Parallel Processing in Spark

Prof. Hyuk-Yoon Kwon

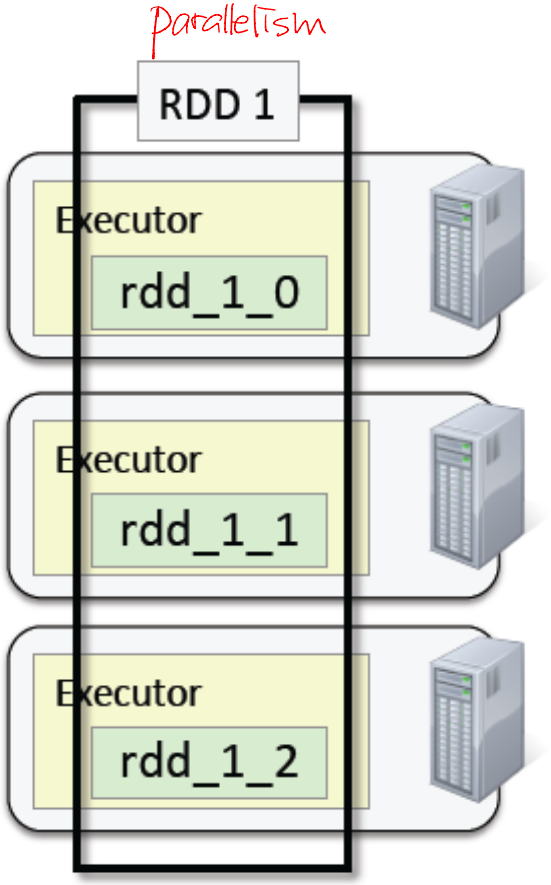
Parallel Programming with Spark

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

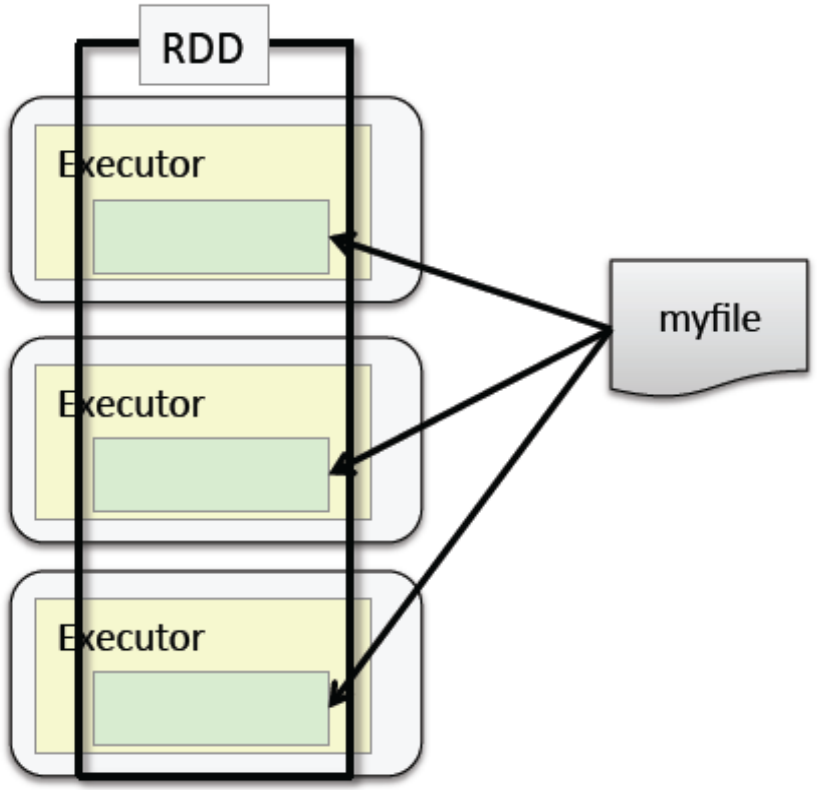


File Partitioning: Single Files

- **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

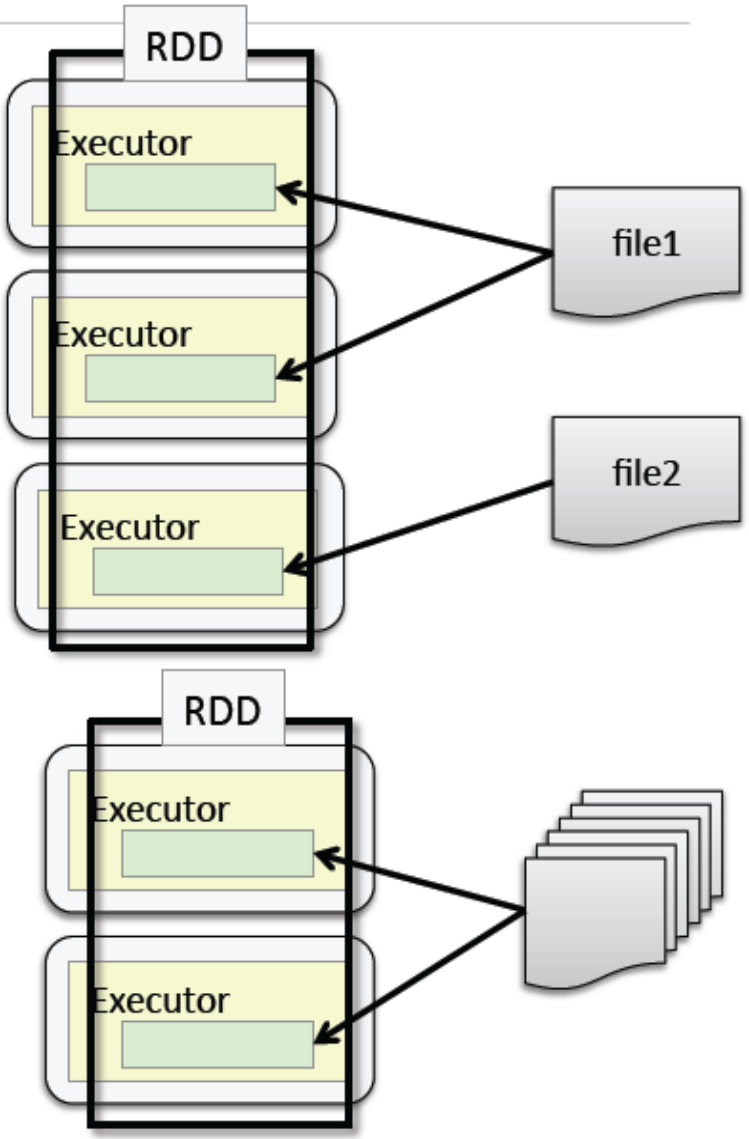
```
sc.textFile("myfile",3)
```



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

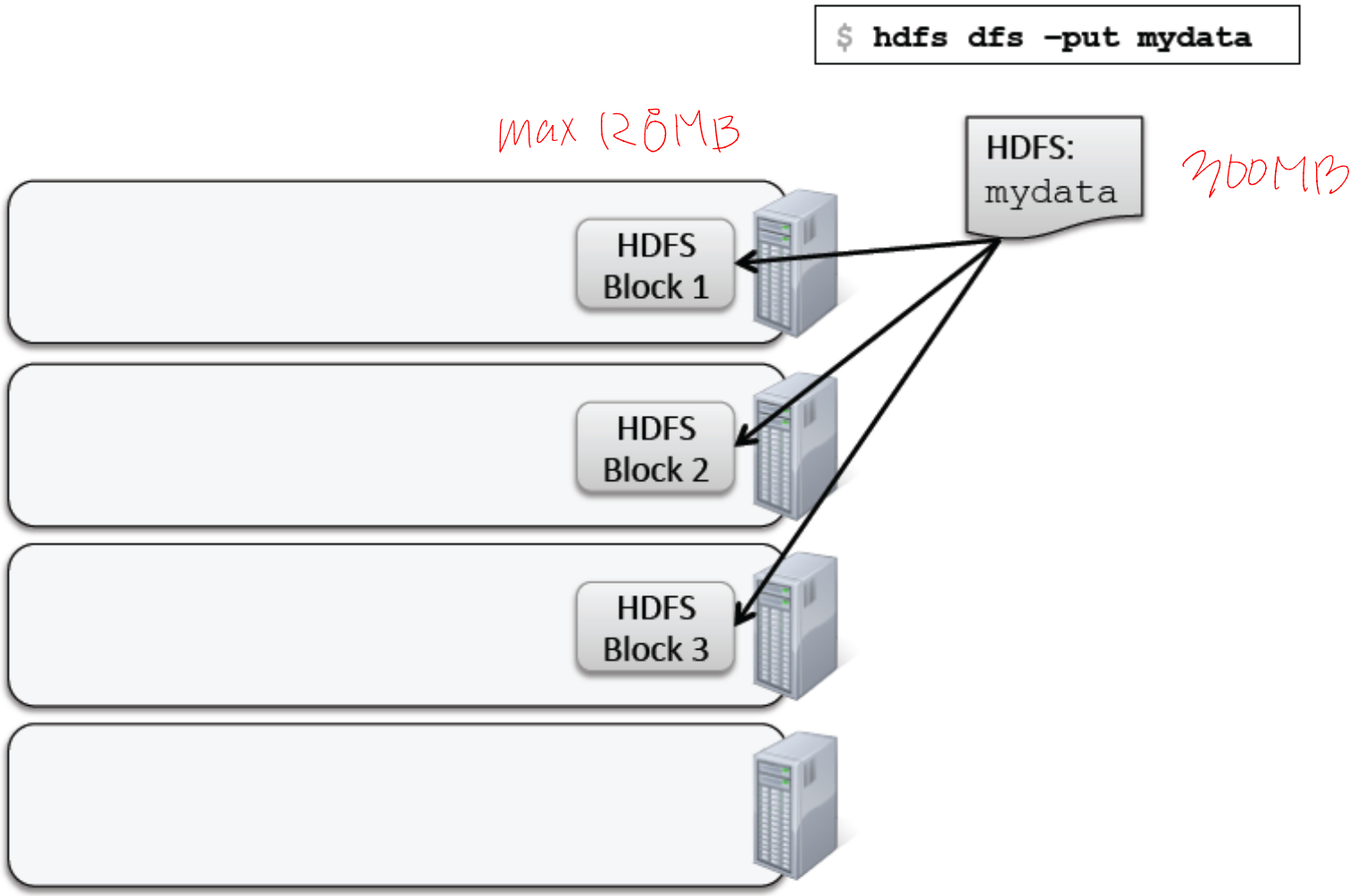
Practice: Explore Partitioning of file-based RDDs

