Parallel Processing in Spark

Prof. Hyuk-Yoon Kwon

Parallel Programming with Spark
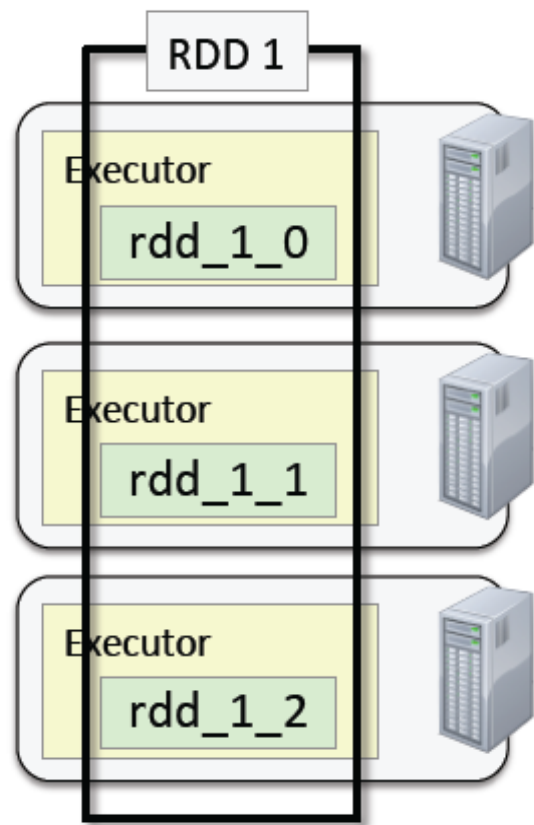
In this chapter you will learn

- How RDDs are distributed across a cluster

- How Spark executes RDD operations in parallel

RDDs on a Cluster

- **Resilient *Distributed* Datasets**
  - Data is *partitioned* across worker nodes

- **Partitioning is done automatically by Spark**
  - Optionally, you can control how many partitions are created

RDD 1

| Executor |
| rdd_1_0 |

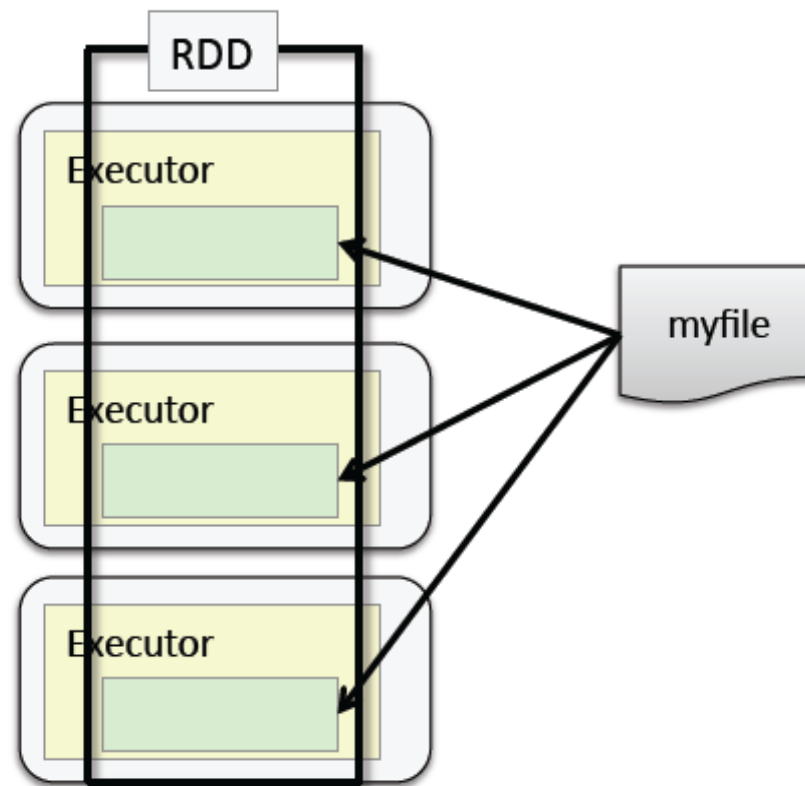| Executor |
| rdd_1_1 |

| Executor |
| rdd_1_2 |

File Partitioning: Single Files
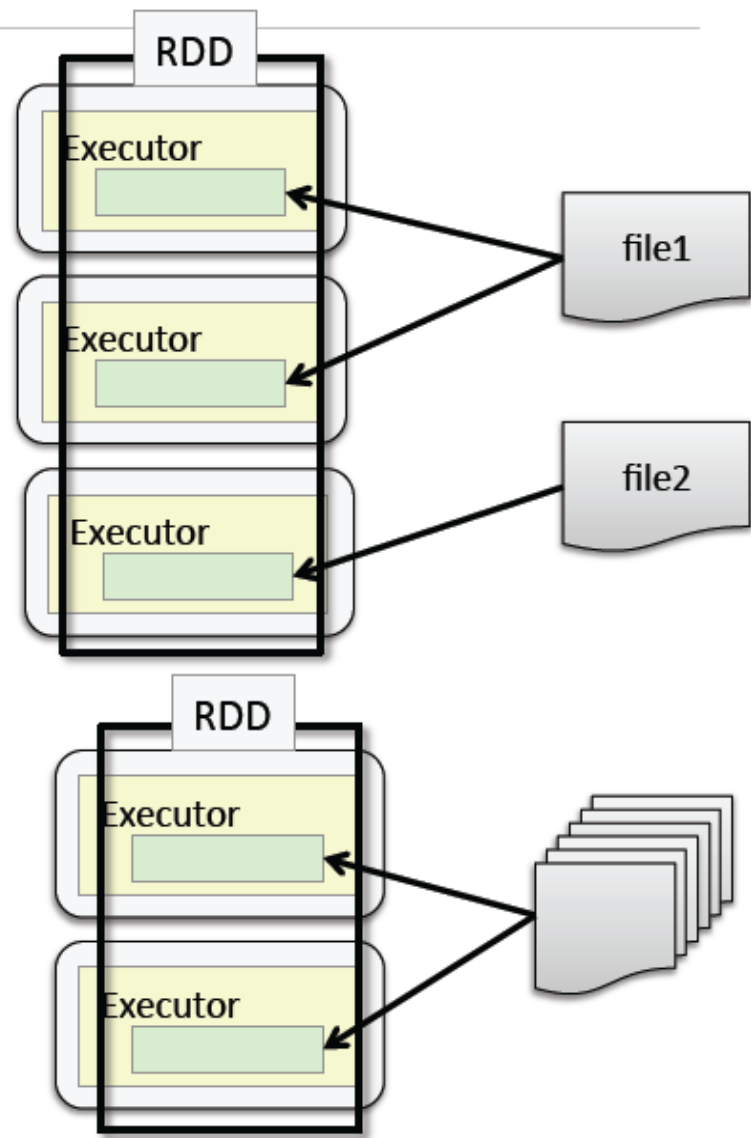
- **Partitions from single files**
  - Partitions based on size
  - You can optionally specify a minimum number of partitions
    `textFile(file, minPartitions)`
  - Default is 2
  - More partitions = more parallelization

`sc.textFile("myfile",3)`

File Partitioning: Multiple Files

- **`sc.textFile("mydir/*")`**
  - Each file becomes (at least) one partition
  - File-based operations can be done per-partition, for example parsing XML

- **`sc.wholeTextFiles("mydir")`**
  - For many small files
  - Creates a key-value PairRDD
    - key = file name
    - value = file contents

Operations on Partitions

- Most RDD operations work on each *element* of an RDD

- A few work on each *partition*
    - **foreachPartition** – call a function for each partition
    - **mapPartitions** – create a new RDD by executing a function on each partition in the current RDD
    - **mapPartitionsWithIndex** – same as **mapPartitions** but includes index of the partition

