



UNIVERSITAT DE  
BARCELONA

# Master in Fundamental Principles of Data Science

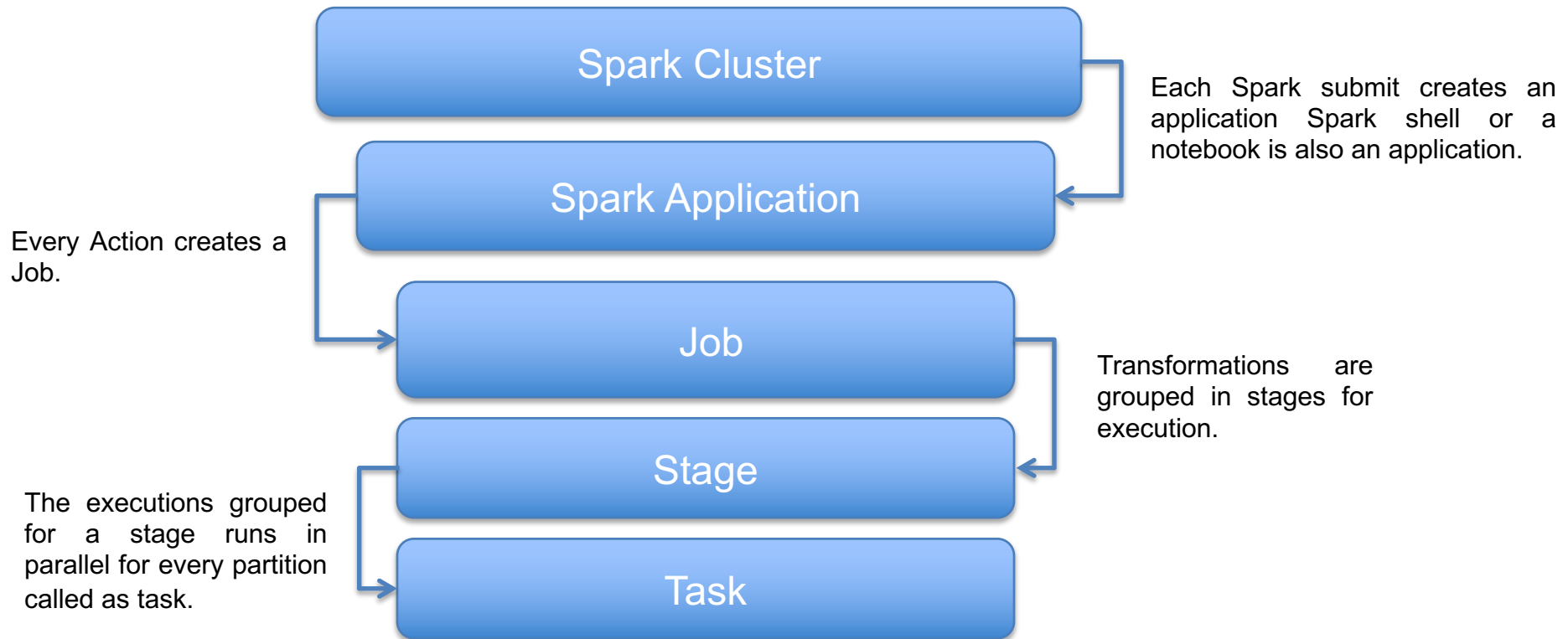
Dr Rohit Kumar



UNIVERSITAT DE  
BARCELONA

# Spark Advanced

# Spark Execution Life cycle

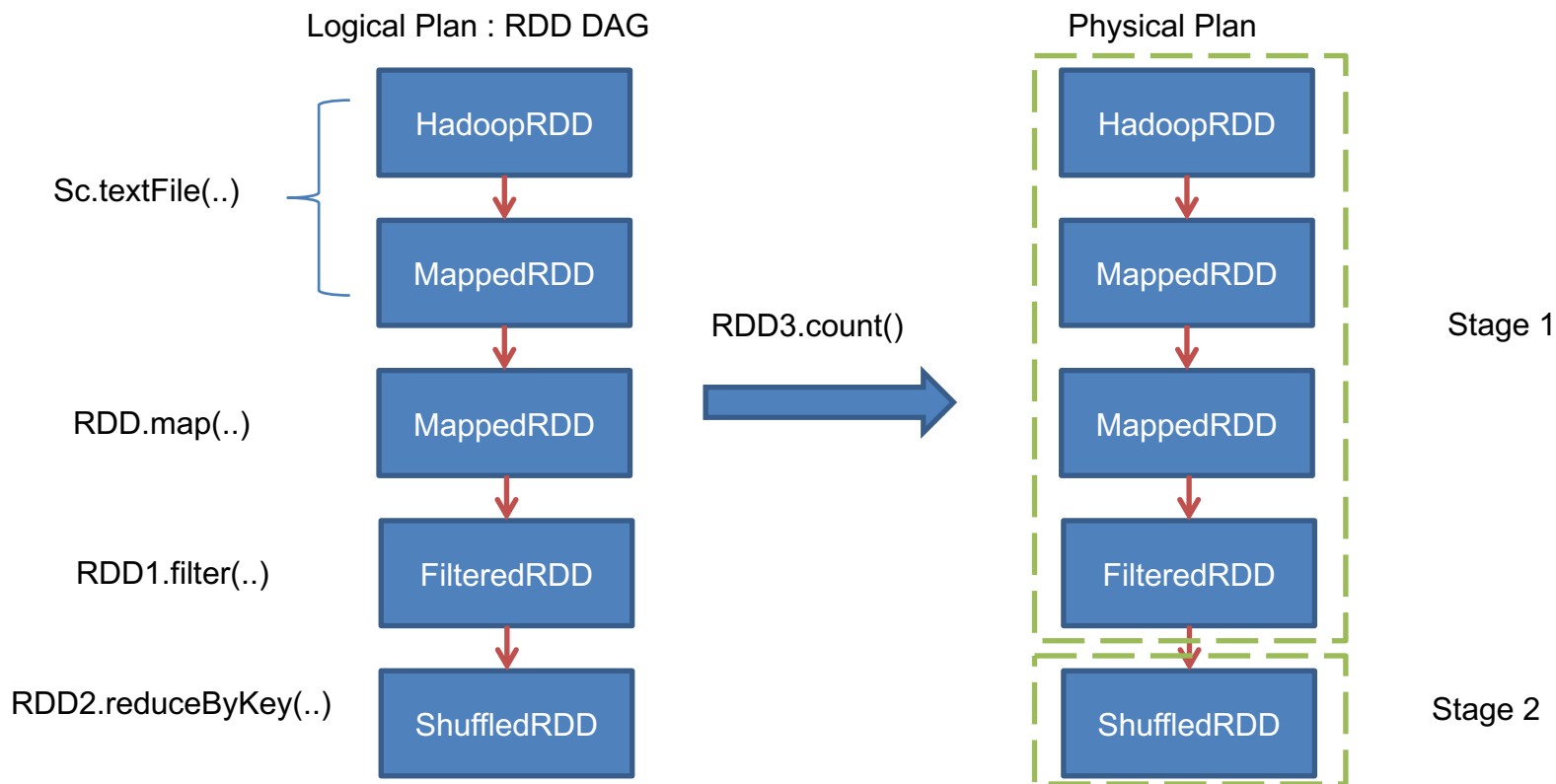


# Monitoring and Debugging Spark

Spark transformations creates a DAG which is a “logical plan” when Spark action is called the DAG is used to create a “physical plan” which consist of grouping transformations together in stages.



