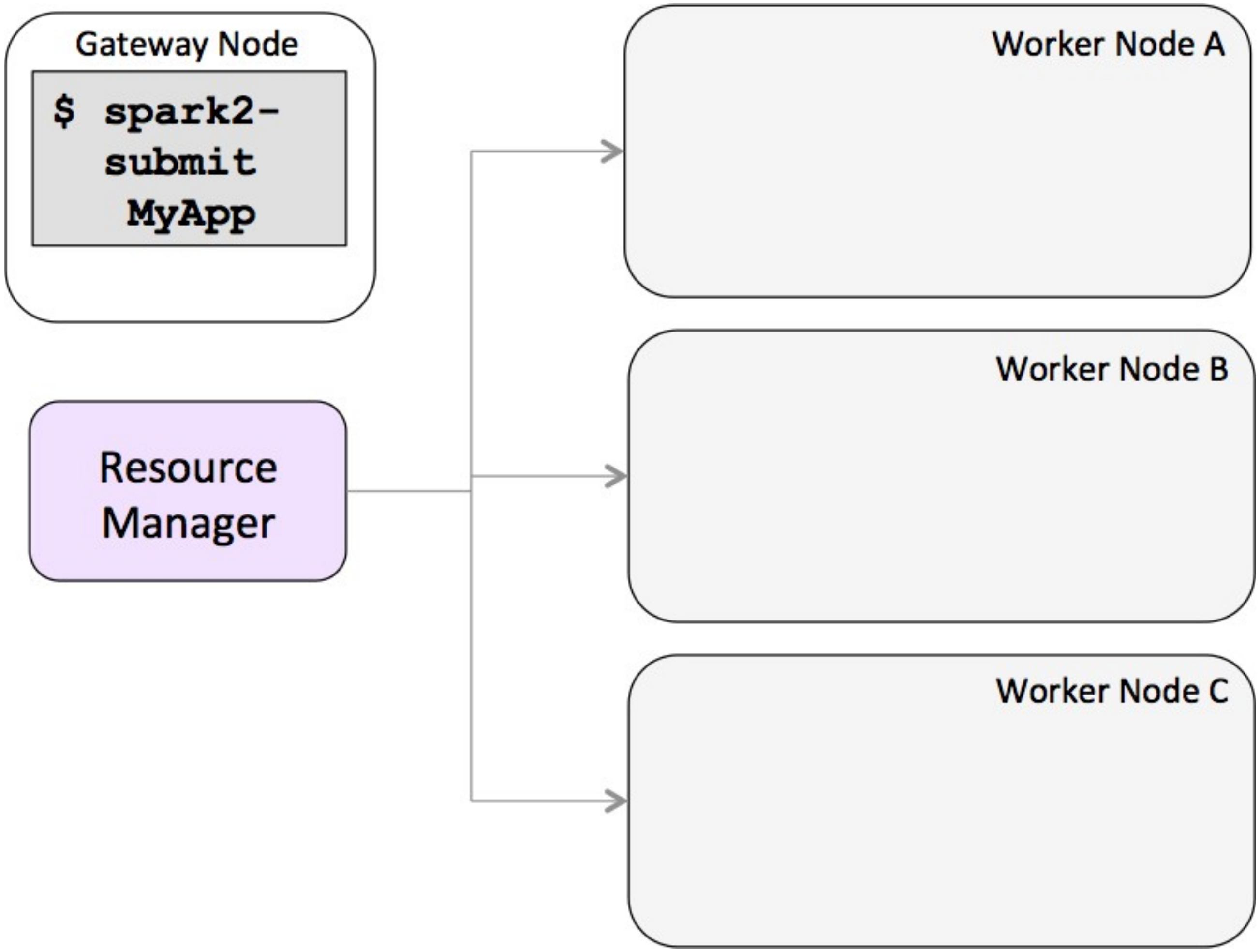
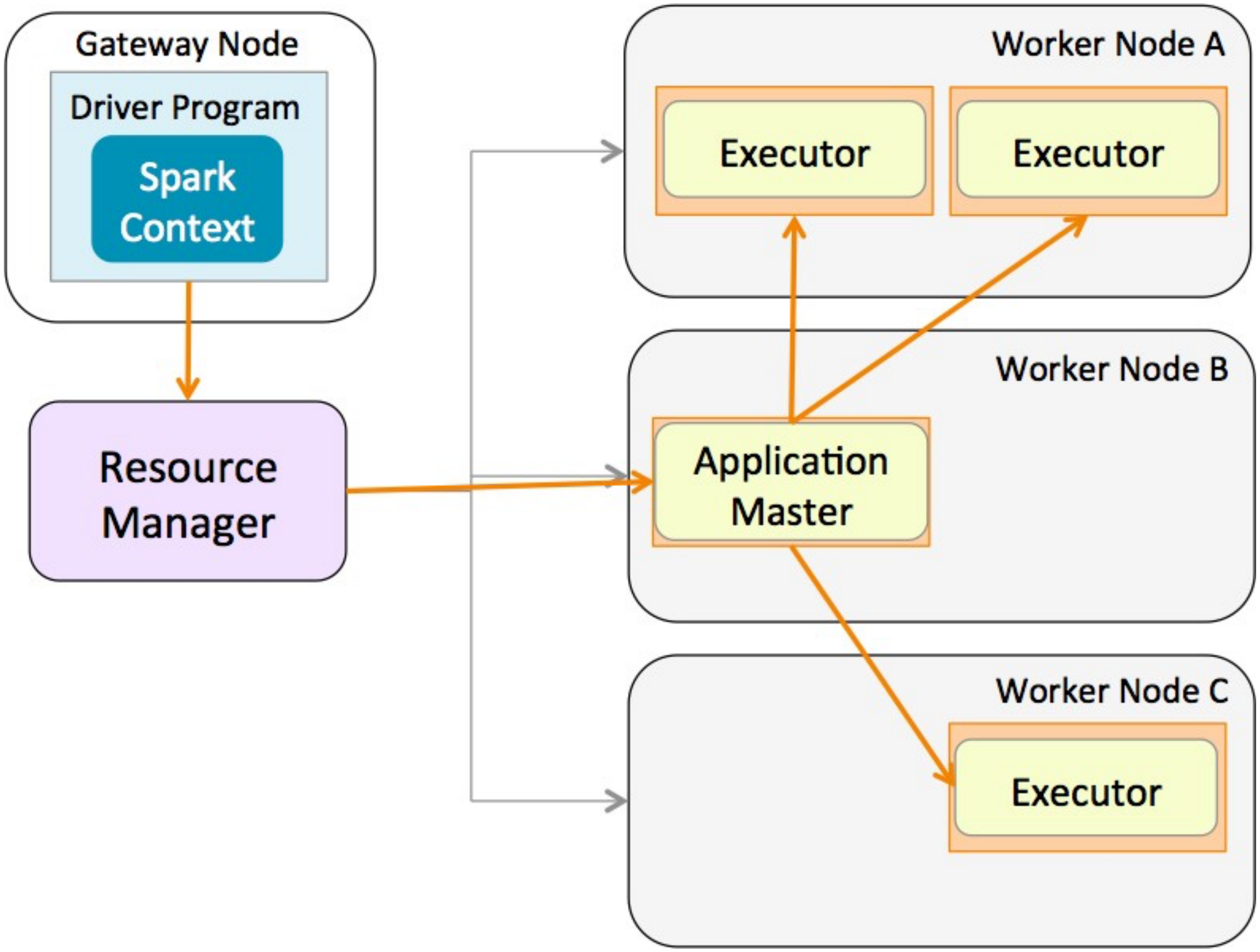
# Distributed Processing