- Functions for partition operations take iterators
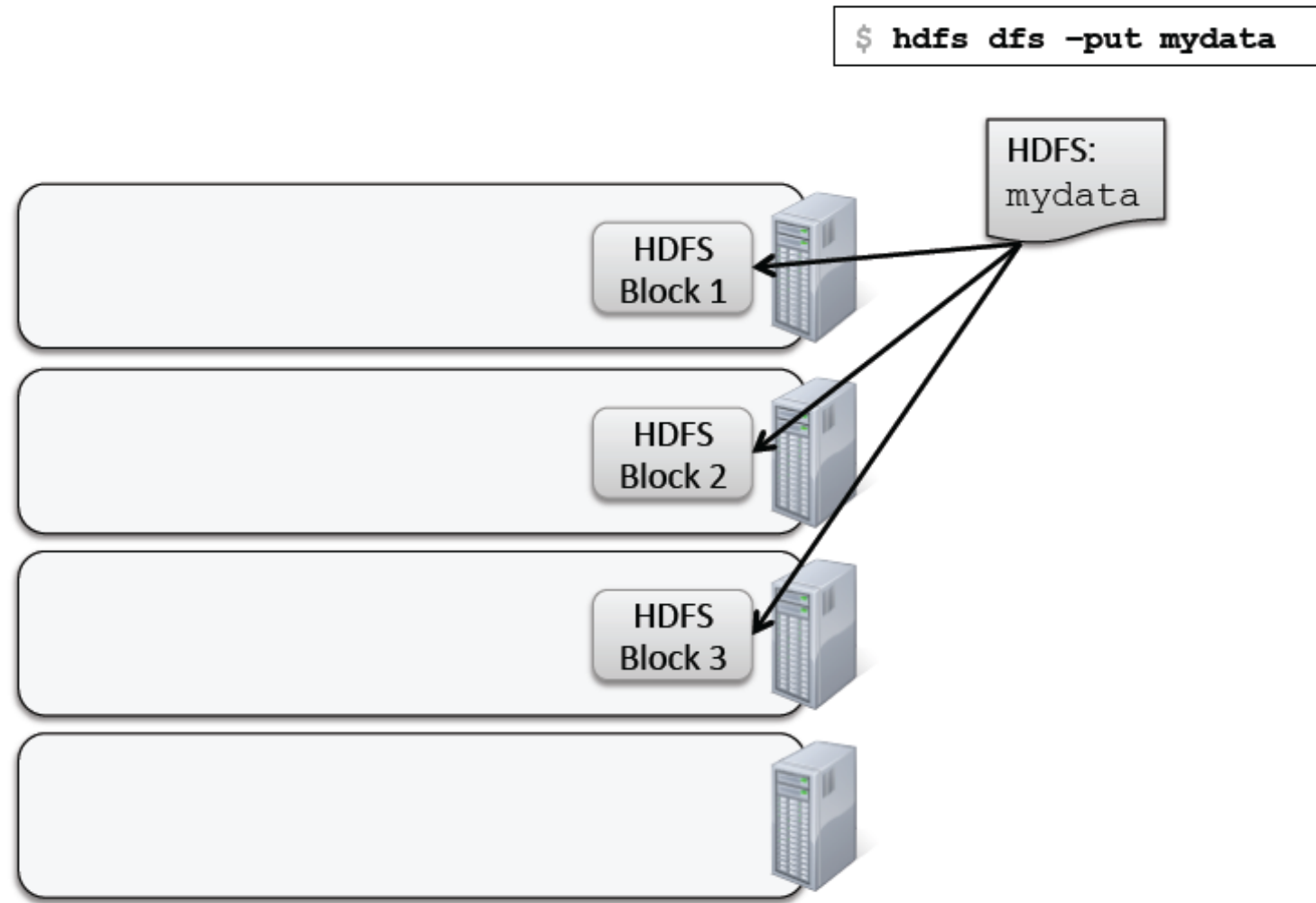
Practice: Explore Partitioning of file-based RDDs

■ **Start the Spark Shell in local mode with 2 threads to simulate a more realistic multi-node cluster.**

■ **Review the accounts dataset (/loudacre/accounts) using Hue or command line. Take note of the number of files**

- If dataset do not exist in HDFS, upload them from the local file system into HDFS

■ **Create an RDD based on a single file in the dataset, e.g., /loudacre/accounts/part-m-00000 and then call toDebugString on the RDD, which displays the number of partitions in parentheses() before the RDD id. How many partitions are in the resulting RDD?**

■ **Repeat this process, but specify a minimum of three partitions: sc.textFile(filename, 3). Does the RDD correctly have three partitions?**

■ **Create an RDD based on all the files in the accounts dataset. How does the number of files in the dataset compare to the number of partitions in the RDD?**
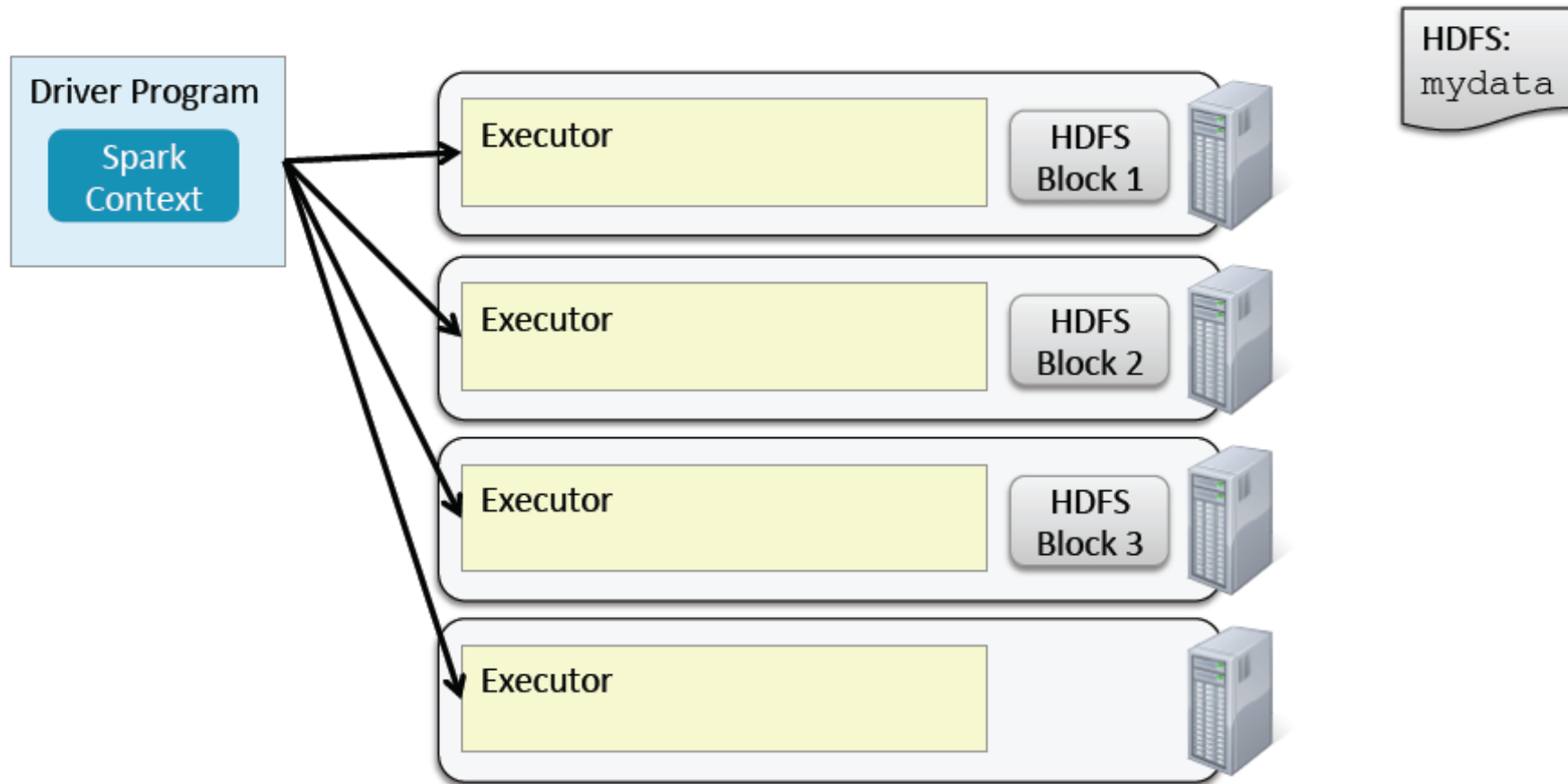
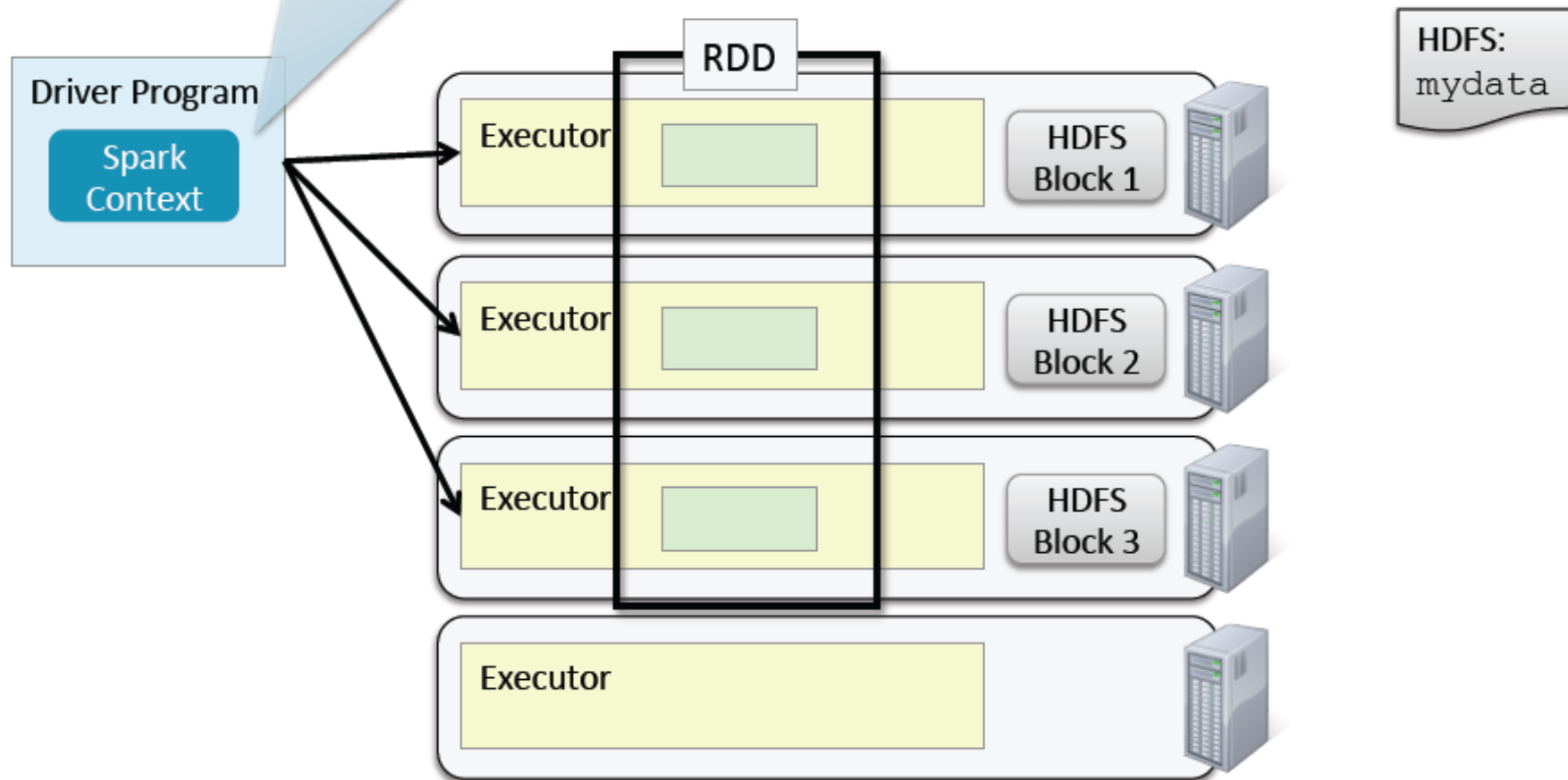# HDFS and Data Locality (1)