- **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?**

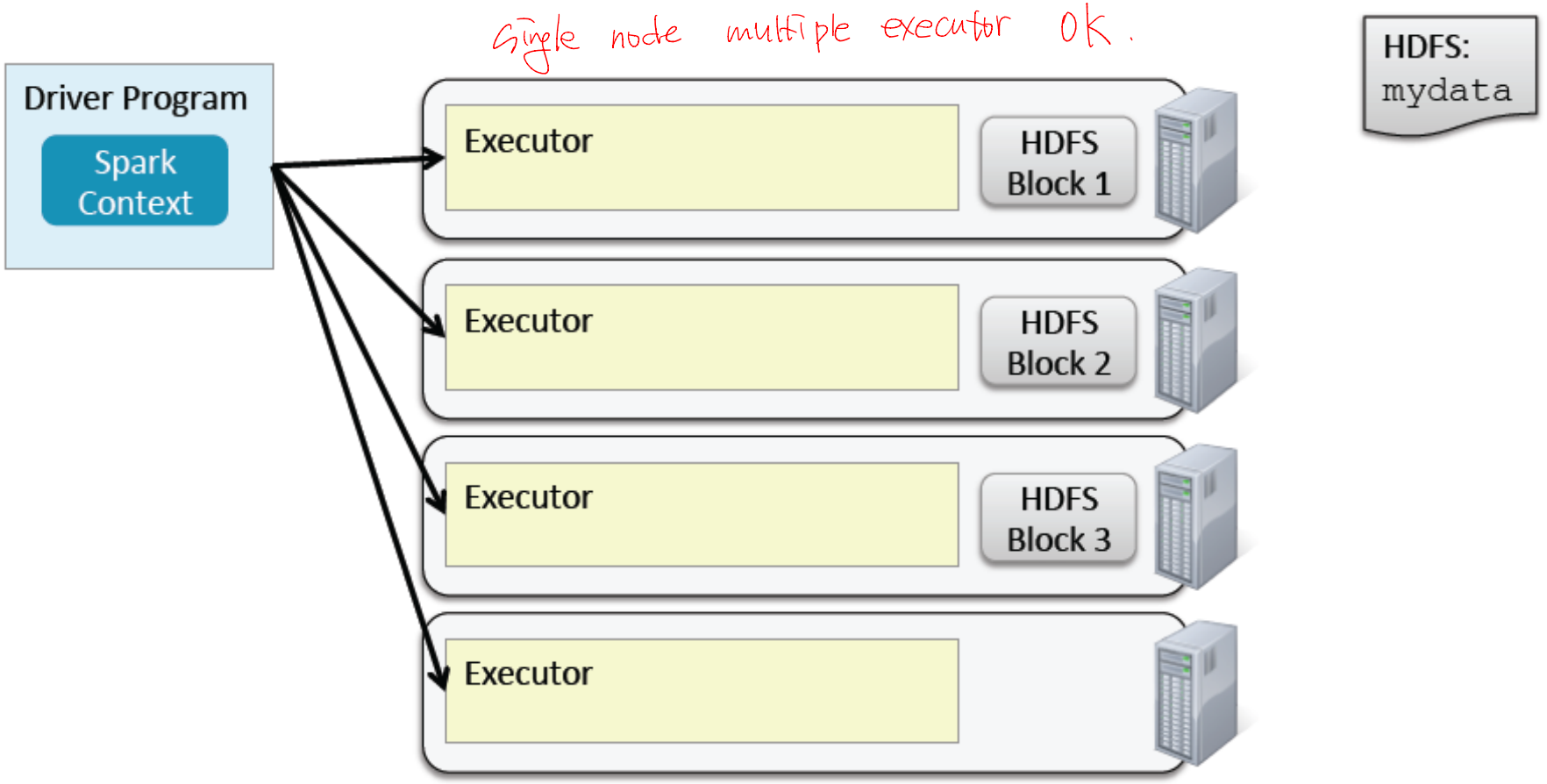
HDFS and Data Locality (1)

```
$ pyspark --master local[2]

> accounts = sc.textFile("/loudacre/accounts/part-m-00000")
> print accounts.toDebugString()

> accounts = sc.textFile("/loudacre/accounts/part-m-00000", 3)
> print accounts.toDebugString()

> accounts = sc.textFile("/loudacre/accounts/*")
> print accounts.toDebugString()
```

HDFS and Data Locality (2)



HDFS and Data Locality (3)

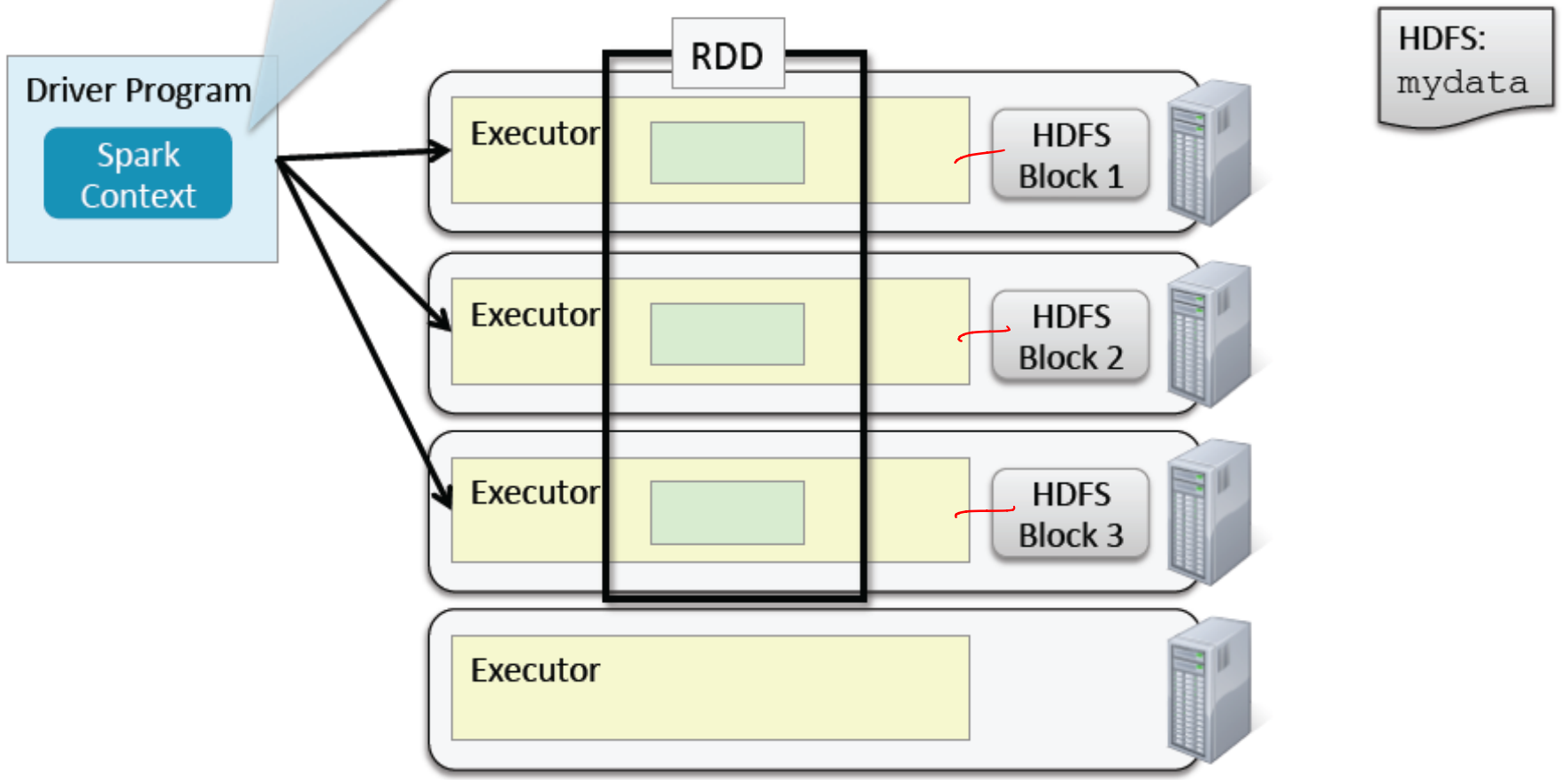


HDFS and Data Locality (4)

lazy

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

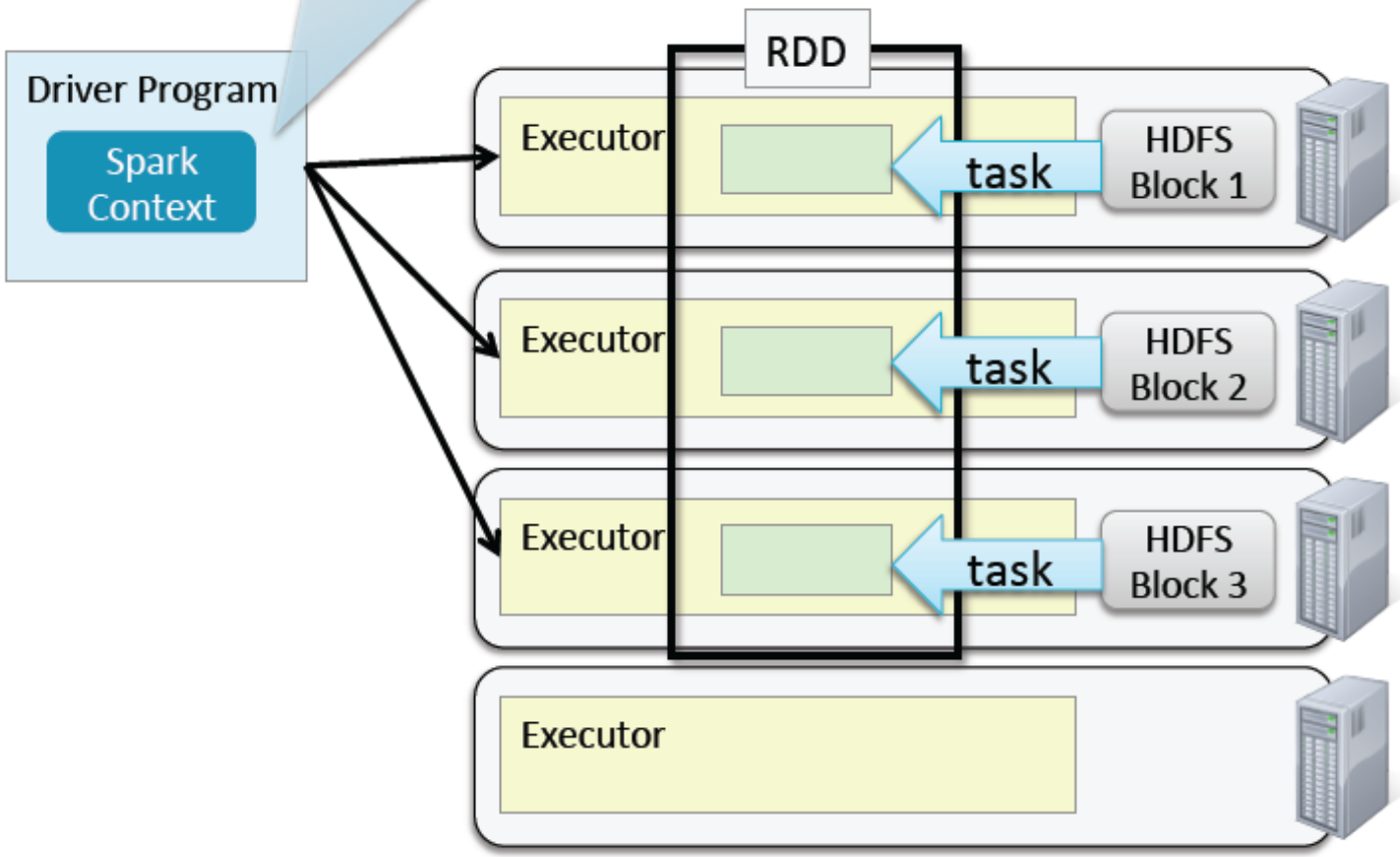
By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.



