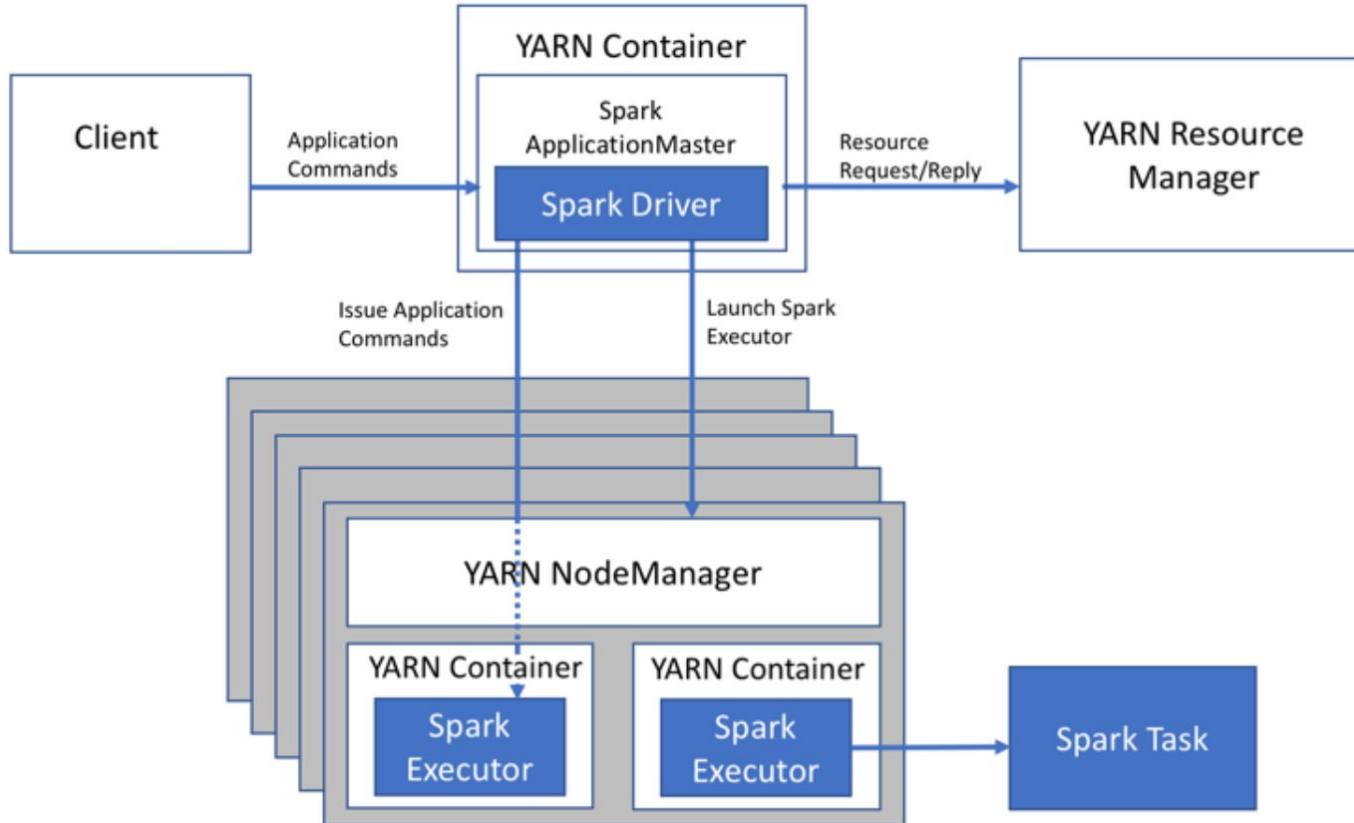
Spark Optimization

- ❑ Running a Spark cluster
- ❑ Managing memory/cores across a Spark cluster

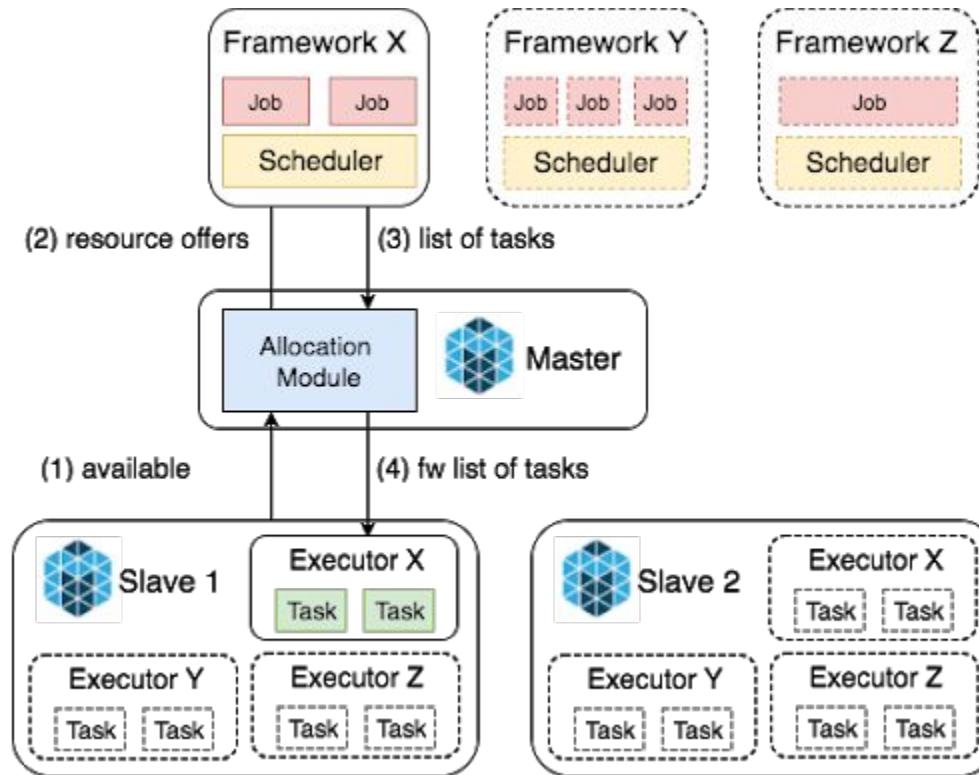
Running a Spark Cluster



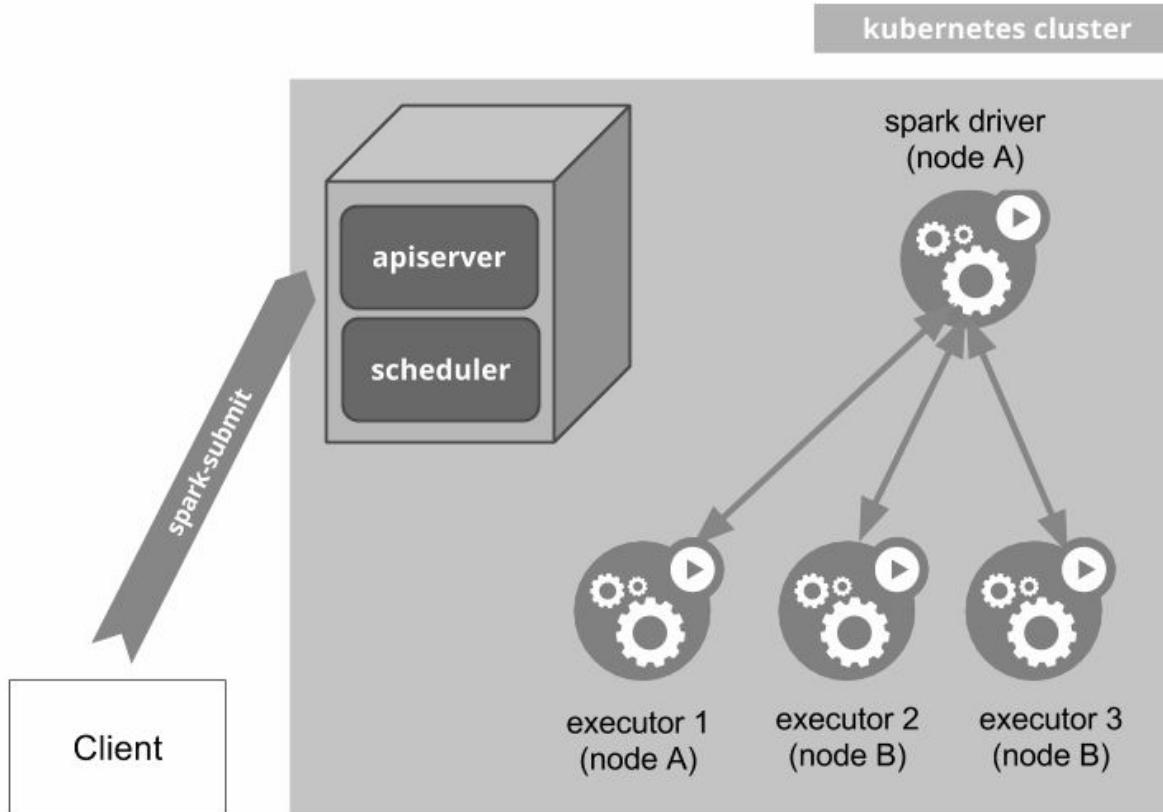
Running a Spark Cluster: Spark on YARN



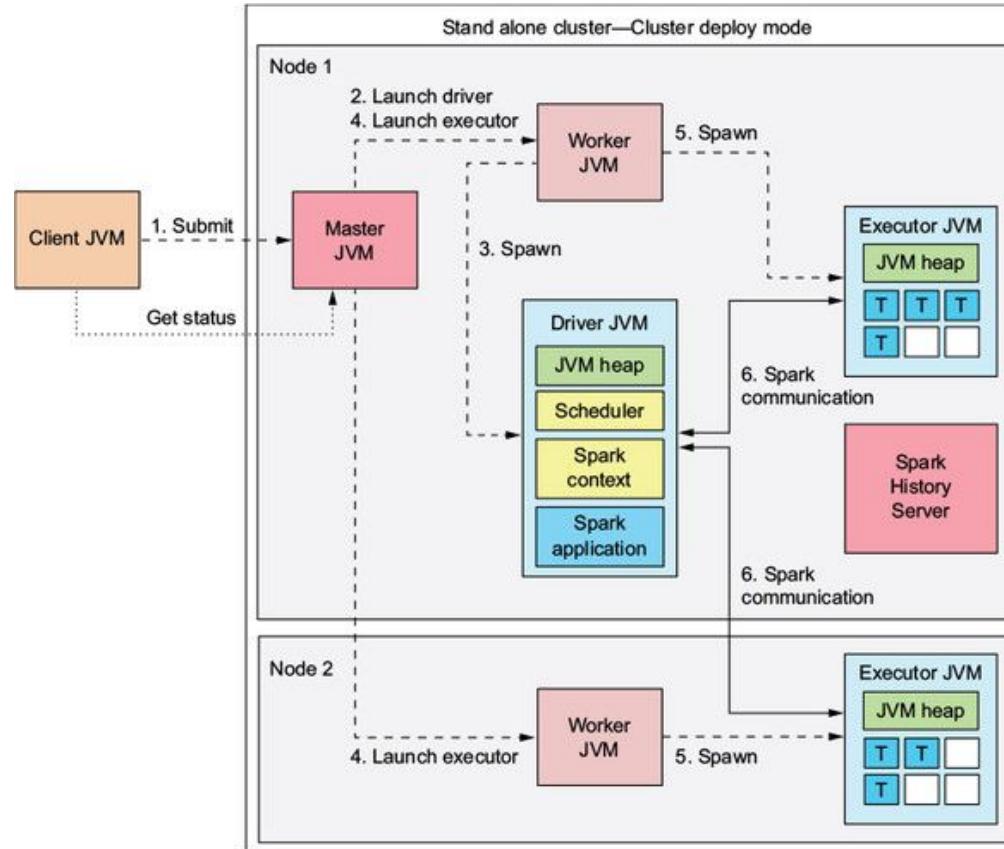
Running a Spark Cluster: Spark on Mesos



Running a Spark Cluster: Spark on Kubernetes



Running a Spark Cluster: Spark Standalone



Running a Spark Cluster

Set the master option to specify cluster type

- ❑ **yarn**
- ❑ **local[*]** runs locally with as many threads as cores (default)
- ❑ **local[n]** runs locally with n threads
- ❑ **local** runs locally with a single thread

```
$ pyspark --master yarn
```

```
$ pyspark --master local[*]
```

```
$ pyspark --master mesos://mesosmaster:5050
```

```
$ Kubectl exec -it <spark pod name> -- /bin/bash
```

Running a Spark Cluster

Set the master option to specify cluster type

- Mesos
- Kubernetes

```
$ spark-submit --master yarn
```

```
$ spark-submit --master local[*]
```

```
$ spark-submit --master mesos://mesosmaster:5050
```

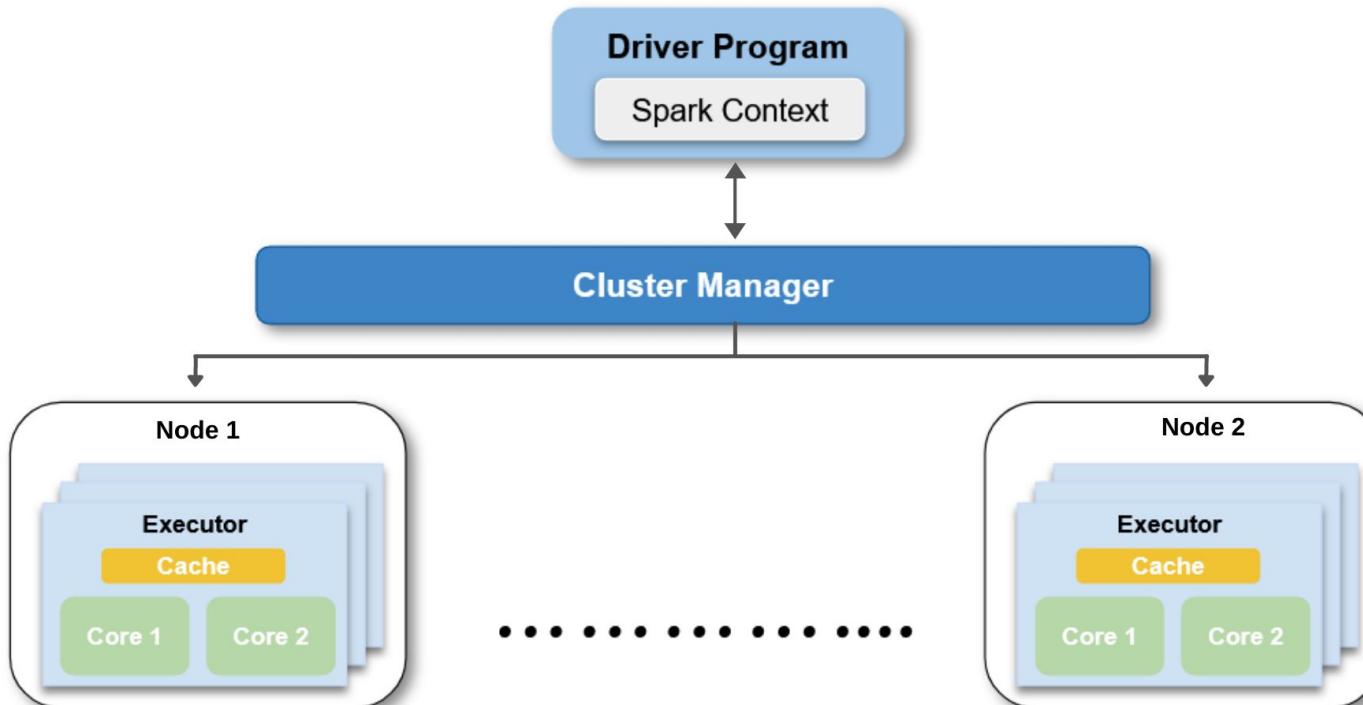
```
$ Kubectl exec -it <spark pod name> -- /bin/bash
```

Spark Optimization

- Running a spark cluster
- Managing memory/cores across a spark cluster**

Clustering with Spark

Managing memory/cores across a Spark cluster



Managing memory/cores across a spark cluster

Similar to managing memory, using **--executor-cores** flag when invoking **spark-submit**, **spark-shell**, and **pyspark** from the command line, or by setting the **spark.executor.cores** property in the **spark-defaults.conf** file or on a **SparkConf** object

Spark Optimization

- Running a spark cluster
- Managing memory/cores across a spark cluster

Agenda:

- Spark Physical Execution
- Clustering with Spark



Optimizations

Agenda:

- ❑ High Performance Spark applications
- ❑ Working with Spark
- ❑ Caching and Checkpointing
- ❑ Joins and more

- ❑ **Performance tuning**
- ❑ Performance tuning metrics
- ❑ SQL performance tuning

Performance tuning techniques

Below are some of the key techniques of tuning Spark applications:

- ❑ Serialization
- ❑ Data Format
- ❑ Caching & Checkpointing
- ❑ Joins
- ❑ Dynamic Resource Allocation
- ❑ Data Skew
- ❑ Partitioning
- ❑ Bucketing
- ❑ Avoid UDFs
- ❑ Spark Performance Properties

Performance tuning: Data Serialization

A **serialization** framework helps you convert objects into a stream of bytes and vice versa in new computing environment. This is very helpful when you try to save objects to disk or send them through networks. **Serializing data in an optimal manner helps in getting good performance**



Performance tuning: Data Serialization

Data Serialization:

Spark supports below serialization libraries:

- Java** Serialization
- Kryo** Serialization
- Tungsten** Serialization

Default one is Java serialization

Performance tuning: Data Serialization

Data Serialization:

Java Serialization:

- It is very easy to use
- It can be implemented using **Serializable** Interface
- But it is very Inefficient

Performance tuning: Data Serialization

Data Serialization:

Kryo Serialization:

- ❑ As Java serializer is inefficient, i.e. why it is advisable to switch to the second supported serializer, [Kryo](#)
- ❑ It can be done by setting **spark.serializer** to **org.apache.spark.serializer.KryoSerializer**
- ❑ Kryo is much more efficient and does not require the classes to implement Serializable
- ❑ However, in very rare cases, Kryo can fail to serialize some classes, which is the sole reason why it is still not Spark's default

Performance tuning: Data Serialization

Data Serialization:

RDD	Average time	Min. time	Max. time
java	65990.9ms	64482ms	68148ms
kryo	30196.5ms	28322ms	33012ms

Data Serialization:

Lineage (Java):

```
(42) MapPartitionsRDD[1] at map at <console>:25 [Disk Serialized 1x Replicated]
|       CachedPartitions: 42; MemorySize: 0.0 B; ExternalBlockStoreSize: 0.0 B; DiskSize: 3.8 GB
|   ParallelCollectionRDD[0] at parallelize at <console>:25 [Disk Serialized 1x Replicated]
```

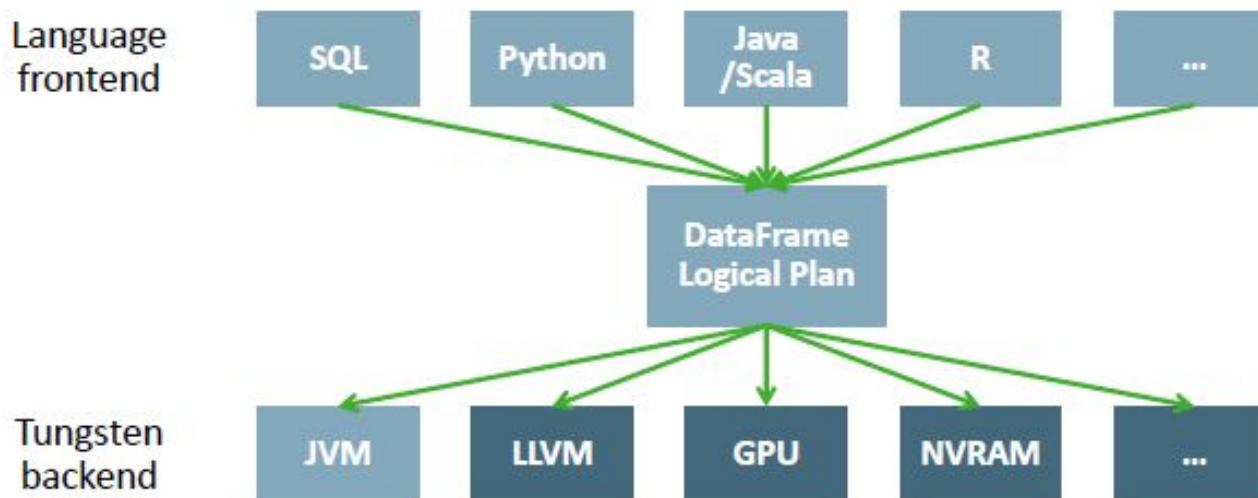
Lineage (Kryo):

```
(42) MapPartitionsRDD[1] at map at <console>:25 [Disk Serialized 1x Replicated]
|       CachedPartitions: 42; MemorySize: 0.0 B; ExternalBlockStoreSize: 0.0 B; DiskSize: 3.1 GB
|   ParallelCollectionRDD[0] at parallelize at <console>:25 [Disk Serialized 1x Replicated]
```

Performance tuning: Data Serialization

Tungsten Serialization

- ❑ Improves Spark execution by optimizing CPU / memory usage
 - Understands and optimizes for hardware architectures
 - Tunes optimizations for Spark's characteristics



Performance tuning: Data Serialization

- ❑ Binary representation of Java objects (Tungsten row format)
 - Different from Java serialization and Kryo
- ❑ Advantages include:
 - Much smaller size than native Java serialization
 - Supports off-heap allocation
 - Structure supports Spark operations without deserialization
 - e.g. You can sort data while it remains in the binary format
 - Avoids GC overhead
- ❑ Result:
 - Much faster, less memory, less CPU
 - Can process much larger datasets

Performance tuning: Data Serialization

- ❑ Modern hardware has many types of memory
 - Main memory
 - Multiple levels of cache (L1, L2, ...)
 - Registers
 - These all have different characteristics
- ❑ Tungsten is aware of the different types of memory and different hardware architectures
 - It generates code that is optimized to utilize modern hardware
- ❑ Result: Much faster performance
 - For instance, it will use a cache-aware sorting algorithm
 - Giving 3x improvement over the non-cache aware version

Performance tuning: Data Serialization

- ❑ Tungsten optimization to improve execution performance
 - Collapses a query expression into a single optimized function
- ❑ For example suppose we were filtering on the following filter('age>25 && 'age<50)
 - Code generation dynamically generates bytecode for this
 - All code contained in one function
 - Instead of classic interpretation
 - With boxing of primitives, polymorphic function calls ...
- ❑ Result:
 - Eliminates virtual function calls
 - Leverages CPU registers for intermediate data
 - Can meet/exceed performance of hand-tuned function for a task