HDFS and Data Locality (2)

HDFS and Data Locality (3)

HDFS and Data Locality (4)

```
sc.textFile("hdfs://…mydata").collect()
```

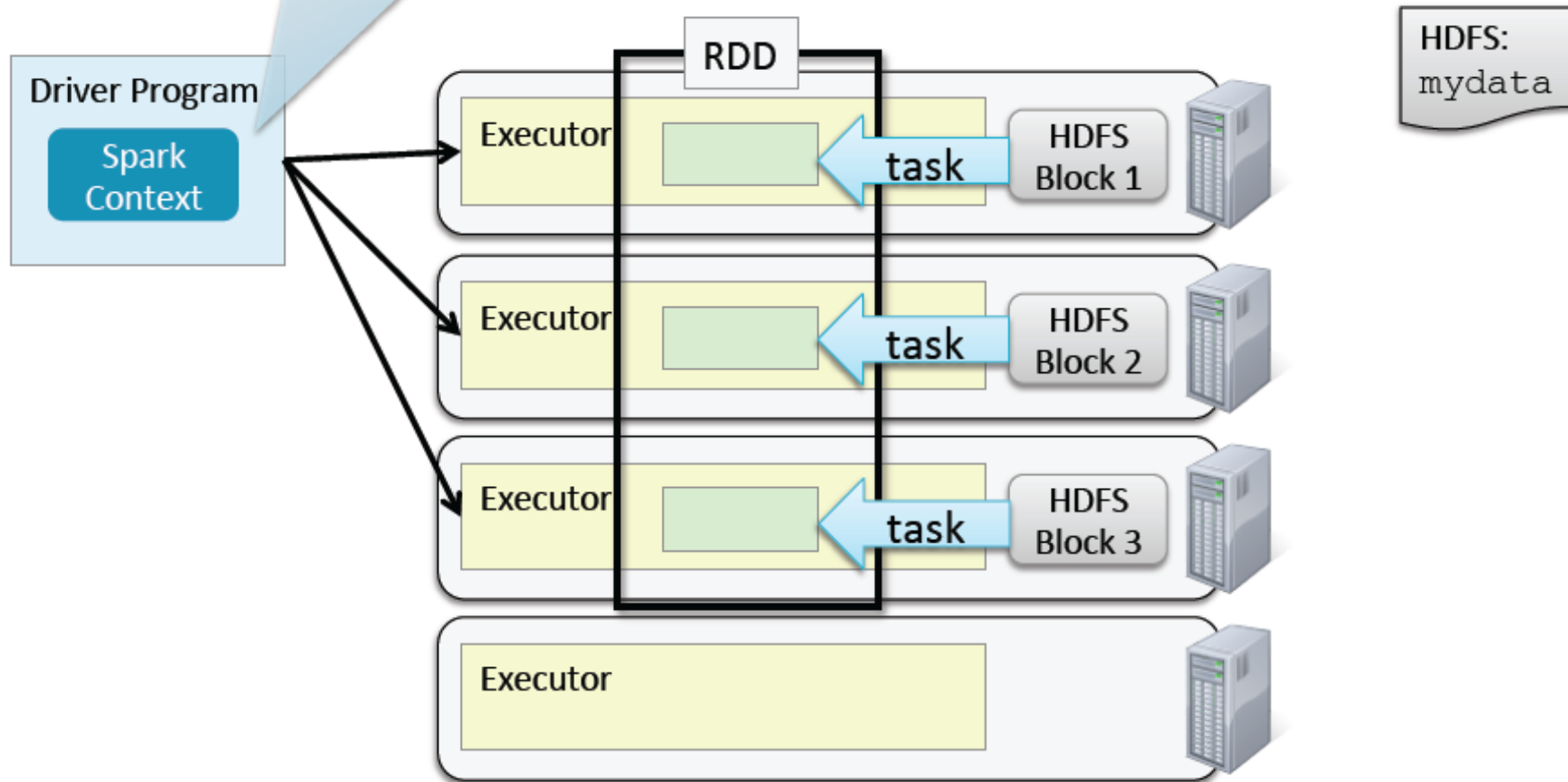By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.

HDFS: mydata

Driver Program

Spark Context

RDD

Executor — HDFS Block 1

Executor — HDFS Block 2

Executor — HDFS Block 3

Executor

HDFS and Data Locality (5)

```
sc.textFile("hdfs://…mydata").collect()
```

An action triggers execution: tasks on executors load data from blocks into partitions

HDFS: mydata

Driver Program

Spark Context

RDD

Executor — task ← HDFS Block 1

Executor — task ← HDFS Block 2

Executor — task ← HDFS Block 3

Executor

HDFS and Data Locality (6)

sc.textFile("hdfs://...mydata").collect()

Data is distributed across executors until an action returns a value to the driver
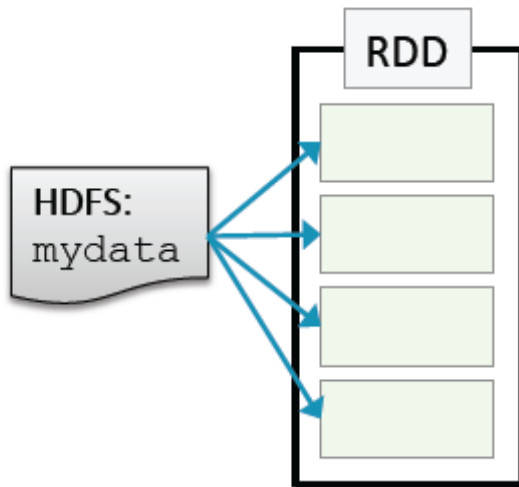
Parallel Operations on Partitions

- RDD operations are executed in parallel on each partition
  - When possible, tasks execute on the worker nodes where the data is in memory