HDFS and Data Locality (5)

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

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

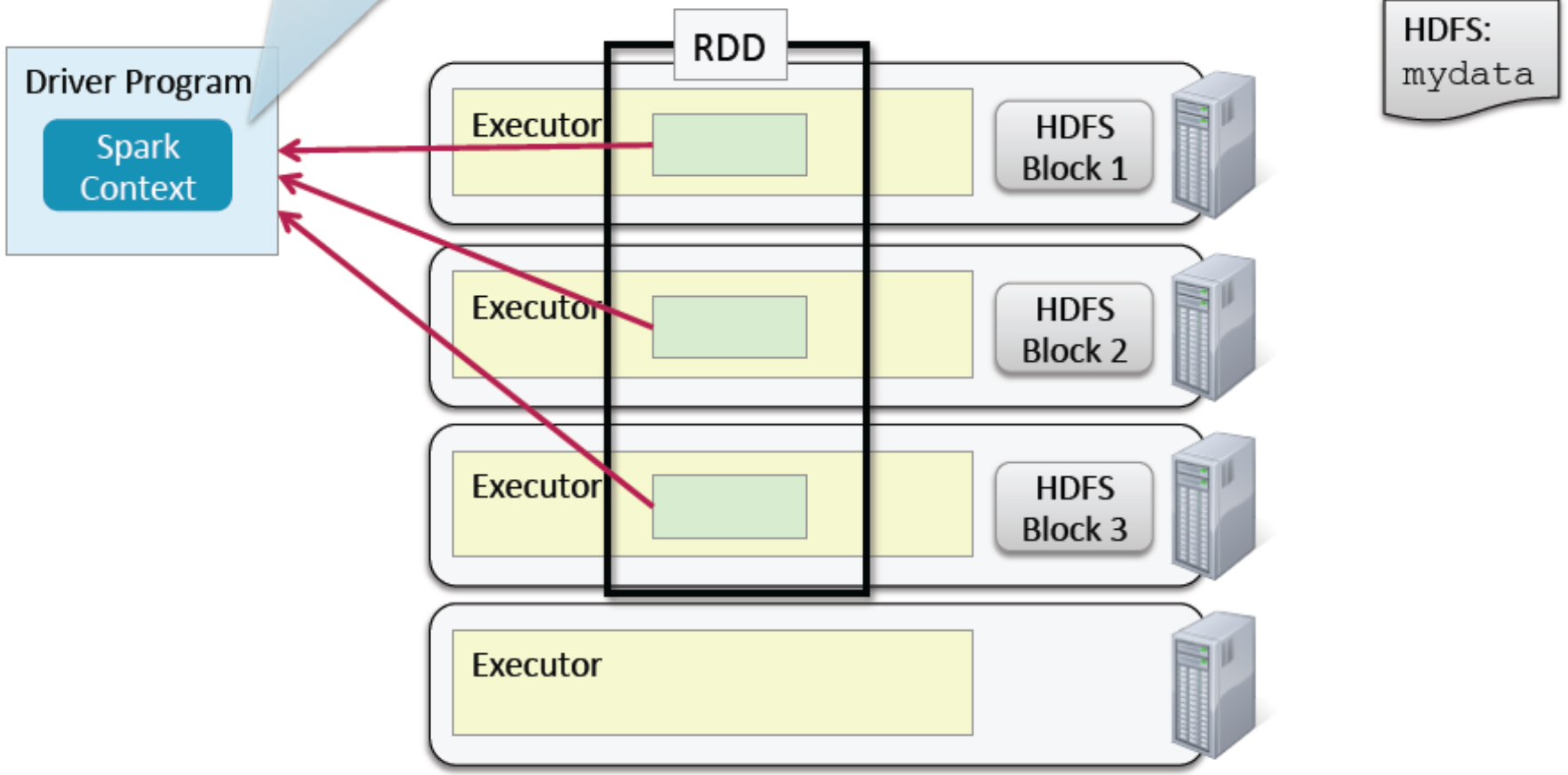


η task
→ map to physically storing node (ideal)
minimal network transfer

HDFS and Data Locality (6)

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

Data is distributed across executors until an action returns a value to the driver



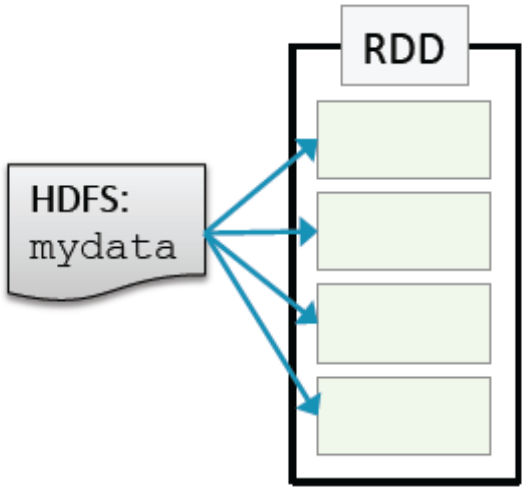
Parallel Operations on Partitions

- **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** no shuffle (재배치)
  - e.g., `map`, `flatMap`, `filter` 각 파티션 개별 처리 가능
- **Some operations repartition** shuffle required
  - e.g., `reduce`, `sort`, `group` 여러 노드에 분산되어 있는 수 있는 정보 사용
    ⇒ 전체 데이터 재분할.

Example: Average Word Length by Letter (1)

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

HDFS ⇒ block 구조 기반. 전체 파일 자동으로 파티션 분할
각 파티션은 독립/병렬 처리

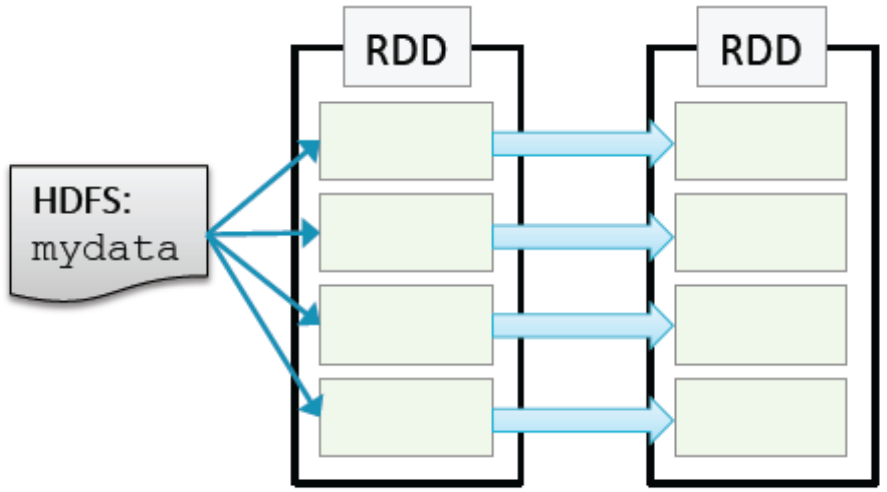


Example: Average Word Length by Letter (2)

```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split())
```

preserve partitions

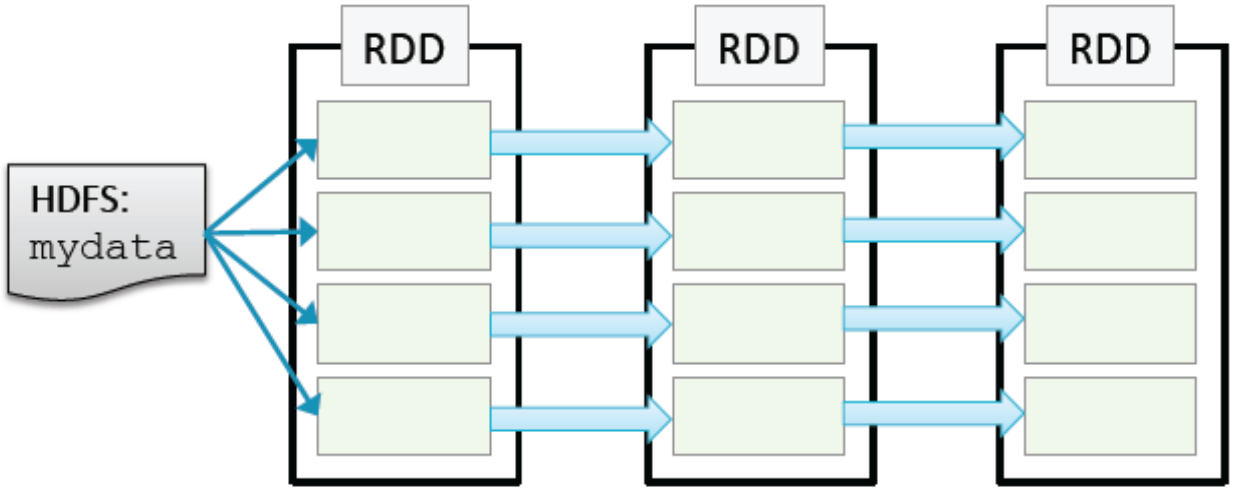
Map 계열 ⇒ 각 파티션 내 개별 레코드에 대해 독립적 작동. (데이터 저장하는 노드에서 병렬 처리)



Example: Average Word Length by Letter (3)