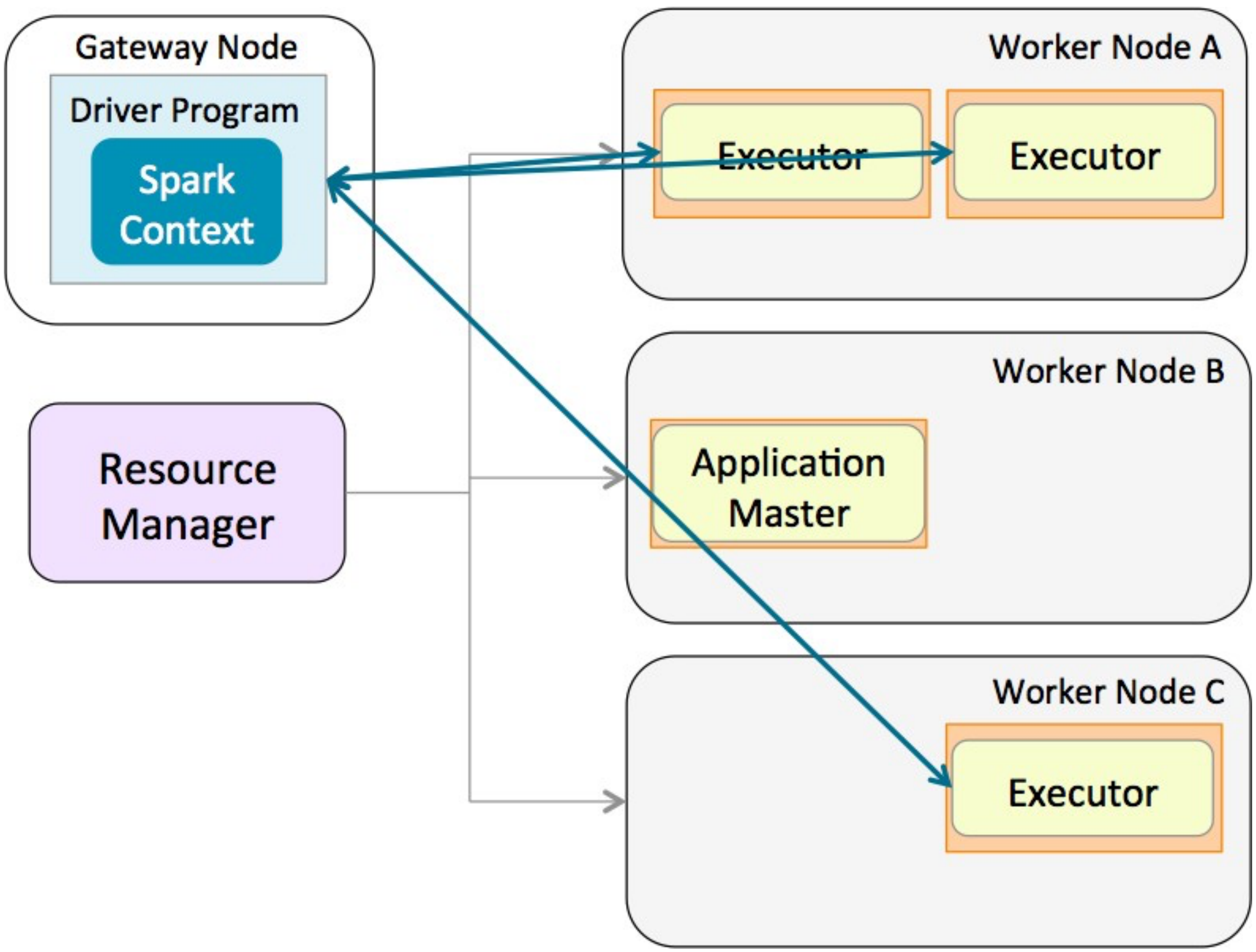
# Review of Spark on YARN (1)



# Review of Spark on YARN (2)

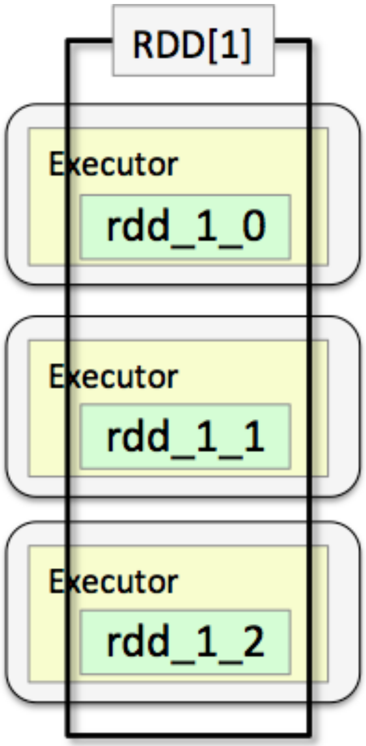


# Review of Spark on YARN (3)



# Data Partitioning (1)

- **Data in Datasets and DataFrames is managed by underlying RDDs**
- **Data in an RDD is *partitioned* across executors**
  - This is what makes RDDs *distributed*
  - Spark assigns tasks to process a partition to the executor managing that partition
- **Data Partitioning is done automatically by Spark**
  - In some cases, you can control how many partitions are created
  - More partitions = more parallelism



# Data Partitioning (2)

- **Spark determines how to partition data in an RDD, Dataset, or DataFrame when**
  - The data source is read
  - An operation is performed on a DataFrame, Dataset, or RDD
  - Spark optimizes a query
  - You call `repartition` or `coalesce`