Performance tuning: Data Serialization

- ❑ Generally, don't think about it much
 - Just enjoy the benefits
- ❑ Ahhh—Except sometimes for lambdas!
 - These require Java objects, and can't be executed with data in the Tungsten format
 - This adds a layer of complexity when they interact with Tungsten
- ❑ Let's look at it in action
 - We can see it at work in the Physical Plan
- ❑ We will look at a DataFrame and Dataset Physical plan
 - Illustrating what Tungsten is doing
 - Exploring some issues with lambdas

Performance tuning: Data Serialization

- ❑ AppendColumnsWithObject converts to Tungsten format
 - After MapPartitions, which must work on Java objects
 - Note that AppendColumnsWithObject does NOT use code gen
 - Less efficient than the previous plan

```
// Convert linesDF to Dataset, set up word count, and look at plan
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
  s).count.explain
== Physical Plan ==
*HashAggregate(keys=[value#496], functions=[count(1)])
+- Exchange hashpartitioning(value#496, 200)
  +- *HashAggregate(keys=[value#496], functions=[partial_count(1)])
    +- *Project [value#496]
      +- AppendColumnsWithObject <function1>, [...]
        +- MapPartitions <function1>, obj#492: java.lang.String
          +- Scan ExternalRDDScan[obj#483]
```

Performance tuning: Data Serialization

- ❑ We've made a trivial change when loading the data
 - Naming our input row "line" instead of the default "value"
 - Note how this adds extra work in our plan
 - There is an extra serialize (to Tungsten format) then deserialize (to Java format) step —the flatMap lambda needs a Java object
 - This is unexpected, and adds extra overhead (1)

```
// Trivial change - name column "line" instead of default "value"
val linesDF = sc.parallelize(Seq("Twinkle twinkle" ... )).toDF("line")
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
s).count.explain
== Physical Plan == // rest of plan as before
+- *Project [value#520]
  +- AppendColumnsWithObject <function1>, ... AS value#520]
  +- MapPartitions <function1>, obj#516: java.lang.String
  +- DeserializeToObject line#509.toString, obj#515: java.lang.String
  +- *Project [value#507 AS line#509]
  +- *SerializeFromObject [... AS value#507]
  +- Scan ExternalRDDScan[obj#506]
```

Performance tuning: Data Serialization

- ❑ AppendColumnsWithObject converts to Tungsten format
 - After MapPartitions, which must work on Java objects
 - Note that AppendColumnsWithObject does NOT use code gen
 - Less efficient than the previous plan

```
// Convert linesDF to Dataset, set up word count, and look at plan
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
  s).count.explain
== Physical Plan ==
*HashAggregate(keys=[value#496], functions=[count(1)])
+- Exchange hashpartitioning(value#496, 200)
  +- *HashAggregate(keys=[value#496], functions=[partial_count(1)])
    +- *Project [value#496]
      +- AppendColumnsWithObject <function1>, [...]
        +- MapPartitions <function1>, obj#492: java.lang.String
          +- Scan ExternalRDDScan[obj#483]
```

Performance tuning: Data Format

Data Format:

When you are designing your datasets for your application, ensure that you are making the best use of the file formats available with Spark. Some things to consider:

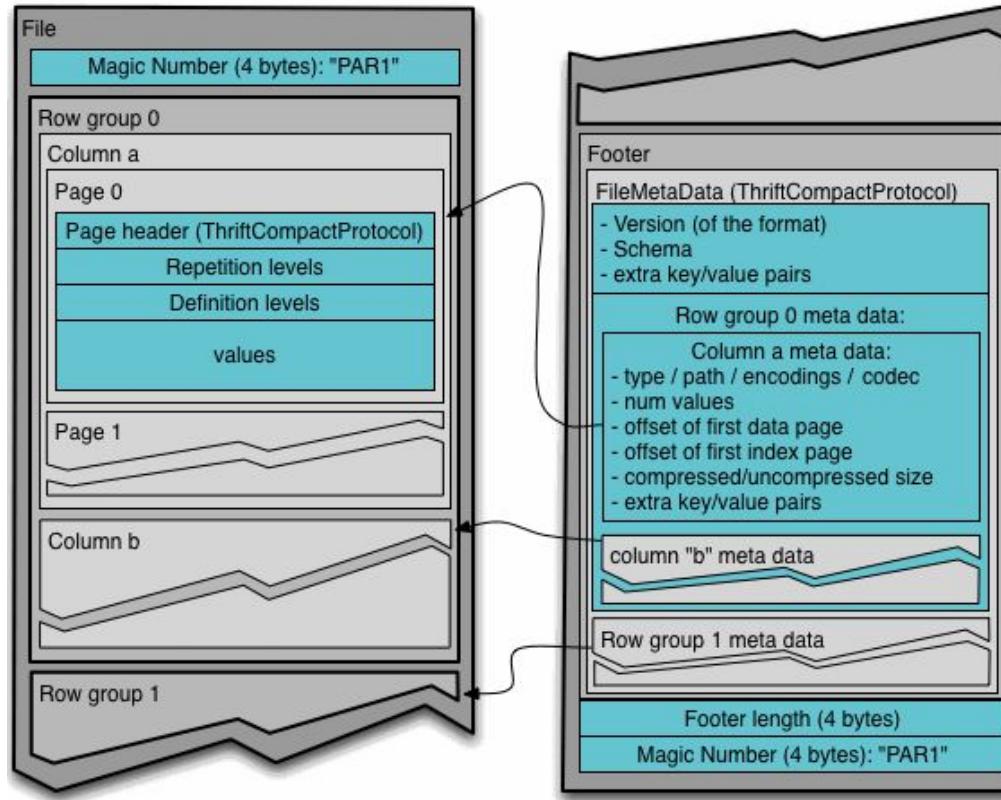
- ❑ Spark is optimized for Apache Parquet and ORC for read throughput. Spark has vectorization support that reduces disk I/O. Columnar formats work well
- ❑ Use the Parquet file format and make use of compression
- ❑ There are different file formats and built-in data sources that can be used in Apache Spark. Use splittable file formats
- ❑ Ensure that there are not too many small files. If you have many small files, it might make sense to do compaction of them for better performance

Data Format: Parquet

- ❑ Apache Parquet is a free and open-source column-oriented data storage format of the Apache Hadoop ecosystem
- ❑ It is similar to the other columnar-storage file formats available in Hadoop namely RCFile and ORC
- ❑ It gives the fastest read performance with Spark
- ❑ *Parquet* arranges data in columns, putting related values close to each other to optimize query performance, minimize I/O, and facilitate compression

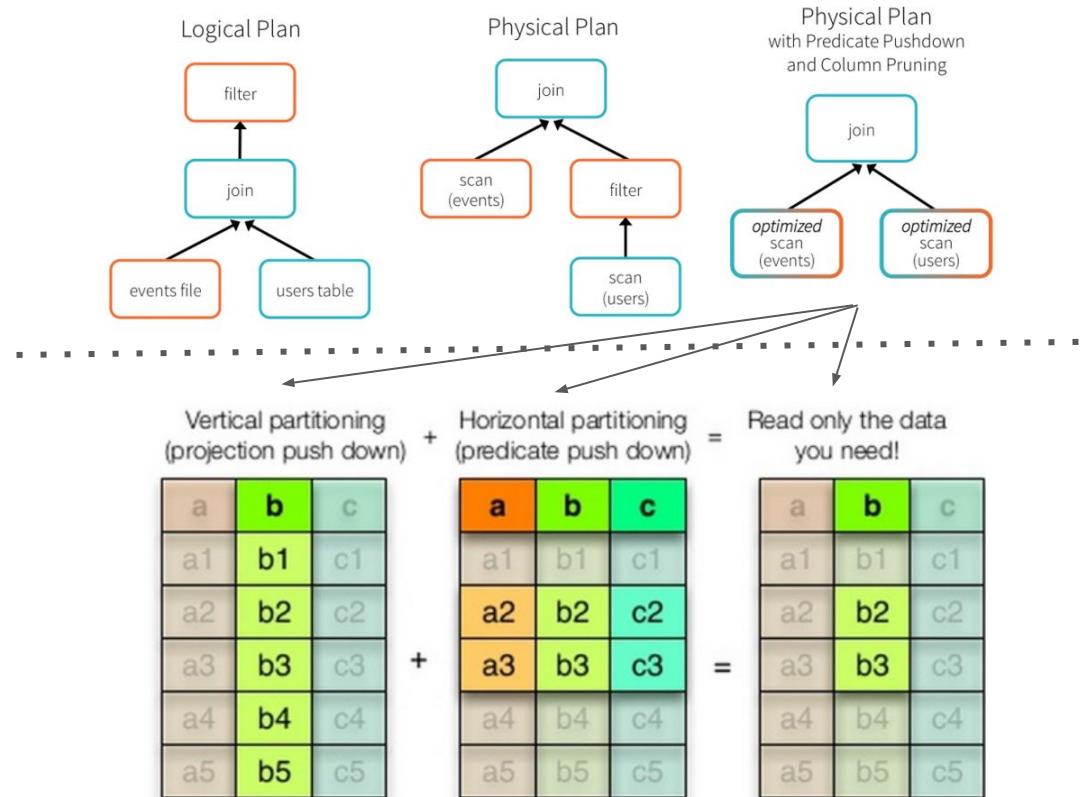
Performance tuning: Data Format

Data Format: Parquet



Performance tuning: Data Format

Data Format: Parquet: How Parquet Work?



Data Format: Parquet: How Parquet Work?

Column Pruning and Predicate Pushdown

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3

+

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3

=

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3

Vertical partitioning
(projection push down)

Horizontal partitioning
(projection push down)

Read only data you need!!

Performance tuning: Data Format

Data Format: Parquet: How Parquet Work?

- ❑ Parquet can implements column pruning and predicate pushdown (filters based on stats) which is simply a process of only selecting the required data for processing when querying a huge table
- ❑ It prevents loading unnecessary parts of the data in-memory and reduces network usage

Performance tuning: Data Format

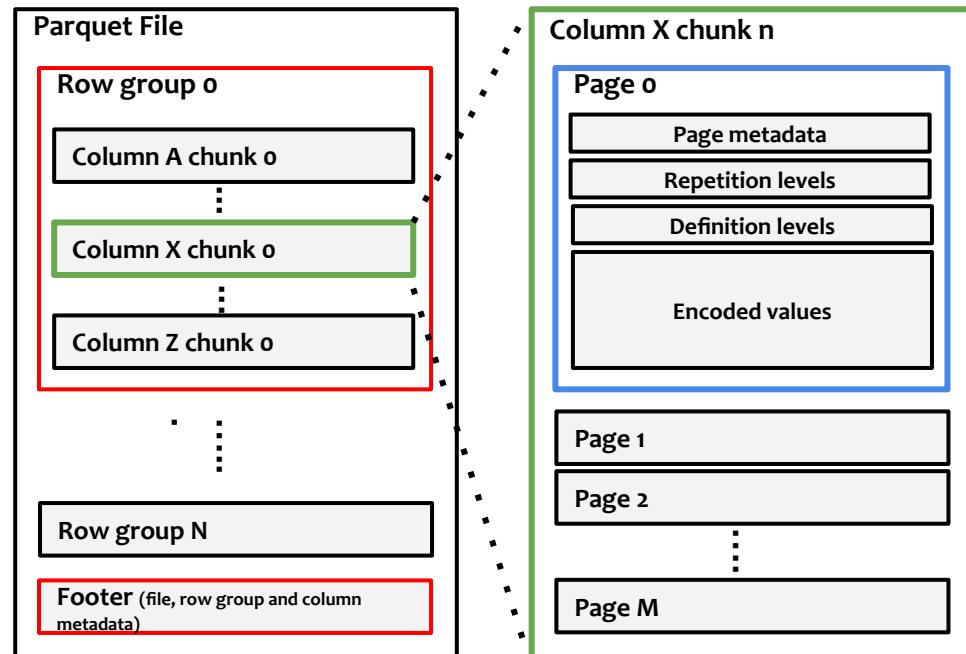
Data Format: Parquet: Why to choose Parquet?

Properties	CSV	JSON	Parquet	AVRO
Columnar	✗	✗	✓	✗
Splittable	✓	✓	✓	✓
Human Readable	✓	✓	✗	✗
Complex Data Structure	✗	✓	✓	✓
Schema Evolution	✗	✗	✓	✓

Data Format: Parquet: How Parquet store Data?

□ Data Organization:

- Row Groups(default 128MB)
- Column Chunks
- Pages(default 1MB)
 - Metadata
 - Min
 - Max
 - Count
 - Rep/Def Level
 - Encoded Values



Performance tuning: Data Format

Data Format: Parquet: How Parquet store Data?

- ❑ Parquet generally, don't store single file on disk usually
- ❑ Logical directory is defined by a root directory
 - ❑ Root dir contains one or multiple files

```
./example_parquet_file/  
./example_parquet_file/part-00000-87439b68-7536-44a2-9eaa-1b40a236163d-c000.snappy.parquet  
./example_parquet_file/part-00001-ae3c183b-d89d-4005-a3c0-c7df9a8e1f94-c000.snappy.parquet
```

- ❑ Or contains subdirectory structure with files in leaf directory

```
./example_parquet_file/  
./example_parquet_file/country=Netherlands/  
./example_parquet_file/country=Netherlands/part-00000-...-475b15e2874d.c000.snappy.parquet  
./example_parquet_file/country=Netherlands/part-00001-...-c7df9a8e1f94.c000.snappy.parquet
```

Data Format: Parquet: Optimizations

PLAIN

- Fixed-width: back-to-back
- Non fixed-width: length prefixed

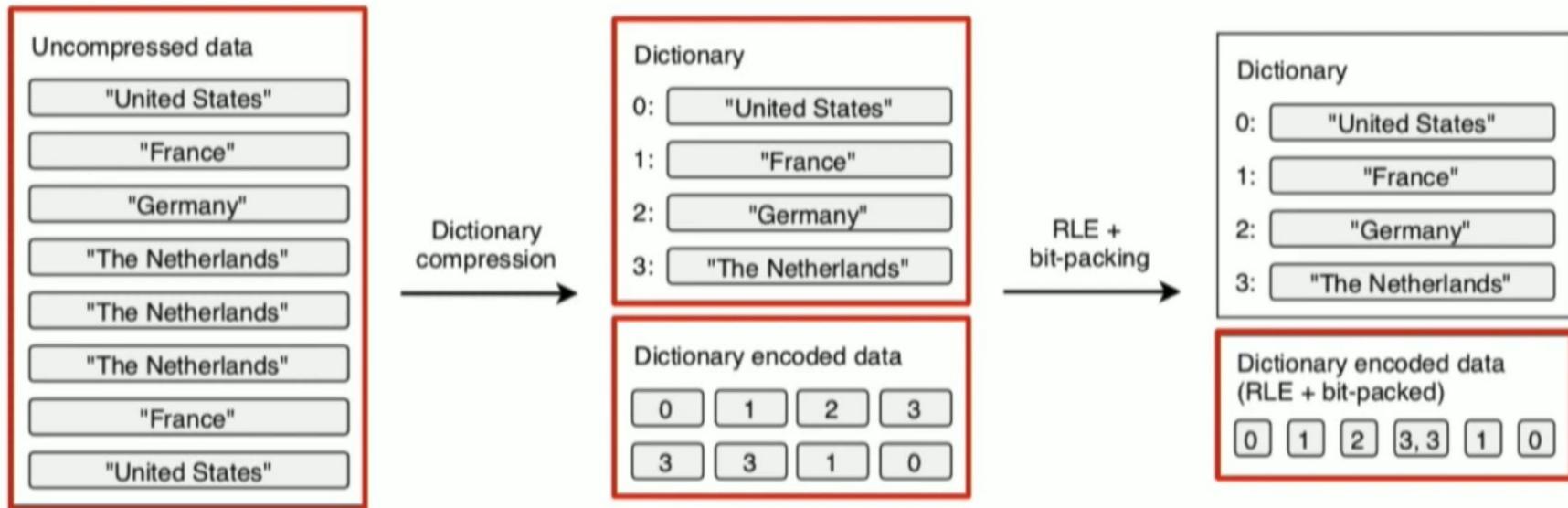
Plain encoded data						
13	"United States"	6	"France"	7	"Germany"	
15	"The Netherlands"	15	"The Netherlands"	15		
"The Netherlands"	6	"France"	13	"United States"		

RLE_DICTIONARY

- Run-length encoding + bit-packing + dictionary compression
- Assumes duplicate and repeated values

Data Format: Parquet: Optimizations

□ RLE_DICTIONARY



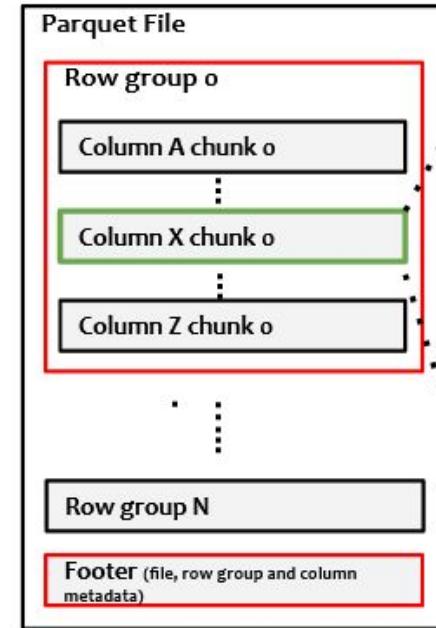
Data Format: Parquet: Optimizations

Dictionary Encoding

- ❑ Smaller files means less I/O
- ❑ Note: single dictionary per column chunk, size limit

Dictionary too big?

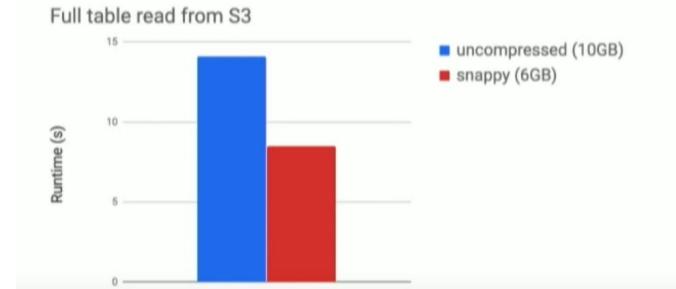
Automatic fallback to PLAIN...



Data Format: Parquet: Optimizations

Page Compression

- ❑ Compression of entire pages
 - ❑ Compression schemes (snappy, gzip, lzo, etc)
 - ❑ `Spark.sql.parquet.compression.codec`
 - ❑ Decompression speed vs I/O saving trade-off



Data Format: Parquet: Optimizations

Predicate Pushdown

```
SELECT * FROM table WHERE x > 5
```

```
Row-group 0: x: [min: 0, max: 9]
```

```
Row-group 1: x: [min: 3, max: 7]
```

```
Row-group 2: x: [min: 1, max: 4]
```

Leverage min/max statistics

spark.sql.parquet.filterPushdown

Data Format: Parquet: Optimizations

Predicate Pushdown

- ❑ Doesn't work well on unsorted data
 - ❑ Large value range within row-group, low min, high max
 - ❑ What to do? Pre-sort data on predicate columns
- ❑ Use typed predicates
 - ❑ Match predicate and column type, don't rely on casting/conversions
 - ❑ Example: use actual longs in predicate instead of ints for long columns

Data Format: Parquet: Optimizations

Partitioning

- ❑ Embed predicates in directory structure

```
df.write.partitionBy("date").parquet(...)
```

```
./example_parquet_file/date=2019-10-15/...
```

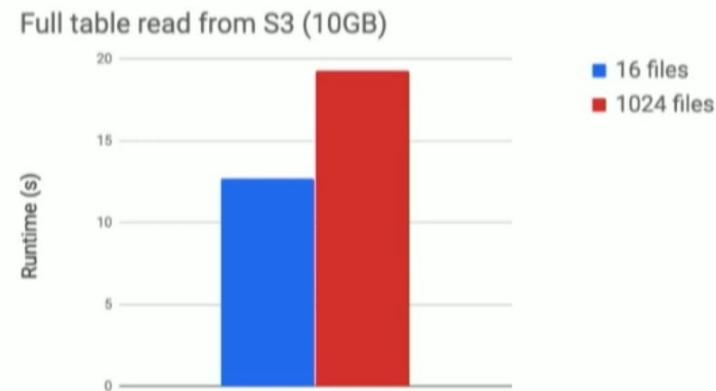
```
./example_parquet_file/date=2019-10-16/...
```

```
./example_parquet_file/date=2019-10-17/part-00000-...-475b15e2874d.c000.snappy.parquet
```

Data Format: Parquet: Optimizations

Avoid many small files

- ❑ For every file
 - ❑ Setup internal data structures
 - ❑ Instantiate reader objects
 - ❑ Fetch file
 - ❑ Parse Parquet metadata



Data Format: Parquet: Optimizations

Avoid few huge files

- ❑ Also avoid having huge files!
- ❑ SELECT count(*) on 250GB dataset
 - ❑ 250 partitions (~1GB each)
 - ❑ 5 mins
 - ❑ 1 huge partition (250GB)
 - ❑ 1 hour
 - ❑ Footer processing not optimized for speed

Performance tuning process: Data Format

Data Format: Parquet: Key Configuration

Configuration	Description
parquet.dictionary.page.size	To increase max dictionary size
parquet.block.size	To decrease row-group size
spark.sql.parquet.compression.codec	To compress the entire schema using snappy, gzip, lzo, etc.
spark.sql.parquet.filterPushdown	To leverage min/max statistics
df.repartition(numPartitions).write.Parquet()	To manually compact using repartition
df.coalesce(numPartitions).write.Parquet()	To manually compact using coalesce

Caching:

Spark supports the caching of datasets in memory. There are different options available:

- Use caching when the same operation is computed multiple times in the pipeline flow
- Use caching using the persist API to enable the required cache setting (persist to disk or not; serialized or not)
- Be aware of lazy loading and prime cache if needed up-front. Some APIs are eager and some are not
- Check out the Spark UI's Storage tab to see information about the datasets you have cached
- It's good practice to unpersist your cached dataset when you are done using them in order to release resources, particularly when you have other people using the cluster as well

Caching to be discussed in detail soon

Checkpointing:

- ❑ *Checkpoint* truncates the execution plan and saves the checkpointed data frame to a temporary location on the disk and reload it back in, which would be redundant anywhere else besides Spark
- ❑ Caching is also an alternative for a similar purpose in order to increase performance
- ❑ It obviously requires much more memory compared to *checkpointing*
- ❑ Spark won't clean up the checkpointed data even after the `sparkContext` is destroyed and the clean-ups need to be managed by the application
- ❑ It is also a good property of *checkpointing* to debug the data pipeline by checking the status of data frames

Checkpointing to be discussed in detail soon

Performance tuning process: Joins

Join is an expensive operation, so optimizing the joins in your application is important. Below are some tips:

- ❑ Join order matters; start with the most selective join. For relations less than `spark.sql.autoBroadcastJoinThreshold`, you can check whether broadcast **HashJoin** is picked up
- ❑ Use SQL hints if needed to force a specific type of join
 - ❑ Example: When joining a small dataset with large dataset, a broadcast join may be forced to broadcast the small dataset
 - ❑ Confirm that Spark is picking up broadcast hash join; if not, one can force it using the SQL hint