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

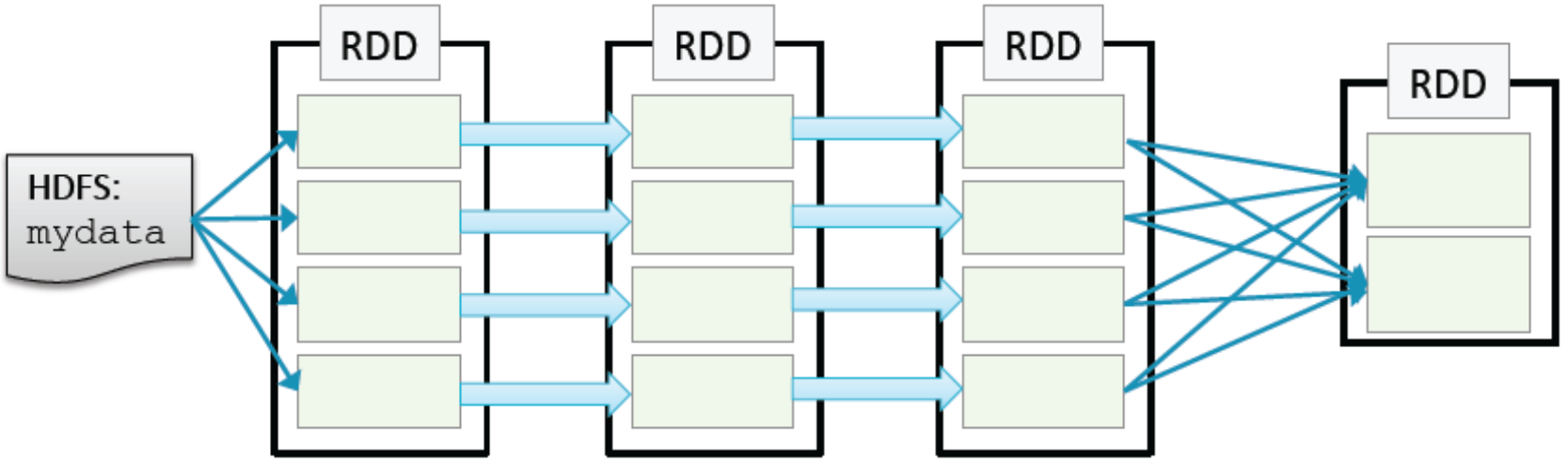
preserve partitioning



Example: Average Word Length by Letter (4)

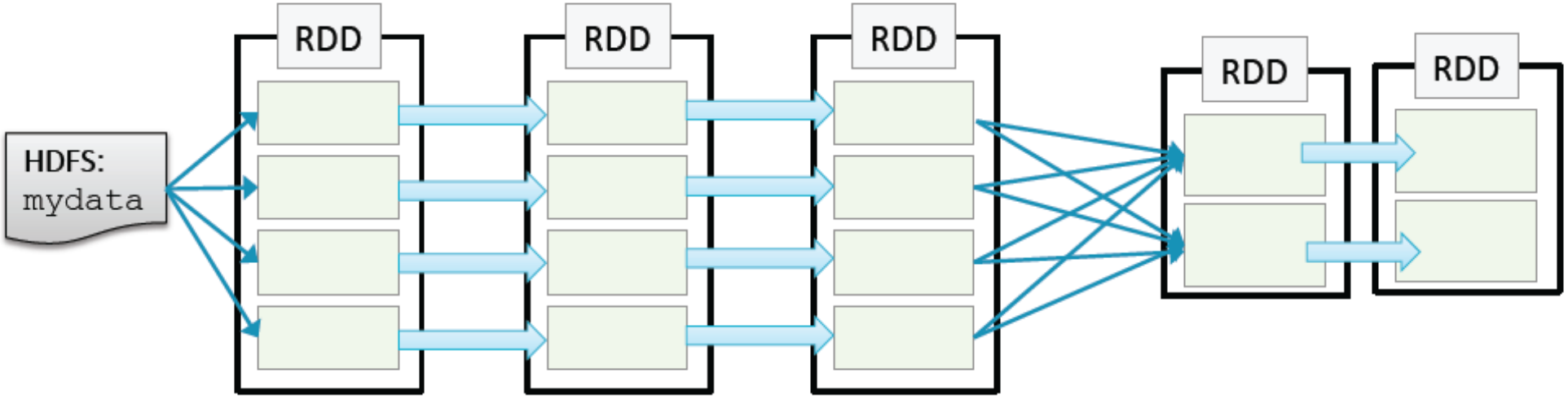
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word))) \
  .groupByKey()
```

reducer type operation ⇒ cluster 전체의 모든 관련 데이터 접근 필요.
⇒ shuffle 필요. (network 활용)



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)))
```

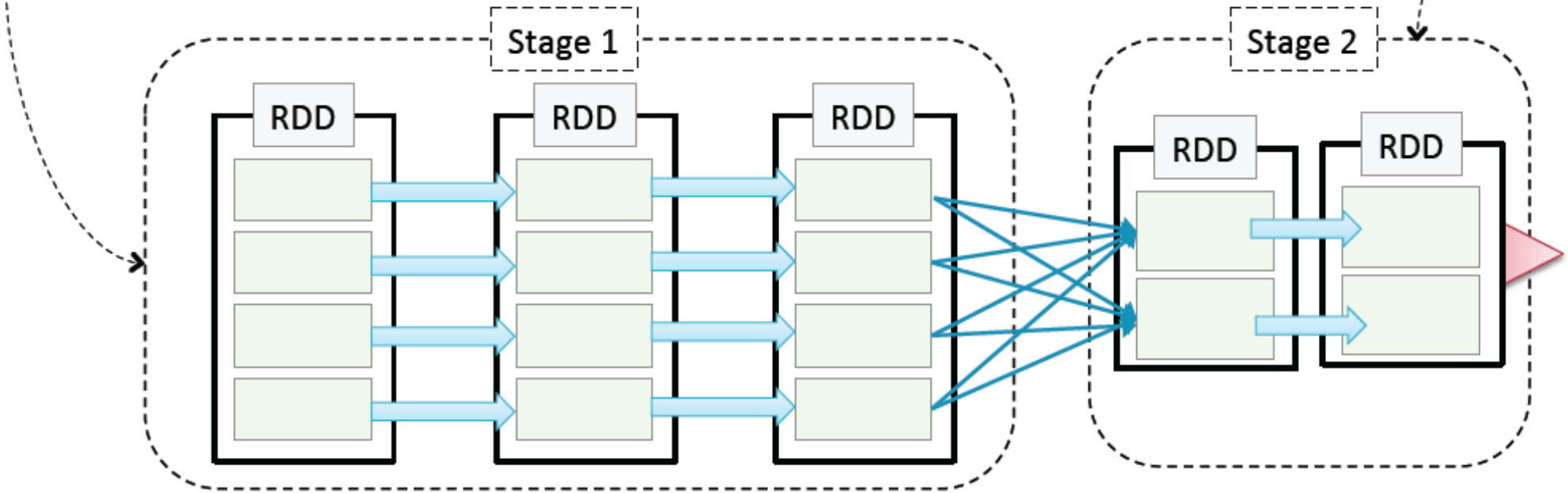


Stages

- Operations that can run on the same partition are executed in *stages*
- Tasks within a stage are pipelined together parallel w/o shuffle
- Developers should be aware of stages to improve performance
  ↳ shuffle 발생 위치 파악 및 최적화 가능.

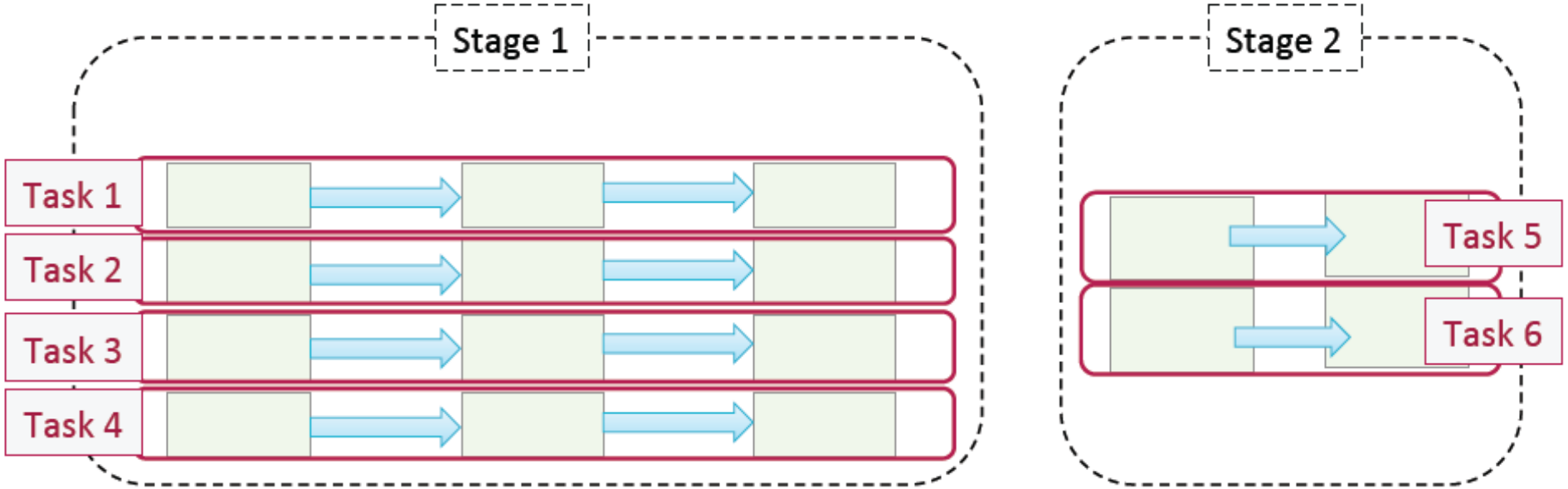
Spark Execution: Stages (1)

```
> 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")
```



