

Cloud Computing (INFS3208)

Lecture 9: RDD Programming

Lecturer: AsPr Sen Wang

School of Electrical Engineering and Computer Science

Faculty of Engineering, Architecture and Information Technology

The University of Queensland

Re-cap

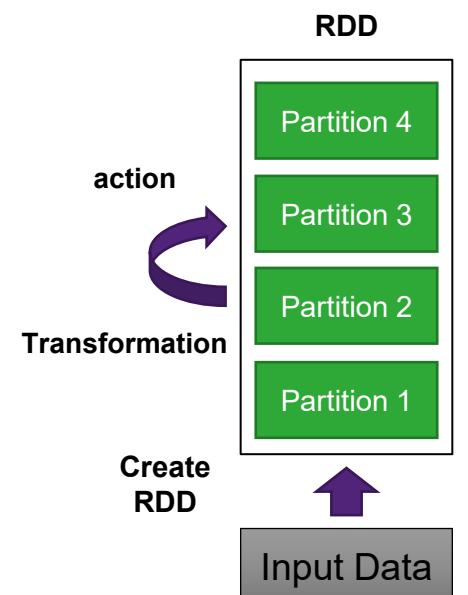
- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

Outline

- • RDD Creation: `textFile` & `parallelize` methods
- RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- Partition for Paired RDDs
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- Loading and Saving for RDD programming

Resilient Distributed Datasets (RDDs) Revisited

- RDD is a fundamental **data structure** of Spark.
- RDD is a **read-only** (i.e. immutable) **distributed** collection of objects/elements.
- Each dataset in RDD is divided into **logical partitions**, which are computed by many worker nodes (computers) in the cluster.
- RDD can be **self recovered** in case of failure (support rebuild if a partition is destroyed).
- In Spark, all work is expressed as
 - **creating** new RDDs or,
 - **transforming** existing RDDs or,
 - **action** on RDDs to compute a result.
- Unstructured data manipulation in Spark is **heavily based** on RDDs.



RDD Creation: `textFile()` method

- Two ways to create RDDs
 - load an external dataset;
 - parallelize a collection of objects (e.g., a list or set) in their driver program
- Spark uses `textFile(URI)` method load data (in filesystems) into newly created RDD
 - URI (Uniform Resource Identifier) can be from any storage source
 - your local file system (a local path on the machine),
 - HDFS (`hdfs://`),
 - Cassandra, HBase, Amazon S3, etc.

```
scala> val distFile = sc.textFile("data.txt")
distFile: org.apache.spark.rdd.RDD[String] = data.txt MapPartitionsRDD[10] at textFile at <console>:26
```