Joins to be discussed in detail soon

Performance tuning process: **Joins**

- ❑ Avoid cross-joins
- ❑ Broadcast **HashJoin** is most performant, but may not be applicable if both relations in join are large
- ❑ Collect statistics on tables for Spark to compute an optimal plan

Joins to be discussed in detail soon

Performance tuning process: Dynamic Resource Allocation

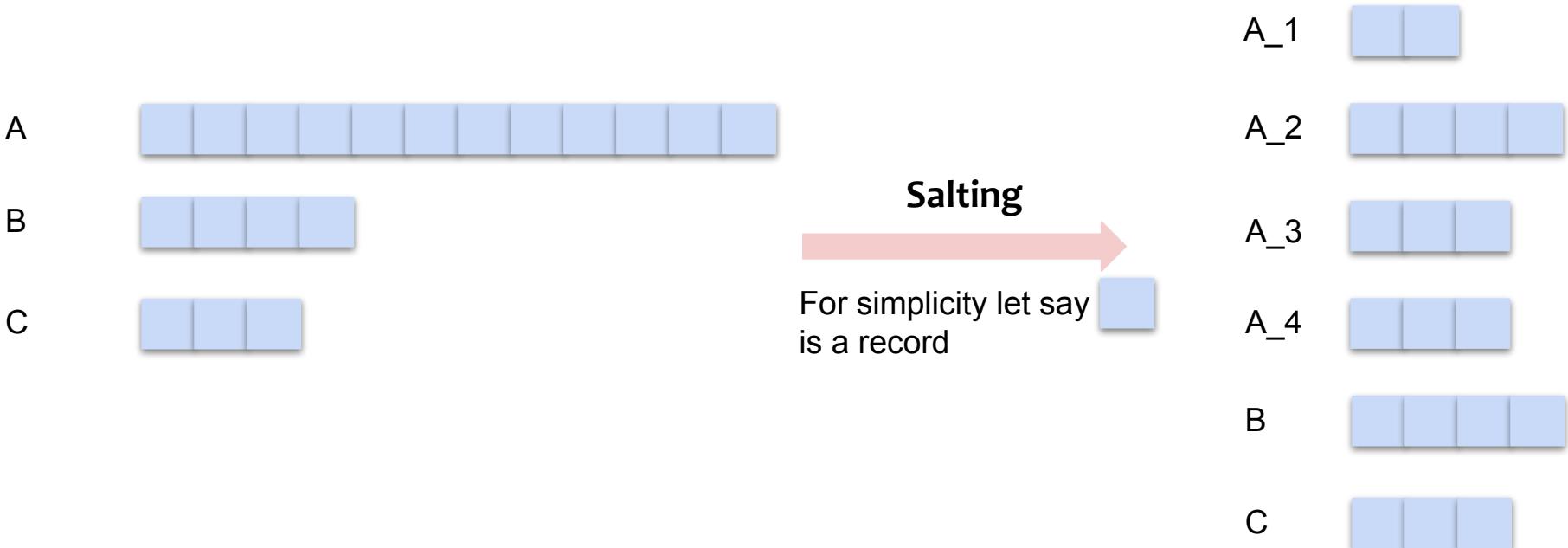
- ❑ Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload
- ❑ This means that your application may give resources back to the cluster if they are no longer used and request them again later when there is demand
- ❑ This feature is particularly useful if multiple applications share resources in your Spark cluster
- ❑ This feature is disabled by default and available on all coarse-grained cluster managers, i.e. [standalone mode](#), [YARN mode](#), [Mesos coarse-grained mode](#) and [K8s mode](#)

Performance tuning process: Data Skew

- ❑ Uneven Distribution of data along with the partitions, is data skewness problem
- ❑ This problem might occur during the intermediate stages of a Spark application
- ❑ Moreover, if the data is highly *skewed*, it might even cause a spill of the data from memory to disk
- ❑ Up to some degree a small amount of skew can be ignored in the range of 10 percent
- ❑ But large skews can result in spill or worse, out of memory errors

Performance tuning process: Data Skew

Salting is one of the solution for Data Skew



Performance tuning process: Data Skew

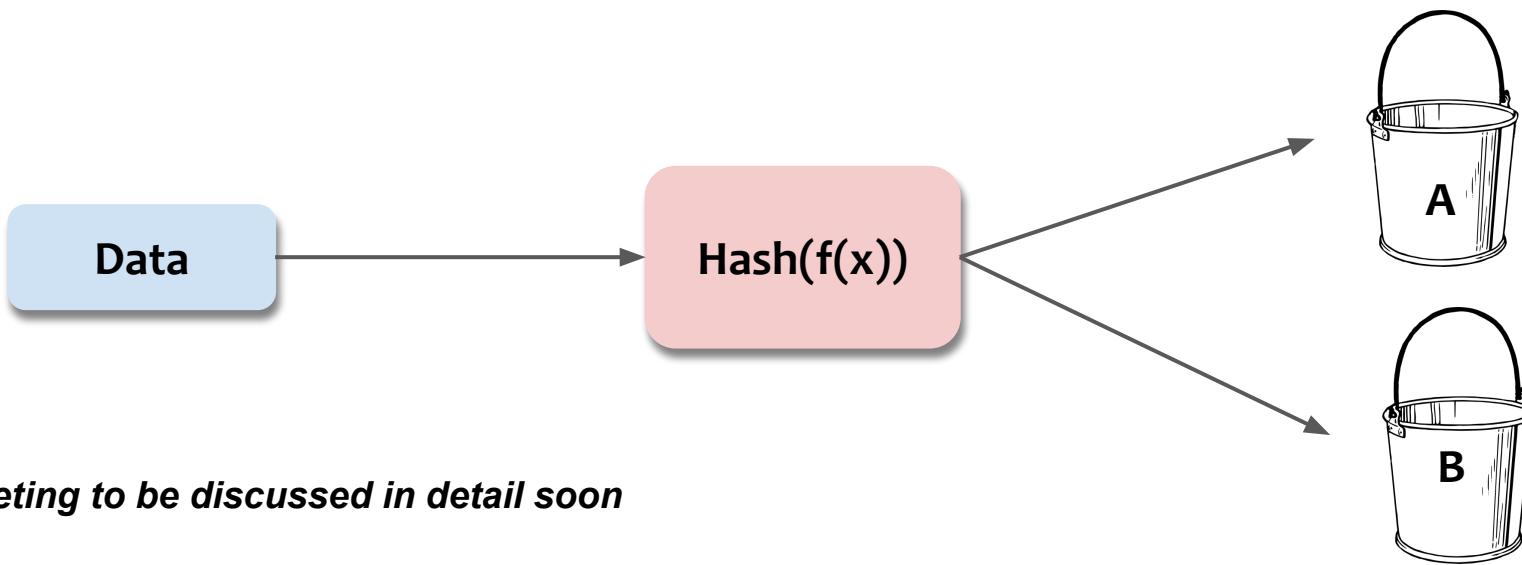
- ❑ **Repartitioning** is the another solution for Data Skew
- ❑ *Repartition* does a full shuffle, creates new partitions, and increases the level of parallelism in the application. More partitions will help to deal with the data skewness problem with an extra cost that is a shuffling of full data as mentioned above
- ❑ It would be very useful if there exists multiple joins or aggregations on these columns in the following steps

Performance tuning process: Partitioning

- ❑ While spark is designed to take advantage of larger numbers of small partitions when dealing with lot of data such as, ***textfile methods***, it can also be used in a way that allows for combining smaller files into a single partition if data is too fragmented, such as ***wholeTextFile***
- ❑ Based on partition behaviour of the operations applied on the data and the minimum partitions configured, following combinations are possible
- ❑ If the partitions are not uniform, we say that the partitioning is skewed. This can happen for a number of reasons and in different parts of our computation

Performance tuning process: **Bucketing**

- ❑ Bucketing is another data organization technique that groups data with the same bucket value
- ❑ It is similar to partitioning, but partitioning creates a directory for each partition, whereas bucketing distributes data across a fixed number of buckets by a hash on the bucket value



Bucketing to be discussed in detail soon

Performance tuning process: **Bucketing**

- ❑ By applying *bucketing* on the convenient columns in the data frames before shuffle required operations, we might avoid multiple probable expensive shuffles
- ❑ *Bucketing* boosts performance by already sorting and shuffling data before performing sort-merge joins
- ❑ It is important to have the same number of buckets on both sides of the tables in the join

Bucketing to be discussed in detail soon

Performance tuning process: **Avoid UDFs**

UDF (user defined function) :

Column-based functions that extend the vocabulary of Spark SQL's DSL

*Use the higher-level standard Column-based functions with Dataset operators whenever possible before reverting to using your own custom UDF functions since UDFs are a **blackbox** for Spark and so it does **not even try** to optimize them*

Try to avoid them and use the Spark's in-built functions

Spark Optimization

- Performance tuning process
- Performance tuning metrics**
- SQL performance tuning

Performance tuning metrics

Task Metrics collected by Spark executors with the granularity of task execution. The metrics can be used for performance troubleshooting and workload characterization

Spark Executor Task Metric name	Short description
executorRunTime	Elapsed time the executor spent running this task. This includes time fetching shuffle data. The value is expressed in milliseconds
executorCpuTime	CPU time the executor spent running this task. This includes time fetching shuffle data. The value is expressed in nanoseconds
executorDeserializeTime	Elapsed time spent to deserialize this task. The value is expressed in milliseconds
executorDeserializeCpuTime	CPU time taken on the executor to deserialize this task. The value is expressed in nanoseconds

Performance tuning metrics

Spark Executor Task Metric name	Short description
resultSize	The number of bytes this task transmitted back to the driver as the TaskResult.
jvmGCTime	Elapsed time the JVM spent in garbage collection while executing this task. The value is expressed in milliseconds
resultSerializationTime	Elapsed time spent serializing the task result. The value is expressed in milliseconds
memoryBytesSpilled	The number of in-memory bytes spilled by this task
diskBytesSpilled	The number of on-disk bytes spilled by this task
inputMetrics.*	Metrics related to reading data from org.apache.spark.rdd.HadoopRDD or from persisted data
bytesRead	Total number of bytes read

Performance tuning metrics

Spark Executor Task Metric name	Short description
recordsRead	Total number of records read
outputMetrics.*	Metrics related to writing data externally (e.g. to a distributed filesystem), defined only in tasks with output
bytesWritten	Total number of bytes written
recordsWritten	Total number of records written
shuffleReadMetrics.*	Metrics related to shuffle read operations
recordsRead	Number of records read in shuffle operations
remoteBlocksFetched	Number of remote blocks fetched in shuffle operations
localBlocksFetched	Number of local (as opposed to read from a remote executor) blocks fetched in shuffle operations

Performance tuning metrics

Executor-level metrics are sent from each executor to the driver as part of the Heartbeat to describe the performance metrics of Executor itself like JVM heap memory, GC information

Executor Level Metric name	Short description
rddBlocks	RDD blocks in the block manager of this executor
memoryUsed	Storage memory used by this executor
diskUsed	Disk space used for RDD storage by this executor
totalCores	Number of cores available in this executor
maxTasks	Maximum number of tasks that can run concurrently in this executor
activeTasks	Number of tasks currently executing
failedTasks	Number of tasks that have failed in this executor

Performance tuning metrics

Executor Level Metric name	Short description
completedTasks	Number of tasks that have completed in this executor
totalTasks	Total number of tasks (running, failed and completed) in this executor
totalDuration	Elapsed time the JVM spent executing tasks in this executor. The value is expressed in milliseconds
totalGCTime	Elapsed time the JVM spent in garbage collection summed in this executor. The value is expressed in milliseconds
totalInputBytes	Total input bytes summed in this executor
totalShuffleRead	Total shuffle read bytes summed in this executor
totalShuffleWrite	Total shuffle write bytes summed in this executor
maxMemory	Total amount of memory available for storage, in bytes

Performance tuning metrics

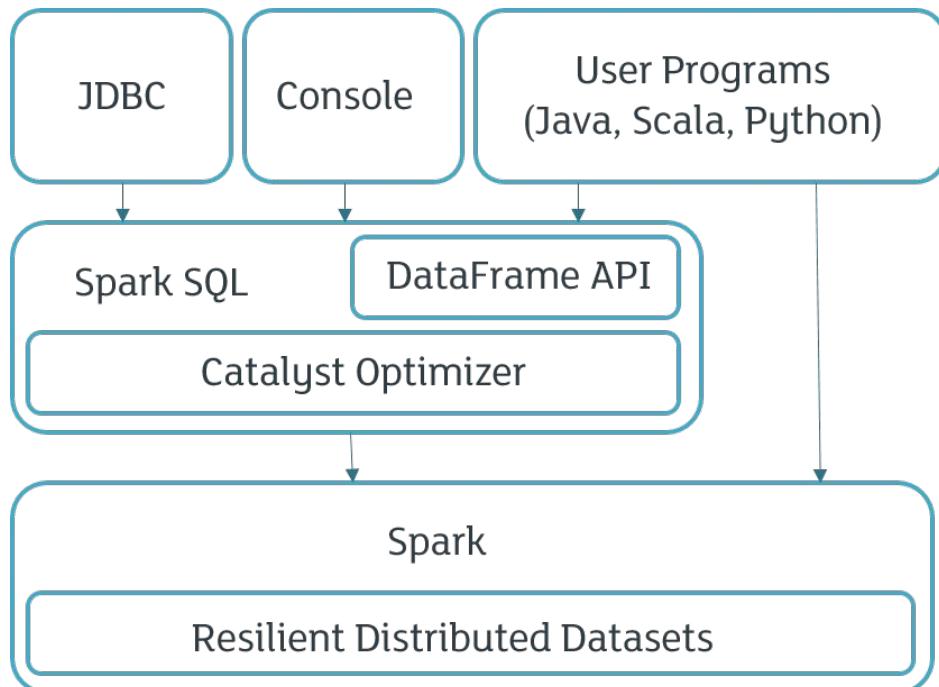
Executor Level Metric name	Short description
memoryMetrics.*	Current value of memory metrics
usedOnHeapStorageMemory	Used on heap memory currently for storage, in bytes
usedOffHeapStorageMemory	Used off heap memory currently for storage, in bytes
totalOnHeapStorageMemory	Total available on heap memory for storage, in bytes. This amount can vary over time, on the MemoryManager implementation
totalOffHeapStorageMemory	Total available off heap memory for storage, in bytes. This amount can vary over time, depending on the MemoryManager implementation
peakMemoryMetrics.*	Peak value of memory (and GC) metrics
JVMHeapMemory	Peak memory usage of the heap that is used for object allocation. The heap consists of one or more memory pools
JVMOFFHeapMemory	Peak memory usage of non-heap memory that is used by the Java virtual machine. The non-heap memory consists of one or more memory pools

Spark Optimization

- Performance tuning process
- Performance tuning metrics
- SQL performance tuning**

SQL performance tuning

- ❑ Spark SQL was designed with an optimizer called Catalyst based on the functional programming of Scala
- ❑ Its two main purposes are: first, to perform query optimization and second, to allow developers to expand and customize the functions of the optimizer



SQL performance tuning

The main components of the Catalyst optimizer are as follows:

Trees

The main data type in Catalyst is the tree. Each tree is composed of nodes, and each node has a nodetype and zero or more children

```
Merge(Attribute(x), Merge(Literal(1), Literal(2)))
```

SQL performance tuning

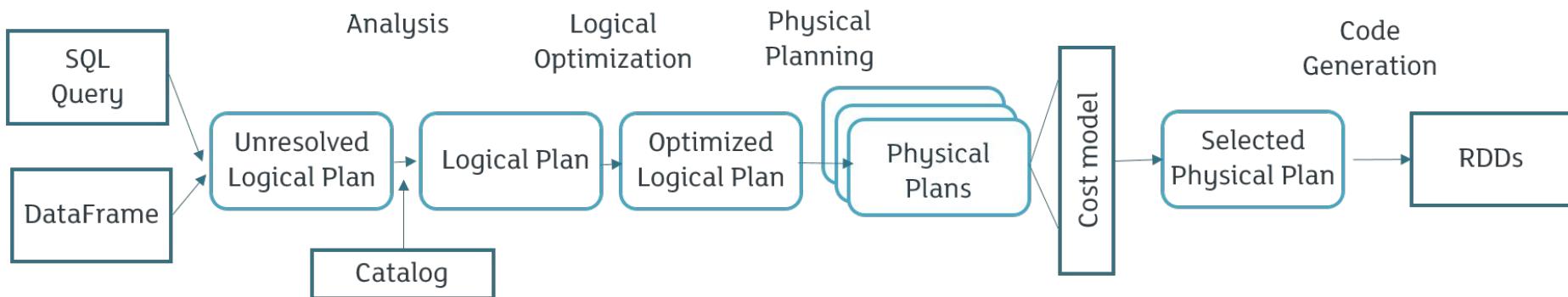
The main components of the Catalyst optimizer are as follows:

Rules

Trees can be manipulated using rules, which are functions of a tree to another tree. The transformation method applies the pattern matching function recursively on all nodes of the tree transforming each pattern to the result

```
tree.transform {  
    case Merge(Literal(c1), Literal(c2)) => Literal(c1) + Literal(c2)  
}
```

Using Catalyst in Spark SQL



Using Catalyst in Spark SQL

Phases

The four phases of the transformation that Catalyst performs are as follows:

1. Analysis

The first phase of Spark SQL optimization is the analysis. Spark SQL starts with a relationship to be processed that can be in two ways. A serious form from an AST (abstract syntax tree) returned by an SQL parser, and on the other hand from a DataFrame object of the Spark SQL API

2. Logical Plan Optimization

The second phase is the logical optimization plan. In this phase, rule-based optimization is applied to the logical plan. It is possible to easily add new rules

Using Catalyst in Spark SQL

Phases

The four phases of the transformation that Catalyst performs are as follows:

3. Physical plan

In the physical plan phase, Spark SQL takes the logical plan and generates one or more physical plans using the physical operators that match the Spark execution engine. The plan to be executed is selected using the cost-based model (comparison between model costs)

4. Code generation

Code generation is the final phase of optimizing Spark SQL. To run on each machine, it is necessary to generate Java code bytecode

Using Catalyst in Spark SQL

```
== Analyzed Logical Plan ==
nombre: string, count: bigint
SubqueryAlias total
+- Aggregate [nombre#4], [nombre#4, count(1) AS count#11L]
  +- LocalRelation [id#3, nombre#4, edad#5]
== Optimized Logical Plan ==
Aggregate [nombre#4], [nombre#4, count(1) AS count#11L]
+- LocalRelation [nombre#4]
== Physical Plan ==
*(2) HashAggregate(keys=[nombre#4], functions=[count(1)], output=[nombre#4, count#11L])
+- Exchange hashpartitioning(nombre#4, 200)
  +- *(1) HashAggregate(keys=[nombre#4], functions=[partial_count(1)], output=[nombre#4])
    +- LocalTableScan [nombre#4]
```

Spark Optimization

- Performance tuning process
- Performance tuning metrics
- SQL performance tuning

Agenda:

- High Performance Spark applications
- Working with Spark**
- Caching and Checkpointing
- Joins and more

Spark Optimization

- ❑ Debugging/troubleshooting Spark apps
- ❑ Developing data workflows
- ❑ Automated Spark builds using Maven

The Spark Shell and Spark Applications

- ❑ The Spark shell allows interactive exploration and manipulation of data
 - ❑ REPL using Python or Scala
- ❑ Spark applications run as independent programs
 - ❑ For jobs such as ETL processing, streaming, and so on
 - ❑ Python, Scala, or Java

The Spark Session and Spark Context

- ❑ Every Spark program needs
 - ❑ One SparkContext object
 - ❑ One or more SparkSession objects
 - ❑ If you are using Spark SQL
- ❑ The interactive shell creates these for you
 - ❑ A SparkSession object called spark
 - ❑ A SparkContext object called sc
- ❑ In a standalone Spark application you must create these yourself
 - ❑ Use a Spark session builder to create a new session
 - ❑ The builder automatically creates a new Spark context as well
 - ❑ Call stop on the session or context when program terminates

Creating a **SparkSession** Object

- ❑ **SparkSession.builder** points to a Builder object
 - ❑ Use the builder to create and configure a **SparkSession** object
- ❑ The **getOrCreate** builder function returns the existing **SparkSession** object if it exists
 - ❑ Creates a new Spark session if none exists
 - ❑ Automatically creates a new **SparkContext** object as **sparkContext** on the **SparkSession** object

Python Example: Name List

```
import sys
from pyspark.sql import SparkSession

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print(sys.stderr, "Usage: spark-submit NameList.py <input-file> <output-file>")
        sys.exit()
    spark= SparkSession.builder.getOrCreate()
    spark.sparkContext.setLogLevel("WARN")
    peopleDF= spark.read.json(sys.argv[1])
    nameDF= PeopleDF.select("firstname","lastname")
        nameDF.write.option("header","true").csv(sys.argv[2])
    spark.stop()
```



Debugging/Troubleshooting

- ❑ For debugging Spark application or job to look at the values in runtime in order to fix issues,
 - ❑ We typically use IntelliJ Idea or Eclipse IDE
 - ❑ To debug locally or remote running applications written in Scala or Java
- ❑ To debug a Scala or Java application, you need to run the application with JVM options
agentlib:jdwp (JDWP: Java Debug Wire Protocol)

Debugging/troubleshooting Spark apps

JDWP (Java Debug Wire Protocol), followed by a comma-separated list of sub-option

```
agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005
```

Debugging/troubleshooting Spark apps

Spark-Submit

+

--conf spark.driver.extraJavaOptions

agentlib:jdwp

=

Debug

Debugging/troubleshooting Spark apps

```
spark-submit \
--name SparkByExamples.com \
--class org.sparkbyexamples.SparkWordCountExample \
--conf
spark.driver.extraJavaOptions=-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005
spark-by-examples.jar
```

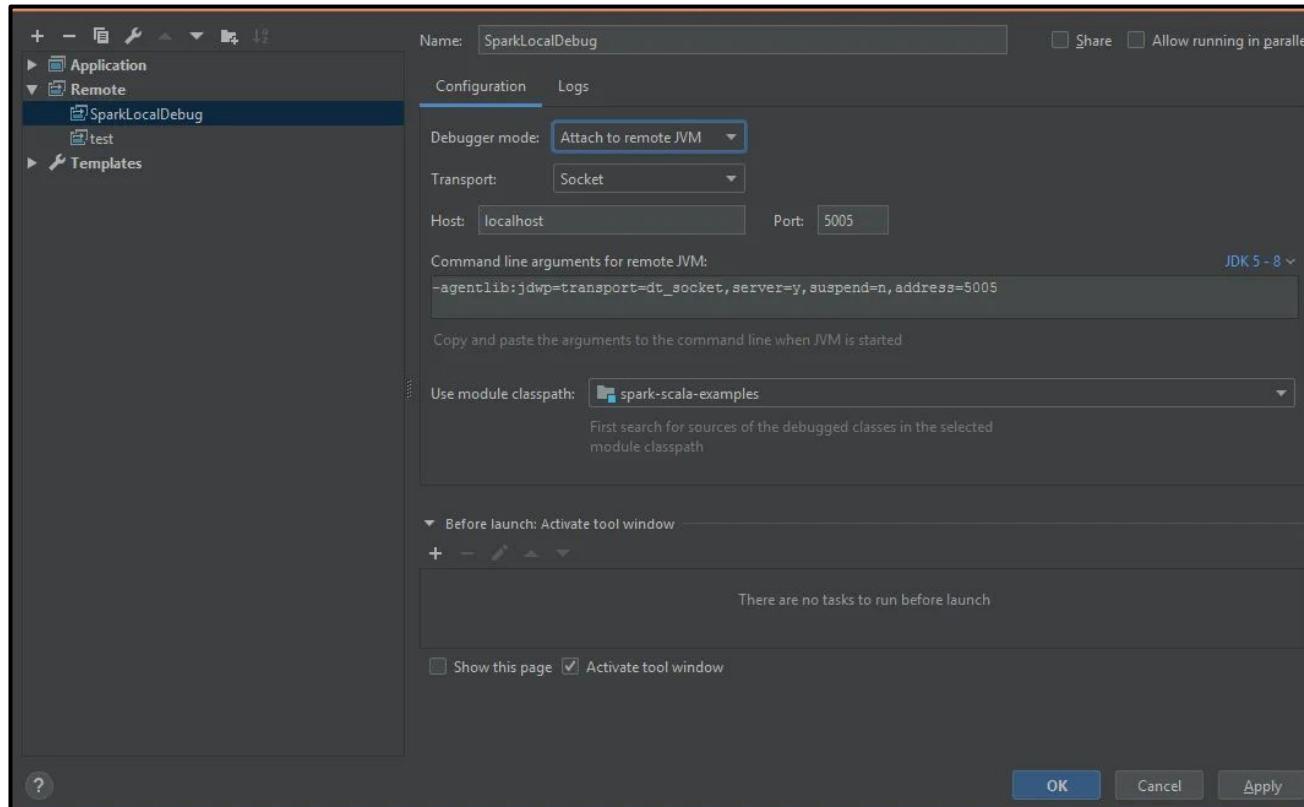
Output:

```
Listening for transport dt_socket at address: 5005
```

Follow the below steps to create Remote application and start to debug.