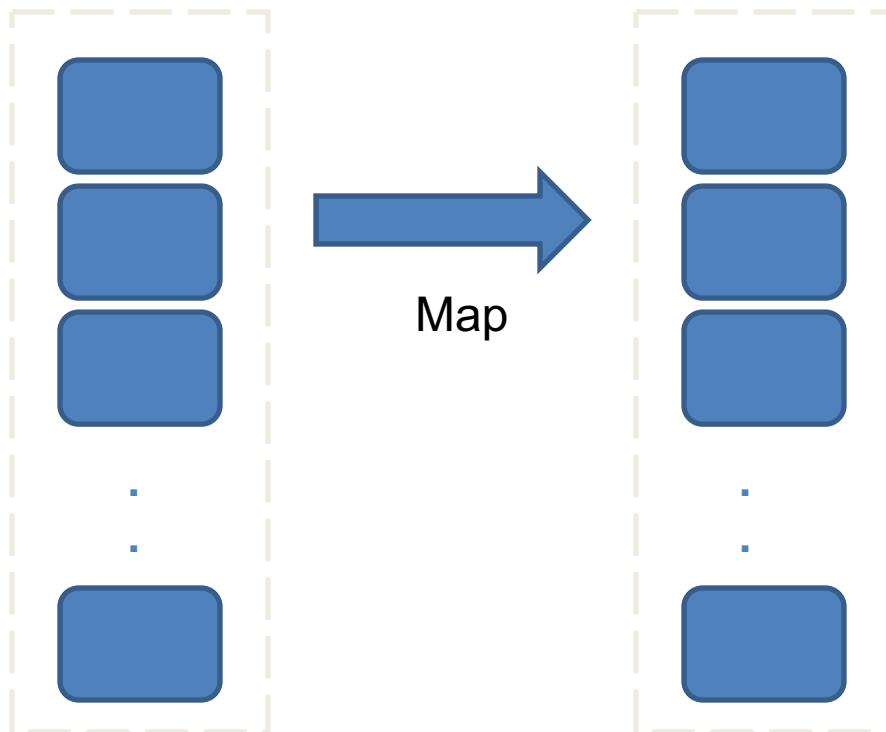
# Logical Plan to Physical plan



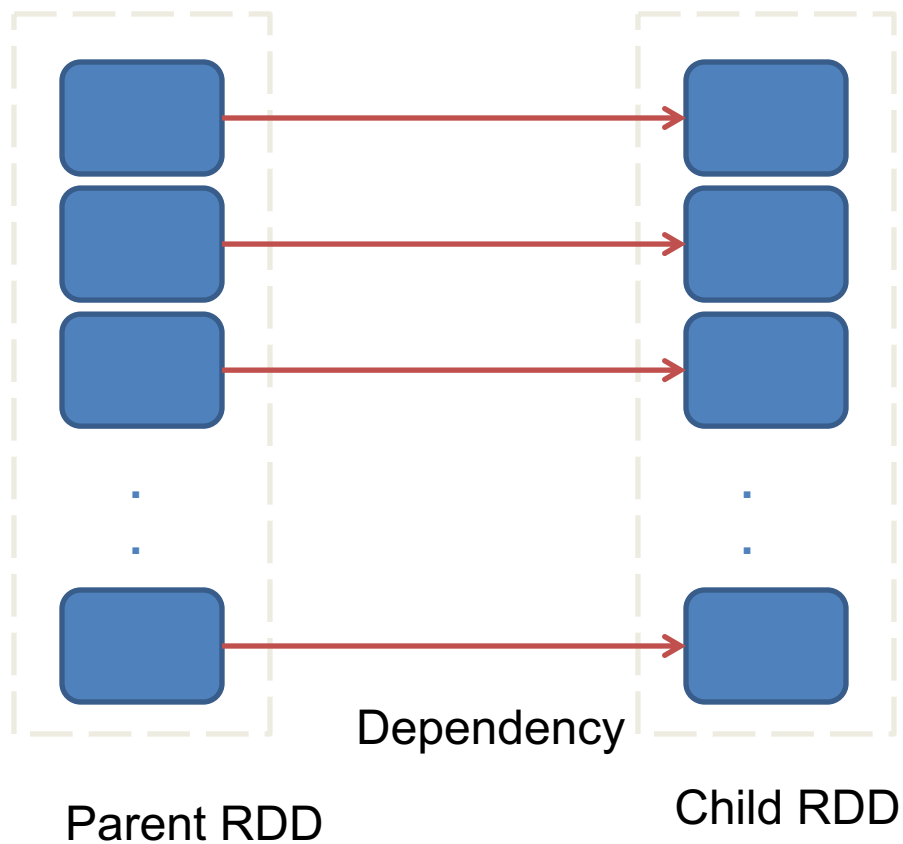
# Pipelining

- Spark tries to run as many transformations in one stage as possible when pipelining is possible.
- New stage is created when data needs to move across different stages like shuffle.

# RDD dependency



# RDD dependency



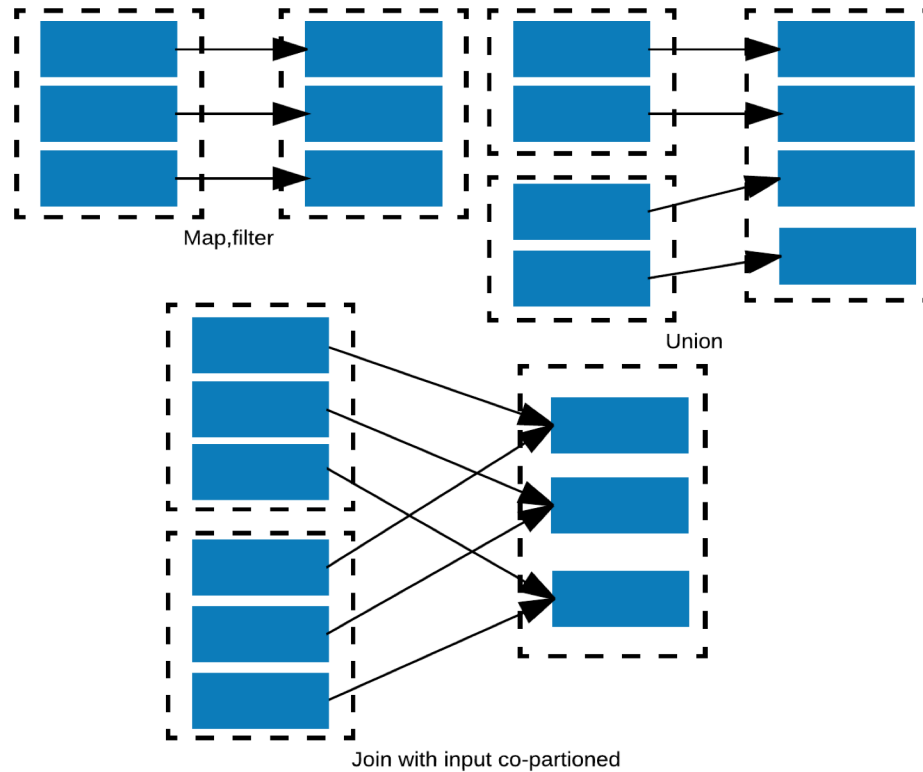




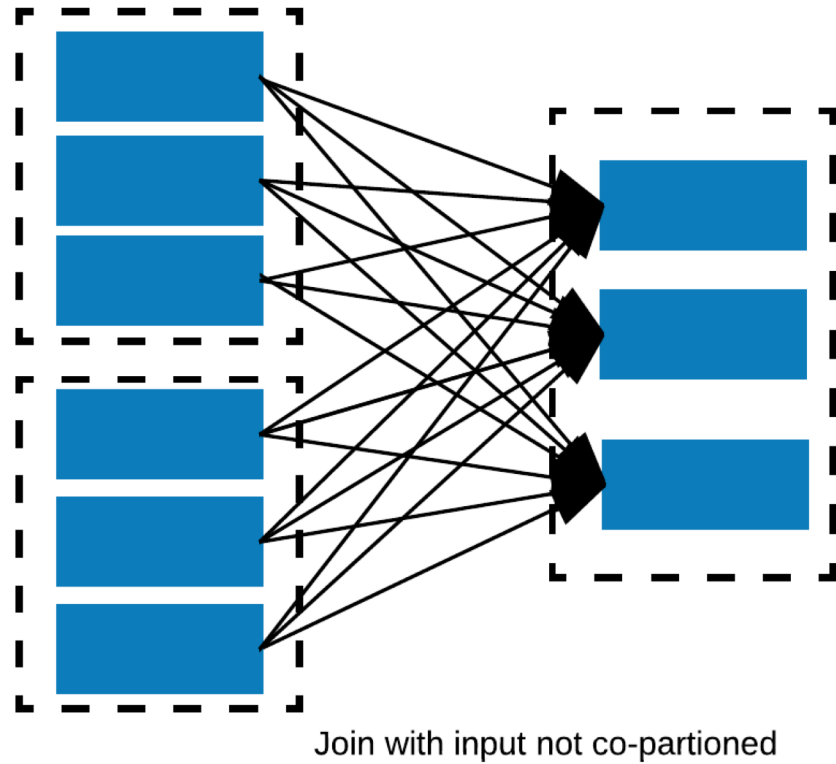
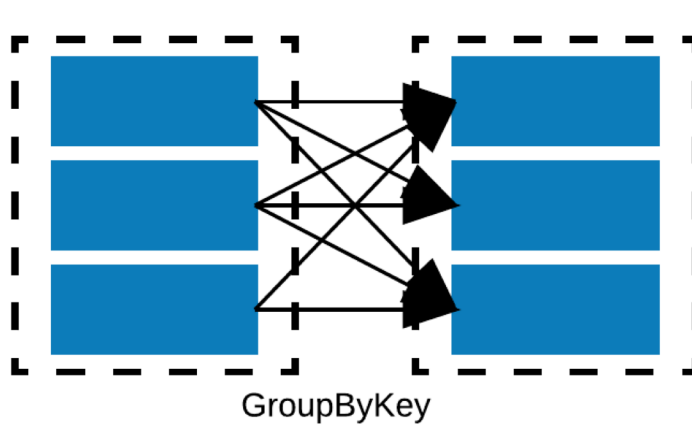
# RDD dependency