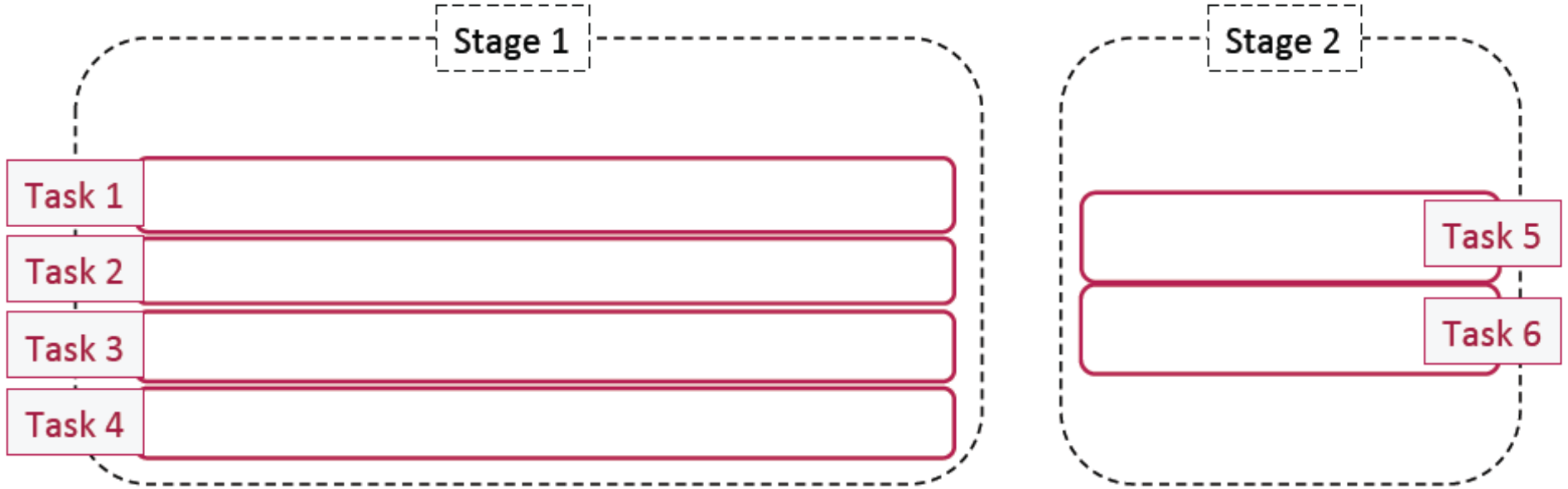
Spark Execution: Stages (2)

```
> 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")
```



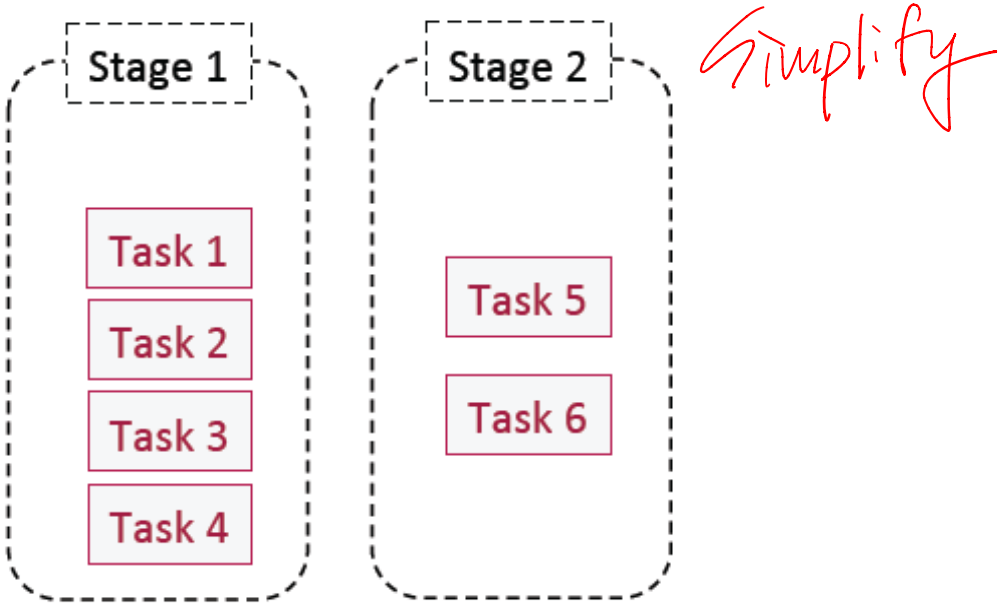
Spark Execution: Stages (3)

```
> 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")
```



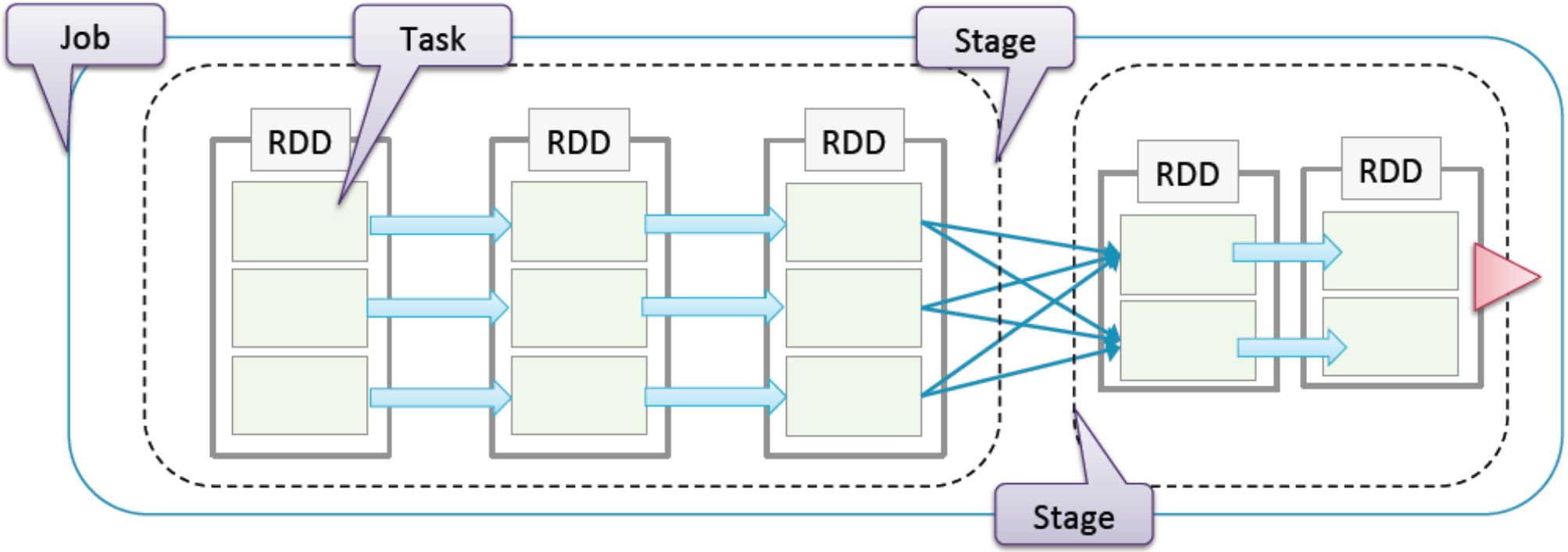
Spark Execution: Stages (4)

```
> 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** ⇒ can be combined into single stage
  - 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** ⇒ movement (shuffle)! start of new stage distinct from prev one
  - 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 Structure.

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?

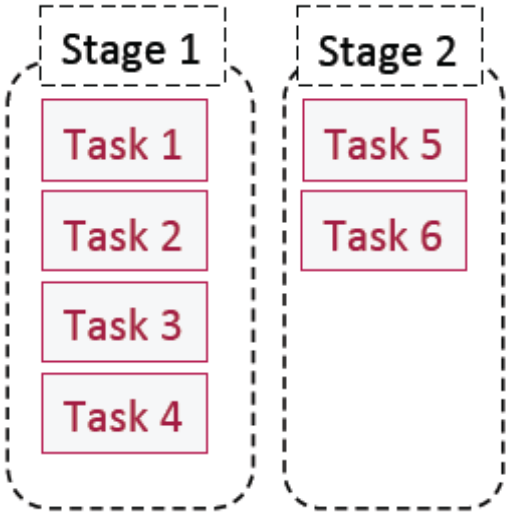
```
> accountsByID = accounts \
  .map(lambda s: s.split(',')) \
  .map(lambda values: (values[0],values[4] + ',' + values[3]))
> usereqs = sc \
  .textFile("/loudacre/weblogs/*6") \
  .map(lambda line: line.split()) \
  .map(lambda words: (words[2],1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
> acounthits = accountsByID.join(userreqs).values()
> print accounthits.toDebugString()
```

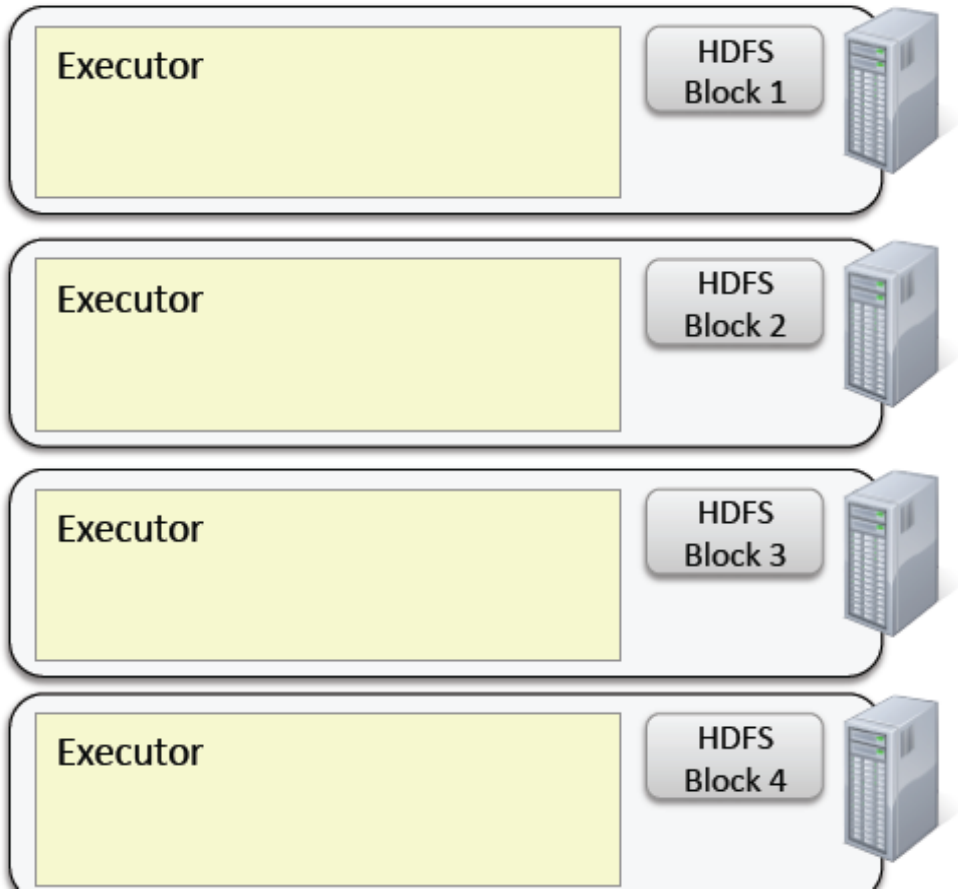
```
In [15]: print accounthits.toDebugString()
(35) PythonRDD[33] at RDD at PythonRDD.scala:42 []
 |   MapPartitionsRDD[32] at mapPartitions at PythonRDD.scala:338 []
 |   ShuffledRDD[31] at partitionBy at NativeMethodAccessorImpl.java:-2 []
 +-(35) PairwiseRDD[30] at join at <ipython-input-14-0bfe05f71275>:1 []
    |   PythonRDD[29] at join at <ipython-input-14-0bfe05f71275>:1 []
    |   UnionRDD[28] at union at NativeMethodAccessorImpl.java:-2 []
    |   PythonRDD[26] at RDD at PythonRDD.scala:42 []
    |   /loudacre/accounts/part-m-00000 MapPartitionsRDD[3] at textFile at NativeMethodAccessorImpl.java:-2 []
    |   /loudacre/accounts/part-m-00000 HadoopRDD[2] at textFile at NativeMethodAccessorImpl.java:-2 []
    |   PythonRDD[27] at RDD at PythonRDD.scala:42 []
    |   MapPartitionsRDD[25] at mapPartitions at PythonRDD.scala:338 []
    |   ShuffledRDD[24] at partitionBy at NativeMethodAccessorImpl.java:-2 []
    +-(32) PairwiseRDD[23] at reduceByKey at <ipython-input-13-6fcdc0b3517b>:1 []
       |   PythonRDD[22] at reduceByKey at <ipython-input-13-6fcdc0b3517b>:1 []
       |   /loudacre/weblogs/*6 MapPartitionsRDD[20] at textFile at NativeMethodAccessorImpl.java:-2 []
       |   /loudacre/weblogs/*6 HadoopRDD[19] at textFile at NativeMethodAccessorImpl.java:-2 []
```

2 stages

Spark Task Execution (1)

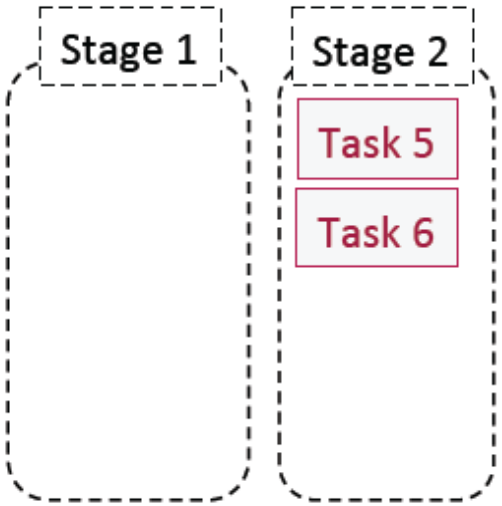
```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```