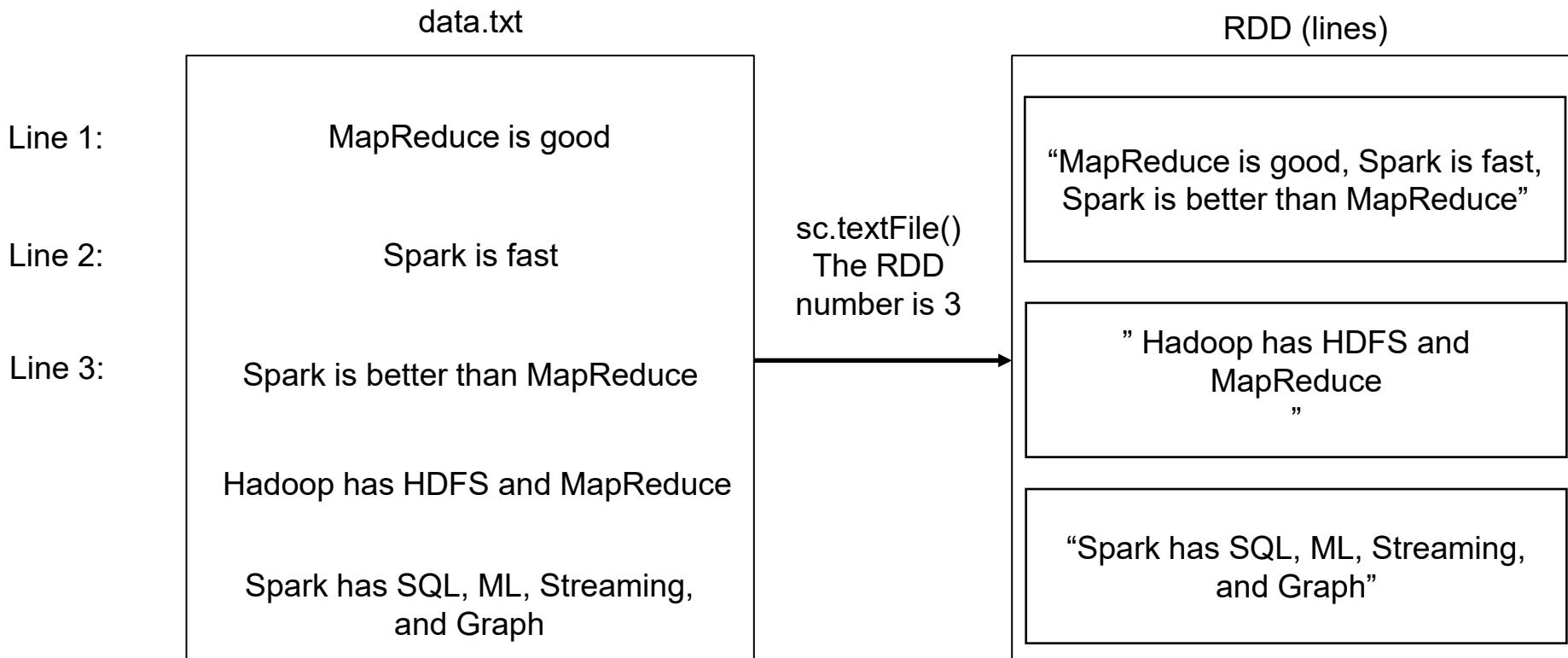
- Once created, `distFile` can be acted on by dataset operations (transformations and actions).

RDD Creation: `textFile()` method

- If using a path on the local filesystem, the file **must also be accessible** at the same path on **worker nodes**.
 - Either **copy** the file to all workers or use a network-mounted **shared file system**.
- All of Spark's file-based input methods, including `textFile`, support running on **directories**, **compressed files**, and **wildcards** as well.
 - For example, you can use:
 - `textFile("/my/directory")`, `textFile("/my/directory/*.txt")`, and `textFile("/my/directory/*.gz")`.
- The `textFile` method also takes an optional **second** argument for controlling the **number of partitions** of the file.
 - By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value.

RDD Creation: `textFile()` method

- Create an RDD from local file system:

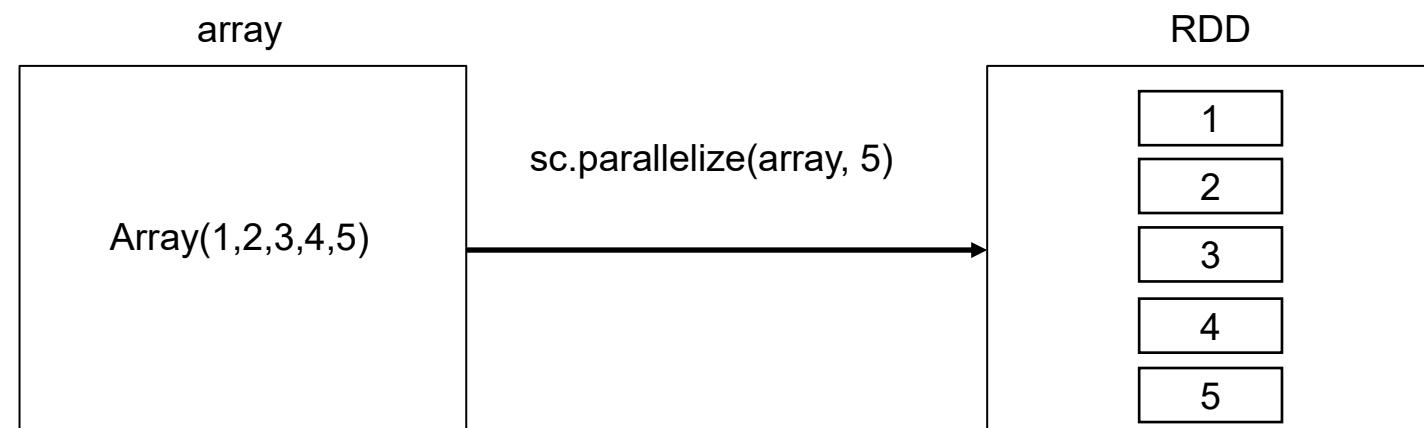


RDD Creation: parallelize () method

- Another simple way to create RDDs is to take an existing collection in your program and pass it to SparkContext's parallelize() method:

```
val data = Array(1, 2, 3, 4, 5)  
val distData = sc.parallelize(data)
```

- This approach is very useful when you are learning Spark, since you can quickly create your own RDDs in the shell and perform operations on them.



RDD Partition

Partition Policy:

- the number of partitions is equal to the CPU cores in the cluster
- For different deploy modes of Spark, i.e. local, Standalone, YARN, Mesos, set the parameter:
 - `Spark.default.parallelism`
 - Default local: the number of CPU cores or `local[N]`
 - Default Mesos: 8 partitions
 - Default Standalone or Apache YARN: `Max(2, total_cpus_in_cluster)`

Create Partition:

- Set number of partitions when [creating RDD](#)
 - `textFile(path, partitionNum)` vs `parallelize(collection, partitionNum)`

RDD Partition

Set Partition using repartition() method:

Repartition actually create a new RDD with the specified number of partitions

```
val data = sc.textFile("word.txt",2)
// this statement will output 2
data.partitions.size
// call repartition(n) method to reset the number of
partitions
rdd = data.repartition(1)
// this statement will output 1
scala> rdd.partitions.size
```

Outline

- RDD Creation: `textFile` & `parallelize` methods
- • RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- Partition for Paired RDDs
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- Loading and Saving for RDD programming

Common Transformation

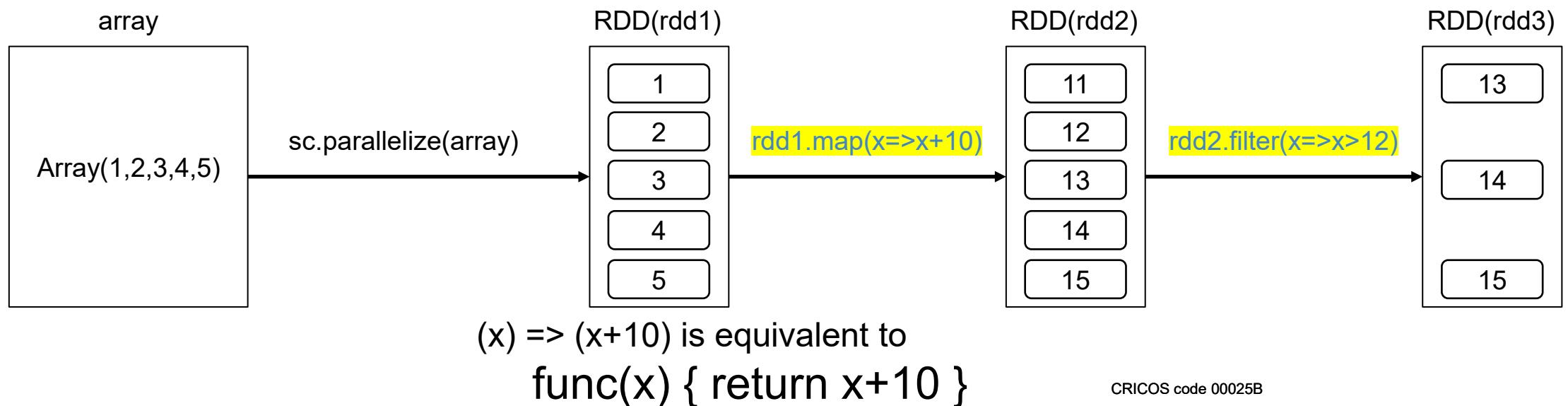
Common Transformations (15+) supported by Spark