- **Narrow dependency:** Each partition of the parent RDD is used by at most one partition of the child RDD. Fast!!
- **Wide dependency:** Each partition of the parent RDD may be used by multiple child partitions. Slow!!

# Narrow Dependency



# Wide Dependency



# Debugging Spark

- Spark web UI : Real time monitoring
- Driver and Executors logs : After the fact.

# Recap

The important components of execution:

- **Task**: a unit of execution that runs on a single machine
- **Stage**: a group of tasks, based on partitions of the input data, which will perform the same computation in parallel
- **Job**: has one or more stages
- **Pipelining**: collapsing of RDDs into a single stage, when RDD transformations can be computed without data movement
- **DAG**: Logical graph of RDD operations
- **RDD**: Parallel dataset with partitions

# Spark web UI

Every SparkContext (spark application) launches a web UI, by default on port 4040 of the driver node, that displays useful information about the application. This includes:

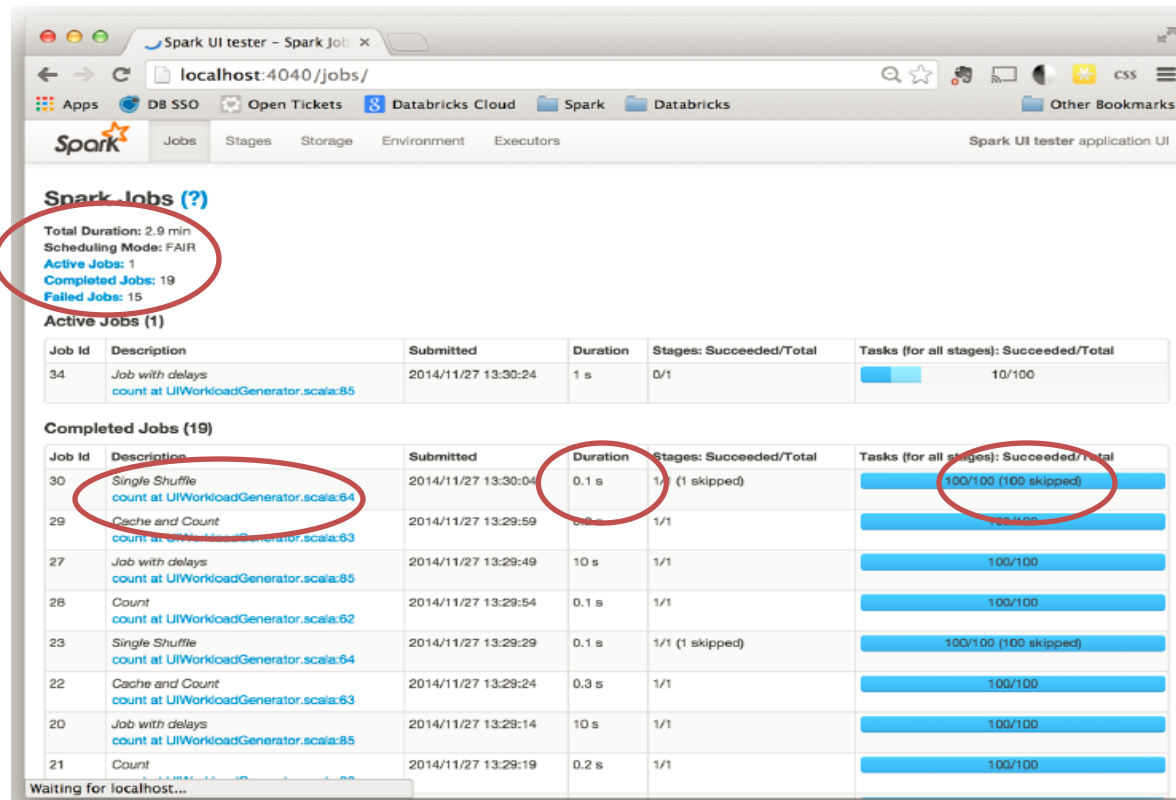
- A list of scheduler stages and tasks
- A summary of RDD sizes and memory usage
- Environmental information.
- Information about the running executors

If multiple host is running in same host they will get assigned to 4041,4042..

# Spark web UI

- Jobs tab: Contains information about the job details of all the jobs.
- Stage tab: Contains information about all the stages details.
- Storage tab: Contains information about cached RDDs.
- Environment tab: Contains information about the spark cluster configuration details.
- Executors tab: Contains information about all the live and dead executors.

# Spark web UI: Jobs



The screenshot shows the Spark web UI interface for the 'Jobs' tab. The browser address bar indicates the URL is `localhost:4040/jobs/`. The page title is 'Spark Jobs (?)'. A summary box on the left shows: 'Total Duration: 2.9 min', 'Scheduling Mode: FAIR', 'Active Jobs: 1', 'Completed Jobs: 19', and 'Failed Jobs: 15'. Below this, the 'Active Jobs (1)' section shows a single job with ID 34. The 'Completed Jobs (19)' section lists 19 jobs, with the first job (ID 30) highlighted by a red circle. The job details for ID 30 are: 'Single Shuffle count at UIWorkloadGenerator.scala:64', submitted at 2014/11/27 13:30:04, duration 0.1 s, stages 1/1 (1 skipped), and tasks 100/100 (100 skipped). The 'Tasks' column for this job is also circled in red.

**Spark Jobs (?)**

Total Duration: 2.9 min  
Scheduling Mode: FAIR  
Active Jobs: 1  
Completed Jobs: 19  
Failed Jobs: 15

**Active Jobs (1)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
34	Job with delays count at UIWorkloadGenerator.scala:85	2014/11/27 13:30:24	1 s	0/1	10/100

**Completed Jobs (19)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
30	Single Shuffle count at UIWorkloadGenerator.scala:64	2014/11/27 13:30:04	0.1 s	1/1 (1 skipped)	100/100 (100 skipped)
29	Cache and Count count at UIWorkloadGenerator.scala:63	2014/11/27 13:29:59	0.3 s	1/1	100/100
27	Job with delays count at UIWorkloadGenerator.scala:85	2014/11/27 13:29:49	10 s	1/1	100/100
28	Count count at UIWorkloadGenerator.scala:62	2014/11/27 13:29:54	0.1 s	1/1	100/100
23	Single Shuffle count at UIWorkloadGenerator.scala:64	2014/11/27 13:29:29	0.1 s	1/1 (1 skipped)	100/100 (100 skipped)
22	Cache and Count count at UIWorkloadGenerator.scala:63	2014/11/27 13:29:24	0.3 s	1/1	100/100
20	Job with delays count at UIWorkloadGenerator.scala:85	2014/11/27 13:29:14	10 s	1/1	100/100
21	Count count at UIWorkloadGenerator.scala:62	2014/11/27 13:29:19	0.2 s	1/1	100/100

Waiting for localhost...