# Partitioning from Data in Files

- **Partitions are determined when files are read**
  - Core Spark determines RDD partitioning based on location, number, and size of files
    - Usually each file is loaded into a single partition
    - Very large files are split across multiple partitions
  - Catalyst optimizer manages partitioning of RDDs that implement DataFrames and Datasets

## Finding the Number of Partitions in an RDD

- **You can view the number of partitions in an RDD by calling the function `getNumPartitions`**

```
myRDD.getNumPartitions
```

**Language**: *Scala*

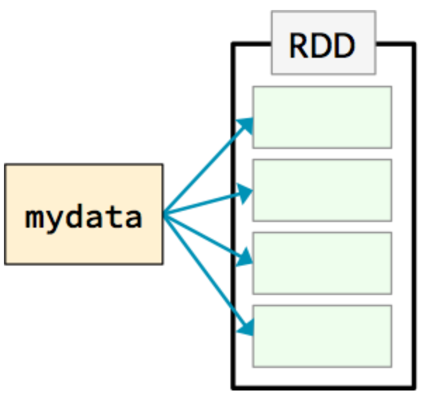
```
myRDD.getNumPartitions()
```

**Language**: *Python*

# Example: Average Word Length by Letter (1)

```
avglens = sc.textFile(mydata)
```

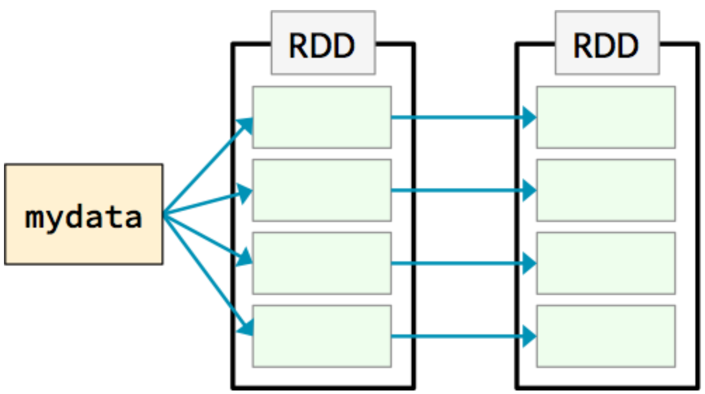
**Language**: *Python*



# Example: Average Word Length by Letter (2)

```
avglens = sc.textFile(mydata) \
  .flatMap(lambda line: line.split(' '))
```

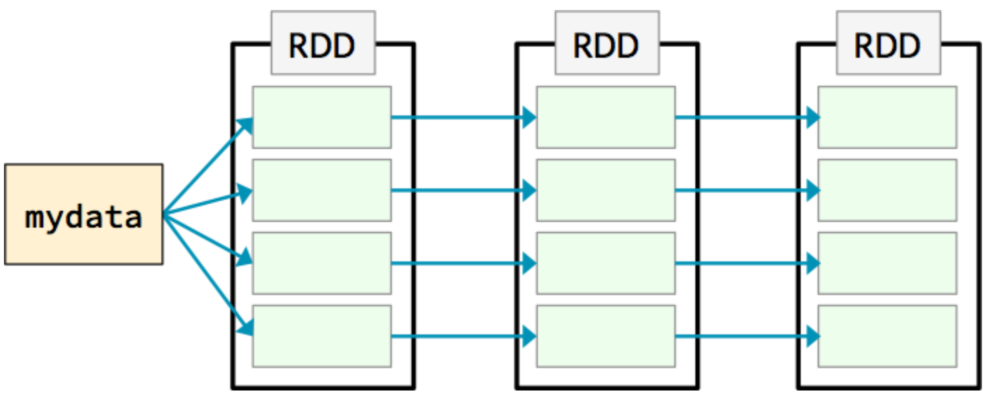
**Language**: *Python*



# Example: Average Word Length by Letter (3)

```
avglens = sc.textFile(mydata) \
  .flatMap(lambda line: line.split(' ')) \
  .map(lambda word: (word[0],len(word)))
```

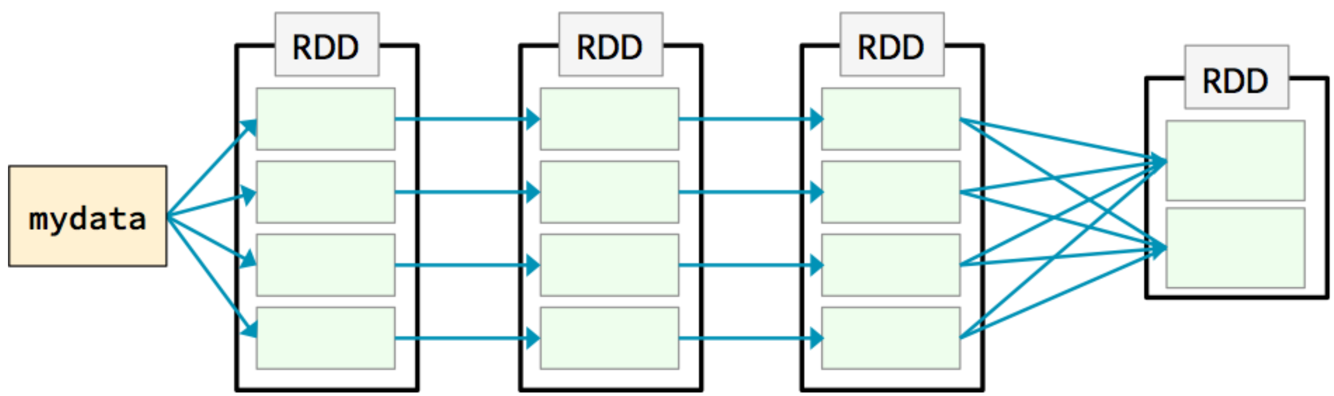
**Language**: *Python*



# Example: Average Word Length by Letter (4)

```
avglens = sc.textFile(mydata) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey()
```