- Some operations preserve partitioning
  - e.g., `map`, `flatMap`, `filter`

- Some operations repartition
  - e.g., `reduce`, `sort`, `group`
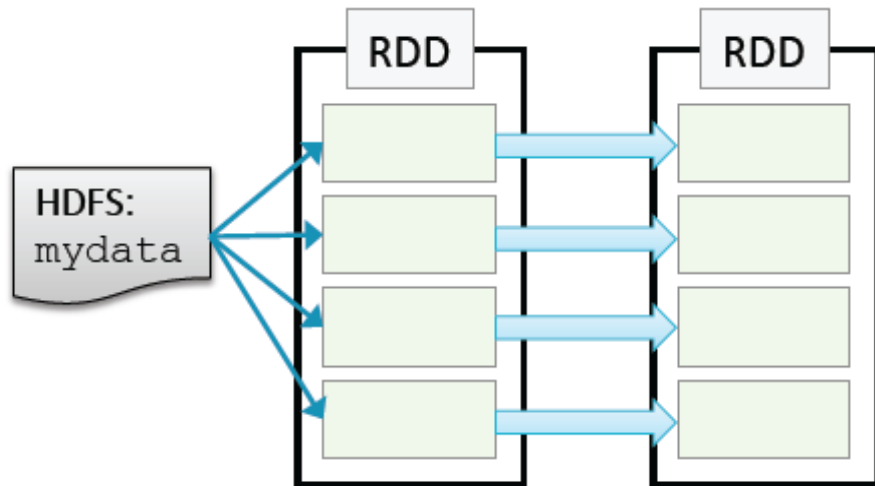
Example: Average Word Length by Letter (1)

```
> avglens = sc.textFile(file)
```
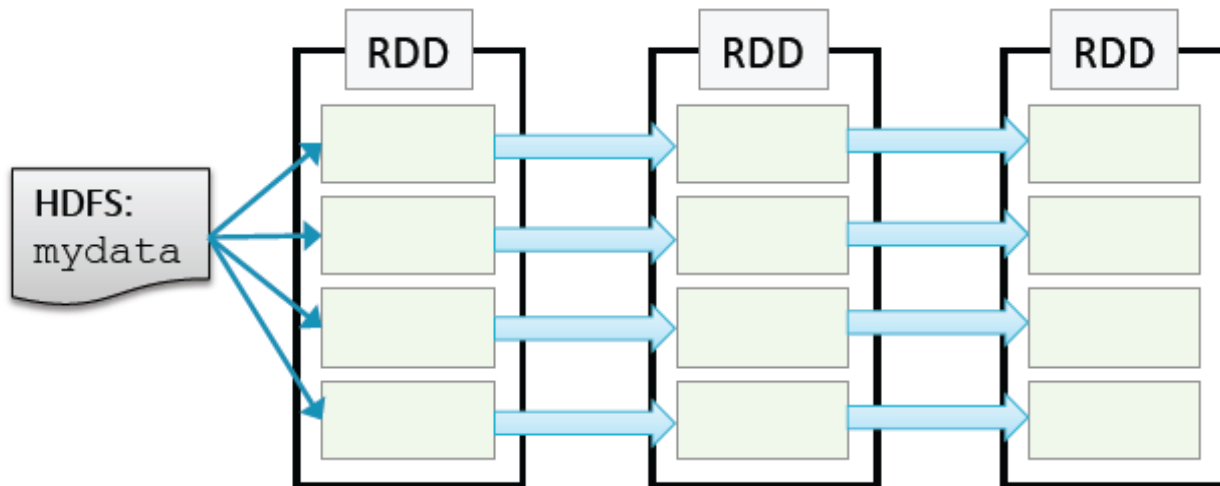

RDD

HDFS:
mydata

Example: Average Word Length by Letter (2)

```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split())
```
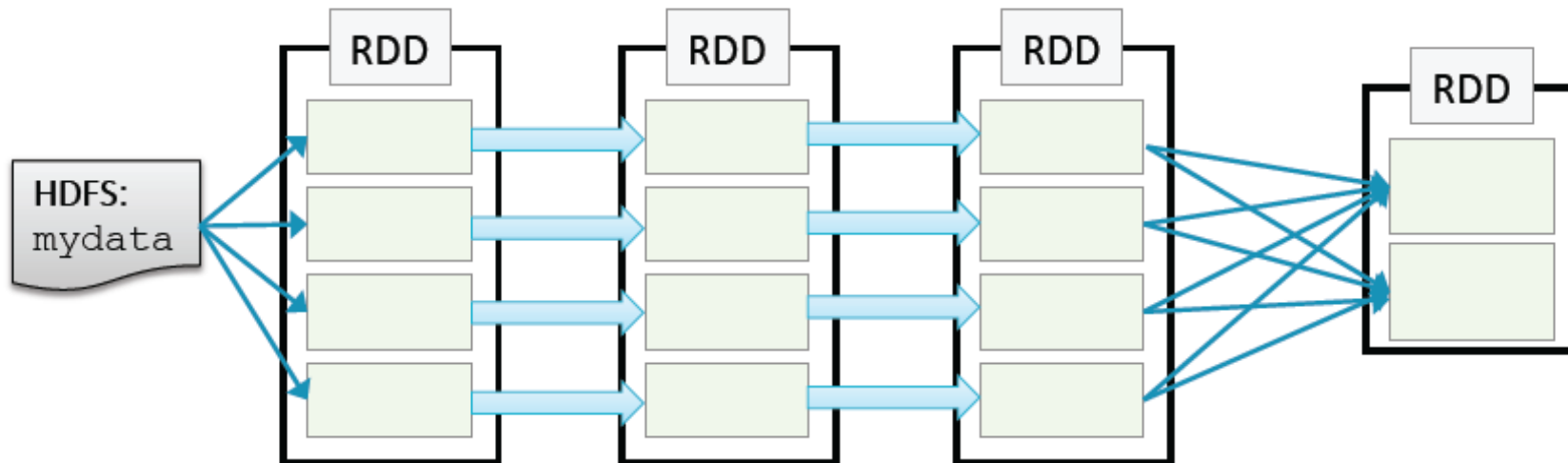
Example: Average Word Length by Letter (3)

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

Example: Average Word Length by Letter (4)

```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word))) \
  .groupByKey()
```
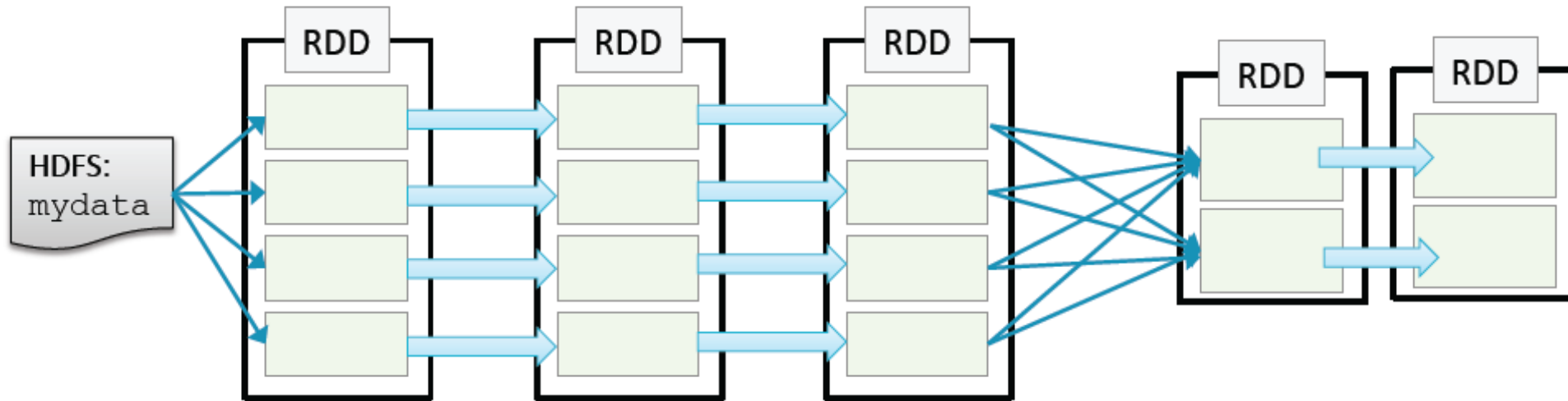
Example: Average Word Length by Letter (5)

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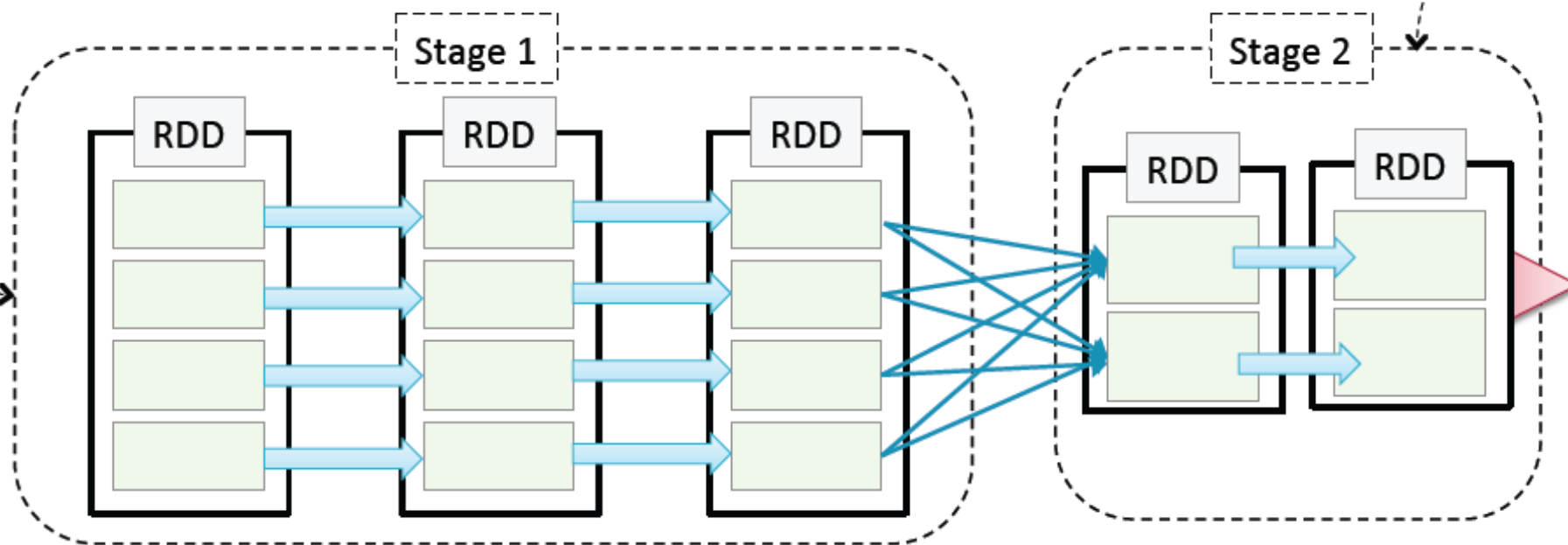
Stages