# Spark web UI: Jobs

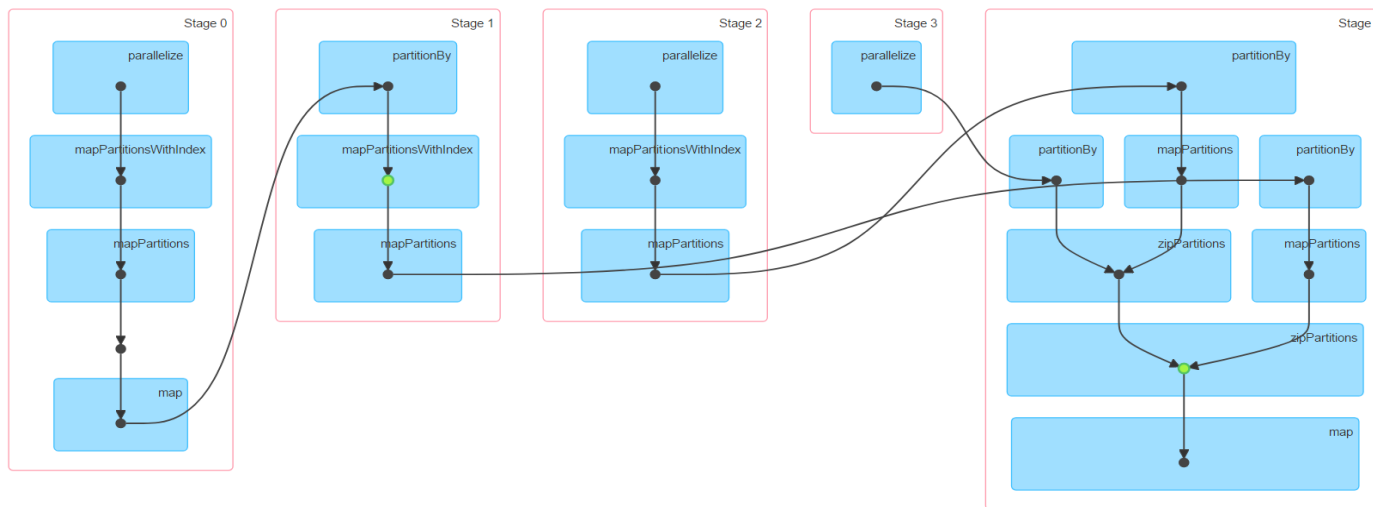
## Details for Job 0

Status: SUCCEEDED

Completed Stages: 5

► Event Timeline

▼ DAG Visualization



Completed Stages: 5



# Spark web UI: Stages



Jobs

Stages

Storage

Environment

Executors

GraphAlgos-SP application UI

## Details for Stage 9 (Attempt 0)

Total Time Across All Tasks: 0.3 s

Locality Level Summary: Process local: 8

Input Size / Records: 503.7 KB / 8

- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▶ Event Timeline

### Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	32 ms	36 ms	36 ms	36 ms	36 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Input Size / Records	60.0 KB / 1	62.6 KB / 1	63.0 KB / 1	64.3 KB / 1	64.9 KB / 1

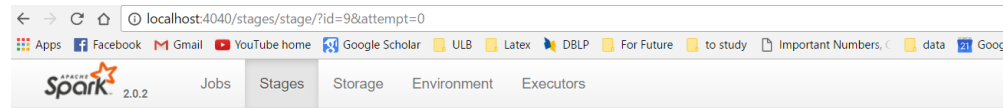
### Aggregated Metrics by Executor

Executor ID ^	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records
driver	192.168.1.49:54684	0.4 s	8	0	8	503.7 KB / 8

### Tasks (8)

Index ^	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	40	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/06/23 22:49:46	36 ms		60.0 KB / 1	

# Spark web UI: Stages



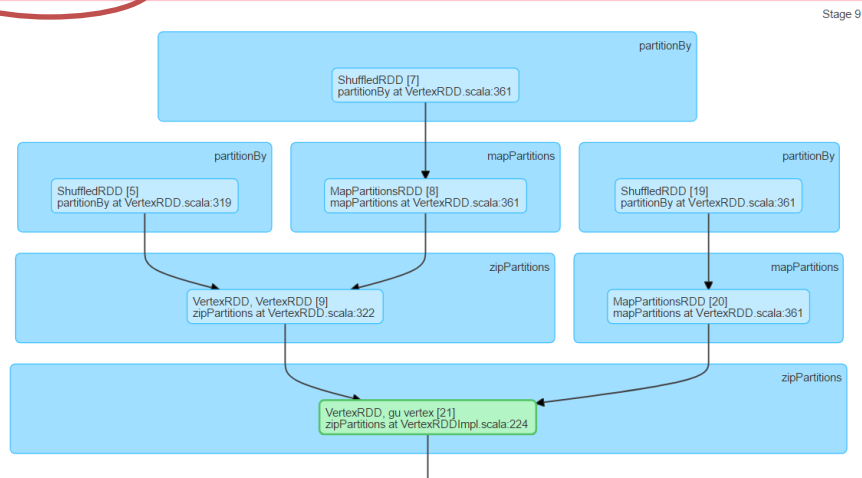
## Details for Stage 9 (Attempt 0)

Total Time Across All Tasks: 0.3 s

Locality Level Summary: Process local: 8

Input Size / Records: 503.7 KB / 8

▼ DAG Visualization



# Spark web UI: Storage

## Storage

### RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
EdgeRDD, gu edges	Memory Deserialized 1x Replicated	8	100%	2.9 MB	0.0 B
EdgeRDD	Memory Deserialized 1x Replicated	8	100%	2.9 MB	0.0 B
VertexRDD	Memory Deserialized 1x Replicated	8	100%	331.4 KB	0.0 B
VertexRDD, gu vertex	Memory Deserialized 1x Replicated	8	100%	503.7 KB	0.0 B
EdgeRDD	Memory Deserialized 1x Replicated	8	100%	2.9 MB	0.0 B

The Fraction cached can show more than 100% in case of more than one replica or when an RDD is Recovered from failure.

Size on Disk will be shown when persist uses disk mode.

You can assign name to your RDDs and it will appear here.



# Spark web UI: Environment

[Jobs](#)[Stages](#)[Storage](#)[Environment](#)[Executors](#)

## Environment

### Runtime Information

Name	Value
Java Home	C:\Program Files\Java\jdk1.7.0_79\jre
Java Version	1.7.0_79 (Oracle Corporation)
Scala Version	version 2.11.7

### Spark Properties

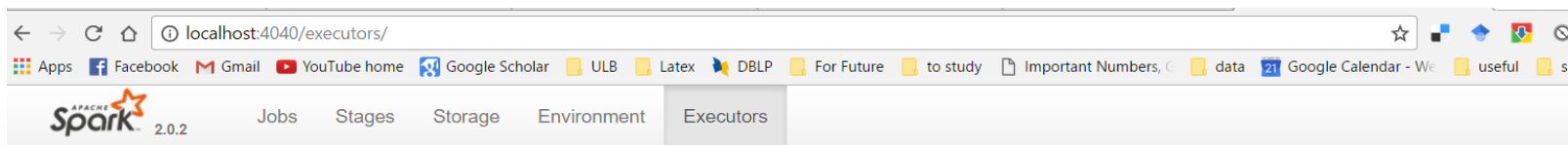
Name	Value
spark.app.id	local-1498250896300
spark.app.name	GraphAlgos-SP
spark.driver.host	192.168.1.49
spark.driver.port	54643
spark.executor.id	driver
spark.kryo.classesToRegister	com.ulb.code.wit.main.VertexPartitioner,com.ulb.code.wit.main.MyPartitionStrategy
spark.master	local[1]
spark.scheduler.mode	FIFO
spark.serializer	org.apache.spark.serializer.KryoSerializer

### System Properties

Name	Value
awt.toolkit	sun.awt.windows.WToolkit



# Spark web UI: Executor



## Executors

### 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
Active(1)	0	0.0 B / 3.3 GB	0.0 B	8	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(1)	0	0.0 B / 3.3 GB	0.0 B	8	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B