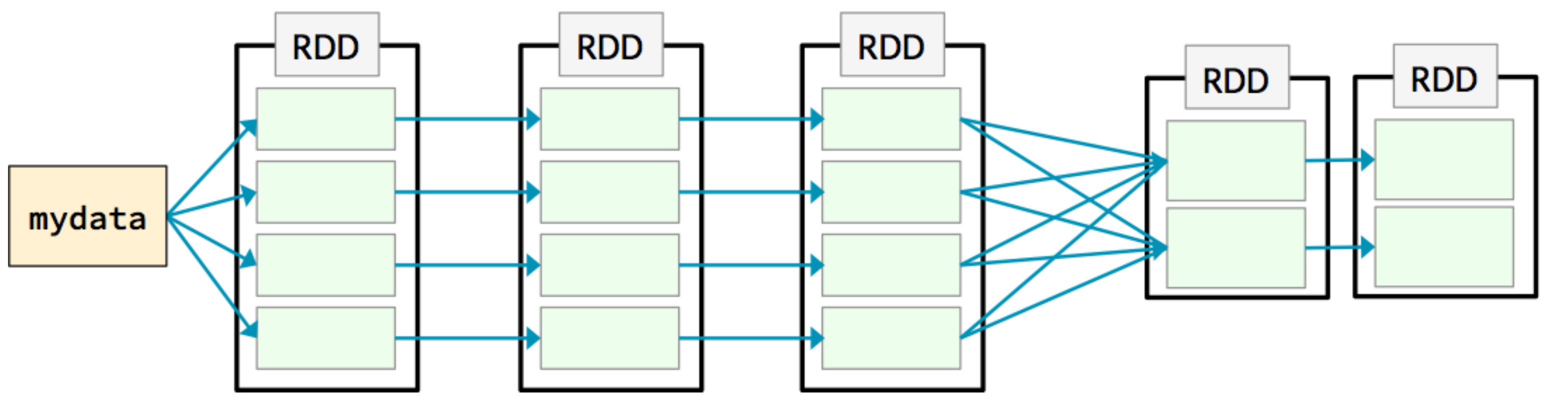
**Language**: *Python*



# Example: Average Word Length by Letter (5)

```
avglens = sc.textFile(mydata) \
  .flatMap(lambda line: line.split(' ')) \
  .map(lambda word: (word[0],len(word))) \
  .groupByKey() \
  .map(lambda (k, values): \
     (k, sum(values)/len(values)))
```

**Language**: *Python*



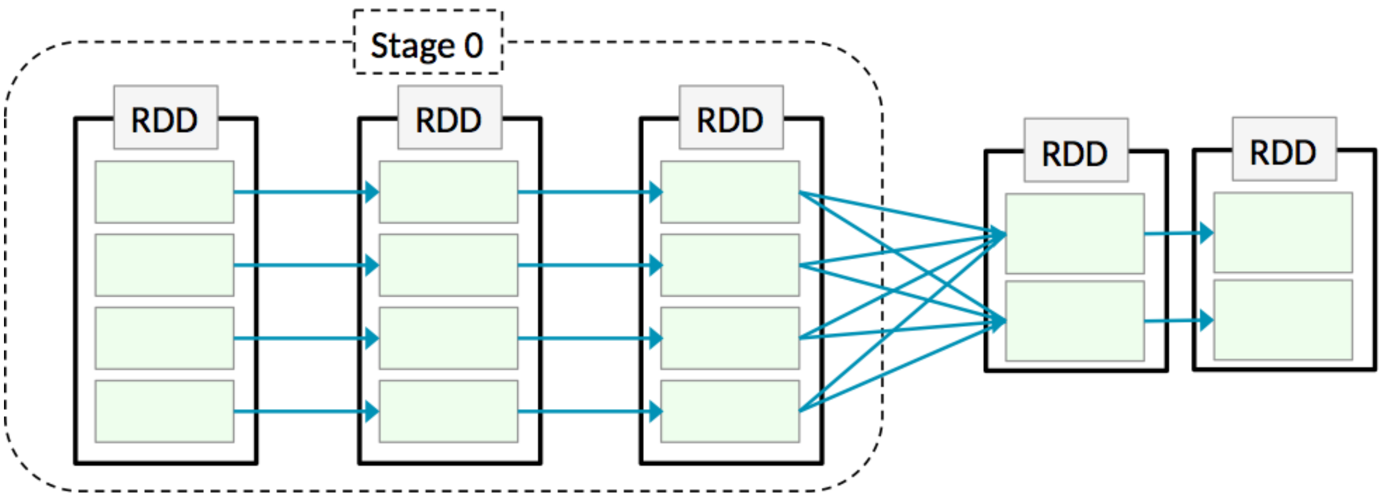
## Stages and Tasks

- **A *task* is a series of operations that work on the same partition and are pipelined together**
- ***Stages* group together tasks that can run in parallel on different partitions of the same RDD**
- ***Jobs* consist of all the stages that make up a query**
- **Catalyst optimizes partitions and stages when using DataFrames and Datasets**
  - Core Spark provides limited optimizations when you work directly with RDDs
    - You need to code most RDD optimizations manually
  - To improve performance, be aware of how tasks and stages are executed when working with RDDs

# Example: Query Stages and Tasks (1)

```
avglens = sc.textFile(mydata) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
      (k, sum(values)/len(values)))

avglens.saveAsTextFile("avglen-output")
```

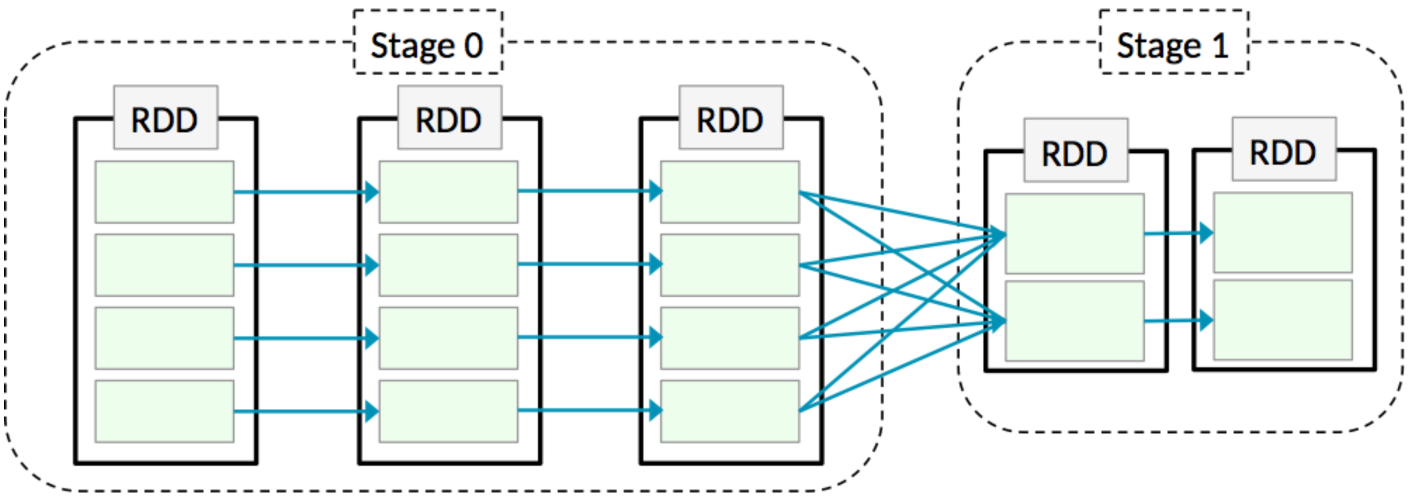
**Language**: *Python*



# Example: Query Stages and Tasks (2)

```
avglens = sc.textFile(mydata) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
      (k, sum(values)/len(values)))

avglens.saveAsTextFile("avglen-output")
```