- Operations that can run on the same partition are executed in *stages*

- Tasks within a stage are pipelined together

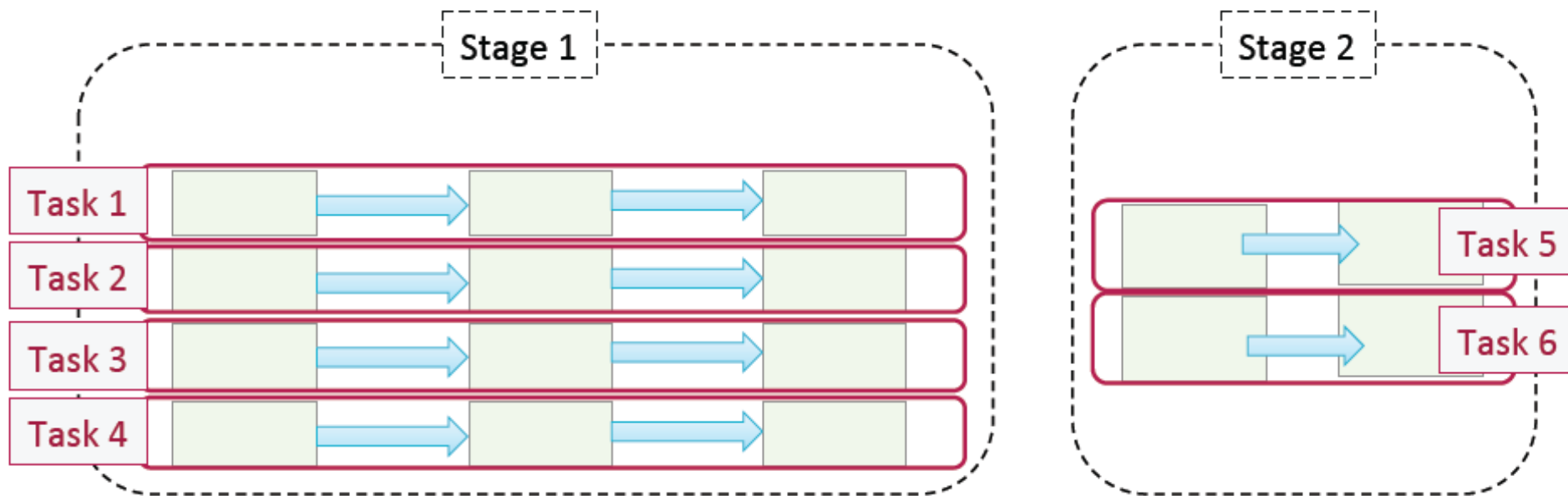- Developers should be aware of stages to improve performance

Spark Execution: Stages (1)

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

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

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

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

Spark Execution: Stages (3)

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```
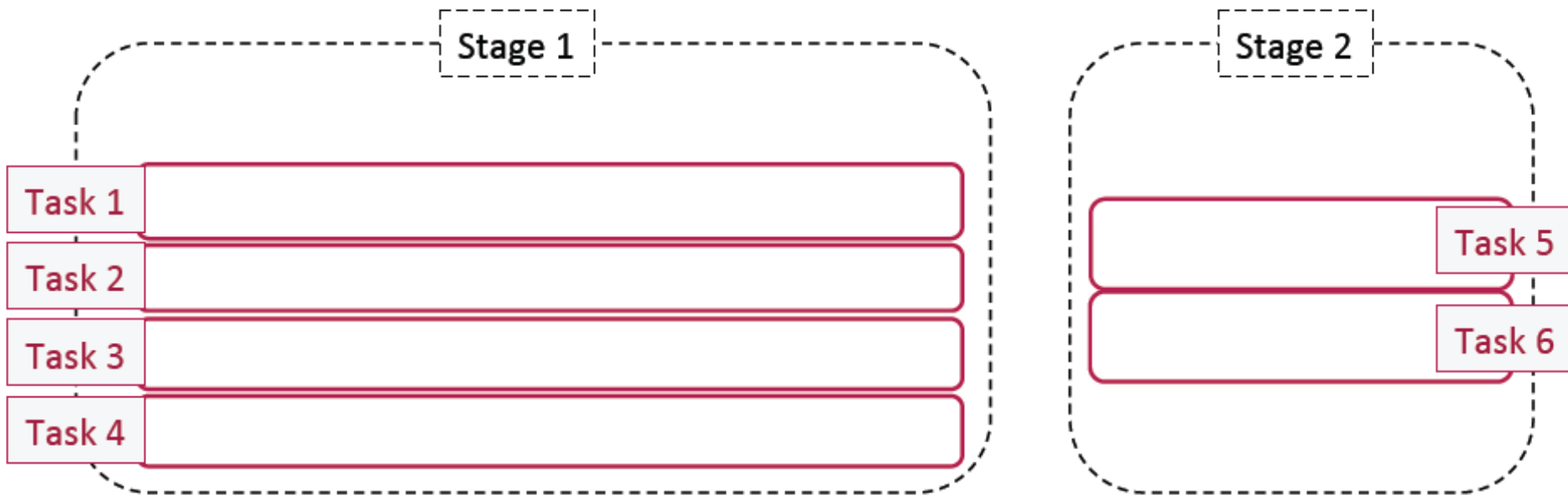
Spark Execution: Stages (4)

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```
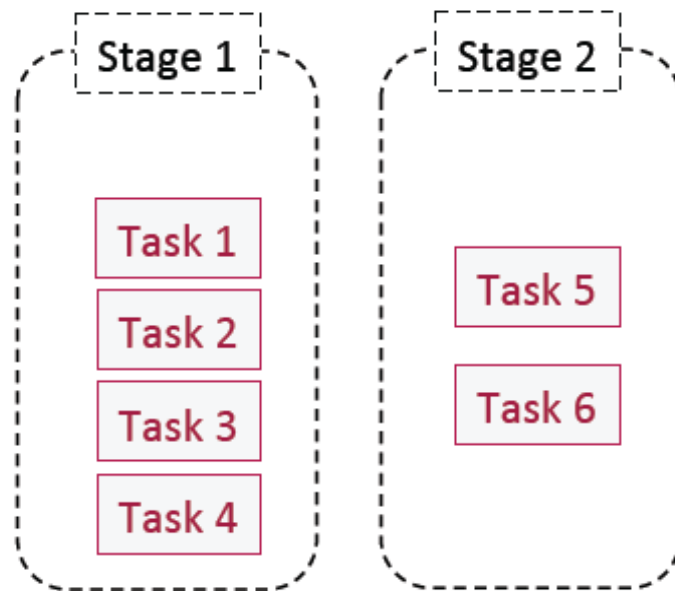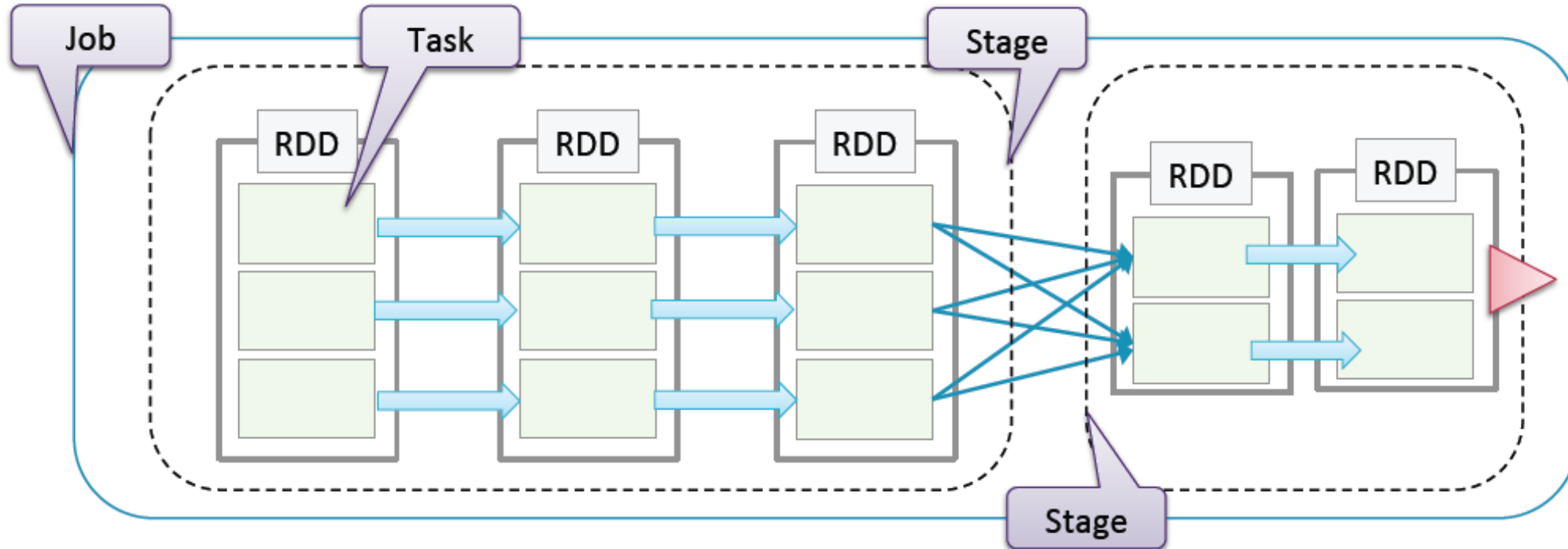
Summary of Spark Terminology