### Executors

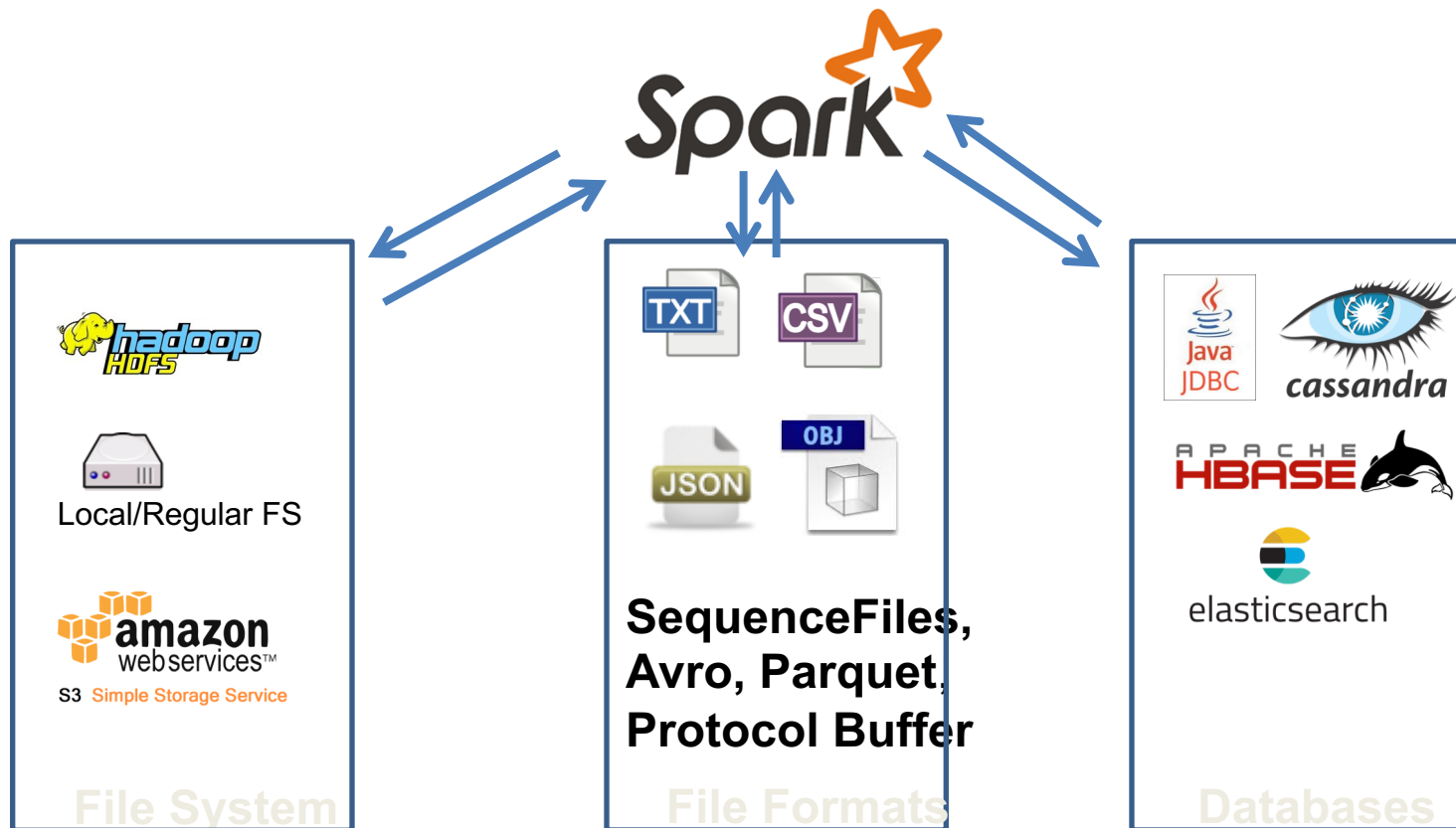
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	192.168.1.49:54684	Active	0	0.0 B / 3.3 GB	0.0 B	8	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	<a href="#">Thread Dump</a>

# Driver and Executor Log

The exact location of Spark's logfiles depends on the deployment mode:

- **In Spark's Standalone mode:** application logs are directly displayed in the standalone master's web UI. They are stored by default in the *work/* directory of the Spark distribution on each worker.
- **In Mesos:** logs are stored in the *work/* directory of a Mesos slave, and accessible from the Mesos master UI.
- **In YARN mode:** the easiest way to collect logs is to use YARN's log collection tool (running `yarn logs -applicationId <app ID>`) to produce a report containing logs from your application.

# Loading and Saving Data in Spark





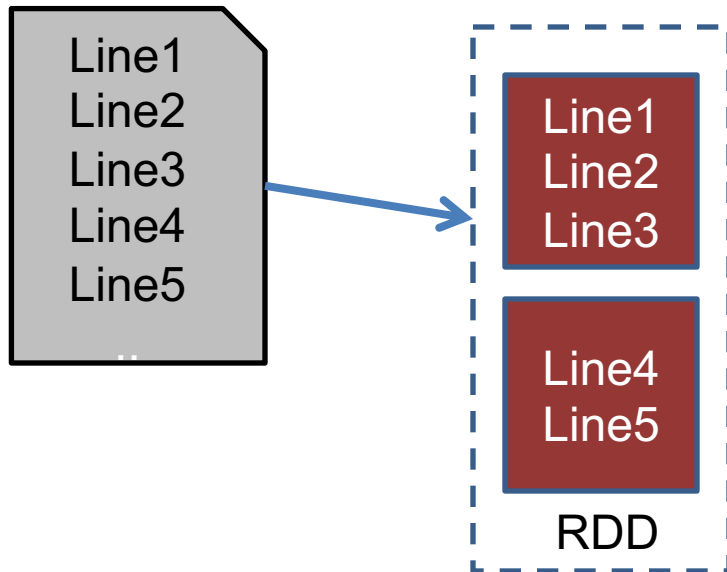
# File Formats

Format	Structured	Comments
Text	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol Buffer	Yes	A fast, space-efficient multilanguage format.
Object	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

# Loading Text Files

Loading a single text file

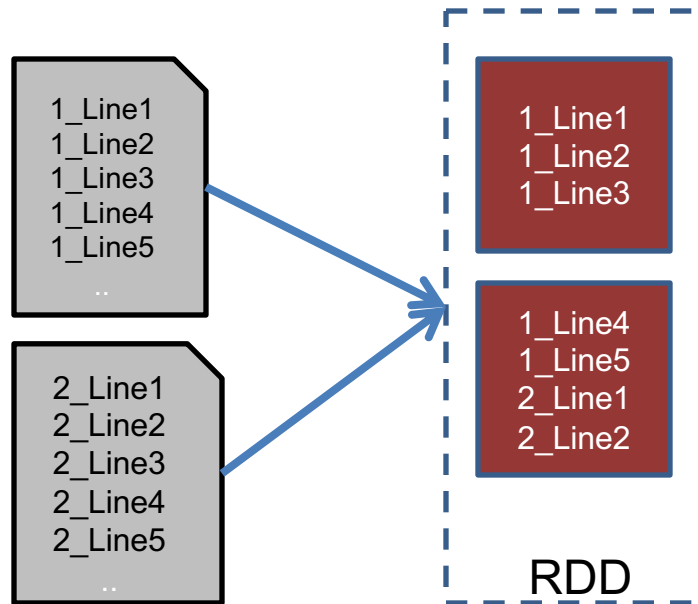
```
input = sc.textFile("file:///fullpath of the file")
```



# Loading Text Files

Loading a multiple text file

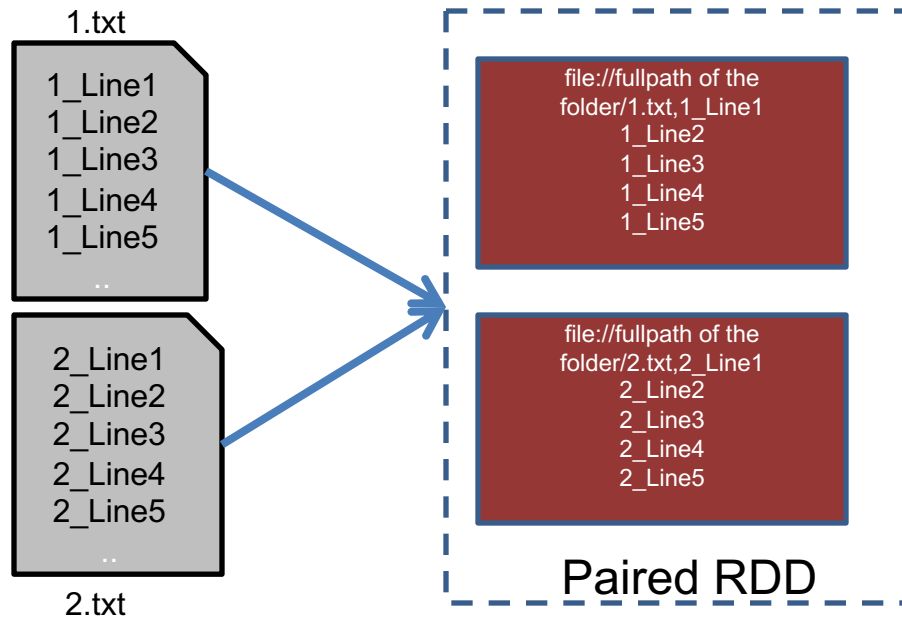
```
input = sc.textFile("file:///fullpath of the folder")
```