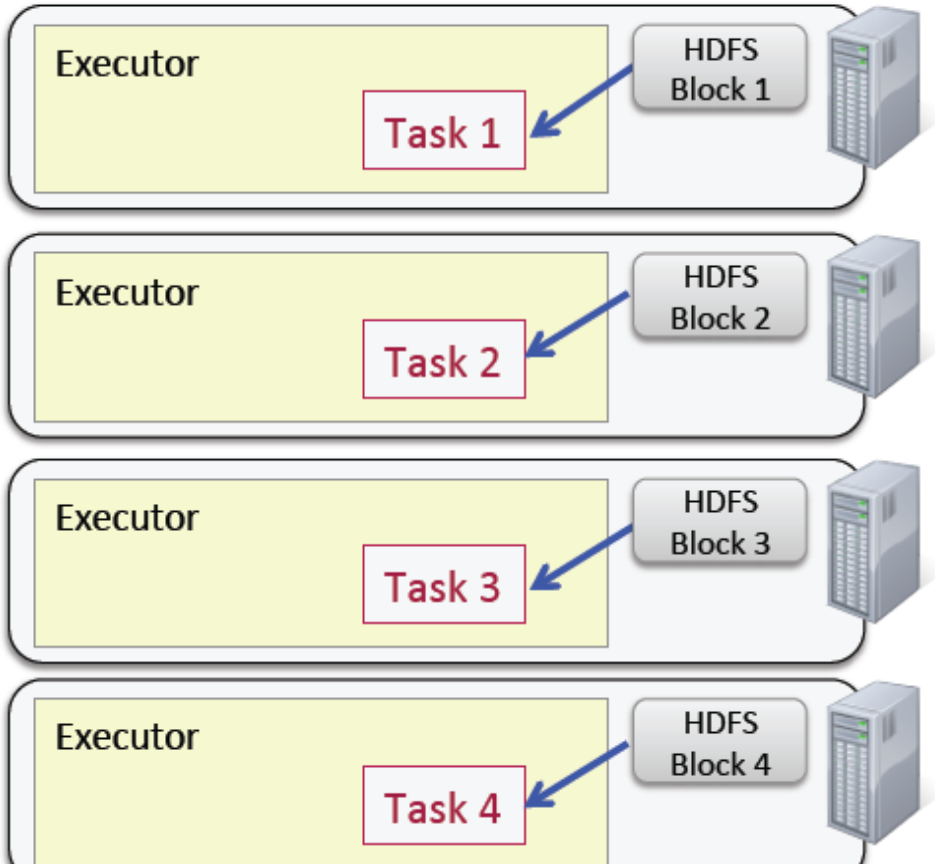




Spark Task Execution (2)

```
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")
```





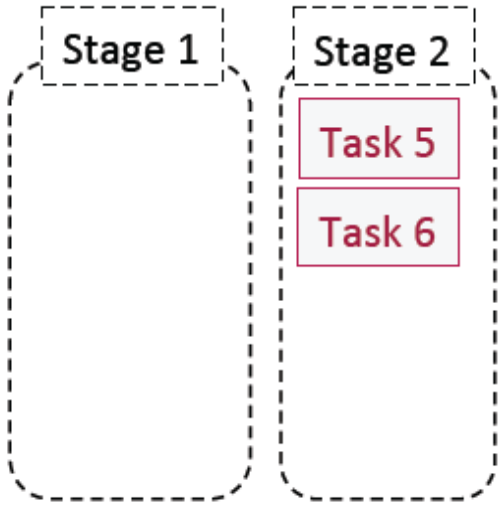
keep data locality.

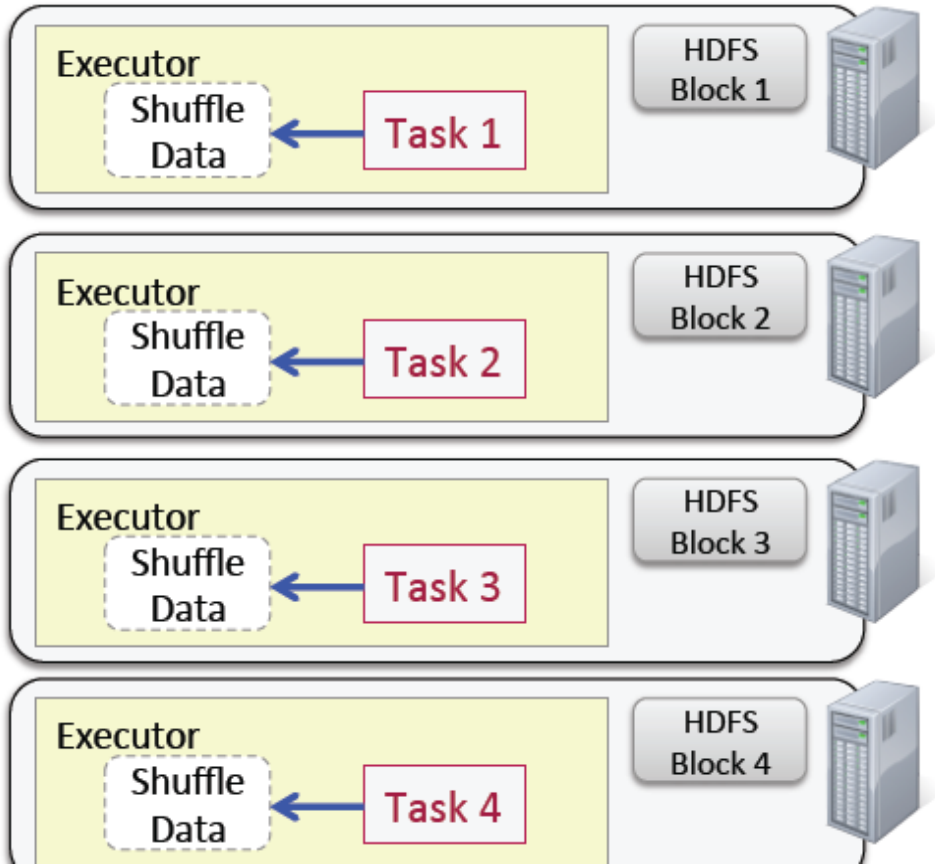
minimize network traffic

데이터 저장 중인 노드가 이미 바쁘다면 다른 노드로 작업 예약
⇒ 불가피하게 데이터 전송 발생 가능.
동적 작업 스케줄링!
bottleneck 방지.