- **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** – can contain any number of jobs managed by a single driver

How Spark Calculates Stages

- **Spark constructs a DAG (Directed Acyclic Graph) of RDD dependencies**

- **_Narrow_ dependencies**
  - Only one child depends on the RDD
  - No shuffle required between nodes
  - Can be collapsed into a single stage
  - e.g., `map`, `filter`, `union`

- **_Wide_ (or _shuffle_) dependencies**
  - Multiple children depend on the RDD
  - Defines a new stage
  - e.g., `reduceByKey`, `join`, `groupByKey`

Viewing the Stages using toDebugString (Python)

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

> print avglens.toDebugString()
(2) PythonRDD[13] at RDD at …
 |  MappedRDD[12] at values at …
 |  ShuffledRDD[11] at partitionBy at …
+-(4) PairwiseRDD[10] at groupByKey at …
    |  PythonRDD[9] at groupByKey at …
    |  myfile MappedRDD[7] at textFile at …
    |  myfile HadoopRDD[6] at textFile at …
```
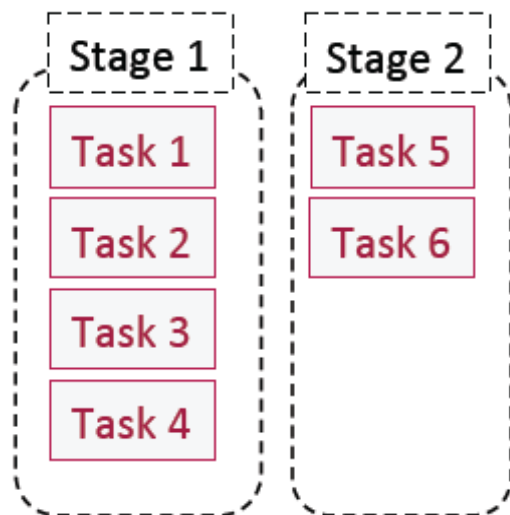
Stage 2

Stage 1

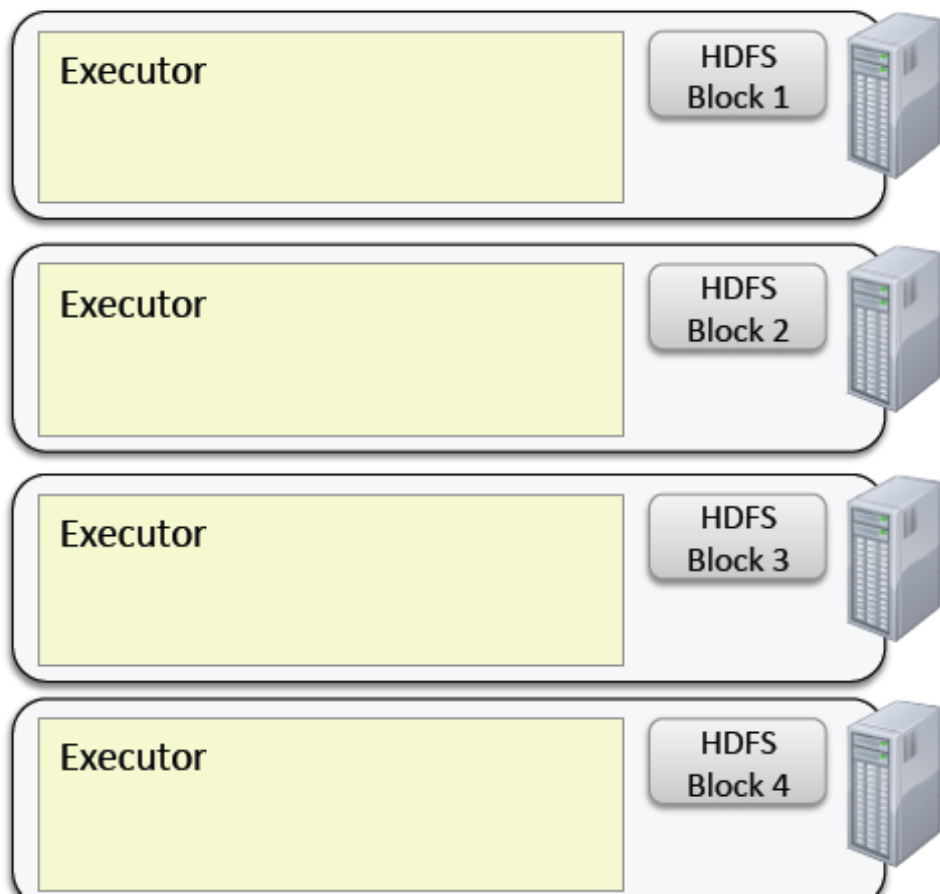Indents indicate stages (shuffle boundaries)

Practice: Set up the job

■ **Create a RDD of accounts, keyed by ID and with first name, last name for the value**

■ **The first step is to construct another RDD with the total number of web hits for each user ID**

■ **Then join the two RDDs by user ID, and construct a new RDD based on first name, last name, and total hits. Name the result RDD as accountHits.**

■ **Print the results of accountHits.toDebugString and review the output. Based on this, see if you can determine**

- How many stages are in this job?
- Which stages are dependent on which?
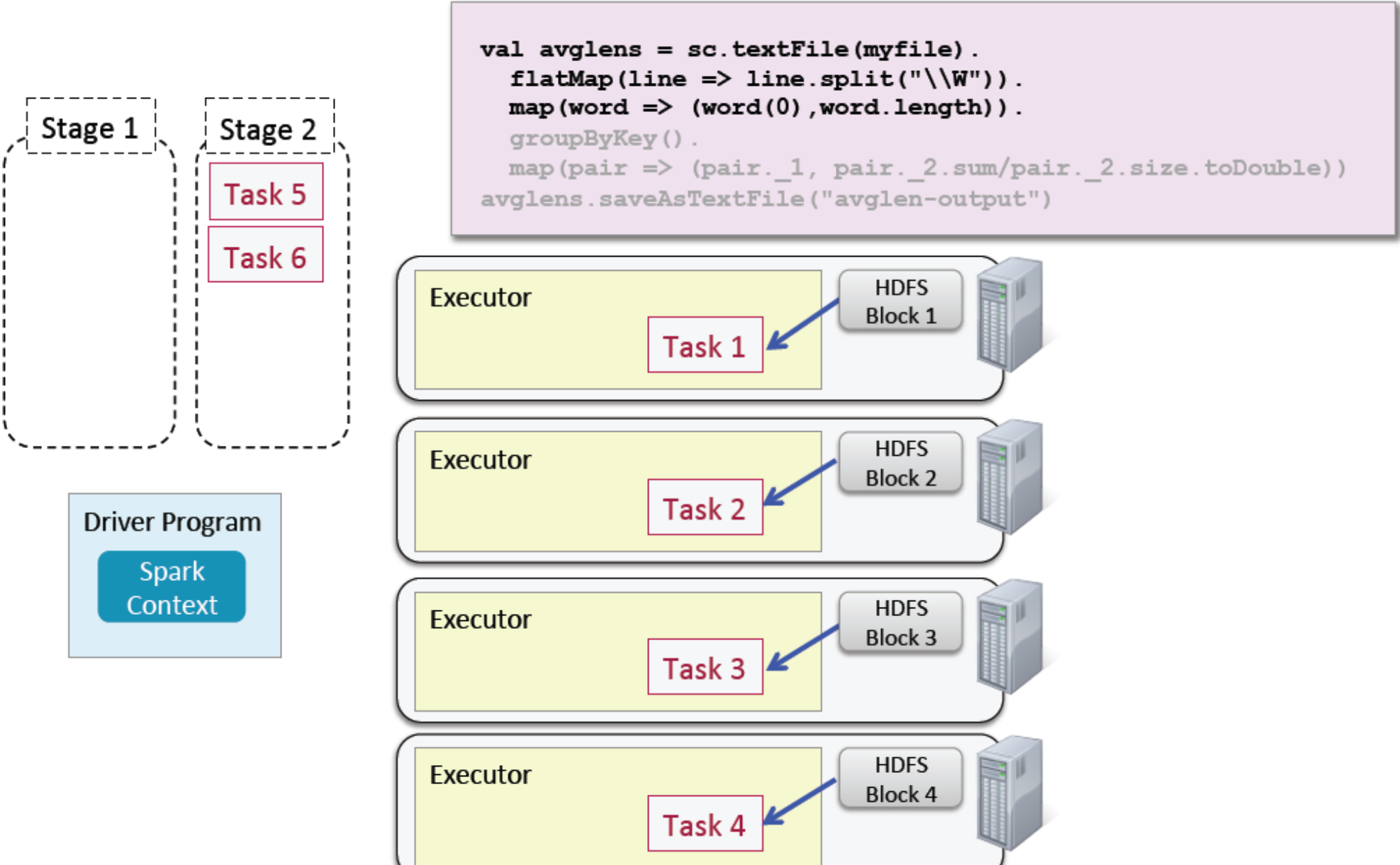- How many tasks will each stage consist of?

Spark Task Execution (1)