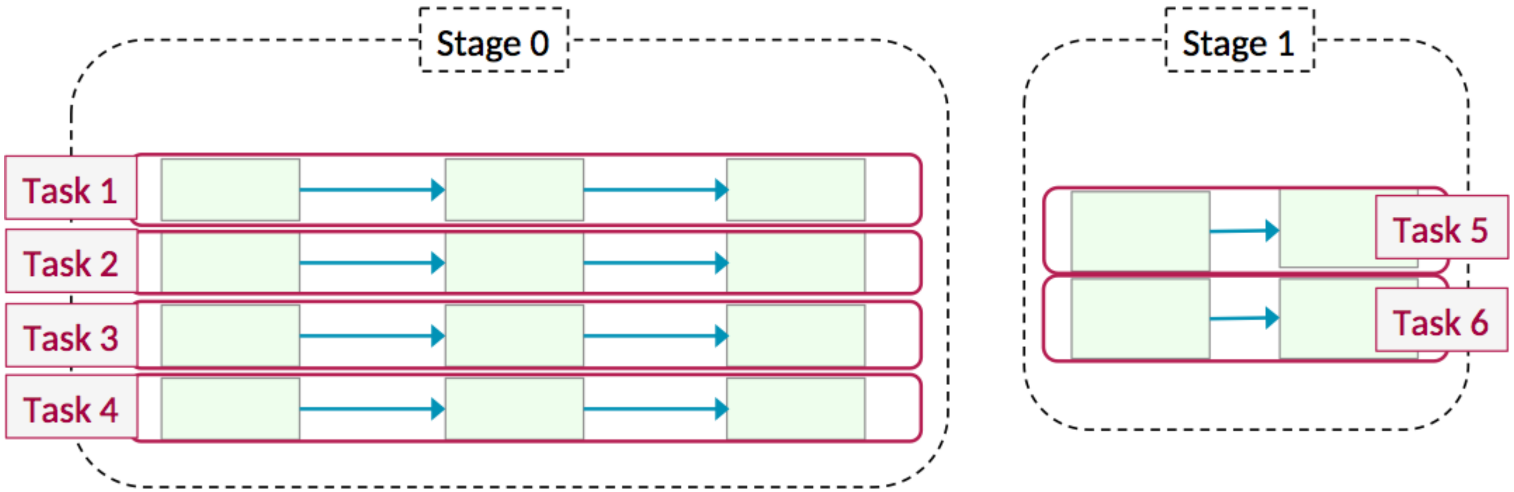
**Language**: *Python*



# Example: Query Stages and Tasks (3)

```
avglens = sc.textFile(mydata) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
      (k, sum(values)/len(values)))

avglens.saveAsTextFile("avglen-output")
```

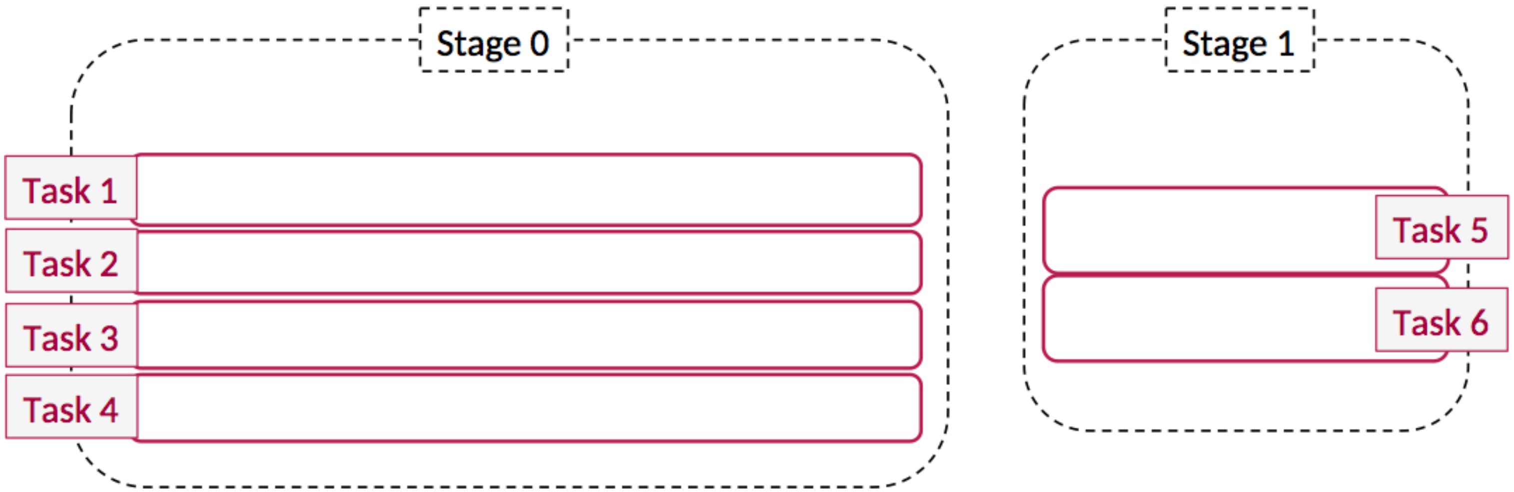
**Language**: *Python*



# Example: Query Stages and Tasks (4)

```
avglens = sc.textFile(mydata) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
      (k, sum(values)/len(values)))

avglens.saveAsTextFile("avglen-output")
```