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")
```





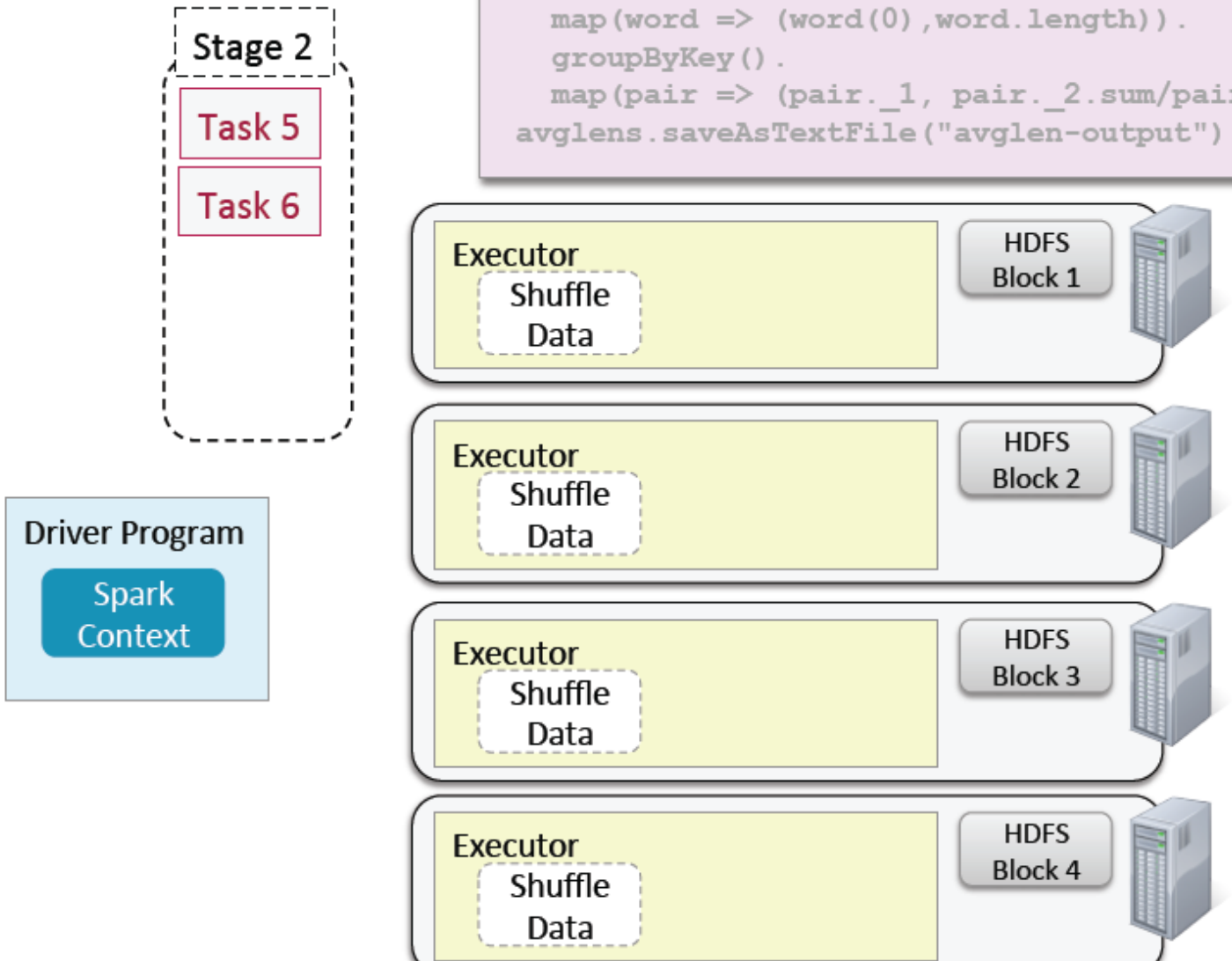
Stage 1 정도

⇒ MEM 여유분에 중간 결과 저장.
Cache로 사용가능. (fast)

⇒ MEM 없음, disk에 저장.
Stage 2가 input으로 전달. (slow)

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")
```

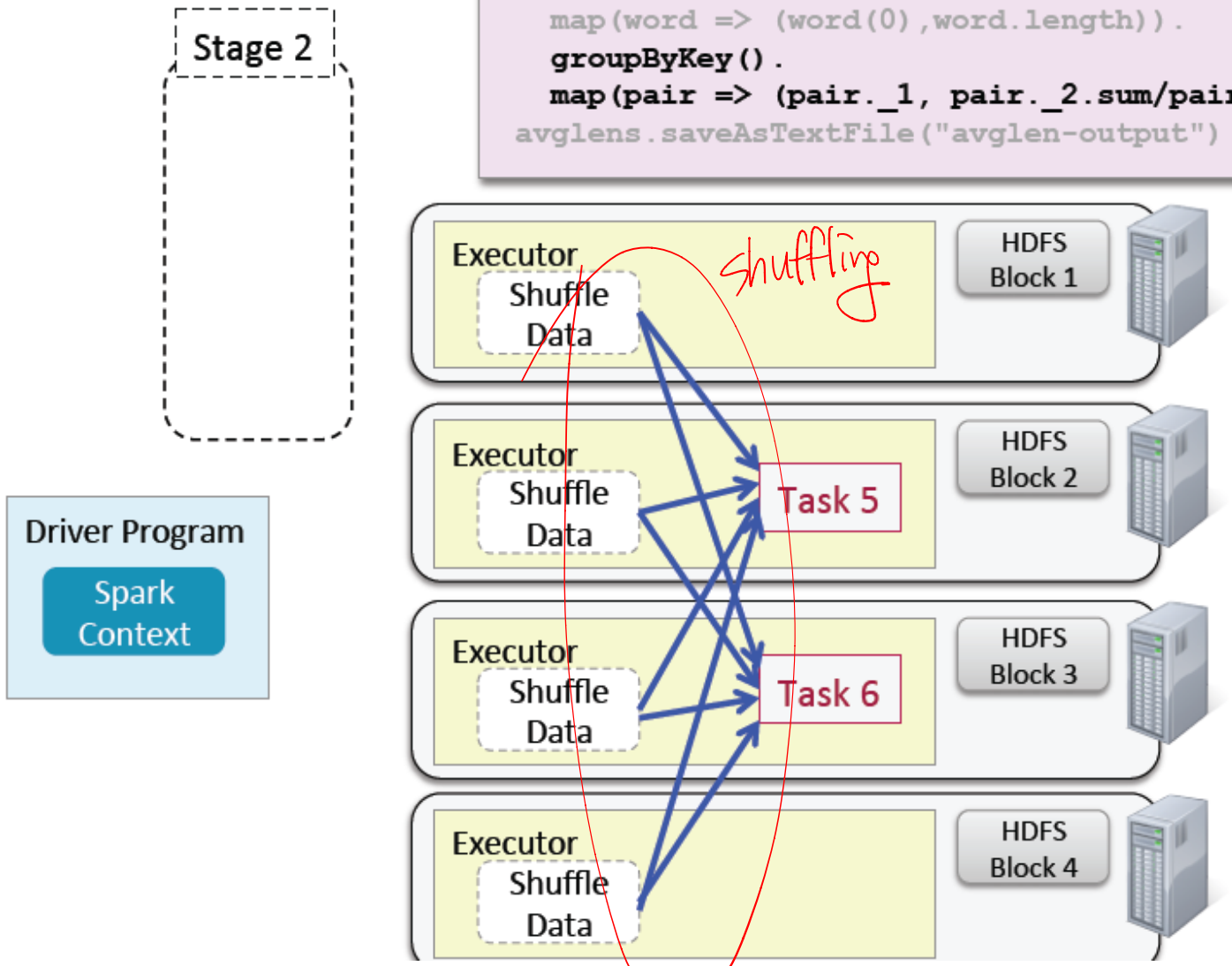


Stage 1 ended.

⇒ able to start next stage

Spark Task Execution (5)

```
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")
```



network 통한 shuffle.

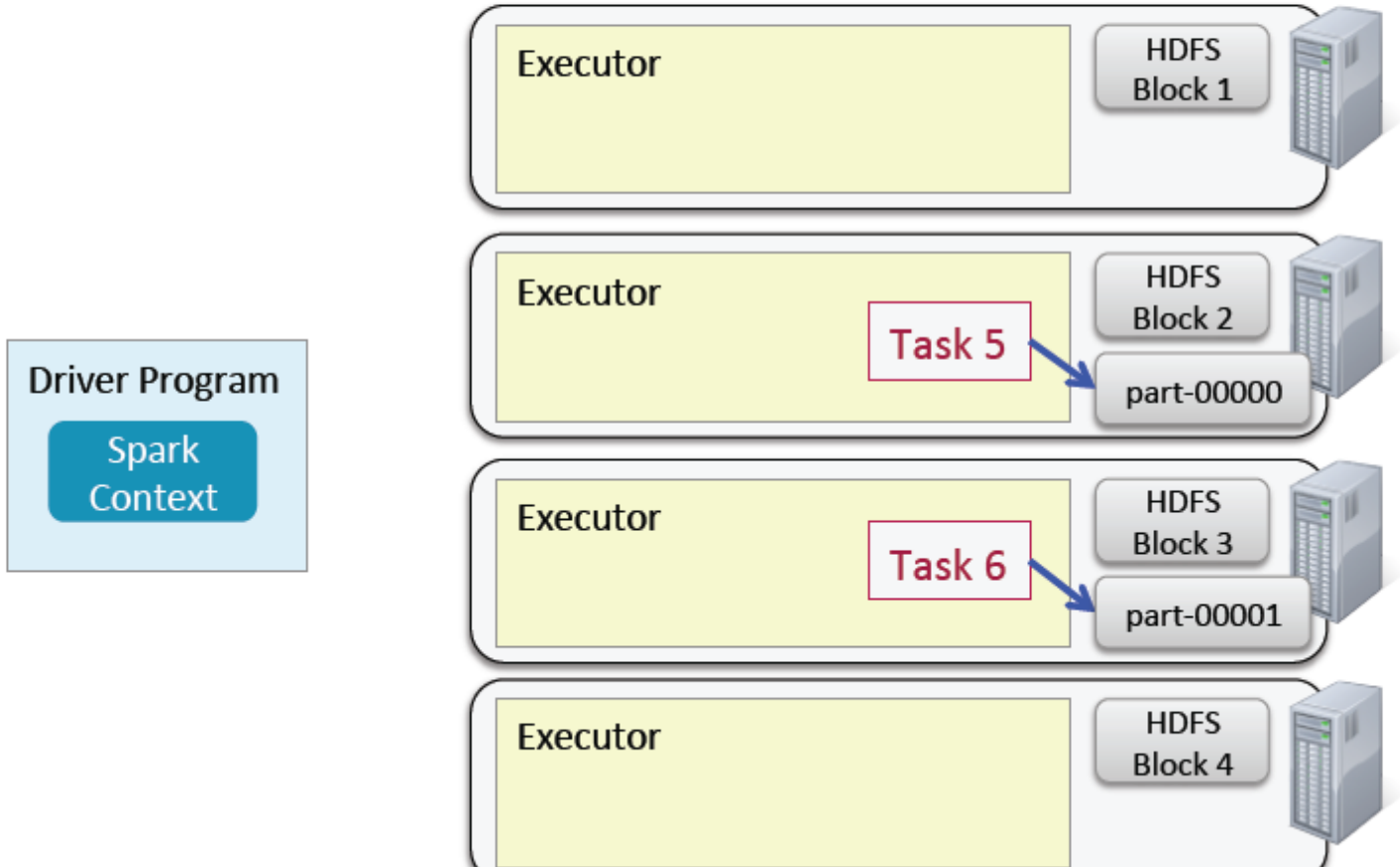
data locality 우선순.