# Loading Text Files

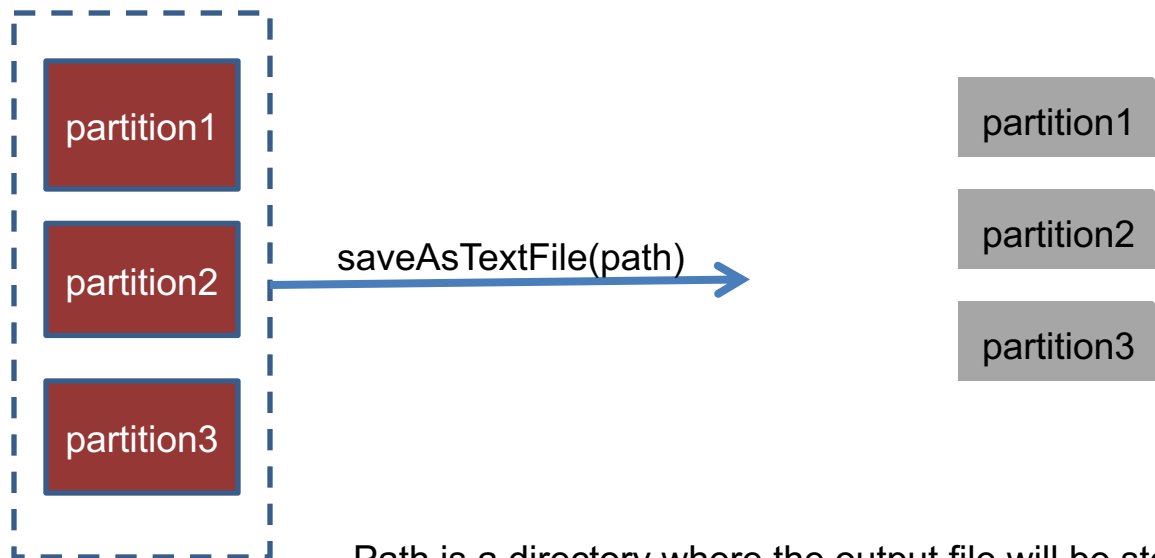
Loading a multiple text file

```
input = sc.wholeTextFiles("file:///fullpath of the folder")
```



Spark supports reading all the files in a given directory and doing wildcard expansion on the input (e.g., part-\*.txt).

# Saving Text File



Path is a directory where the output file will be stored. 1 file per partition.

# Loading JSON files

## line delimited JSON

```
{"string":"string1","int":1,"array":[1,2,3],"dict":{"key": "value1"}}  
{"string":"string2","int":2,"array":[2,4,6],"dict":{"key": "value2"}}  
{"string":"string3","int":3,"array":[3,6,9],"dict":{"key": "value3", "extra_key": "extra_value3"}}
```

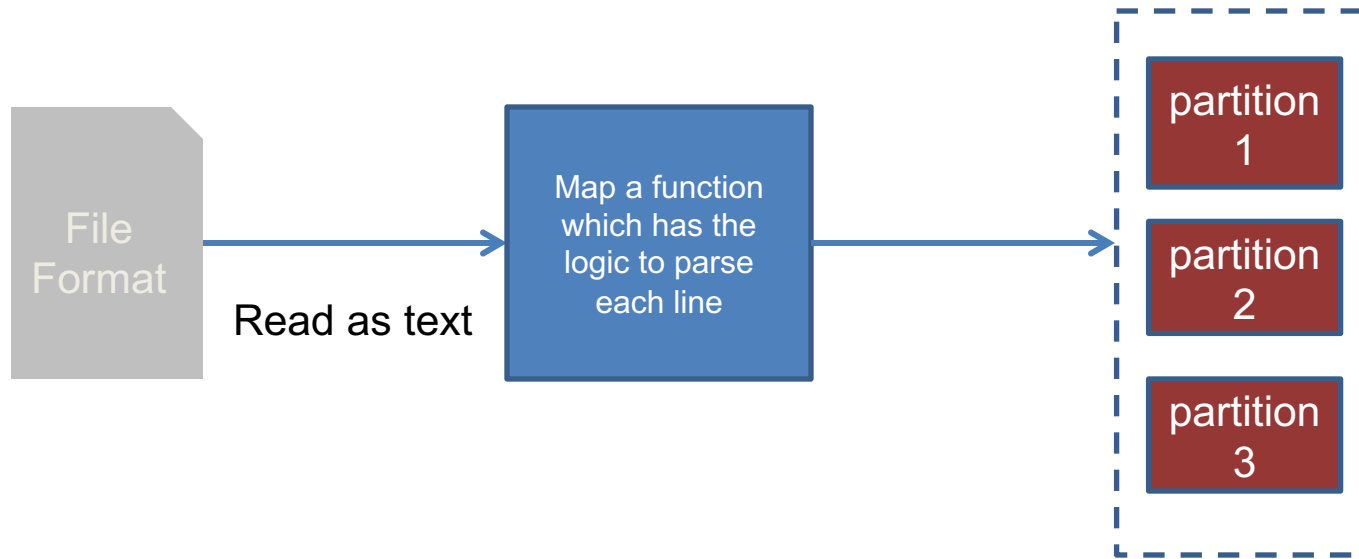
```
import json
```

```
input = sc.textFile("file://fullpath of the file")
```

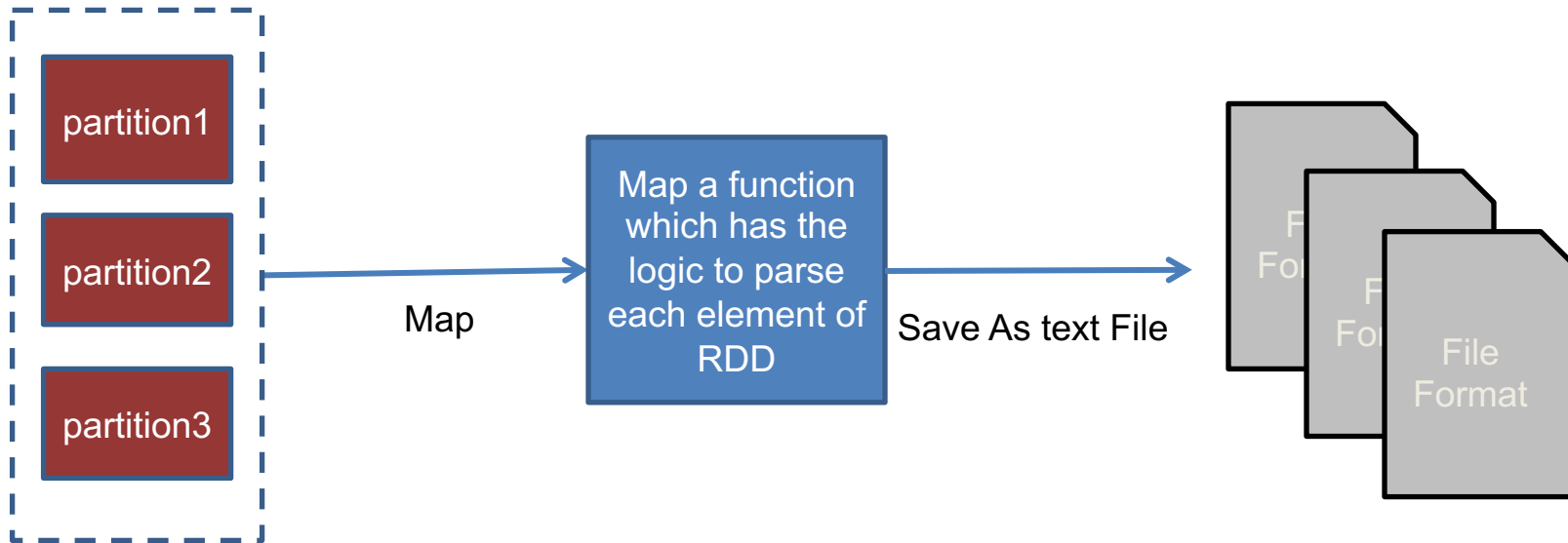
```
data = input.map(lambda x: json.loads(x))
```