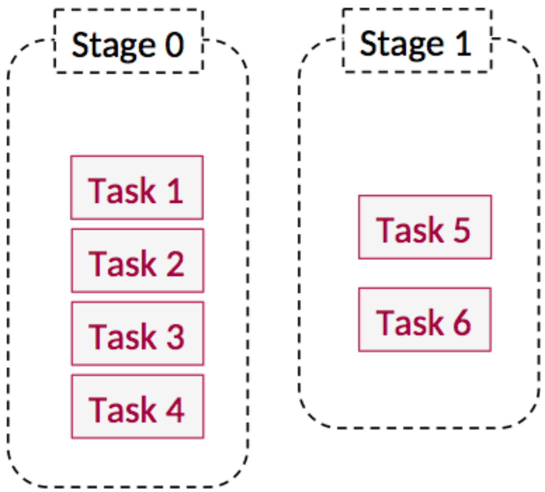
**Language**: *Python*



# Example: Query Stages and Tasks (5)

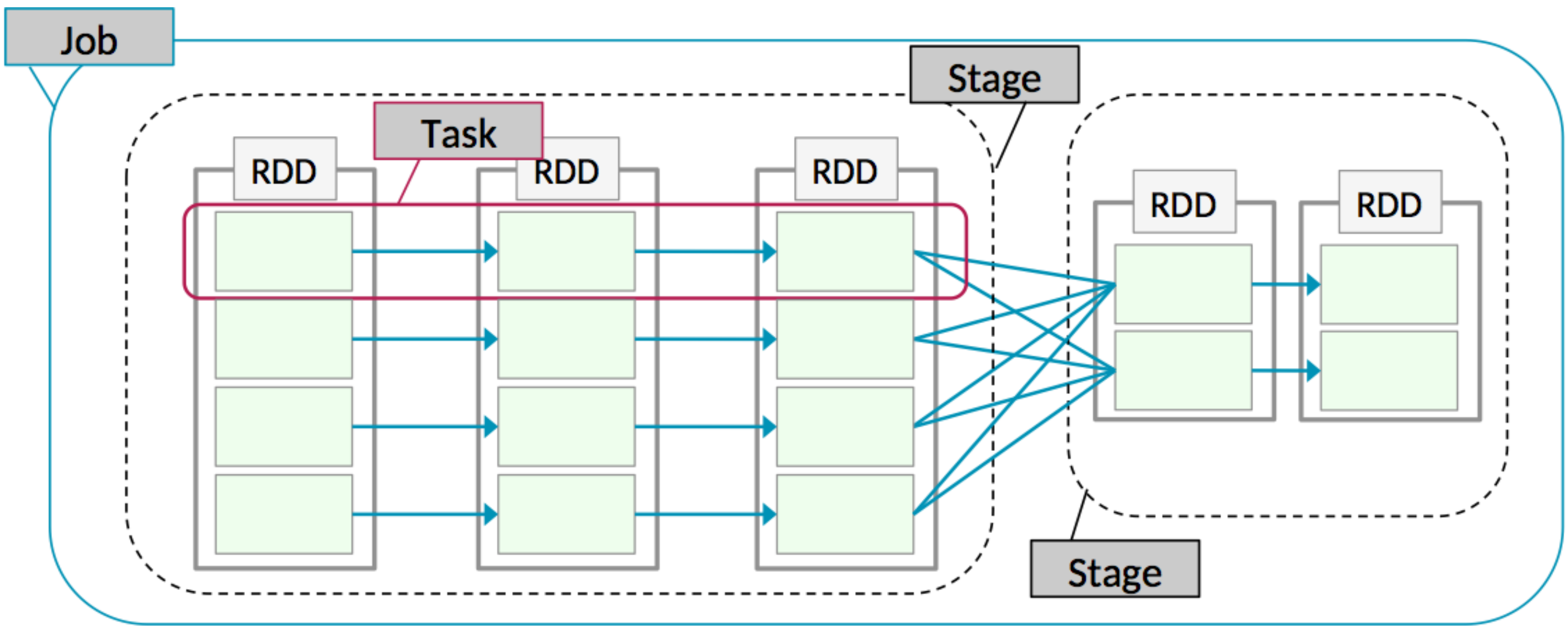
```
avglens = sc.textFile(mydata) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
      (k, sum(values)/len(values)))

avglens.saveAsTextFile("avglen-output")
```

**Language**: *Python*



## Summary of Spark Terminology

- **Job—a set of tasks executed as a result of an *action***
- **Stage—a set of tasks in a job that can be executed in parallel**
- **Task—an individual unit of work sent to one executor**
- **Application—the set of jobs managed by a single driver**

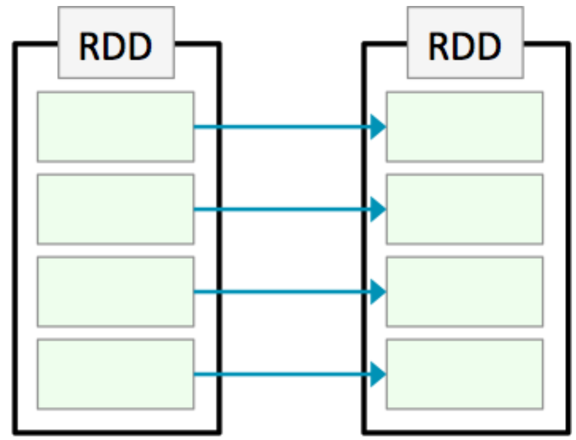


## Execution Plans

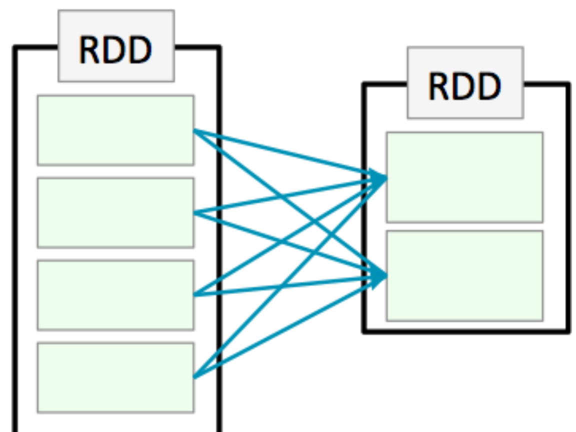
- **Spark creates an execution plan for each job in an application**
- **Catalyst creates SQL, Dataset, and DataFrame execution plans**
  - Highly optimized
- **Core Spark creates execution plans for RDDs**
  - Based on RDD lineage
  - Limited optimization

# How Execution Plans are Created

- **Spark constructs a DAG (Directed Acyclic Graph) based on RDD dependencies**
- ***Narrow* dependencies**
  - Each partition in the child RDD depends on just one partition of the parent RDD
  - No shuffle required between executors
  - Can be pipelined into a single stage
  - Examples: `map`, `filter`, and `union`




- ***Wide* (or *shuffle*) dependencies**
  - Child partitions depend on multiple partitions in the parent RDD
  - Defines a new stage
  - Examples: `reduceByKey`, `join`, and `groupByKey`