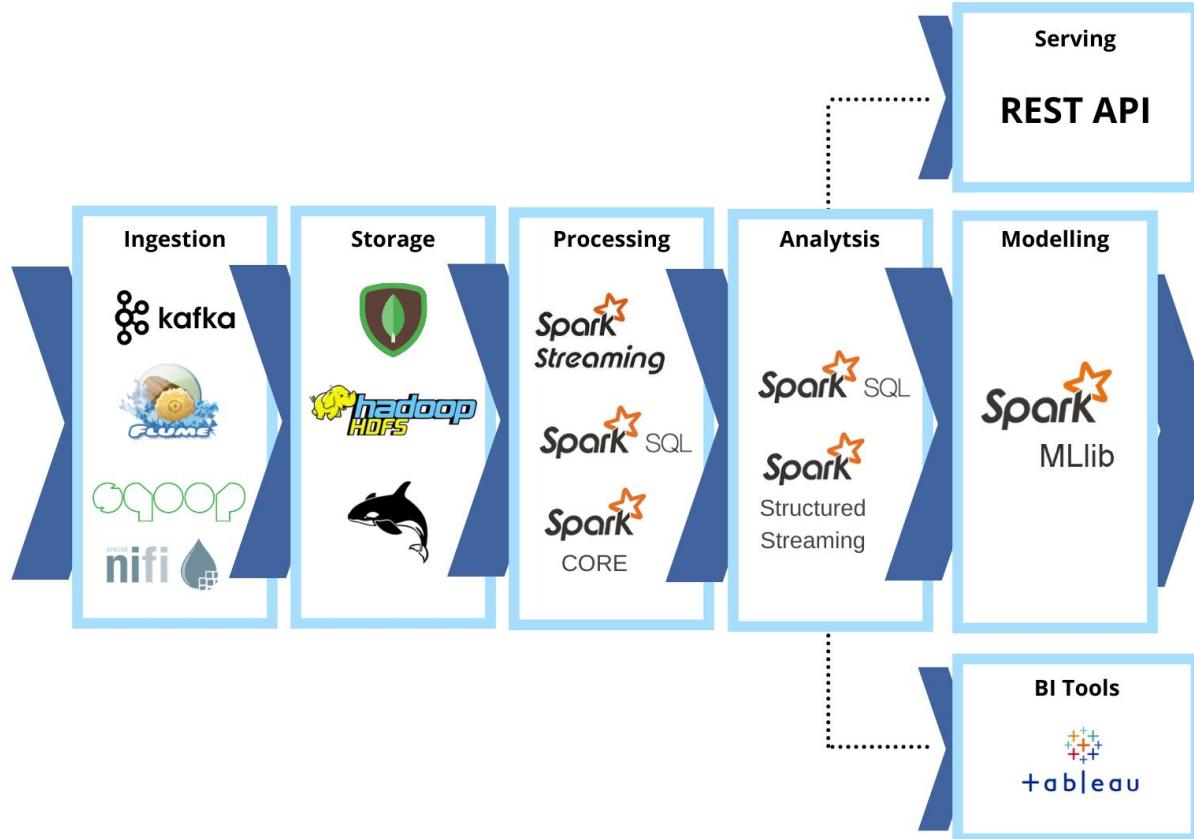
- Open your Spark application you wanted to debug in IntelliJ Idea IDE
- Access **Run -> Edit Configurations**, this brings you **Run/Debug Configurations** window
- Now select **Applications** and select + sign from the top left corner and select **Remote** option
- Enter your debugger name for **Name** field. for example, enter **SparkLocalDebug**
- For **Debugger mode** option select **Attach to local JVM**
- For **Transport**, select **Socket** (this selected by default)
- For **Host**, enter **localhost** as we are debugging Local and enter the port number for **Port**. For our example, we are using **5005**
- Finally, select **OK**. This just creates the Application to debug but it doesn't start

Debugging/troubleshooting Spark apps



- Debugging/troubleshooting Spark apps
- Developing data workflows**
- Automated Spark builds using Maven

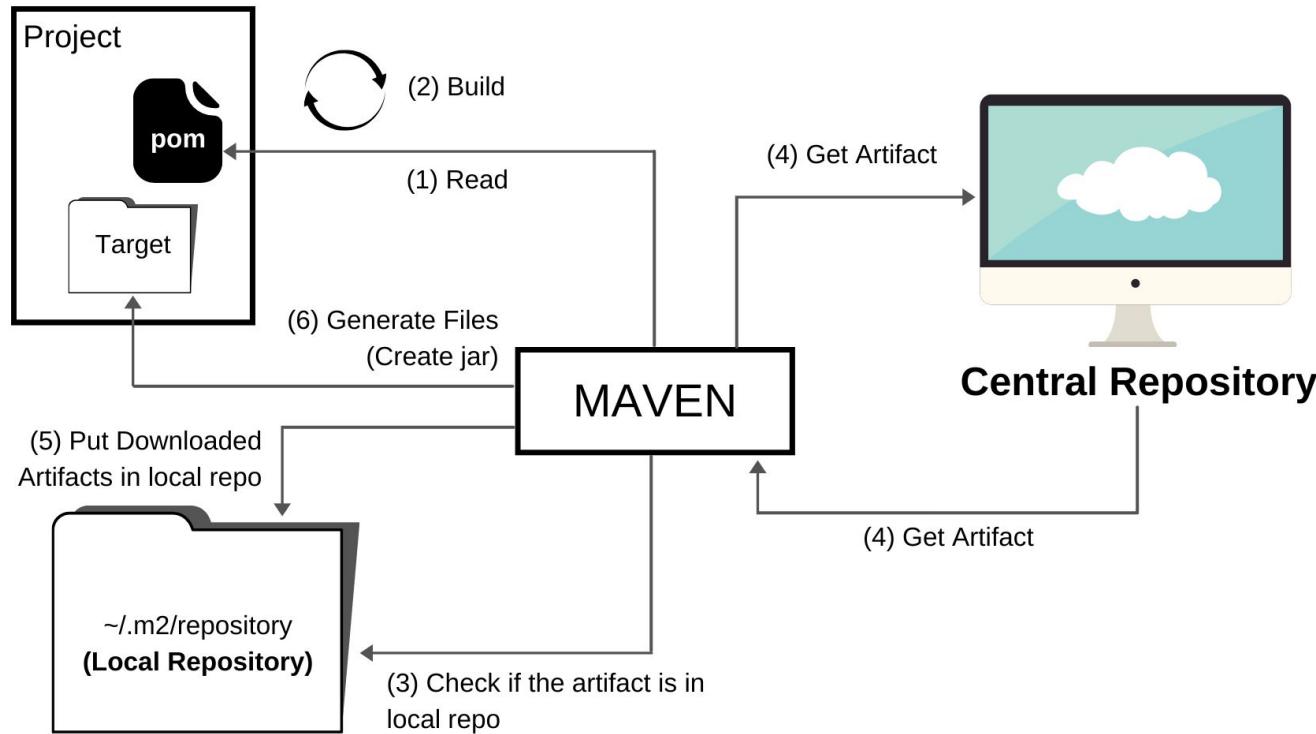
Developing Data Workflow



Spark Optimization

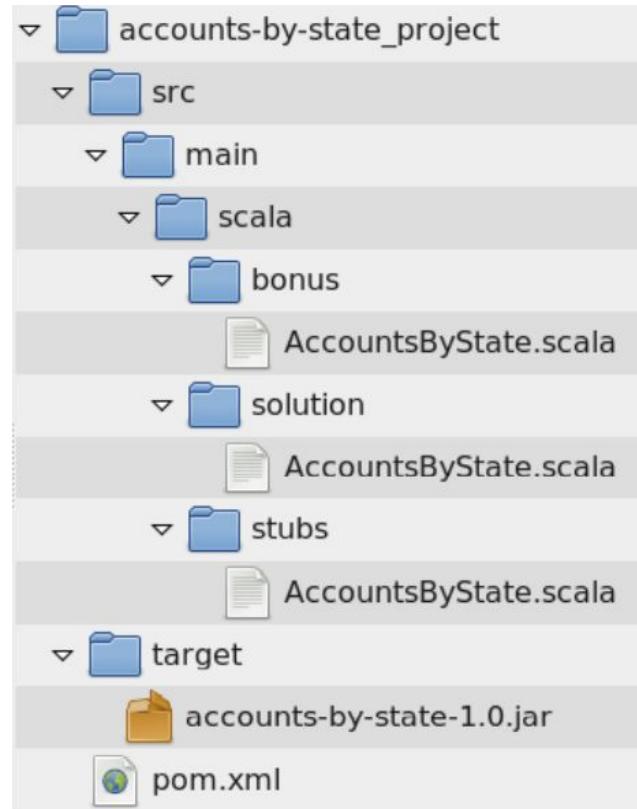
- Debugging/troubleshooting Spark apps
- Developing data workflows
- Automated Spark builds using Maven**

Automated Spark builds using Maven



Automated Spark builds using Maven

- ❑ Basic Apache Maven projects are provided in the exercise directory
 - ❑ stubs: starter Scala files—do exercises here
 - ❑ solution: exercise solutions
 - ❑ bonus: bonus solutions
- ❑ Build command: mvn package



Automated Spark builds using Maven

```
<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core -->  
<dependency>  
    <groupId>org.apache.spark</groupId>  
    <artifactId>spark-core_2.12</artifactId>  
    <version>3.0.0</version>  
</dependency>
```

Spark Optimization

- Debugging/troubleshooting Spark apps
- Developing data workflows
- Automated Spark builds using Maven

Agenda:

- High Performance Spark applications
- Working with Spark
- Caching and Checkpointing**
- Joins and more

- ❑ When to Cache?
- ❑ How Caching helps?
- ❑ Caching Strategies
- ❑ How does the Spark plan change when Caching is on?
- ❑ Visualizing Cached Dataset in Spark UI
- ❑ Working with On Heap and Off Heap Caching
- ❑ Checkpointing
- ❑ How is Caching different from Checkpointing?

When to Cache?

Spark and the Fine Art of Caching

The **cache** method makes the DataFrame or RDD for **Caching** in memory/disk, but this only happens once when an action is performed on the DataFrame/RDD, and that too in a lazy evaluation fashion, i.e. if you want to read only 100 rows, then only those 100 rows are **cached**

When to Cache?

Caching is recommended in the following situations:

- For RDD re-use in iterative machine learning applications
- For RDD re-use in standalone Spark applications
- When RDD computation is expensive, caching can help in reducing the cost of recovery in the case one executor fails

- When to Cache?
- How Caching helps?**
- Caching Strategies
- How does the Spark plan change when Caching is on?
- Visualizing Cached Dataset in Spark UI
- Working with On Heap and Off Heap Caching
- Checkpointing
- How is Caching different from Checkpointing?

How Cache is Helpful?

- ❑ Caching is an optimization technique for iterative and interactive computations
- ❑ Caching helps in saving interim, partial results so they can be reused in subsequent stages of computation
- ❑ These interim results are stored as RDDs (Resilient Distributed Datasets) and are kept either in memory (by default) or on disk
- ❑ Data cached in memory is faster to access, of course, but cache space is always limited

- When to Cache?
- How Caching helps?
- Caching Strategies**
- How does the Spark plan change when Caching is on?
- Visualizing Cached Dataset in Spark UI
- Working with On Heap and Off Heap Caching
- Checkpointing
- How is Caching different from Checkpointing?

Caching Strategies

In Spark, there are two function calls for caching an RDD:

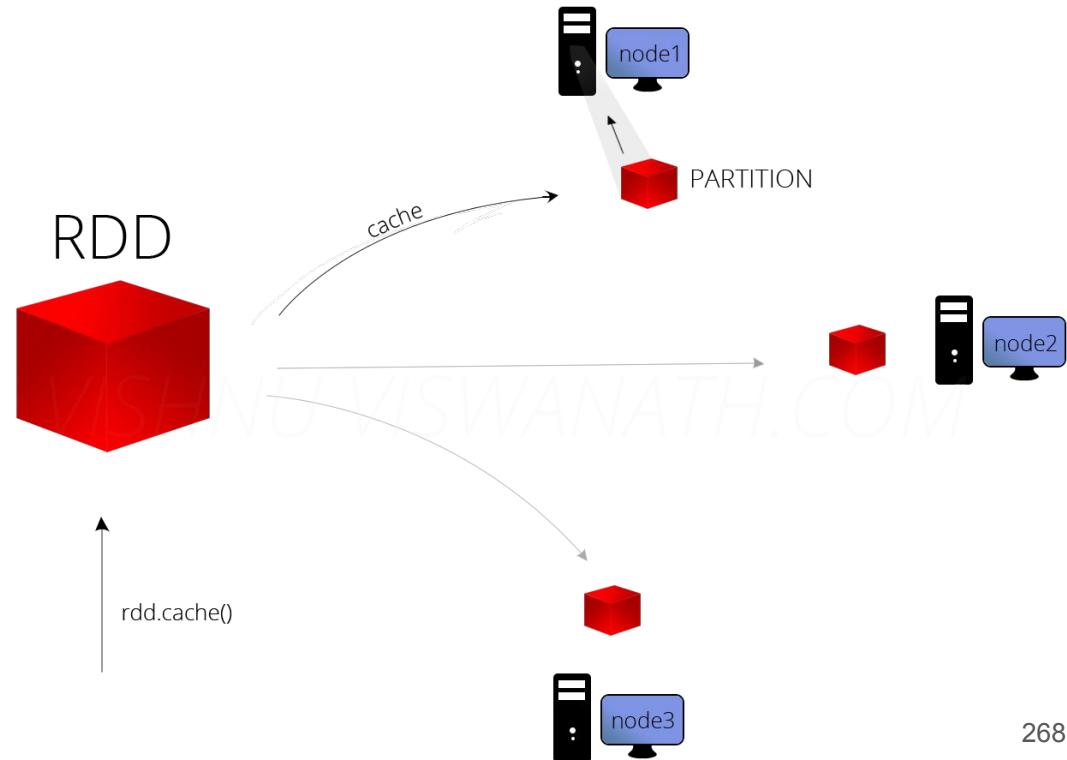
1. **cache()**
2. **persist (level: StorageLevel)**

Caching Strategies

1. With **cache()**, you use only the default storage level :
 - MEMORY_ONLY for **RDD**
 - MEMORY_AND_DISK for **Dataset**
2. With **persist()**, you can specify which storage level you want for both **RDD** and **Dataset**

Caching Strategies

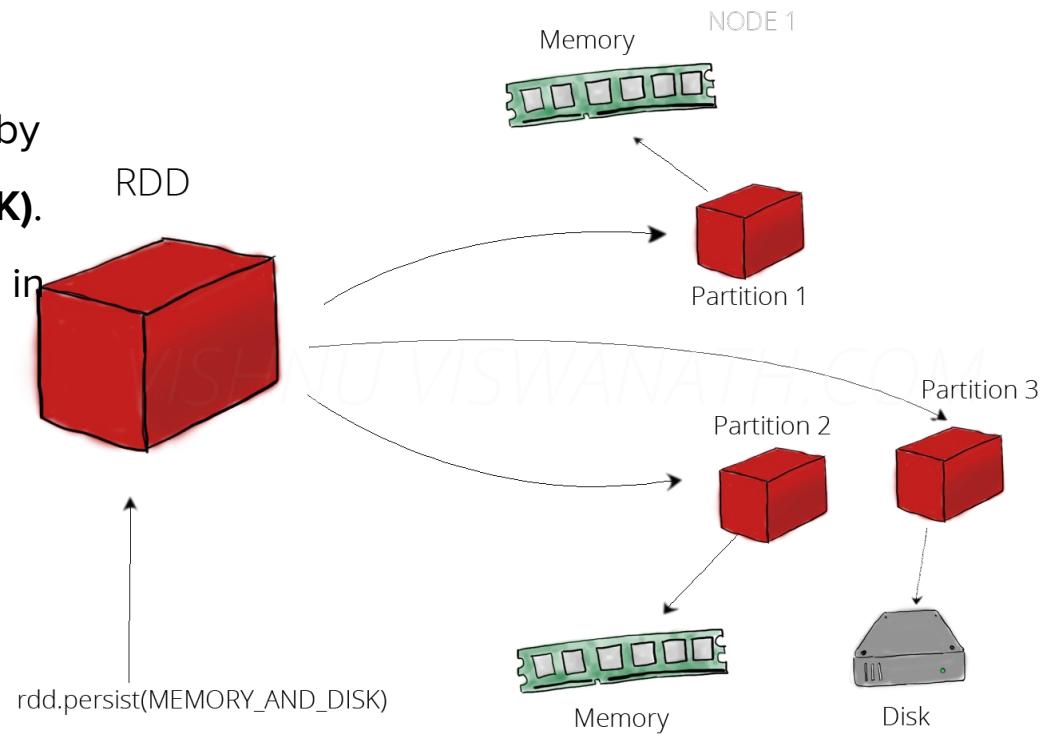
You can cache an RDD in memory by calling `rdd.cache()`. When you cache an RDD, its Partitions are loaded into memory of the nodes that hold it



Caching Strategies

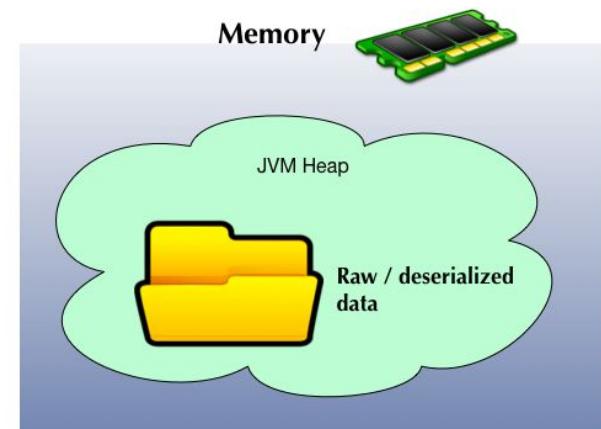
You can persist an RDD in memory by calling **rdd.persist(MEMORY_AND_DISK)**.