**Easy way is to use Spark SQL**

# General Principle Reading



# General Principle Writing



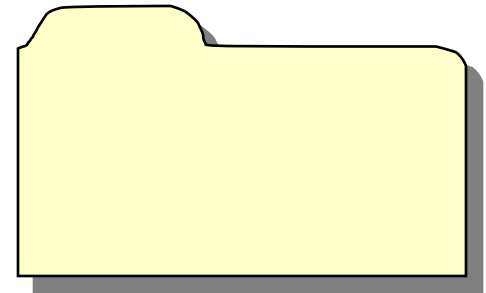


# Writable classes

Scala	Java	Hadoop Writable
Int	Integer	IntWritable
Long	Long	LongWritable
Float	Float	FloatWritable
Double	Double	DoubleWritable
Boolean	Boolean	BooleanWritable
Array[Byte]	byte[]	BytesWritable
String	String	Text
Array[T]	T[]	ArrayWritable<TW>3
List[T]	List<T>	ArrayWritable<TW>3
Map[A,B]	Map<A,B>	MapWritable<AW,BW>

The Python Spark API knows only how to convert the basic Writables available in Hadoop to Python, and makes a best effort for other classes based on their available getter methods.

# File Compression



Spark can read from compressed data source but not all type could be distributed.

# Compression Options

Format	Splittable	Average compression speed	Effectiveness on text	Hadoop compression codec	Comments
gzip	N	Fast	High	org.apache.hadoop.io.compress.GzipCodec	
lzo	Y	Very Fast	Medium	com.hadoop.compression.lzo.LzoCodec	LZO requires installation on every worker node
bzip2	Y	Slow	Very High	org.apache.hadoop.io.compress.BZip2Codec	Uses pure Java for splittable version
zlib	N	Slow	Medium	org.apache.hadoop.io.compress.DefaultCodec	Default compression codec for Hadoop.

# File System

- Spark supports many file systems will go through some of them.



Local/Regular FS



# Local/Regular FS

```
rdd = sc.textFile("file://path to the file")
```

The **Path** could be a shared NFS system or local system.

# HDFS

```
sc.textFile("hdfs://master:port/path")
```

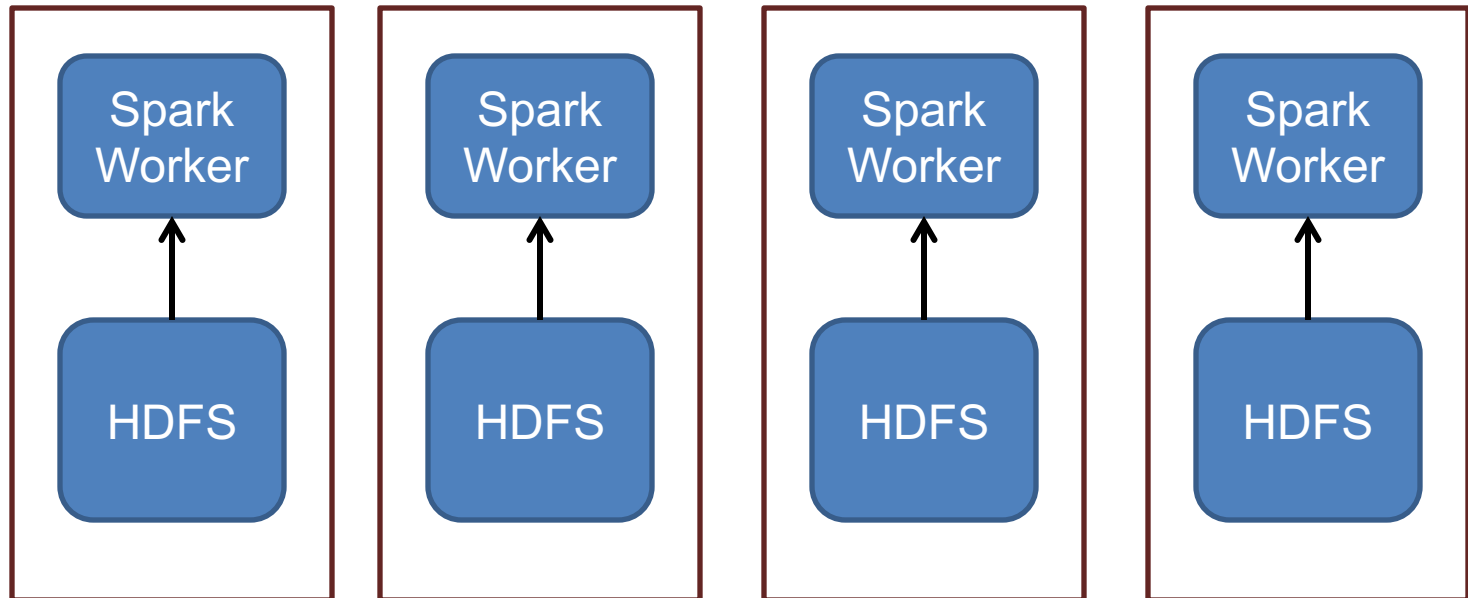


- Spark and HDFS can be collocated on the same machines, and Spark can take advantage of this data locality to avoid network overhead



The HDFS protocol changes across Hadoop versions, so if you run a version of Spark that is compiled for a different version it will fail. If you build from source, you can specify `SPARK_HADOOP_VERSION=` as an environment variable to build against a different version; or you can download a different precompiled version of Spark.

# Data locality



Every worker will load data from local hdfs partition in parallel.

# Databases

- Spark can access several popular databases using either Hadoop connectors or custom Spark Connectors.

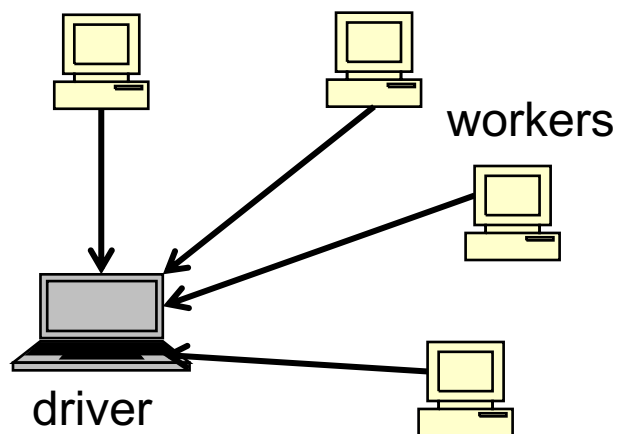


You can write your own connector for a database!