# Controlling the Number of Partitions in RDDs (1)

- **Partitioning determines how queries execute on a cluster**
  - More partitions = more parallel tasks
  - Cluster will be under-utilized if there are too few partitions
    - But too many partitions will increase overhead without an offsetting increase in performance
- **Catalyst controls partitioning for SQL, DataFrame, and Dataset queries**
- **You can control how many partitions are created for RDD queries**

# Controlling the Number of Partitions in RDDs (2)

- **Specify the number of partitions when data is read**
  - Default partitioning is based on size and number of the files (minimum is two)
  - Specify a different minimum number when reading a file

```
myRDD = sc.textFile(myfile,5)
```

- **Manually repartition**
  - Create a new RDD with a specified number of partitions using `repartition` or `coalesce`
    - `coalesce` reduces the number of partitions without requiring a shuffle
    - `repartition` shuffles the data into more or fewer partitions

```
newRDD = myRDD.repartition(15)
```

# Controlling the Number of Partitions in RDDs (3)

- **Specify the number of partitions created by transformations**
  - Wide (shuffle) operations such as `reduceByKey` and `join` repartition data
  - By default, the number of partitions created is based on the number of partitions of the parent RDD(s)
  - Choose a different default by configuring the `spark.default.parallelism` property

```
spark.default.parallelism  15
```

  - Override the default with the optional `numPartitions` operation parameter

```
countRDD = wordsRDD. \
  reduceByKey(lambda v1, v2: v1 + v2, 15)
```

## Catalyst Optimizer

- **Catalyst can improve SQL, DataFrame, and Dataset query performance by optimizing the DAG to**
  - Minimize data transfer between executors
    - Such as *broadcast* joins—small data sets are pushed to the executors where the larger data sets reside
  - Minimize wide (shuffle) operations
    - Such as unioning two RDDs—grouping, sorting, and joining do not require shuffling
  - Pipeline as many operations into a single stage as possible
  - Generate code for a whole stage at run time
  - Break a query job into multiple jobs, executed in a series

## Catalyst Execution Plans

- **Execution plans for DataFrame, Dataset, and SQL queries include the following phases**
  - **Parsed logical plan**—calculated directly from the sequence of operations specified in the query
  - **Analyzed logical plan**—resolves relationships between data sources and columns
  - **Optimized logical plan**—applies rule-based optimizations
  - **Physical plan**—describes the actual sequence of operations
  - **Code generation**—generates bytecode to run on each node, based on a cost model

# Viewing Catalyst Execution Plans

- **You can view SQL, DataFrame, and Dataset (Catalyst) execution plans**
  - Use DataFrame/Dataset `explain`
    - Shows only the physical execution plan by default
    - Pass `true` to see the full execution plan
  - Use **SQL** tab in the Spark UI or history server
    - Shows details of execution after job runs

## Example: Catalyst Execution Plan (1)

```
peopleDF = spark.read. \
  option("header","true").csv("people.csv")
pcodesDF = spark.read. \
  option("header","true").csv("pcodes.csv")
joinedDF = peopleDF.join(pcodesDF, "pcode")
joinedDF.explain(True)

== Parsed Logical Plan ==
'Join UsingJoin(Inner,ArrayBuffer('pcode))
:- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
+- Relation[pcode#9,city#10,state#11] csv
```

**Language**: *Python*

*continued on next slide...*

# Example: Catalyst Execution Plan (2)

```
== Analyzed Logical Plan ==
pcode: string, lastName: string, firstName: string, age:
 string, city: string, state: string
Project [pcode#0, lastName#1, firstName#2, age#3, city#10,
 state#11]
+- Join Inner, (pcode#0 = pcode#9)
   :- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
   +- Relation[pcode#9,city#10,state#11] csv

== Optimized Logical Plan ==
Project [pcode#0, lastName#1, firstName#2, age#3, city#10,
 state#11]
+- Join Inner, (pcode#0 = pcode#9)
   :- Filter isnotnull(pcode#0)
   :  +- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
   +- Filter isnotnull(pcode#9)
      +- Relation[pcode#9,city#10,state#11] csv
```

**Language**: *Python*
*continued on next slide…*

## Example: Catalyst Execution Plan (3)

```
== Physical Plan ==
*Project [pcode#0, lastName#1, firstName#2, age#3, city#10,
 state#11]
+- *BroadcastHashJoin [pcode#0], [pcode#9], Inner, BuildRight
   :- *Project [pcode#0, lastName#1, firstName#2, age#3]
   :  +- *Filter isnotnull(pcode#0)
   :     +- *Scan csv [pcode#0,lastName#1,firstName#2,age#3]
 Format: CSV, InputPaths: file:/home/training/people.csv,
 PushedFilters: [IsNotNull(pcode)], ReadSchema:
 struct<pcode:string,lastName:string,firstName:string,age:string>
   +- BroadcastExchange
 HashedRelationBroadcastMode(List(input[0, string, true]))
      +- *Project [pcode#9, city#10, state#11]
         +- *Filter isnotnull(pcode#9)
            +- *Scan csv [pcode#9,city#10,state#11]
 Format: CSV, InputPaths: file:/home/training/pcodes.csv,
 PushedFilters: [IsNotNull(pcode)], ReadSchema:
 struct<pcode:string,city:string,state:string></
pcode:string,city:string,state:string>
```