Here some partitions are persisted in Memory and some partitions in Disk



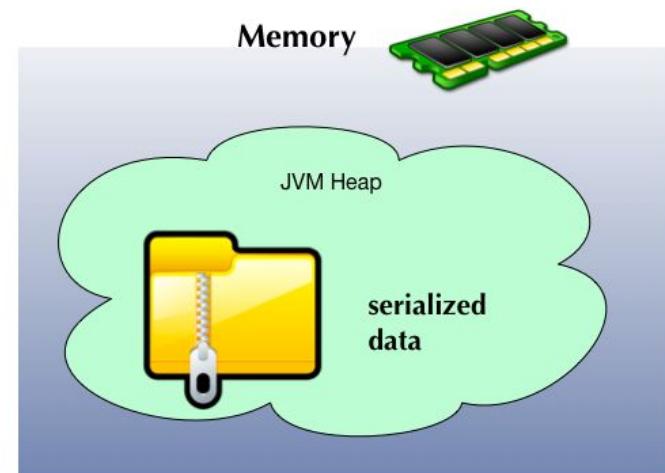
Cache Strategies

- ❑ cache() or persist(MEMORY_ONLY)
 - Most CPU efficient
 - Data stored as ‘raw’ / serialized
 - Takes up memory (3x –5x)
 - 1G raw data uses 3G –5G memory



Cache Strategies

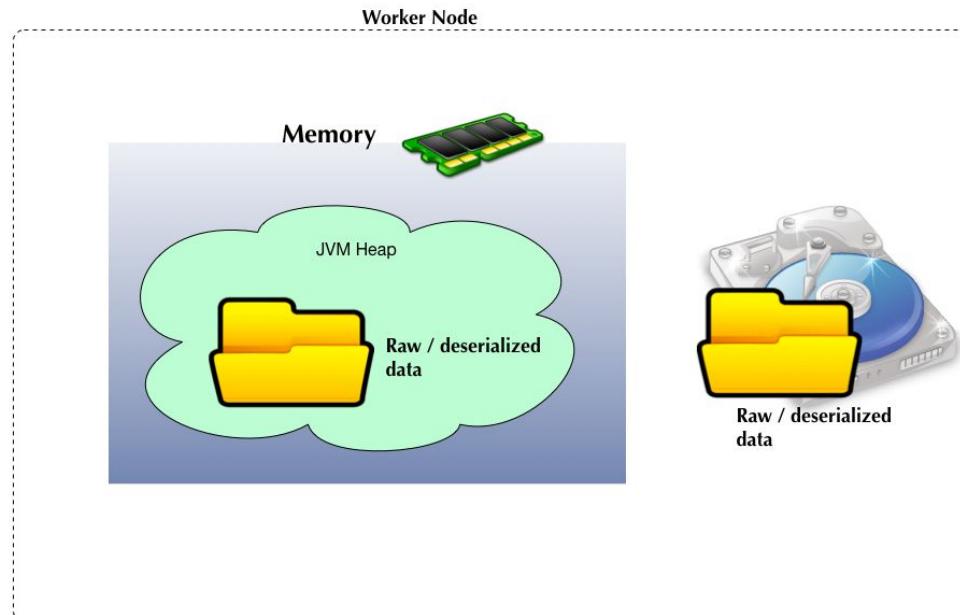
- ❑ persist(MEMORY_ONLY_SER)
 - Most memory efficient option
 - Little memory overhead
 - CPU intensive (to serialize / deserialize)
 - Default Java serializer is OK —use ‘kryo’ serializer for high performance



Cache Strategies

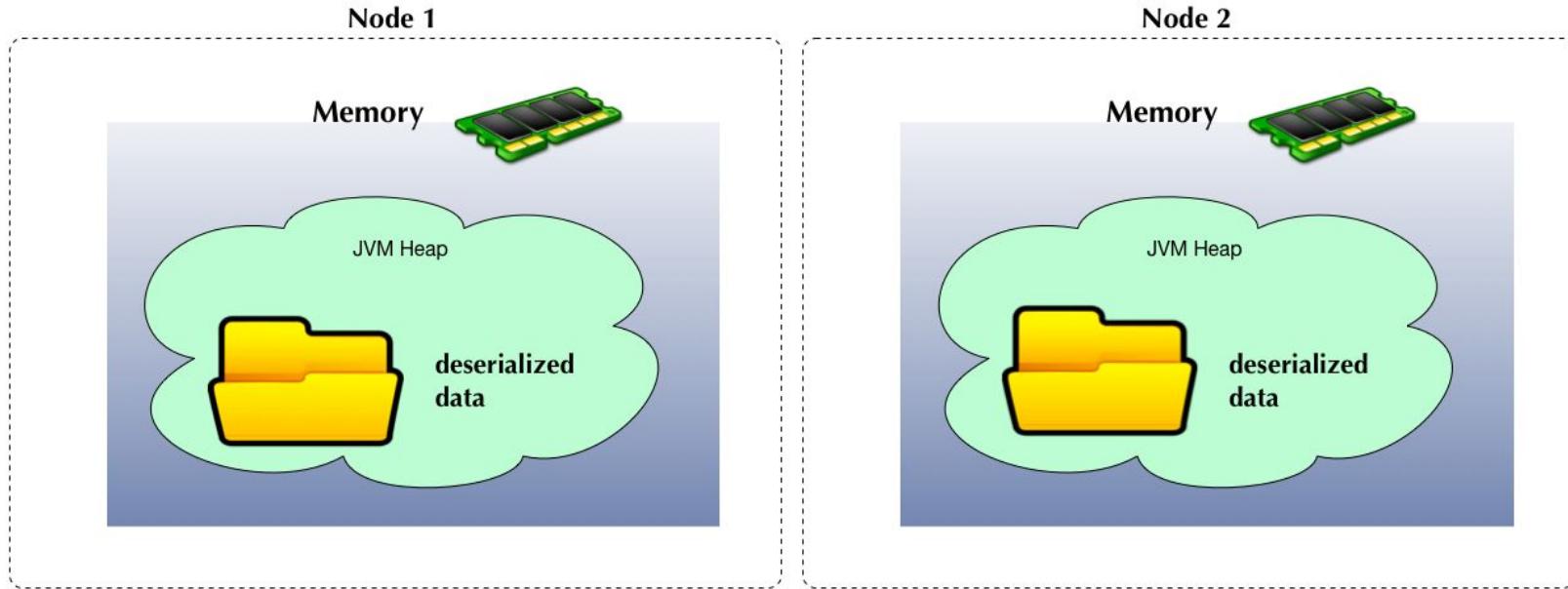
- ❑ persist(MEMORY_AND_DISK)

- Both in memory & disk
 - Can survive memory eviction



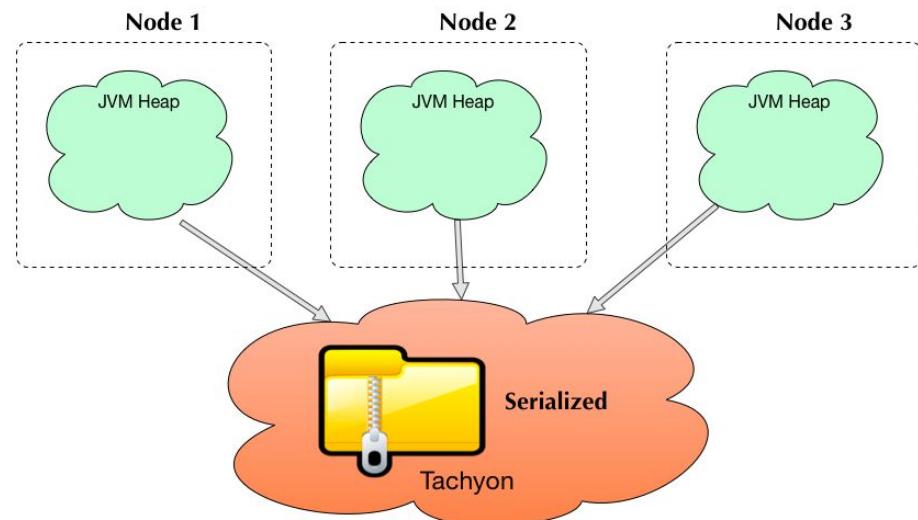
Cache Strategies

- ❑ persist(MEMORY_ONLY_2)
 - Survives a node failure



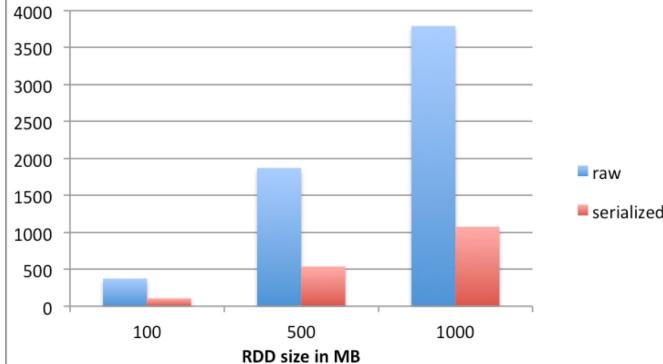
Cache Strategies

- ❑ In memory caching (Uses Tachyon/Alluxio)
- ❑ Cached data distributed across nodes
(scale out)
- ❑ Doesn't use JVM for storage
 - No need to worry about Garbage Collection (GC)
 - Can accommodate very large data sets

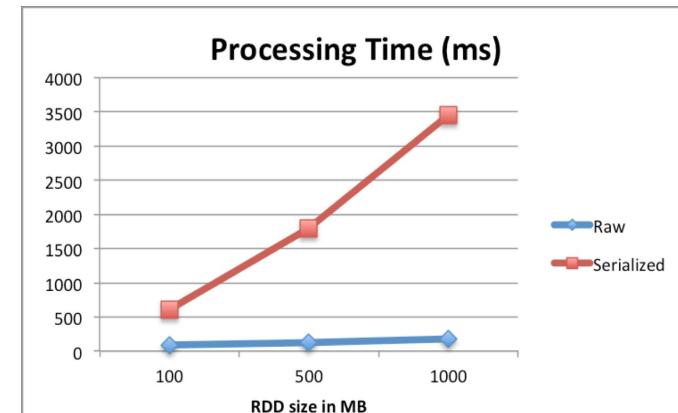


Cache Strategies

Memory Footprint (MB)



Processing Time (ms)



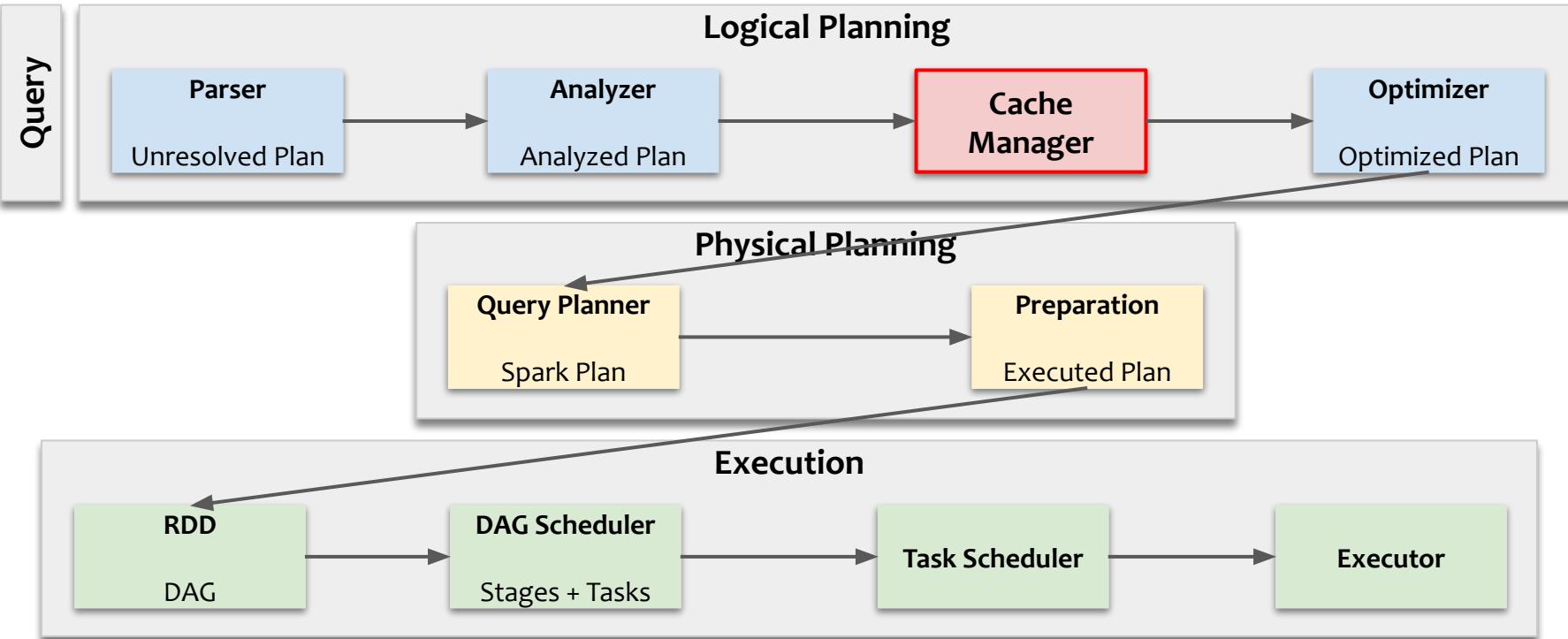
- ❑ Raw caching consumes more memory (2-5x)
 - But is faster to process
 - MEMORY_ONLY
- ❑ Serialized caching uses less memory
 - Processing time is more
 - MEMORY_ONLY_SER

- When to Cache?
- How Caching helps?
- Caching Strategies
- How does the Spark plan change when Caching is on?**
- Visualizing Cached Dataset in Spark UI
- Working with On Heap and Off Heap Caching
- Checkpointing
- How is Caching different from Checkpointing?

Caching and Checkpointing

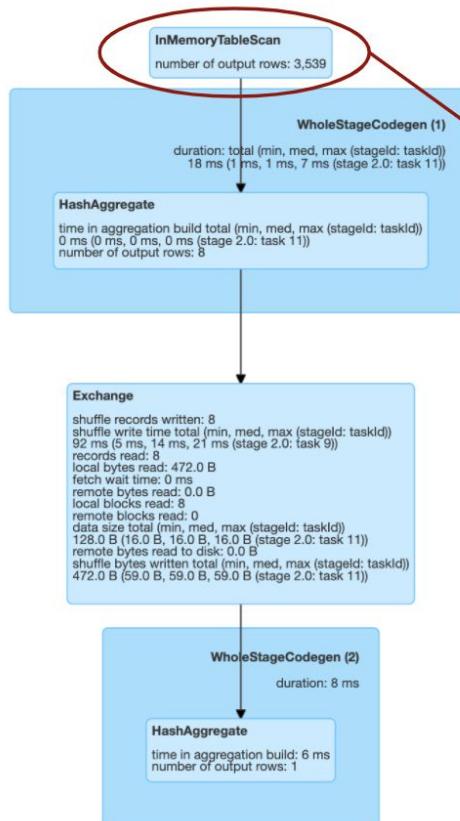
How does the Spark plan change when Caching is on?

Cache Manager



Caching and Checkpointing

How does the Spark plan change when Caching is on?



== Physical Plan ==

```
*(2) HashAggregate(keys=[], functions=[count(1)]  
+- Exchange SinglePartition, true, [id=#59]  
  +- *(1) HashAggregate(keys=[], functions=[par-  
    +- InMemoryTableScan  
    +- InMemoryRelation [user_id#0L, dis-  
      +- *(1) Project [user_id#0L, d-  
        +- *(1) Filter (isnotnull(v-  
          +- *(1) ColumnarToRow  
            +- FileScan parquet de-
```

This computation is cached.

- When to Cache?
- How Caching helps?
- Caching Strategies
- How does the Spark plan change when Caching is on?
- Visualizing Cached Dataset in Spark UI**
- Working with On Heap and Off Heap Caching
- Checkpointing
- How is Caching different from Checkpointing?

Visualizing Cached Dataset in Spark UI

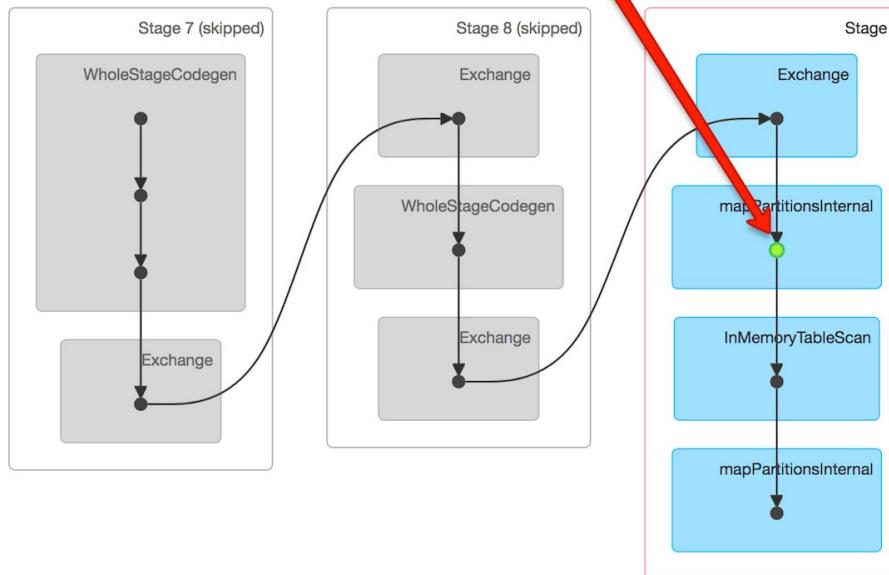
Details for Job 5

Status: SUCCEEDED

Completed Stages: 1

Skipped Stages: 2

- ▶ Event Timeline
- ▼ DAG Visualization

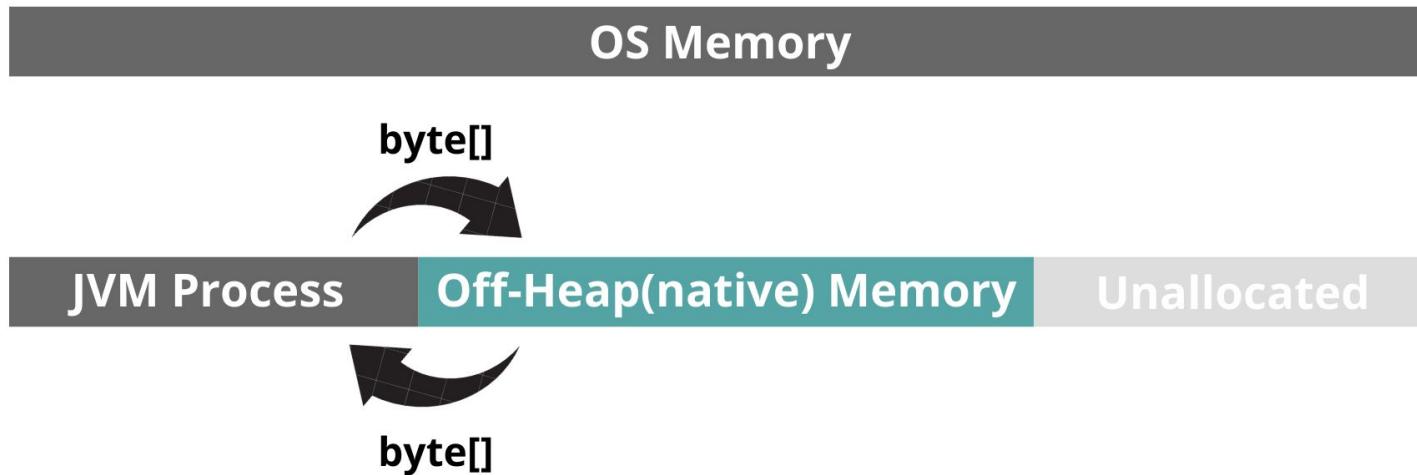


- When to Cache?
- How Caching helps?
- Caching Strategies
- How does the Spark plan change when Caching is on?
- Visualizing Cached Dataset in Spark UI
- Working with On Heap and Off Heap Caching**
- Checkpointing
- How is Caching different from Checkpointing?

On-Heap Caching

- ❑ On-heap caching is useful in scenarios when you do a lot of cache reads on server nodes that work with cache entries in [binary form](#) or invoke cache entries' deserialization
- ❑ When on-heap caching is enabled, you can use one of the on-heap eviction policies to manage the growing on-heap cache

Off-Heap Caching



- When to Cache?
- How Caching helps?
- Caching Strategies
- How does the Spark plan change when Caching is on?
- Visualizing Cached Dataset in Spark UI
- Working with On Heap and Off Heap Caching
- Checkpointing**
- How is Caching different from Checkpointing?

Checkpointing

- ❑ Checkpoint is used to freeze the content of data frame before something else happens
- ❑ It can be in the scenario of iterative algorithms (as mentioned in the Javadoc) but also in recursive algorithms or simply branching out a data frame to run different kinds of analytics on both
- ❑ Spark has been offering checkpoints on streaming since earlier versions (at least v1.2.0), but checkpoints on data frames are a different beast

Types of Checkpoints

You can create two kinds of checkpoints:

1. Eager
2. Non-Eager

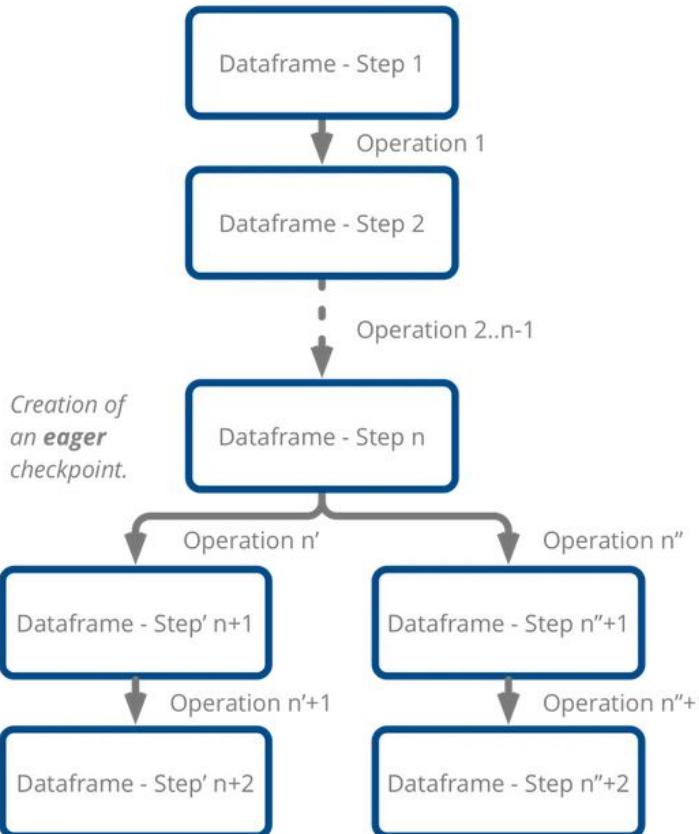
Eager Checkpoint

- ❑ An eager checkpoint will cut the lineage from previous data frames and will allow you to start “fresh” from this point on
- ❑ In clear, Spark will dump your data frame in a file specified by **setCheckpointDir()** and will start a fresh new data frame from it
- ❑ You will also need to wait for completion of the operation

Checkpointing

Lineage of operation is split: Catalyst will not optimize the whole process.

The lineage starts fresh after the checkpoint.

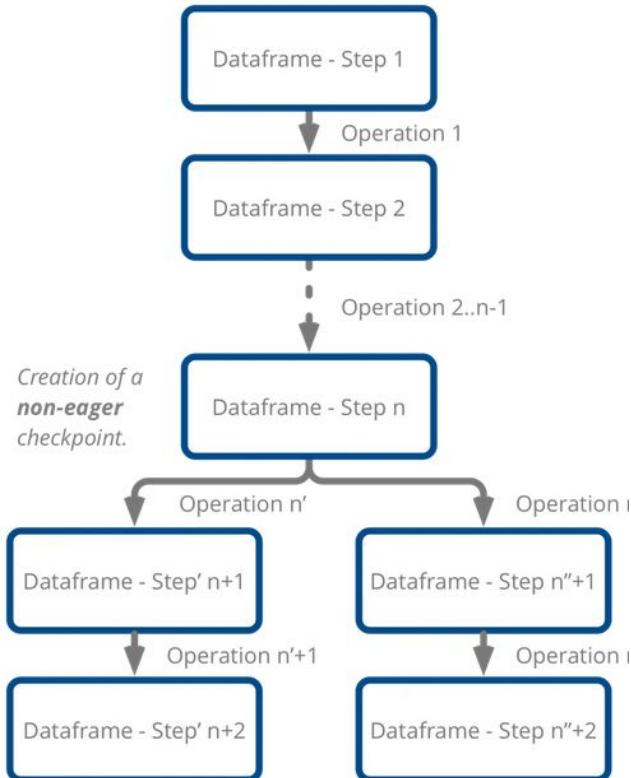


Non-Eager Checkpoint

On the other hand, a non-eager checkpoint will keep the lineage from previous operations in the data frame

Checkpointing

Lineage of operation is kept: Catalyst will optimize the whole process.