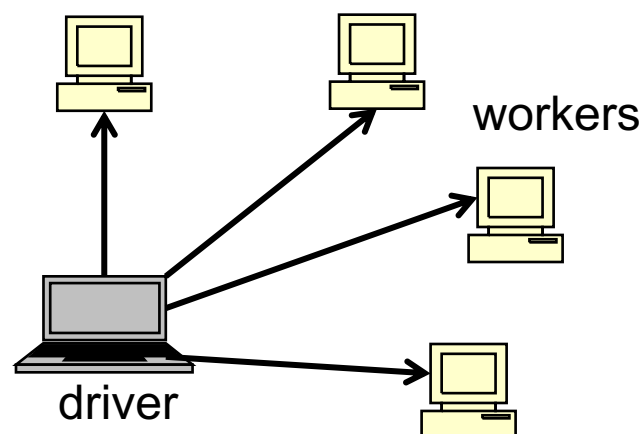


# Spark Programming Advanced

## Accumulators



## Broadcast Variables

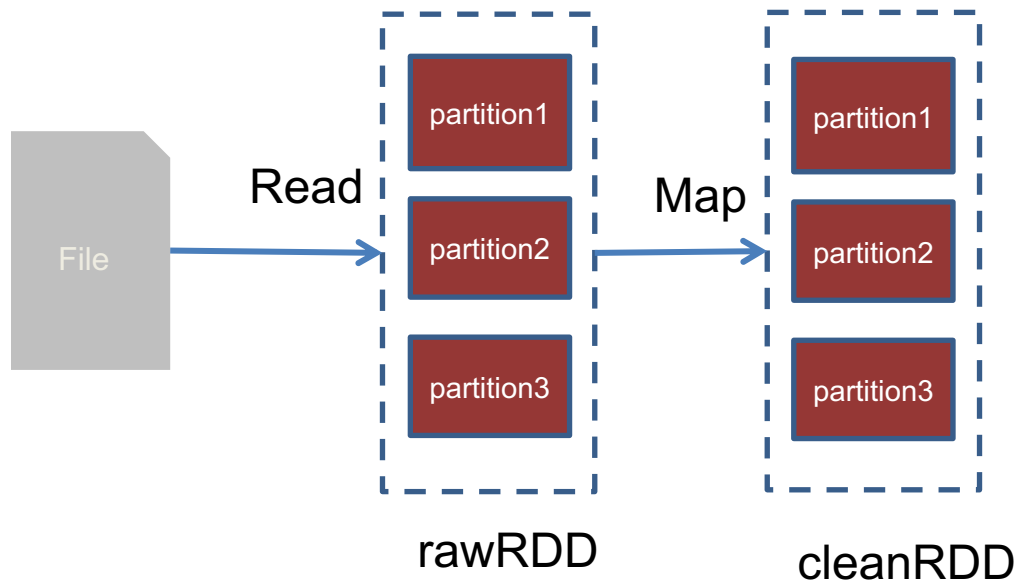


# Accumulators

Provides a simple syntax for aggregating values from worker nodes back to the driver program

**Example:** Need to keep count of badly formatted lines.

# Accumulator Example



## Code:

```
rawRDD= sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
corruptLines = sc.accumulator(0)
def extractCallSigns(line):
    # Make the global variable accessible
    global corruptLines
    if (isFormattedWell(line)):
        corruptLines += 1
    return line.split(" ")
cleanRDD = rawRDD.flatMap(extractCallSigns)
cleanRDD.saveAsTextFile(outputDir + "/Data")
print "Corrupt lines: %d" % corruptLines.value
```

# Accumulators access

Driver

Create the accumulator **A**

Update **A**

Read **A**

Workers

Update **A**

✗ Read **A**

Update **A**

✗ Read **A**



# Accumulators and Fault Tolerance

Action

Accumulator  
Updates only  
Once

Transformation

Accumulator  
may do multiple  
run on re  
execution.

# Custom Accumulators

- Spark supports Int, Double, Long and Float accumulators.
- Extend AccumulatorParam to define your custom accumulator.
- Register your custom accumulator with Spark context.
- You can use any operation inside accumulator, provided that operation is ***commutative*** and ***associative***.

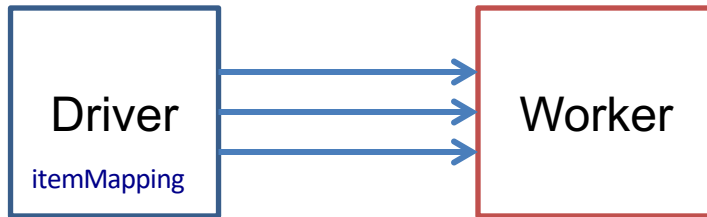


# Broadcast Variable

```
itemMapping= loadItemTable()
def processPurchase(sale):
    country = lookupCountry(sale.StateCode, itemMapping)
    count = sale.Count
    return (country, count)
countrySaleCounts = (saleRDD
    .map(processSignCount)
    .reduceByKey((lambda x, y: x+ y)))
```

# Broadcast Variable

Without BroadCast

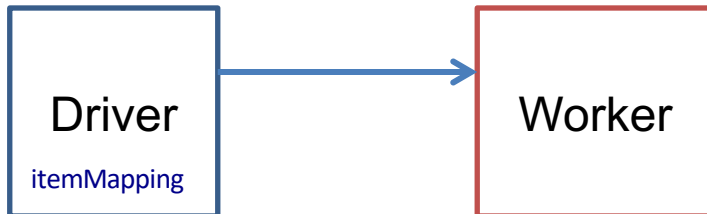


```
saleRDD.map(processSignCount)
```

```
saleRDD.map(processSignCount) -> with  
new data
```

```
returnRDD.map(processSignCount) ->  
with other data
```

With BroadCast





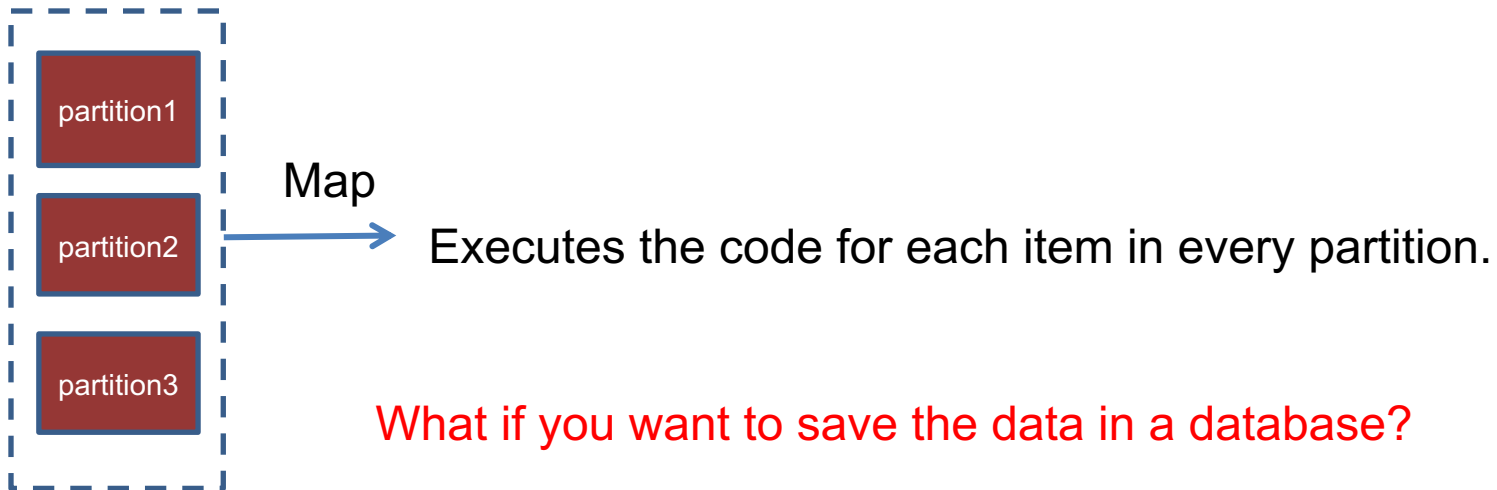


# Broadcast Variable

```
itemMapping= sc.broadcast(loadItemTable())  
def processPurchase(sale):  
    country = lookupCountry(sale.StateCode, itemMapping.value)  
    count = sale.Count  
    return (country, count)  
countrySaleCounts = (saleRDD  
    .map(processSignCount)  
    .reduceByKey(lambda x, y: x+ y)))
```



# Working on a Per-Partition Basis





# Working on a Per-Partition Basis



mapPartitions

Executes the code for each partitions only once.

**Use it for setup work like opening database connection**



# Working on a Per-Partition Basis

Function name	Called with	Signature
mapPartitions()	Iterator of the elements in that partition	$f: (\text{Iterator}[T]) \rightarrow \text{Iterator}[U]$
mapPartitionsWithIndex()	Integer of partition number, and Iterator of the elements in that partition	$f: (\text{Int}, \text{Iterator}[T]) \rightarrow \text{Iterator}[U]$
foreachPartition()	Iterator of the elements	$f: (\text{Iterator}[T]) \rightarrow \text{Unit}$