```
val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```

Stage 1

Task 1

Task 2

Task 3

Task 4

Stage 2

Task 5

Task 6

Driver Program

Spark Context

Executor — HDFS Block 1

Executor — HDFS Block 2

Executor — HDFS Block 3

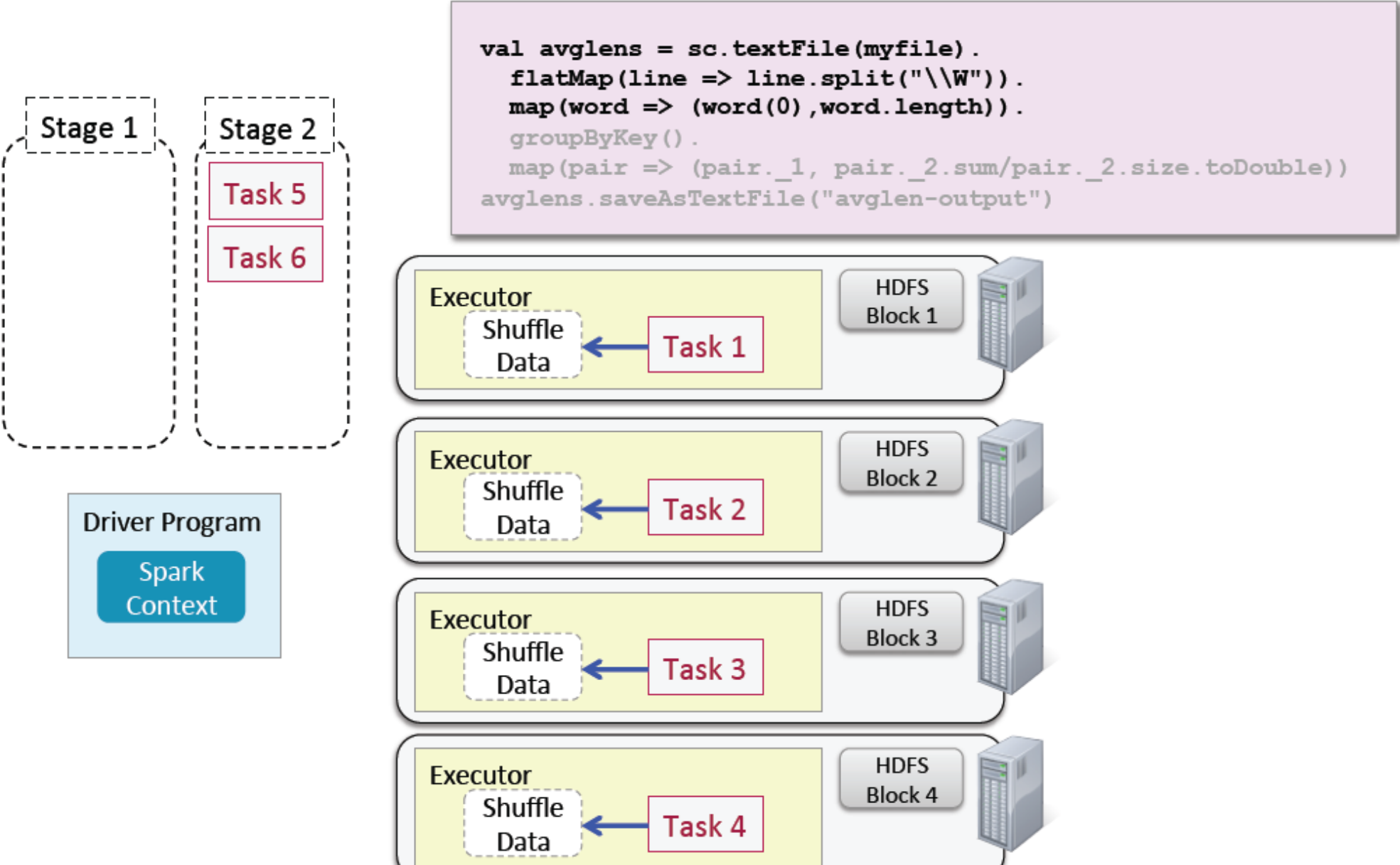Executor — HDFS Block 4

Spark Task Execution (2)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```
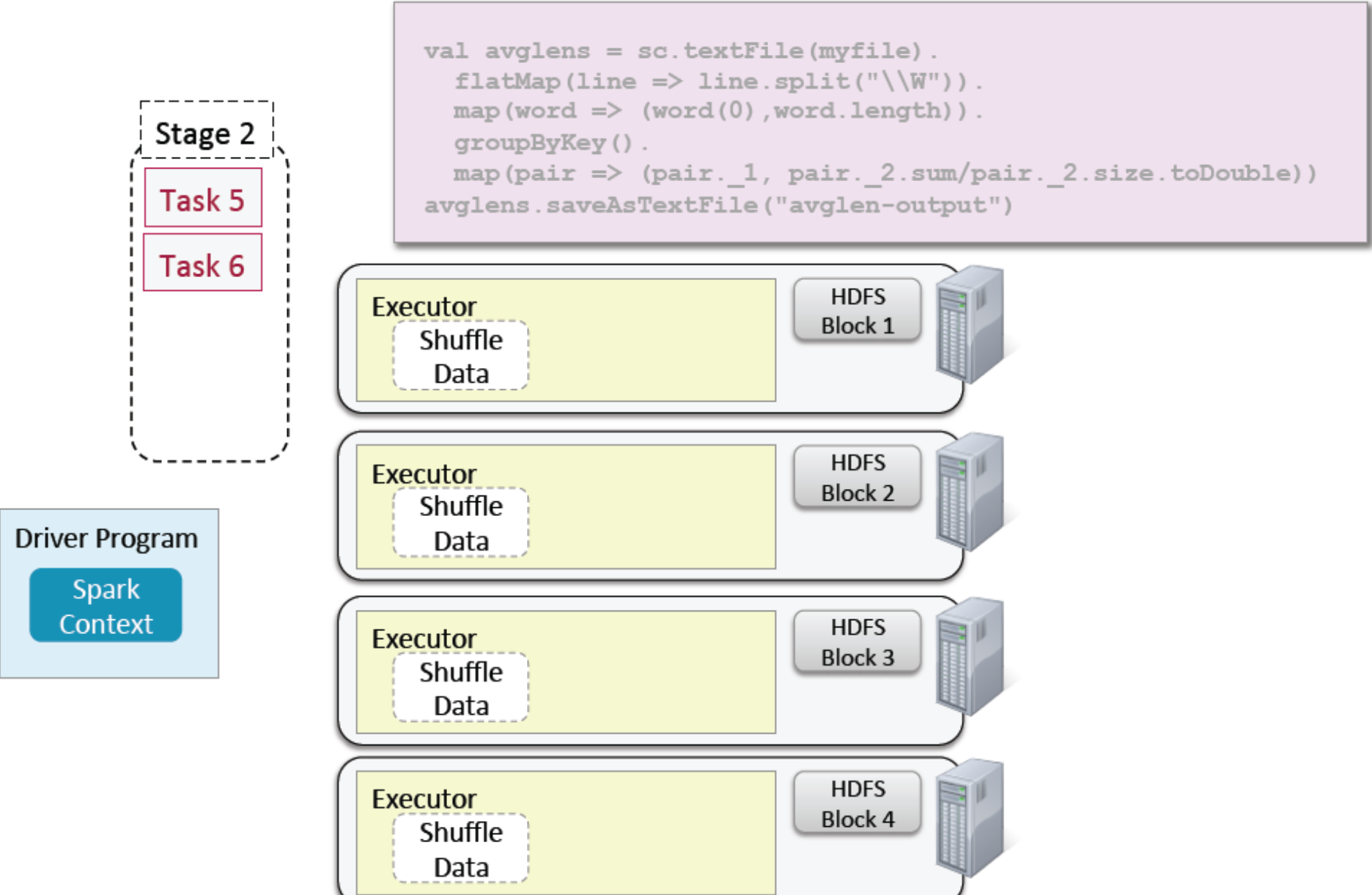