**Language**: *Python*

# Example: Catalyst Execution Plan (4)

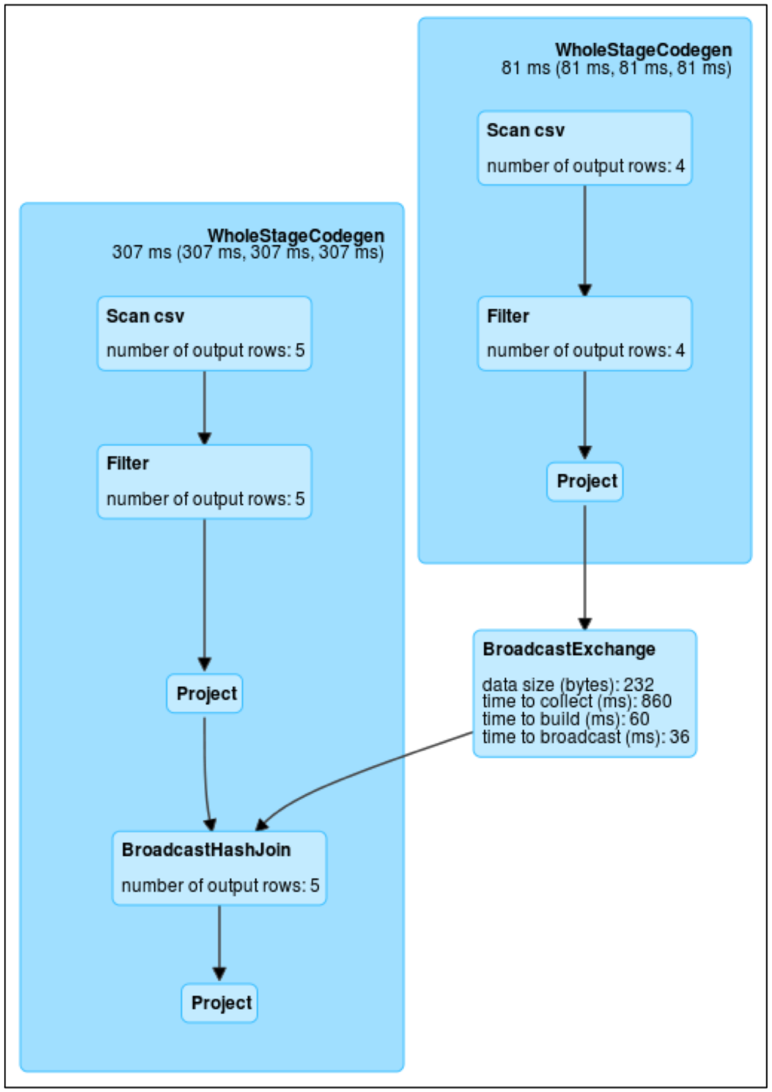
Jobs Stages Storage Environment Executors SQL

## SQL

### Completed Queries

| ID | Description | | Submitted | Duration | Jobs |
|---|---|---|---|---|---|
| 0 | collect at <ipython-input-65-31b1b37d0504>:1 | +details | 2017/05/24 08:45:13 | 1 s | 31<br>32 |

# Example: Catalyst Execution Plan (5)



# Viewing RDD Execution Plans

- **You can view RDD (lineage-based) execution plans**
  - Use the RDD `toDebugString` function
  - Use **Jobs** and **Stages** tabs in the Spark UI or history server
    - Shows details of execution after job runs
- **Note that plans may be different depending on programming language**
  - Plan optimization rules vary

# Example: RDD Execution Plan (1)

```
val peopleRDD = sc.textFile("people2.csv").keyBy(s => s.split(',')(0))
val pcodesRDD = sc.textFile("pcodes2.csv").keyBy(s => s.split(',')(0))
val joinedRDD = peopleRDD.join(pcodesRDD)
joinedRDD.toDebugString
(2) MapPartitionsRDD[8] at join at …  ①
 |  MapPartitionsRDD[7] at join at …
 |  CoGroupedRDD[6] at join at …
 +-(2) MapPartitionsRDD[2] at keyBy at …  ②
 |  |  people2.csv MapPartitionsRDD[1] at textFile at
 …
 |  |  people2.csv HadoopRDD[0] at …
 +-(2) MapPartitionsRDD[5] at keyBy at …  ③
    |  pcodes2.csv MapPartitionsRDD[4] at textFile at
 ④ …
    |  pcodes2.csv HadoopRDD[3] at textFile at …
```

**Language:** *Scala*

1. **Stage 2**
2. **Stage 1**
3. **Stage 0**
4. **Indents indicate stages (shuffle boundaries)**

# Example: RDD Execution Plan (2)

Jobs Stages Storage Environment Executors SQL

## Spark Jobs (?)

**User:** training
**Total Uptime:** 55 s
**Scheduling Mode:** FIFO
**Completed Jobs:** 1

▸ Event Timeline

### Completed Jobs (1)

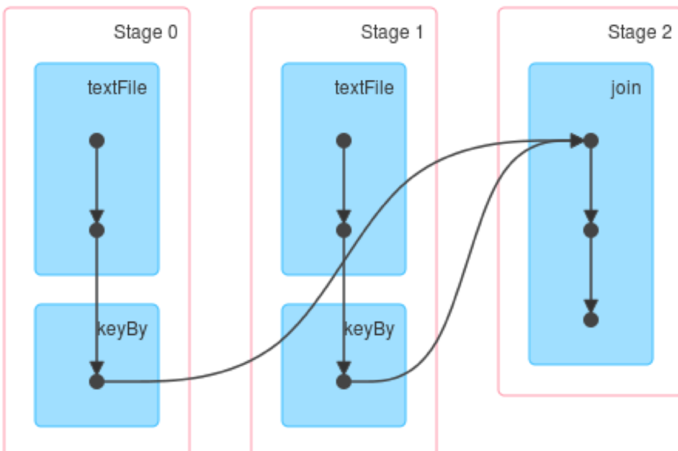
| Job Id ▾ | Description | Submitted | Duration | Stages: Succeeded/Total | Tasks (for all stages): Succeeded/Total |
|---|---|---|---|---|---|
| 0 | collect at <console>:31 | 2017/05/24 11:09:51 | 1 s | 3/3 | 6/6 |

# Example: RDD Execution Plan (3)

## Details for Job 0

**Status:** SUCCEEDED
**Completed Stages:** 3

▸ Event Timeline
▾ DAG Visualization




### Completed Stages (3)

| Stage Id ▾ | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|---|---|
| 2 | collect at <console>:31 +details | 2017/05/24 11:09:52 | 0.2 s | 2/2 | | | 586.0 B | |
| 1 | keyBy at <console>:24 +details | 2017/05/24 11:09:51 | 99 ms | 2/2 | 170.0 B | | | 339.0 B |
| 0 | keyBy at <console>:24 +details | 2017/05/24 11:09:51 | 0.7 s | 2/2 | 105.0 B | | | 247.0 B |

# Essential Points

- **Spark partitions split data across different executors in an application**
- **Executors execute query tasks that process the data in their partitions**
- **Narrow operations like `map` and `filter` are pipelined within a single stage**
  - Wide operations like `groupByKey` and `join` shuffle and repartition data between stages
- **Jobs consist of a sequence of stages triggered by a single action**
- **Jobs execute according to execution plans**
  - Core Spark creates RDD execution plans based on RDD lineages
  - Catalyst builds optimized query execution plans
- **You can explore how Spark executes queries in the Spark Application UI**

# Hands-On Exercise: Jobs Monitoring : Using Web UI.

- **In this exercise, you will explore how Spark plans and executes RDD and DataFrame/Dataset queries**
  - Please refer to the Hands-On Exercise Manual for instructions