Spark Optimization

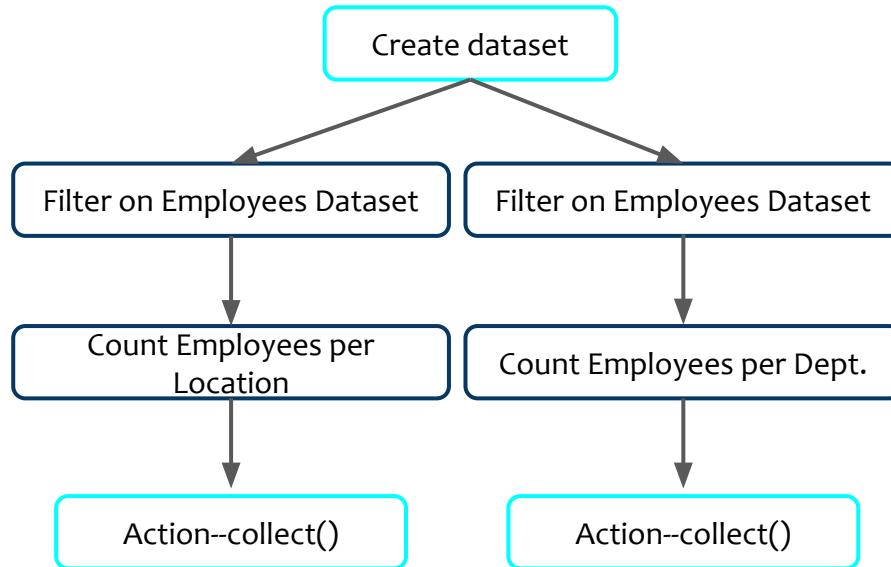
- When to Cache?
- How Caching helps?
- Caching Strategies
- How does the Spark plan change when Caching is on?
- Visualizing Cached Dataset in Spark UI
- Working with On Heap and Off Heap Caching
- Checkpointing
- How is Caching different from Checkpointing?**

How is Caching different from Checkpointing?

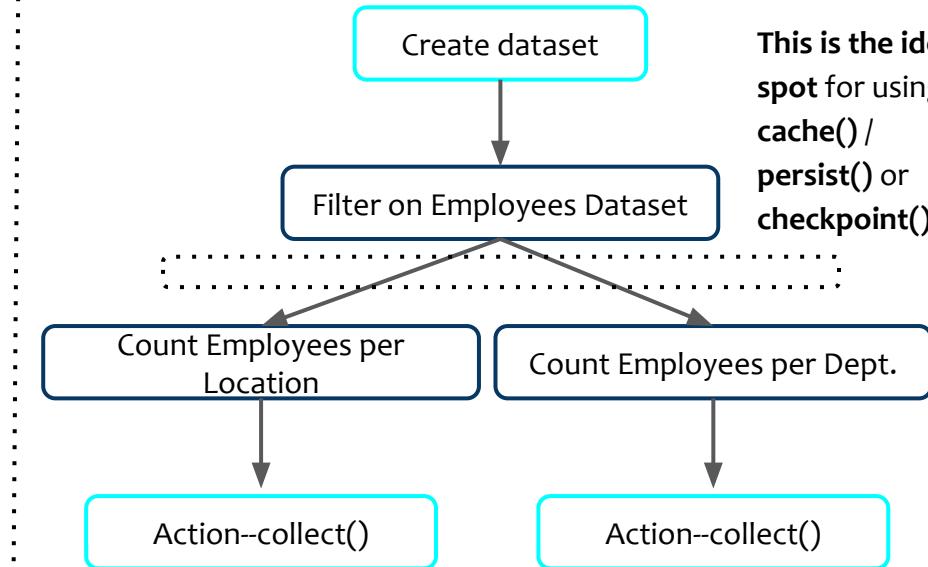
Caching	Checkpointing
Caching computes and materializes an RDD in memory while keeping track of its lineage (dependencies)	Checkpointing stores the rdd physically to hdfs and destroys the lineage (eager)
After the application terminates, the cache is cleared or file destroyed	Checkpointed file won't be deleted even after the Spark application terminated
Caching remembers an RDD's lineage, Spark can recompute loss partitions in the event of node failures	Checkpoint files can be used in subsequent job run or driver program Checkpointing an RDD causes double computation

Caching and Checkpointing

How is Caching different from Checkpointing?



Not using caching/checkpointing: the filter operation is performed for each action



Caching/checkpointing: the filter operation is performed for each action

Spark Optimization

- When to Cache?
- How Caching helps?
- Caching Strategies
- How does the Spark plan change when Caching is on?
- Visualizing Cached Dataset in Spark UI
- Working with On Heap and Off Heap Caching
- Checkpointing
- How is Caching different from Checkpointing?



Best Practices and more

Agenda:

- High Performance Spark applications
- Working with Spark
- Caching and Checkpointing
- Joins and more**

- ❑ Types of Joins
- ❑ Quick Recap of MapReduce MapSide and Reduce Side Joins
- ❑ Broadcasting
- ❑ Optimizing Sort Merge Join
- ❑ Bucketing
- ❑ Common Production Issues

Types of Joins

1. Sort-Merge Join
2. Broadcast Join
3. Shuffle-Hash Join
4. Broadcast Nested Loop Join
5. Skew Join

Types of Joins

Sort-Merge Join:

Sort-Merge join is composed of 2 steps:

1. The first step is to sort the data
2. The second operation is to merge the sorted data in the partition by iterating over the elements and according to the join key join the rows having the same value

The internal parameter '**spark.sql.join.preferSortMergeJoin**' can be set as True/False, for using Sort-Merge Join

Types of Joins

Sort-Merge Joins:

Customers table

Id	Login
2	User#2
1	User#1
4	User#4
3	User#3

Orders table

Id	User id
1001	2
1002	4
1003	4
1004	1

Sorting

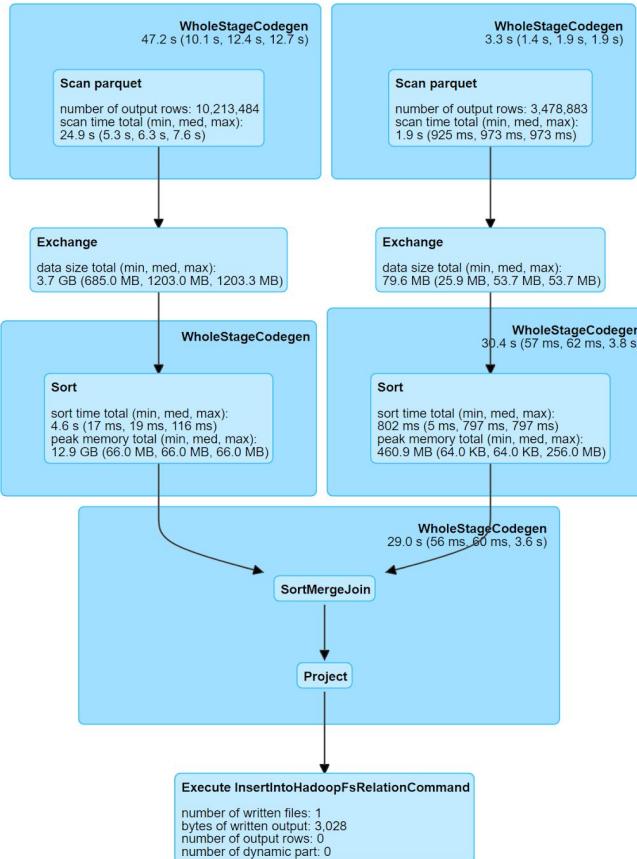
Id	Login
1	User#1
2	User#2
3	User#3
4	User#4

Id	User id
1004	1
1001	2
1002	4
1003	4

Merging

Types of Joins

Sort-Merge Joins:



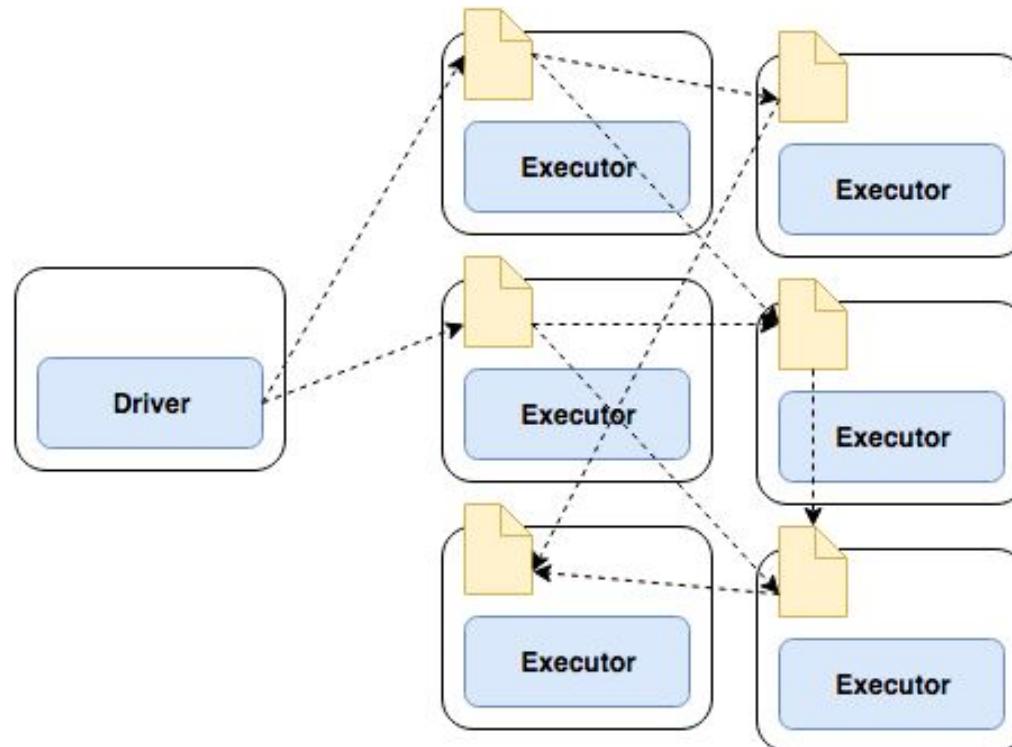
Types of Joins

Broadcast Join:

- ❑ Broadcast joins are the ones which yield the maximum performance in Spark
- ❑ In broadcast join, the smaller table will be broadcasted to all worker nodes
- ❑ Thus, when working with one large table and another smaller table always makes sure to broadcast the smaller table
- ❑ We can hint spark to broadcast a table
- ❑ Spark also internally maintains a threshold of the table size to automatically apply broadcast joins
- ❑ The threshold can be configured using “**spark.sql.autoBroadcastJoinThreshold**” which is by default 10mb

Types of Joins

Broadcast Join:



Types of Joins

Broadcast Join:

```
import org.apache.spark.sql.functions.broadcast  
  
val dataframe = largedataframe.join(broadcast(smalldataframe), "key")
```

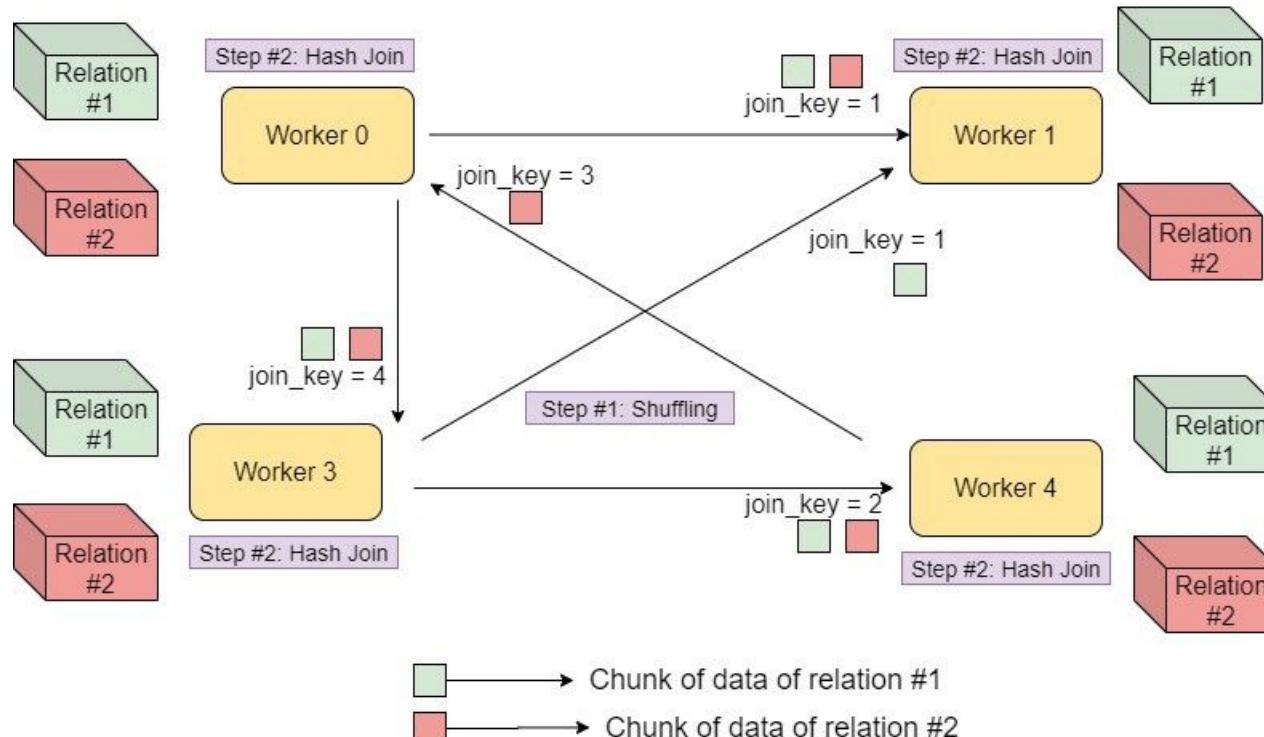
Types of Joins

Shuffle Hash Join:

- ❑ Shuffle Hash join works based on the concept of mapreduce
- ❑ Map through the data frames and use the values of the join column as output key
- ❑ Shuffles the data frames based on the output keys and join the data frames in the reduce phase as the rows from the different data frame with the same keys will end up in the same machine
- ❑ Spark chooses Shuffle Hash join when Sort merge join is turned off or if the key is not suitable and also based on the accompanying two functions

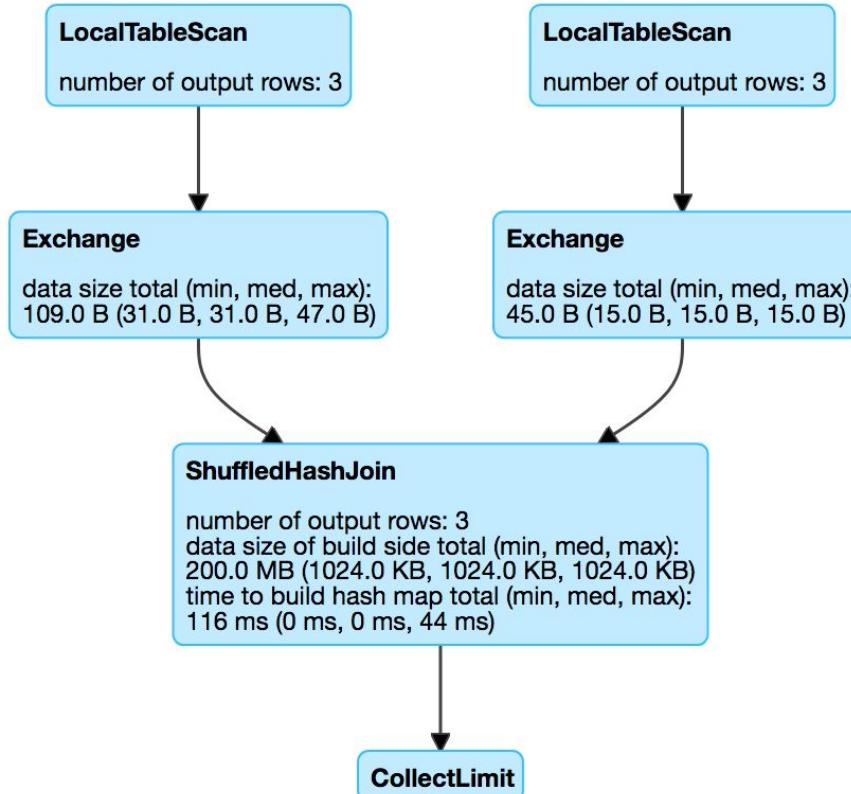
Types of Joins

Shuffle Hash Join:



Types of Joins

Shuffle Hash Join:



Types of Joins

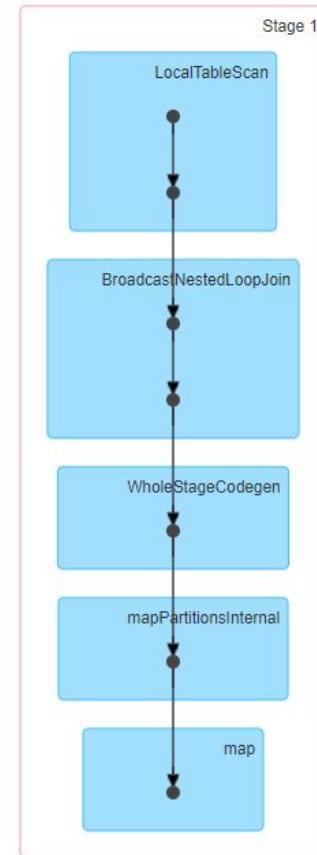
Broadcast Nested Loop Join:

Broadcast Nested Loop join works by broadcasting one of the entire datasets and performing a nested loop to join the data. So essentially every record from dataset 1 is attempted to join with every record from dataset 2

Broadcast Nested Loop is not preferred and could be quite slow. It works for both equi and non-equi joins and it is picked by default when you have a non-equi join

Types of Joins

Broadcast Nested Loop Join:



Types of Joins

Skew Join:

Joins between big tables require shuffling data and the skew can lead to an extreme imbalance of work in the cluster. It's likely that data skew is affecting a query if a query appears to be stuck finishing very few tasks. To verify that data skew is affecting a query:

1. Click the stage that is stuck and verify that it is doing a join
2. After the query finishes, find the stage that does a join and check the task duration distribution
3. Sort the tasks by decreasing duration and check the first few tasks. If one task took much longer to complete than the other tasks, there is skew

Types of Joins

Skew Join:

Say you have to join two tables A and B on $A.id = B.id$. Let's assume that table A has skew on $id=1$

i.e. *select A.id from A join B on A.id = B.id*

There are two basic approaches to solve the skew join issue:

Approach 1:

Break your query/dataset into 2 parts - one containing only skew and the other containing non skewed data. In the above example. query will become -

1. select A.id from A join B on A.id = B.id where A.id <> 1;
2. select A.id from A join B on A.id = B.id where A.id = 1 and B.id = 1;

Types of Joins

Skew Join:

Approach 2:

Also mentioned by LeMuBei above, the 2nd approach tries to randomize the join key by appending extra column. Steps:

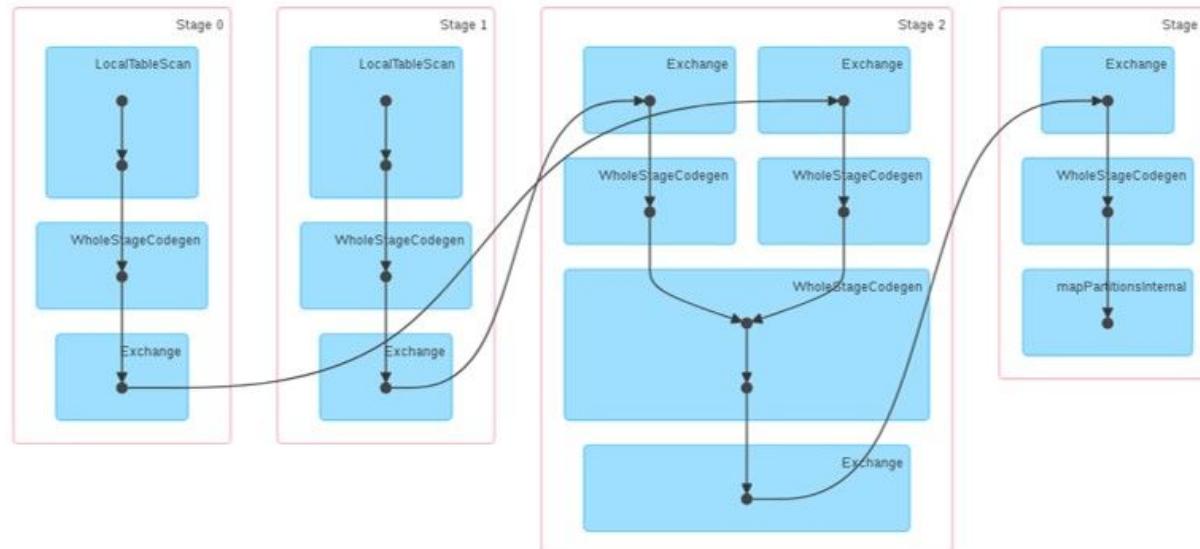
1. Add a column in the larger table (A), say *skewLeft* and populate it with random numbers between 0 to N-1 for all the rows
2. Add a column in the smaller table (B), say *skewRight*. Replicate the smaller table N times. So values in new *skewRight* column will vary from 0 to N-1 for each copy of original data. For this, you can use the explode sql/dataset operator

After 1 and 2, join the 2 datasets/tables with join condition updated to-

```
*A.id = B.id && A.skewLeft = B.skewRight*
```

Types of Joins

Skew Join:



Active Stages (1)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total
2	Collect at <console>:56	+details (kill)	2018/07/13 10:16:24	1.1 min

- Types of Joins
- Quick Recap of MapReduce MapSide and Reduce Side Joins**
- Broadcasting
- Optimizing Sort Merge Join
- Bucketing
- Common Production Issues

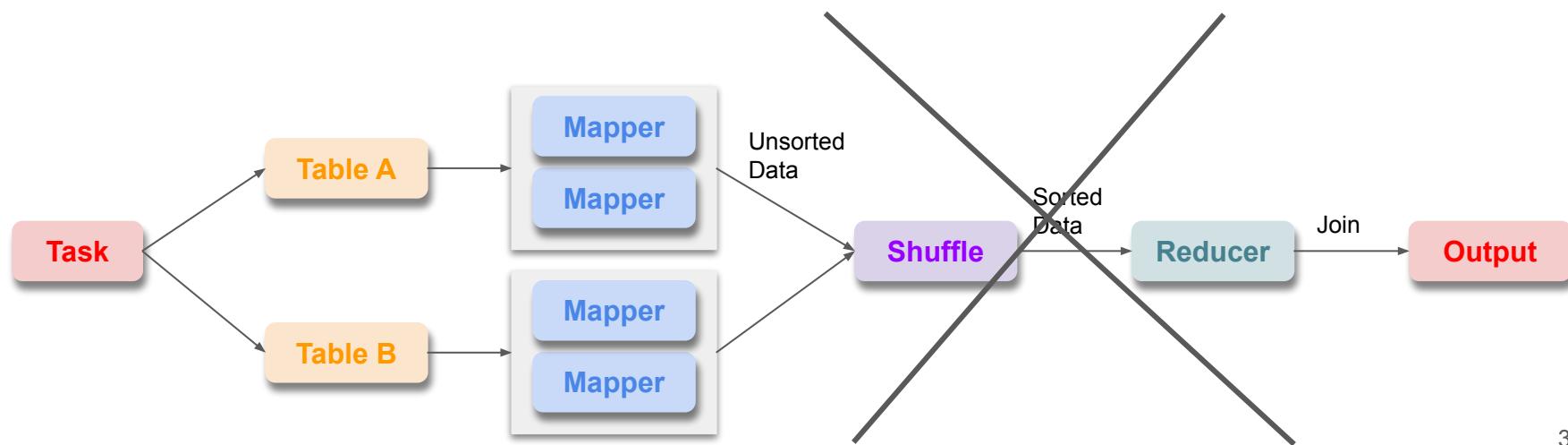
Joins in MapReduce

2 types of join operations in MapReduce:

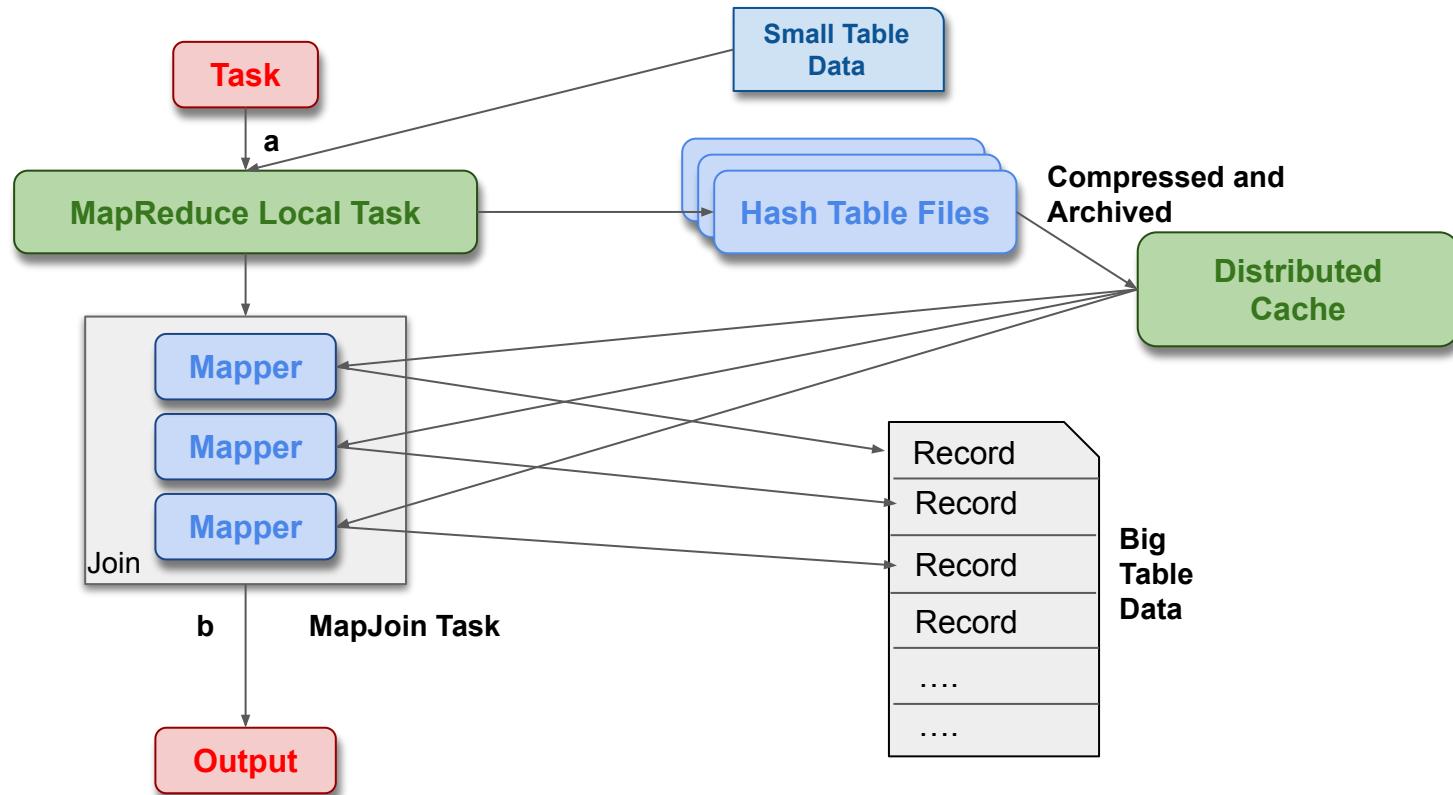
1. Map Side Join
2. Reduce Side Join

Joins in MapReduce

1. **Map Side Join:** Map-side Join is similar to a join but all the task will be performed by the mapper alone. The Map-side Join will be mostly suitable for small tables to optimize the task



Joins in MapReduce



Joins in MapReduce

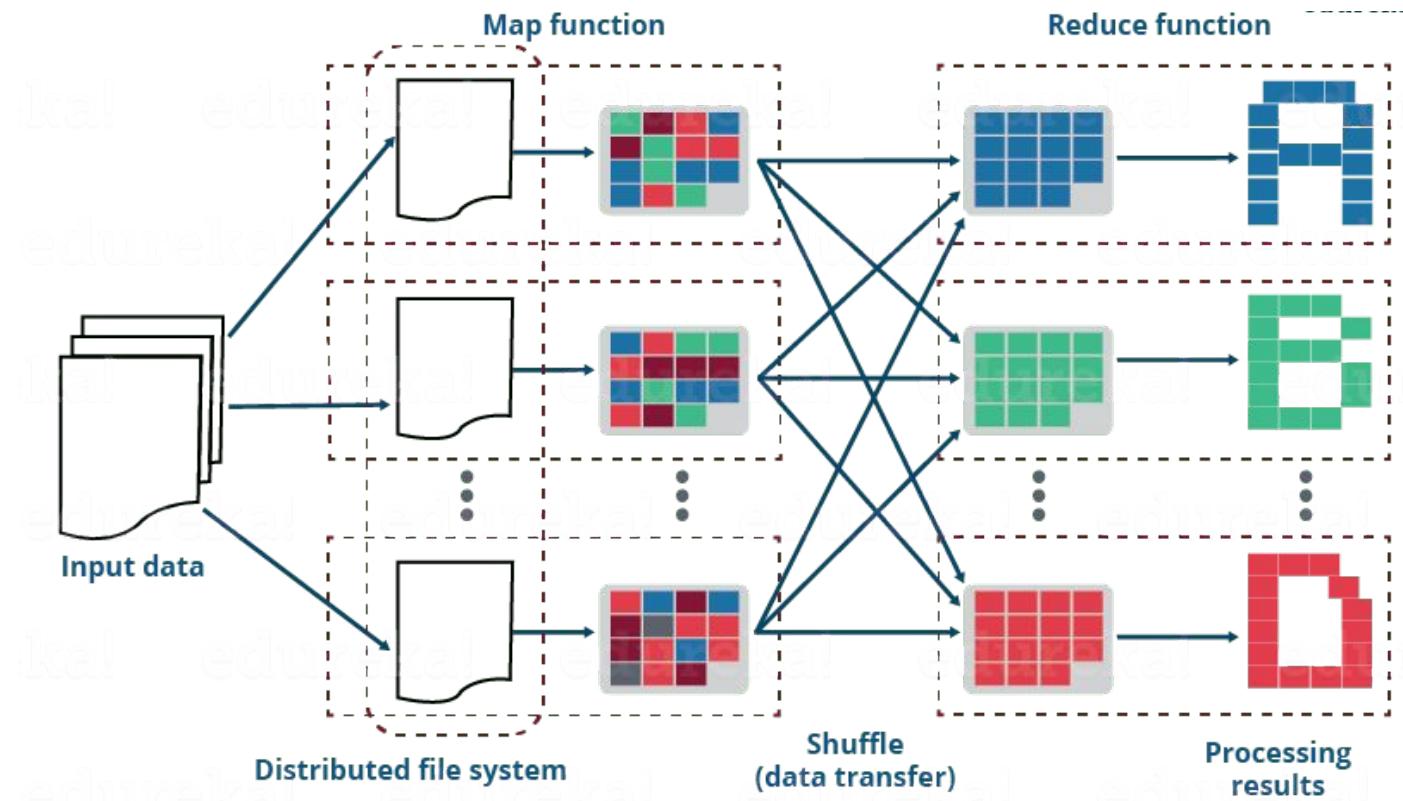
- ❑ **Reduce Side Join:** As the name suggests, in the reduce side join, the reducer is responsible for performing the join operation

Joins in MapReduce

The reduce side join takes place in the following manner:

- ❑ Mapper reads the input data which are to be combined based on common column or join key
- ❑ The mapper processes the input and adds a tag to the input to distinguish the input belonging from different sources or data sets or databases
- ❑ The mapper outputs the intermediate key-value pair where the key is nothing but the join key
- ❑ After the sorting and shuffling phase, a key and the list of values is generated for the reducer
- ❑ Now, the reducers join the values present in the list with the key to give the final aggregated output

Joins in MapReduce



- Types of Joins
- Quick Recap of MapReduce MapSide and Reduce Side Joins
- Broadcasting**
- Optimizing Sort Merge Join
- Bucketing
- Common Production Issues

Broadcasting

- ❑ Broadcast Variables cache read-only data on each node
 - And can be reused at no additional cost
 - Create it via `SparkContext.broadcast()`
 - Access as `variable.value`
 - Broadcast variables are immutable
 - Changes are NOT propagated anywhere (1)
 - We illustrate a broadcast variable sharing the `smallArray`

```
// LargeDF smallArray, and case class Result declared as previously
// Create Broadcast Variable
> val smallArrayBC = sc.broadcast(smallArray)
// Use it in the map
> val mappedDF = largeDF.map(r => {
  if (r.getInt(1) == smallArrayBC.value(0).bit) {
    Result(r.getInt(0), r.getInt(1), smallArrayBC.value(0).bitName)
  } else {
    Result(r.getInt(1), r.getInt(1), smallArrayBC.value(1).bitName)
  } })
```

Broadcasting: Catalyst and Broadcasting

- ❑ Catalyst can handle the broadcast for you
 - You can give it a hint to broadcast (shown below)
 - broadcast() is a function that does this
- ❑ Catalyst uses a BroadcastHashJoin
 - Which broadcasts the smallArrayDF
 - Also —no serializing
 - This is the most efficient plan, by far

```
// largeDF and smallDF declared as previously
import org.apache.spark.sql.functions.broadcast
> largeDF.join(broadcast(smallDF),
    largeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*BroadcastHashJoin [bit#3], [bit#60], Inner, BuildRight
:- *Filter isnotnull(bit#3)
:  +- Scan ExistingRDD[num#2,bit#3]
+- BroadcastExchange HashedRelationBroadcastMode(...))
   +- LocalTableScan [bit#60, bitName#61]
```

Broadcasting: Catalyst Automatic Broadcasting

- ❑ Catalyst can automatically choose a broadcast (see at bottom)
 - If your dataframe size is under the value of config param
spark.sql.autoBroadcastJoinThreshold (10MB default)
 - And catalyst can tell the size of your data (1)
 - We previously disabled this by setting the threshold to -1
 - To illustrate the non-broadcast plan in earlier examples

```
// LargeDF and smallDF declared as previously
// Set broadcast threshold explicitly to 10MB
> spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 1024*1024*10)

> LargeDF.join(smallDF, LargeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*BroadcastHashJoin [bit#3], [bit#8], Inner, BuildRight
:- *Filter isnotnull(bit#3)
:  +- Scan ExistingRDD[num#2,bit#3]
+- BroadcastExchange HashedRelationBroadcastMode(...)
   +- LocalTableScan [bit#8]
```

Broadcasting: Accumulators for Calculations

- ❑ Accumulators are variables that can be added to in parallel
 - Added to via special associative operators ($+=$, add)
 - Workers can add to the variable, only the driver can read it
 - Can be used for counters or sums
 - We illustrate a simple example below
 - Use cases : counting website traffic / transactions per minute .etc

```
> val accum = sc.accumulator(0, "My Accumulator")
accum: org.apache.spark.Accumulator[Int] = 0

> sc.parallelize(Array(1, 2, 3, 4)).
    foreach(x => accum += x)

> accum.value
res5: Int = 10
```

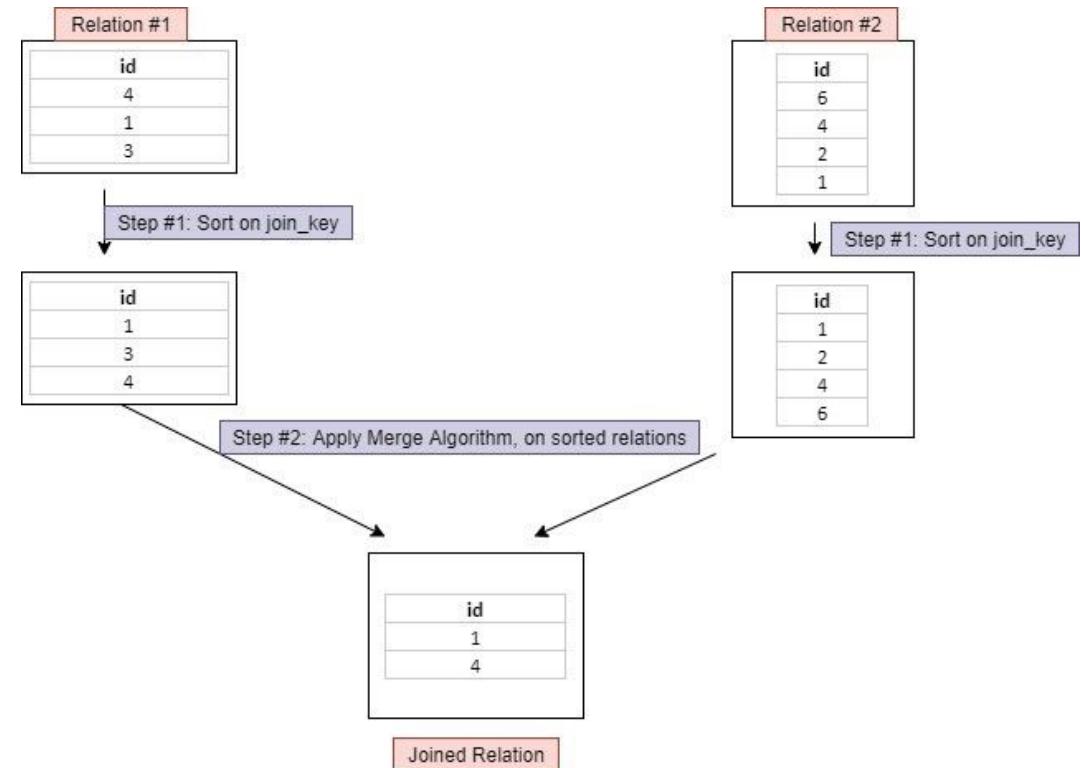
Broadcasting: Summary

- ❑ Useful to reduce shuffling
 - Remember —broadcast data has to fit in memory
 - And if it's not being used more than once, it may be fine to pass it in the transformation instead of broadcasting
- ❑ Catalyst can help you!
 - It has optimizations that will automatically broadcast when appropriate
 - Or you can do it manually
- ❑ Even if you broadcast, you might have other issues
 - e.g. the serialization we saw with the shared data approach

- Types of Joins
- Quick Recap of MapReduce MapSide and Reduce Side Joins
- Broadcasting
- Optimizing Sort Merge Join**
- Bucketing
- Common Production Issues

Optimizing Sort Merge Join

Sort join involves, first sorting the relations based on join keys and then merging both the datasets(think of merge step of merge sort)



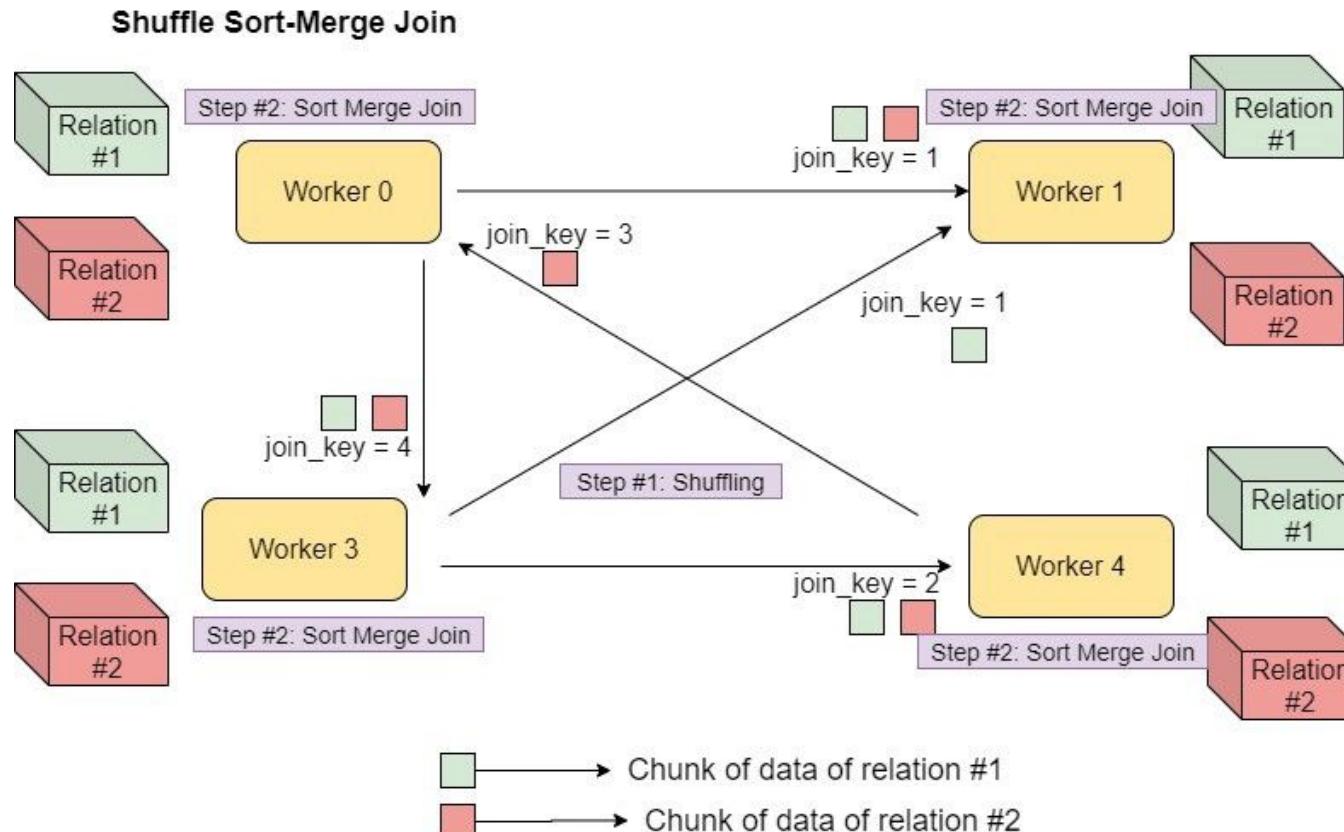
Optimizing Sort Merge Join

Shuffle sort-merge join involves, shuffling of data to get the same join_key with the same worker, and then performing sort-merge join operation at the partition level in the worker nodes

Things to Note:

- Since spark 2.3, this is the default join strategy in spark and can be disabled with **spark.sql.join.preferSortMergeJoin**
- Only supported for '=' join
- The join keys need to be sortable(obviously)
- Supported for all join types

Optimizing Sort Merge Join



Handling data with Skew

Skewed Join: Spark Framework also allows you to use skewed join. For example using Spark SQL. To specify a skewed table, we have to use hints for our SQL queries

```
SELECT /*+ SKEW('orders') */ *
      FROM orders AS o, customers AS c
     WHERE o.customer_id = c.customer_id;
```

- Types of Joins
- Quick Recap of MapReduce MapSide and Reduce Side Joins
- Broadcasting
- Optimizing Sort Merge Join
- Bucketing**
- Common Production Issues

Bucketing

Bucketing is an optimization method that breaks down data into more manageable parts (buckets) to determine the data partitioning while it is written out. The motivation for this method is to make successive reads of the data more performant for downstream jobs if the SQL operators can make use of this property



Bucketing

- ❑ Bucketing is on by default
- ❑ Spark uses the configuration property `spark.sql.sources.bucketing.enabled` to control whether or not it should be enabled and used to optimize requests
- ❑ Bucketing determines the physical layout of the data, so we shuffle the data beforehand because we want to avoid such shuffling later in the process

Bucketing

```
df.write\  
    .bucketBy(16, 'key') \  
    .sortBy('value') \  
    .saveAsTable('bucketed', format='parquet')
```

```
t2 = spark.table('bucketed')  
t3 = spark.table('bucketed')  
  
# bucketed - bucketed join.  
# Both sides have the same bucketing, and no shuffles are needed.  
t3.join(t2, 'key').explain()
```

```
== Physical Plan ==
*(3) Project [key#14L, value#15, value#30]
+- *(3) SortMergeJoin [key#14L], [key#29L], Inner
  :- *(1) Sort [key#14L ASC NULLS FIRST], false, 0
    :  +- *(1) Project [key#14L, value#15]
    :    +- *(1) Filter isnotnull(key#14L)
    :      +- *(1) FileScan parquet default.bucketed[key#14L,value#15] Batched:
+- *(2) Sort [key#29L ASC NULLS FIRST], false, 0
  +- *(2) Project [key#29L, value#30]
    +- *(2) Filter isnotnull(key#29L)
      +- *(2) FileScan parquet default.bucketed[key#29L,value#30] Batched:
```

- Types of Joins
- Quick Recap of MapReduce MapSide and Reduce Side Joins
- Broadcasting
- Optimizing Sort Merge Join
- Bucketing
- Common Production Issues**

Common Issues and their Solutions -

1. **Issue:** Your application runs out of heap space on the executors

Error messages:

- ❑ *java.lang.OutOfMemoryError*: Java heap space on the executors nodes
- ❑ *java.lang.OutOfMemoryError*: GC overhead limit exceeded
- ❑ *org.apache.spark.shuffle.FetchFailedException*

Possible Causes and Solutions

- ❑ An executor might have to deal with partitions requiring more memory than what is assigned. Consider increasing the executor memory or the executor memory overhead to a suitable value for your application
- ❑ Shuffles are expensive operations since they involve disk I/O, data serialization, and network I/O. Avoid shuffles whenever possible. Shuffle operations can result in large partitions where the data is not evenly distributed across them. Also, the shuffle block size is limited to two gigabytes