Stage 1 Stage 2

Task 5

Task 6

Driver Program

Spark Context

Executor — Task 1 — HDFS Block 1

Executor — Task 2 — HDFS Block 2

Executor — Task 3 — HDFS Block 3

Executor — Task 4 — HDFS Block 4

Spark Task Execution (3)

```
val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```
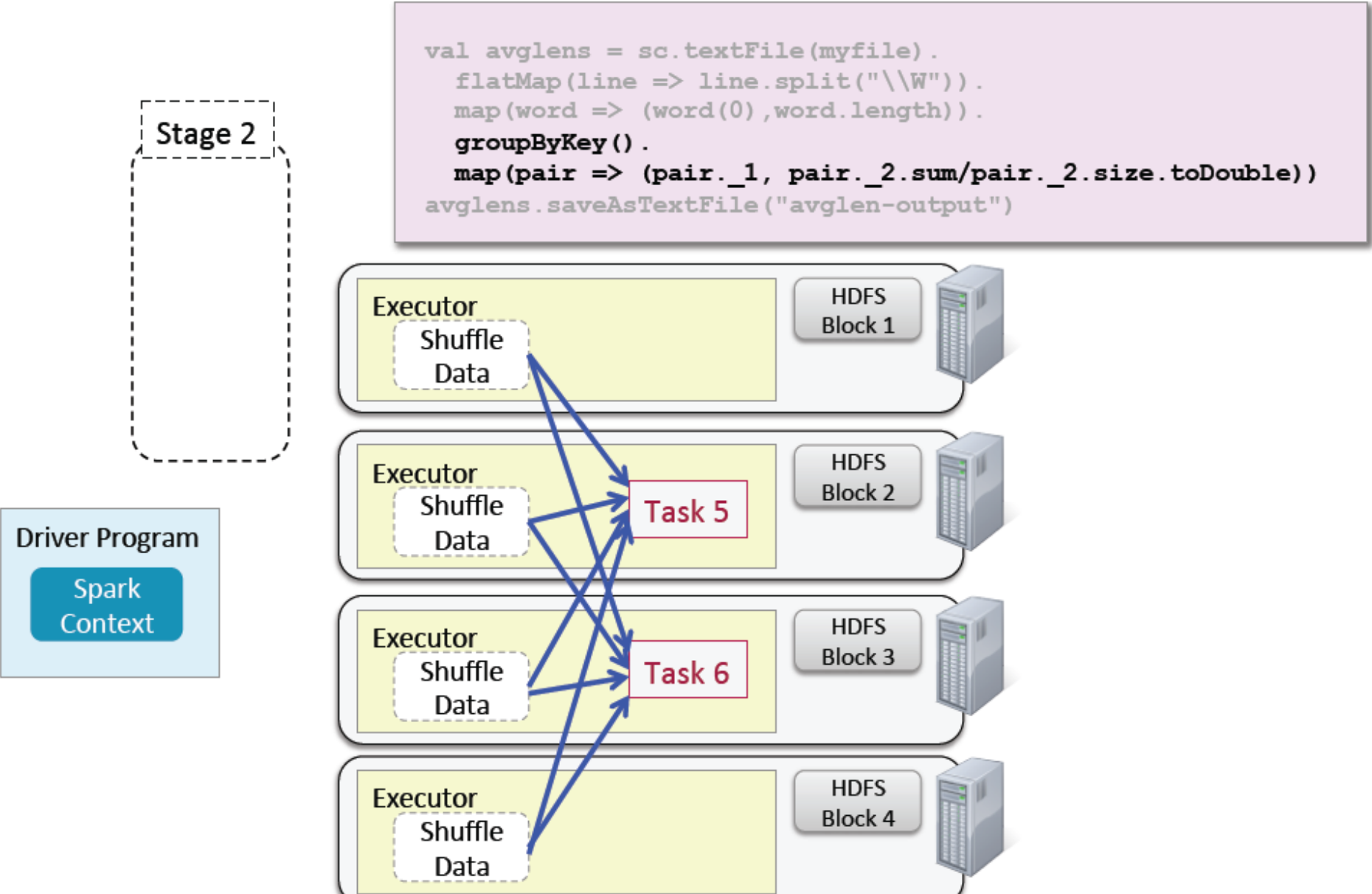
Spark Task Execution (4)


```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```

Stage 2

Task 5

Task 6

Driver Program

Spark Context

Executor
Shuffle Data

HDFS Block 1

Executor
Shuffle Data

HDFS Block 2

Executor
Shuffle Data

HDFS Block 3

Executor
Shuffle Data

HDFS Block 4

Spark Task Execution (5)

Spark Task Execution (6)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```

Driver Program

Spark Context

Executor — HDFS Block 1

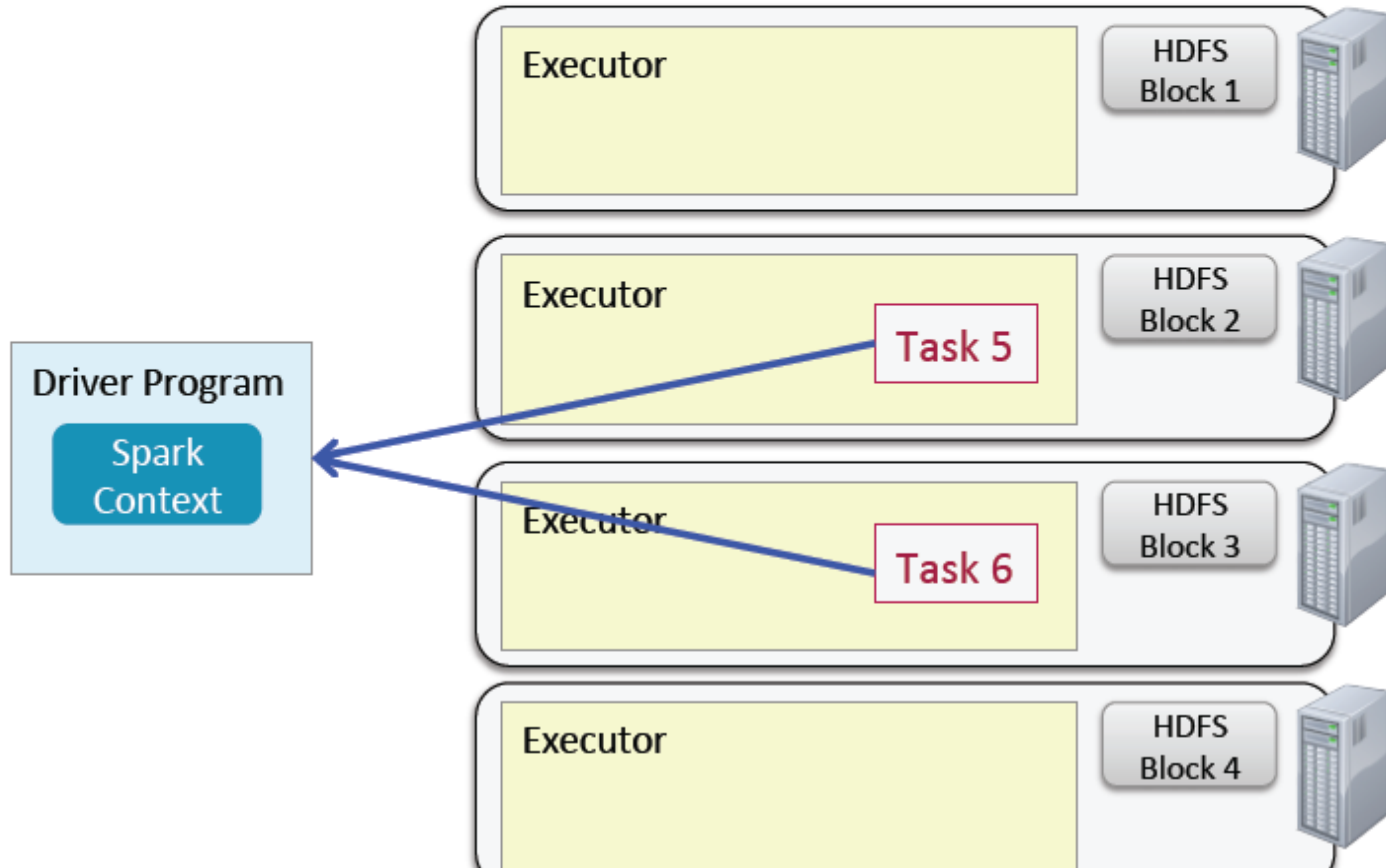Executor — HDFS Block 2 — Task 5 → part-00000

Executor — HDFS Block 3 — Task 6 → part-00001

Executor — HDFS Block 4

Spark Task Execution (alternate ending)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.collect()
```

Controlling the Level of Parallelism

- **"Wide" operations (e.g., `reduceByKey`) partition result RDDs**
  - More partitions = more parallel tasks
  - Cluster will be under-utilized if there are too few partitions

- **You can control how many partitions**
  - Configure with the **`spark.default.parallelism`** property

  ```
  spark.default.parallelism      10
  ```

  - Optional **`numPartitions`** parameter in function call

  ```
  > words.reduceByKey(lambda v1, v2: v1 + v2, 15)
  ```

Viewing Stages in the Spark Application UI (1)

- You can view jobs and stages in the Spark Application UI

Viewing Stages in the Spark Application UI (2)

- Select the job to view execution stages

Essential Points

- RDDs are stored in the memory of Spark executor JVMs

- Data is split into partitions – each partition in a separate executor

- RDD operations are executed on partitions in parallel

- Operations that depend on the same partition are pipelined together in stages
    - e.g., `map`, `filter`

- Operations that depend on multiple partitions are executed in separate stages
    - e.g., `join`, `reduceByKey`