계산 필요 노드에 적절한 데이터 전달

⇒ cost: wide > narrow

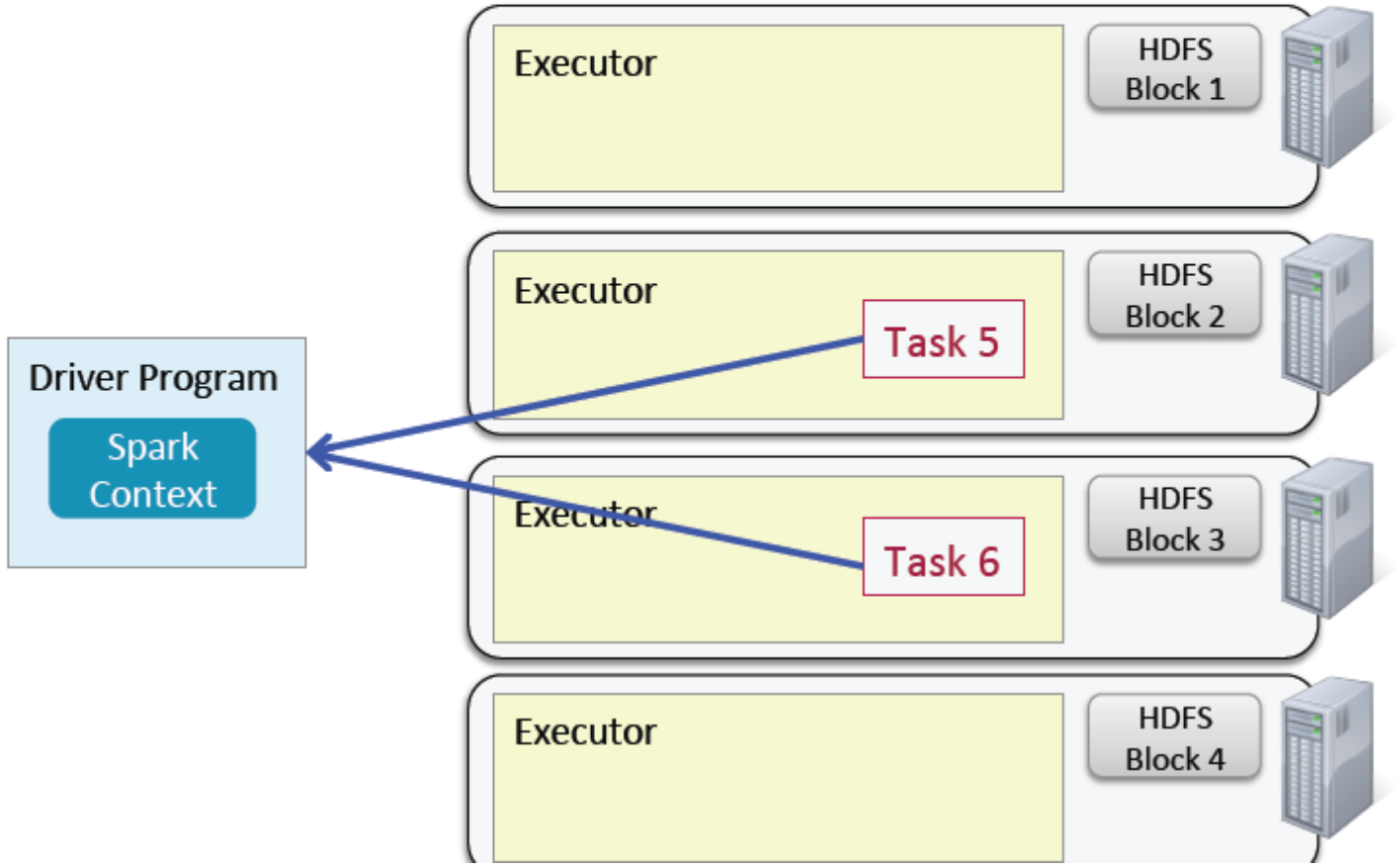
Spark Task Execution (6)

```
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")
```



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

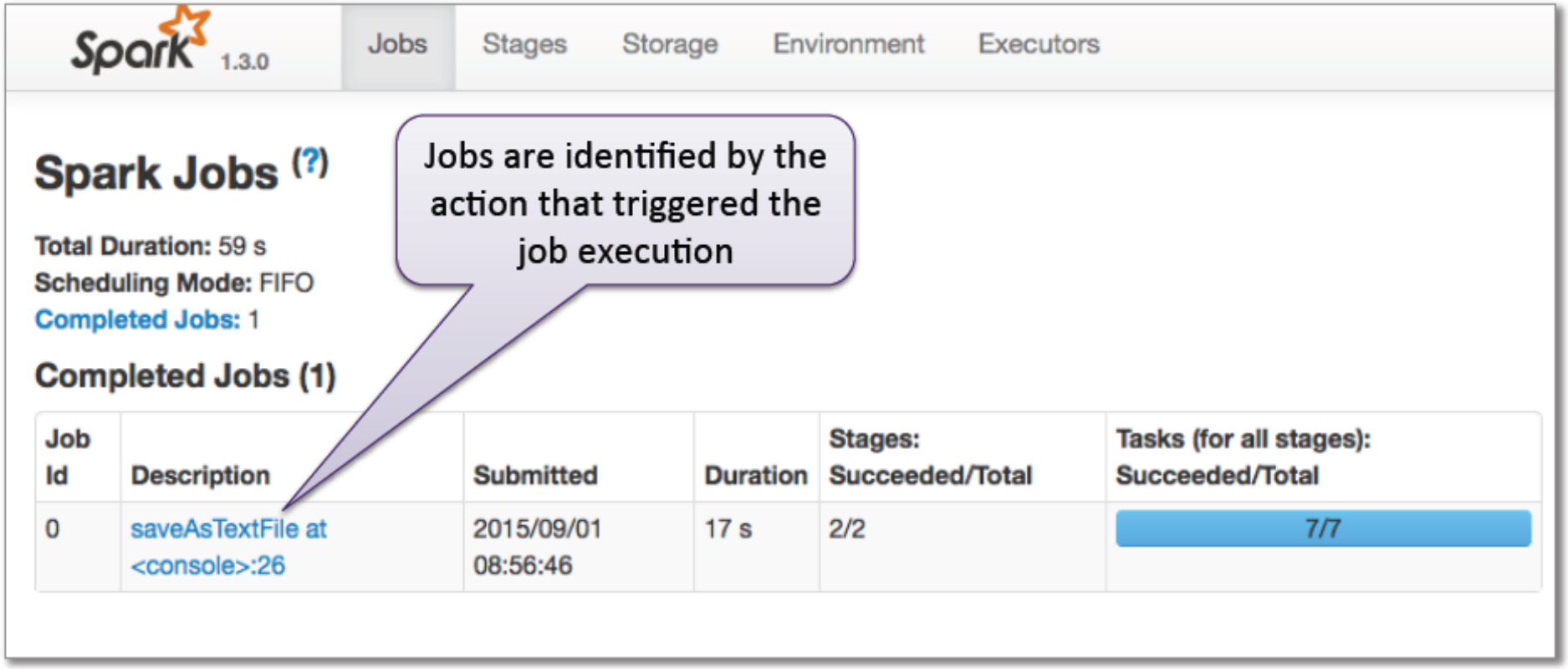
```
spark.default.parallelism      10
```

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

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

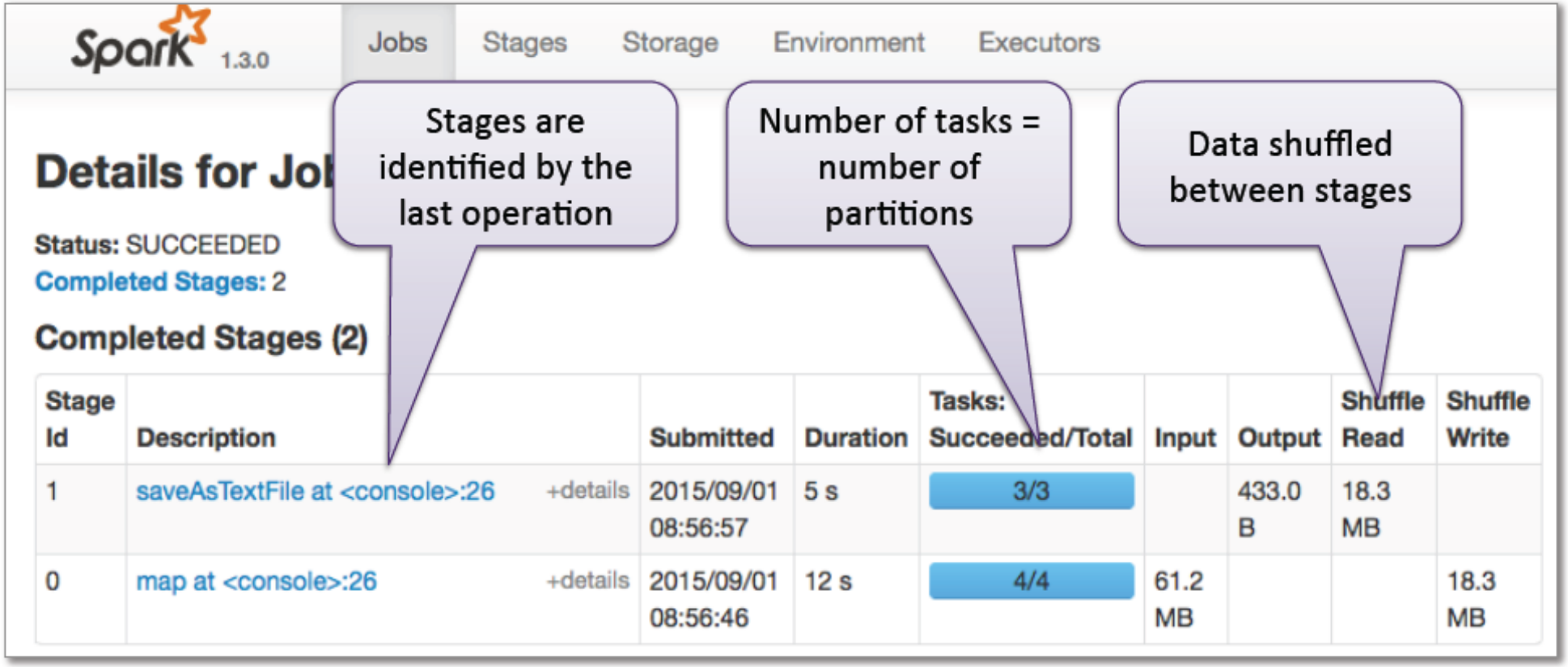
Viewing Stages in the Spark Application UI (1)

- 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`