Common Production Issues

- ❑ Very large partitions may end up with writing blocks greater than two gigabytes. In cases like these, consider increasing the number of partitions in your job. This ensures that there is lesser data per partition
- ❑ If the data that you are dealing with has skewed partitions i.e. certain partitions having huge amount of data compared to the rest, then append some hash value to the end of your key. This will lead to better distribution of your data and you can have an additional aggregate step to remove the appended hash and get back all values for that key

Common Production Issues

- ❑ Another interesting point to remember while re-partitioning is that Spark highly compresses the data if the number of partitions is greater than 2,000. So, if your job is pretty close to having about 2,000 partitions, repartition the data to 2,001 or more partitions
- ❑ While working with RDDs, avoid using groupByKeys. GroupBy keys tend to keep all values for a given key in memory. Keys having a very large value list that cannot be kept in memory will result in OOMs as they aren't spilled to disk. One solution is to replace groupByKeys with reduceByKeys that does a map side combine and decreases the amount of data that is passed to the reducer

Common Issues and their Solutions -

2. **Issue:** Your application runs out of heap space on the driver node

Error messages:

- ❑ *java.lang.OutOfMemoryError*: Java heap space on the driver node

Possible Causes and Solutions

- ❑ First and foremost, consider increasing –driver-memory or driver memory overhead if set to a very low value
- ❑ When you perform actions on the data that ends up collecting the result on the driver end, for example, applying a collect() on the data is an unsafe operation that can lead to an OOM if the collected size is larger than what the driver can house in its memory. If the goal is to save the data, then instead of bringing back all the data to the driver, it is preferable to let the executors parallelly write down each of the resultant partitions into separate files

Common Production Issues

- ❑ When your application has a really large number of tasks (partitions), an OOM on the driver end can occur easily. Every task sends a mapStatus object back to the driver. When there is a shuffle (re-partition, coalesce, reduceByKey, groupByKey, sortByKey) operation involved, data gets re-distributed across executors and leads to the generation of map and reduce tasks
- ❑ Intermediate files are written to disk in the shuffle stage to avoid re-computation. The map status contains location information for the tasks which is sent back to the driver. A large number of tasks would lead to the driver receiving a lot of data and also having to deal with multiple Map output status requests in the reduce phase

Common Issues and their Solutions -

3. **Issue:** Cluster runs out of disk space

Error messages:

- ❑ *java.io.IOException: No space left on device*

Possible Causes and Solutions

- ❑ Check the executors for logs like this `UnsafeExternalSorter: Thread 75 spilling sort data of 141.0 MB to disk (90 times so far)`. This is an indication that the storage data is continuously evicted to disk. Exceptions like this occur when data becomes larger than what is configured to be stored on the node
- ❑ Ensure that the `spark.memory.fraction` isn't too low. The default being 0.6 of the heap space, setting it to a higher value will give more memory for both execution and storage data and will cause lesser spills

Common Production Issues

- ❑ Shuffles involve writing data to disk at the end of the shuffle stage. As a general practice avoid spilling to disk unless the cost of computation is much higher than reading from the disk. One such example to avoid shuffles is to broadcast the smaller table while joining or even partition both the datasets with the same hash partitioner so that keys with the same hash from both tables reside in the same partition
- ❑ If you are running the job with minimal number of nodes, consider adding more nodes to increase the DFS for the cluster

Common Production Issues

Common Issues and their Solutions -

4. **Issue:** Application takes too long to complete or is indefinitely stuck and does not show progress

Possible Causes and Solutions

- ❑ Long running tasks often referred to as stragglers are mostly a result of skewed data. This means there are a few partitions that have more data than the rest causing tasks that deal with them to run for a longer time. The right solution here is to re-partition your data to ensure even distribution among tasks

Common Production Issues

- ❑ Tasks are scheduled to the executors based on the lowest locality level of data. However, a task that is set to run on a particular executor will be handed over to another executor if it is free. This brings in the possibility of the re-assigned task to read data from over the network and might change the locality level of the data for the task
- ❑ Tasks dealing with higher locality levels will face delays due to the increase in network I/O. If you encounter this scenario where most of your stragglers are because of the higher locality levels, consider increasing spark.locality.wait to let the task wait a little longer before it gets reassigned to a free executor

Common Production Issues

Apart from the above errors that revolve under what operations you perform on your data and how efficiently you use the memory resources available, one might also encounter network issues where an executor's heartbeat times out. In such cases, consider increasing your `spark.network.timeout` and `spark.executor.heartbeatInterval`

While avoiding pitfalls is essential, one is always interested in making jobs more performant. Stay tuned for the next post in the series that dives deeper into Spark's memory configuration, on how to set the right parameters for your job and the best practices one must adopt

- Types of Joins
- Quick Recap of MapReduce MapSide and Reduce Side Joins
- Broadcasting
- Optimizing Sort Merge Join
- Bucketing
- Common Production Issues

Agenda

- ❑ Best Practices for writing Spark SQL code
- ❑ Performance Tuning
- ❑ Best Practises

Tuning your applications for SparkSQL:

1. Reduce the Network I/O
2. Reduce Disk I/O
3. Improve/optimize CPU utilization by reducing any unnecessary computation, including filtering out unnecessary data, and ensuring that your CPU resources are getting utilized efficiently
4. Benefit from Spark's in-memory computation, including caching when appropriate

Parallelism

- ❑ Spark decides on the number of partitions based on the file size input. At times, it makes sense to specify the number of partitions explicitly
 - ❑ The read API takes an optional number of partitions
 - ❑ **spark.sql.files.maxPartitionBytes**, available in Spark v2.0.0, for **Parquet**, **ORC**, and **JSON**
- ❑ The shuffle partitions may be tuned by setting **spark.sql.shuffle.partitions**, which defaults to **200**. This is really small if you have large dataset sizes

Reduce shuffle

- ❑ Tune the **spark.sql.shuffle.partitions**
- ❑ Partition the input dataset appropriately so each task size is not too big
- ❑ Use the Spark UI to study the plan to look for opportunity to reduce the shuffle as much as possible
- ❑ Formula recommendation for **spark.sql.shuffle.partitions**:
 - ❑ For large datasets, aim for anywhere from **100MB** to less than **200MB** task target size for a partition (use target size of 100MB, for example)
 - ❑ **spark.sql.shuffle.partitions = quotient (shuffle stage input size/target size)/total cores) * total cores**

Filter/Reduce dataSet size

- ❑ Use appropriate filter predicates in your SQL query so Spark can push them down to the underlying datasource; selective predicates are good
- ❑ Use them as appropriate
- ❑ Use partition filters if they are applicable
- ❑ Look for opportunities to filter out data as early as possible in your application pipeline
- ❑ If there is a filter operation and you are only interested in doing analysis for a subset of the data, apply this filter early

Cache appropriately

- ❑ Use caching when the same operation is computed multiple times in the pipeline flow
- ❑ Use caching using the persist API to enable the required cache setting (persist to disk or not; serialized or not)
- ❑ Be aware of lazy loading and prime cache if needed up-front. Some APIs are eager and some are not
- ❑ Check out the Spark UI's Storage tab to see information about the datasets you have cached
- ❑ It's good practice to unpersist your cached dataset when you are done using them in order to release resources, particularly when you have other people using the cluster as well

Join

- ❑ Join order matters; start with the most selective join. For relations less than `spark.sql.autoBroadcastJoinThreshold`, you can check whether broadcast **HashJoin** is picked up
- ❑ Use SQL hints if needed to force a specific type of join
 - ❑ **Example:** When joining a small dataset with large dataset, a broadcast join may be forced to broadcast the small dataset
 - ❑ Confirm that Spark is picking up broadcast hash join; if not, one can force it using the SQL hint
- ❑ Avoid **cross-joins**
- ❑ Broadcast **HashJoin** is most performant, but may not be applicable if both relations in join are large
- ❑ Collect statistics on tables for Spark to compute an optimal plan

Tune cluster resources

- ❑ Tune the resources on the cluster depending on the resource manager and version of Spark
- ❑ Tune the available memory to the driver: **spark.driver.memory**
- ❑ Tune the number of executors and the memory and core usage based on resources in the cluster: **executor-memory**, **num-executors**, and **executor-cores**

Avoid expensive operations

- ❑ Avoid **order by** if it is not needed
- ❑ When you are writing your queries, instead of using **select *** to get all the columns, only retrieve the columns relevant for your query
- ❑ Don't call count unnecessarily

Data skew

- ❑ Ensure that the partitions are equal in size to avoid data skew and low CPU-utilization issues
- ❑ Repartition will cause a shuffle, and shuffle is an expensive operation, so this should be evaluated on an application basis
- ❑ Use the Spark UI to look for the partition sizes and task duration

UDFs

- ❑ Spark has a number of built-in user-defined functions (UDFs) available
- ❑ For performance, check to see if you can use one of the built-in functions since they are good for performance
- ❑ Custom UDFs in the Scala API are more performant than Python UDFs
- ❑ If you have to use the Python API, use the newly introduced pandas UDF in Python that was released in Spark 2.3
- ❑ The pandas UDF (vectorized UDFs) support in Spark has significant performance improvements as opposed to writing a custom Python UDF

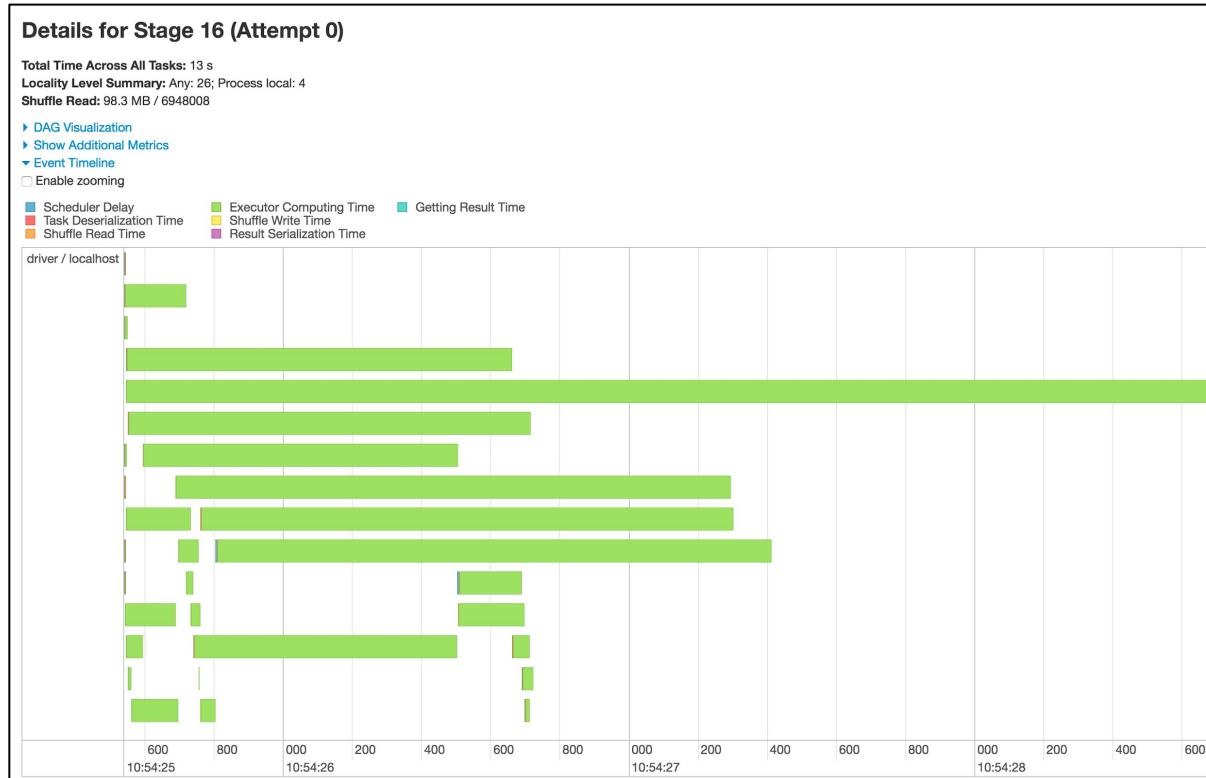
Agenda

- Best Practices for writing Spark SQL code
- Performance Tuning**
- Best Practises

Performance Tuning

- ❑ It will point you to problem areas
 - We've seen it in many labs and slides (port 4040)
- ❑ Things to look at
 - Task execution time
 - Amount of shuffle data
 - Partitions and overall data volume
 - GC time
 - DAG lineage
- ❑ Can run the history server for stats of completed jobs
- ❑ Pulls info from application event logs
- ❑ Displays it in typical Spark UI form

Performance Tuning: Task Execution Time



Performance Tuning: Amount of Shuffle Data

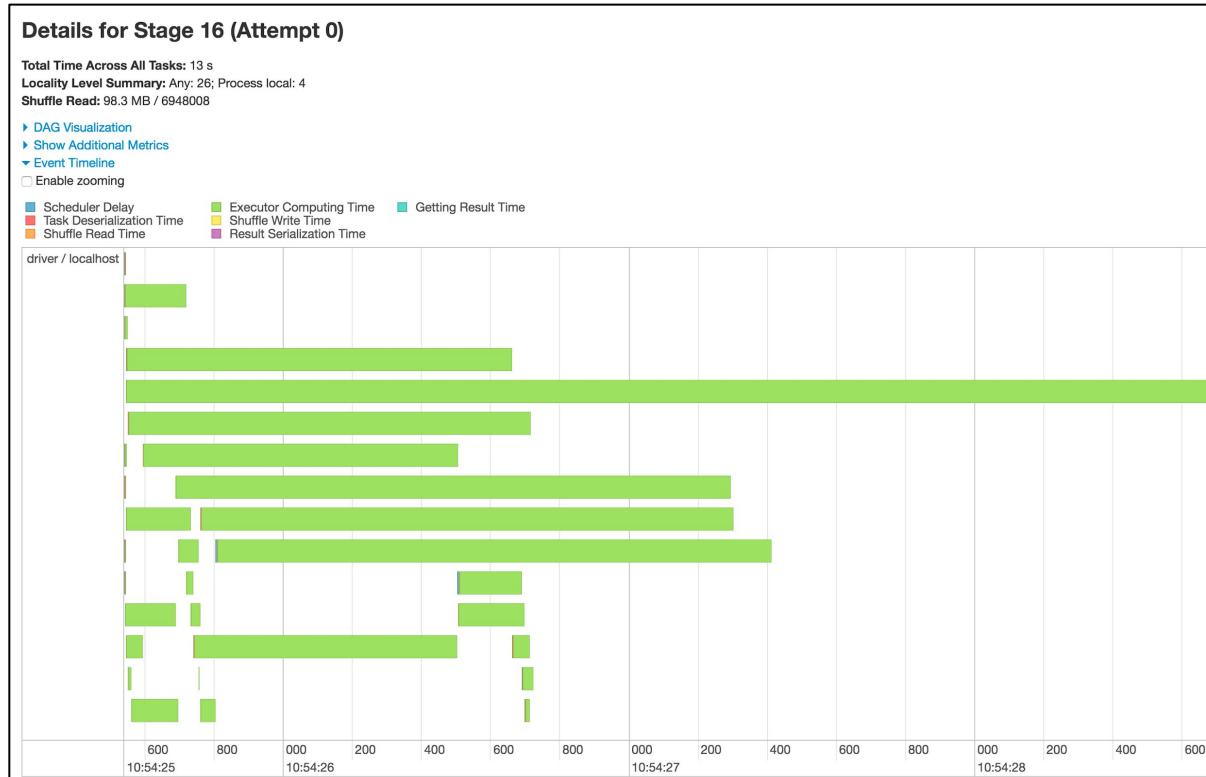
- ❑ One technique is to remove the shuffle completely
 - By sending data from the client (the driver program)
 - We illustrate this below

```
> val largeDF = // ... As previously - has 1M rows
> case class Bit(bit:Int, bitName: String)
> val smallArray = List (Bit(0,"zero"), Bit(1,"one"))
> case class Result(num:Int, bit:Int, bitName:String)
> val mappedDF = largeDF.map(r => {
    if ( r.getInt(1) == smallArray(0).bit ) {
        Result(r.getInt(0), r.getInt(1), smallArray(0).bitName)
    } else {
        Result(r.getInt(1), r.getInt(1), smallArray(1).bitName)
    } } )
> mappedDF.limit(2).show
+---+---+
|num|bit|bitName|
+---+---+
| 1| 1| one|
| 2| 0| zero|
```

Performance Tuning

- ❑ It will point you to problem areas
 - We've seen it in many labs and slides (port 4040)
- ❑ Things to look at
 - Task execution time
 - Amount of shuffle data
 - Partitions and overall data volume
 - GC time
 - DAG lineage
- ❑ Can run the history server for stats of completed jobs
- ❑ Pulls info from application event logs
- ❑ Displays it in typical Spark UI form

Performance Tuning: Task Execution Time

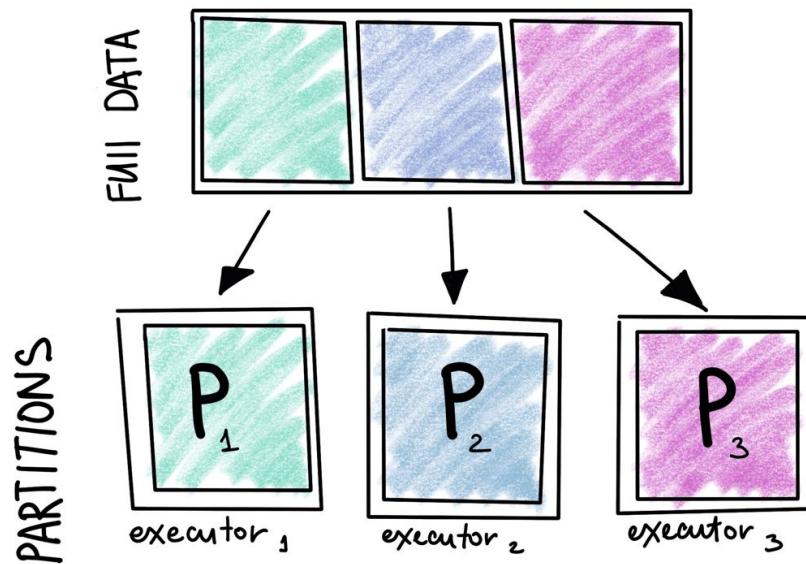


Performance Tuning: Amount of Shuffle Data

- ❑ One technique is to remove the shuffle completely
 - By sending data from the client (the driver program)
 - We illustrate this below

```
> val largeDF = // ... As previously - has 1M rows
> case class Bit(bit:Int, bitName: String)
> val smallArray = List (Bit(0,"zero"), Bit(1,"one"))
> case class Result(num:Int, bit:Int, bitName:String)
> val mappedDF = largeDF.map(r => {
    if ( r.getInt(1) == smallArray(0).bit ) {
        Result(r.getInt(0), r.getInt(1), smallArray(0).bitName)
    } else {
        Result(r.getInt(1), r.getInt(1), smallArray(1).bitName)
    } } )
> mappedDF.limit(2).show
+---+---+
|num|bit|bitName|
+---+---+
| 1| 1| one|
| 2| 0| zero|
```

Performance Tuning: Partition and Overall Data Volume



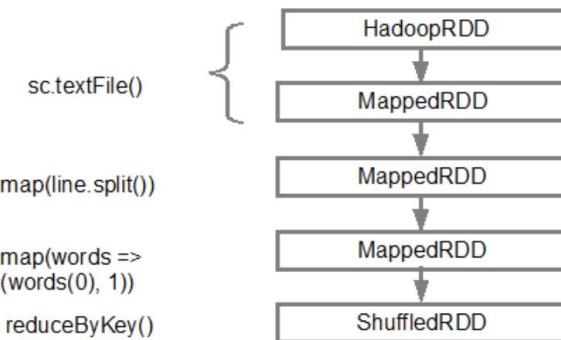
Performance Tuning: GC Time

Spark can store large amounts of data in memory, it has a major reliance on Java's memory management and garbage collection (**GC**)

Garbage Collector	Running Time for 88GB Heap
Parallel GC	6.5min
CMS GC	9min
G1 GC	7.6min

Performance Tuning: DAG Lineage

- ❑ Once an Action is called on RDD, Spark creates the DAG and submits it to the DAG scheduler
- ❑ The DAG scheduler divides operators into stages of tasks
- ❑ The DAG scheduler pipelines operators together e.g. Many map operators can be scheduled in a single stage
- ❑ The final result of a DAG scheduler is a set of stages



Agenda

- Best Practices for writing Spark SQL code
- Performance Tuning
- Best Practises**

Best Practices: Use Efficient Transformations

- ❑ Transformation choices will greatly affect your performance
 - We've seen that many times
 - A few guidelines will help
- ❑ Use DataFrames/Datasets for Catalyst/Tungsten benefits
- ❑ Minimize shuffles and use effective joins
 - As seen earlier —Catalyst will help you with this
 - Consider broadcast variables and accumulators
- ❑ Beware of lambda functions —they raise issues with Catalyst/Tungsten
- ❑ Be aware of your data characteristics
 - e.g. is all your data on one partition —problem!

Best Practices: Filter Early

- ❑ Filter out unused data as early as possible
 - ❑ Don't process it, then throw it out
 - ❑ More network traffic, CPU usage, storage requirements, etc
 - ❑ Catalyst will help you with this, but not always
 - ❑ Depending on your transformations
- ❑ Filter runs parallel on all partitions
 - ❑ Narrow dependency —very efficient
- ❑ Filtered partitions can be unbalanced
 - ❑ With one much larger than another
 - ❑ Beware of this —consider coalesce or repartition
 - ❑ Be judicious—these can be expensive
 - ❑ Check the resource usage (e.g. Web UI)

Best Practices: Use Good Data Storage

- ❑ Spark needs a reliable data store for data
 - Several choices
- ❑ HDFS is often a good choice
 - Cheap(er), Reliable, Scalable
 - Proven infrastructure
- ❑ NoSQL data stores
 - Good for real time work loads
 - Cassandra, Couchbase, Aerospike, etc
 - Many are already integrated with Spark
- ❑ Exploit data locality
 - Process data on same node —supporting high throughput
 - Works well with HDFS, as Spark is understands its storage

Best Practices: Monitor, Monitor, Monitor

- ❑ Put resources into planning your monitoring
 - Will help you diagnose performance issues
- ❑ Monitor at the system level
 - CPU, Memory
 - JVM, Garbage collection
- ❑ Monitor at application level
 - Transformation times and resources
 - Should be checked while developing —often by developer
- ❑ Collect and graph metrics
 - Codahale, Graphite, Graphana

Best Practices: Don't Reinvent the Wheel

- ❑ Spark has a large community
 - Someone should be a part of it to gain from its experience
- ❑ There are a lot of resources
 - There are a number of excellent books by Spark contributors
 - This is true even for Spark 3
 - Use them, follow their expert advice
- ❑ Don't start from scratch with a square wheel !
 - Hey all —this didn't work, you should go another route



Agenda

- Best Practices for writing Spark SQL code
- Performance Tuning
- Best Practises



Thank You!!