Transformation	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
sortByKey([ascending], [numPartitions])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

RDD Operations – Transformation

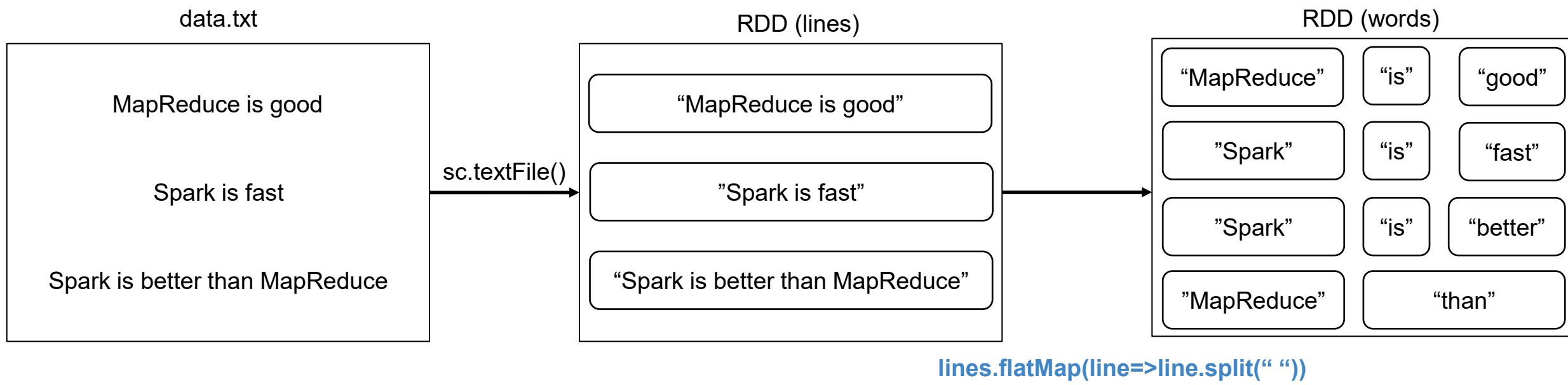
Element-wise transformation:

- **map(func)**: Return a new RDD formed by **passing each element** of the source through a function func.
- **filter(func)**: Return a new RDD formed by **selecting those elements** of the source on which func returns **true**.



RDD Operations – Transformation

- Sometimes we want to produce multiple output elements for each input element. The operation to do this is called `flatMap()`.
 - `flatMap(func)`: Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
 - A simple usage of `flatMap()` is splitting up an input `string` into `words`,



`lines.flatMap(line=>line.split(" "))`

CRICOS code 00025B

RDD Operations – Transformation

Pseudo set transformation:

- `distinct()`: produces a new RDD with only distinct items, but is expensive due to `shuffle`
- `union()`: gives back an RDD consisting of the data from both sources
 - Different from the mathematical union, if there are duplicates in the input RDDs, the result of Spark's `union()` will contain duplicates (which we can fix if desired with `distinct()`).
 - `intersection()`: returns only elements in both RDDs. `intersection()` also removes all duplicates (including duplicates from a single RDD)
 - `intersection()` and `union()` are two similar concepts,
 - the performance of `intersection()` is much worse
 - since it requires a `shuffle` over the network to identify common elements.
 - `subtract(other)` function takes in another RDD and returns an RDD that has only values present in the first RDD and not the second RDD (`shuffle`).

RDD1
{coffee, coffee,
panda, monkey, tea}

RDD2
{coffee, monkey,
kitty}

RDD1.distinct()
{coffee, panda,
monkey, tea}

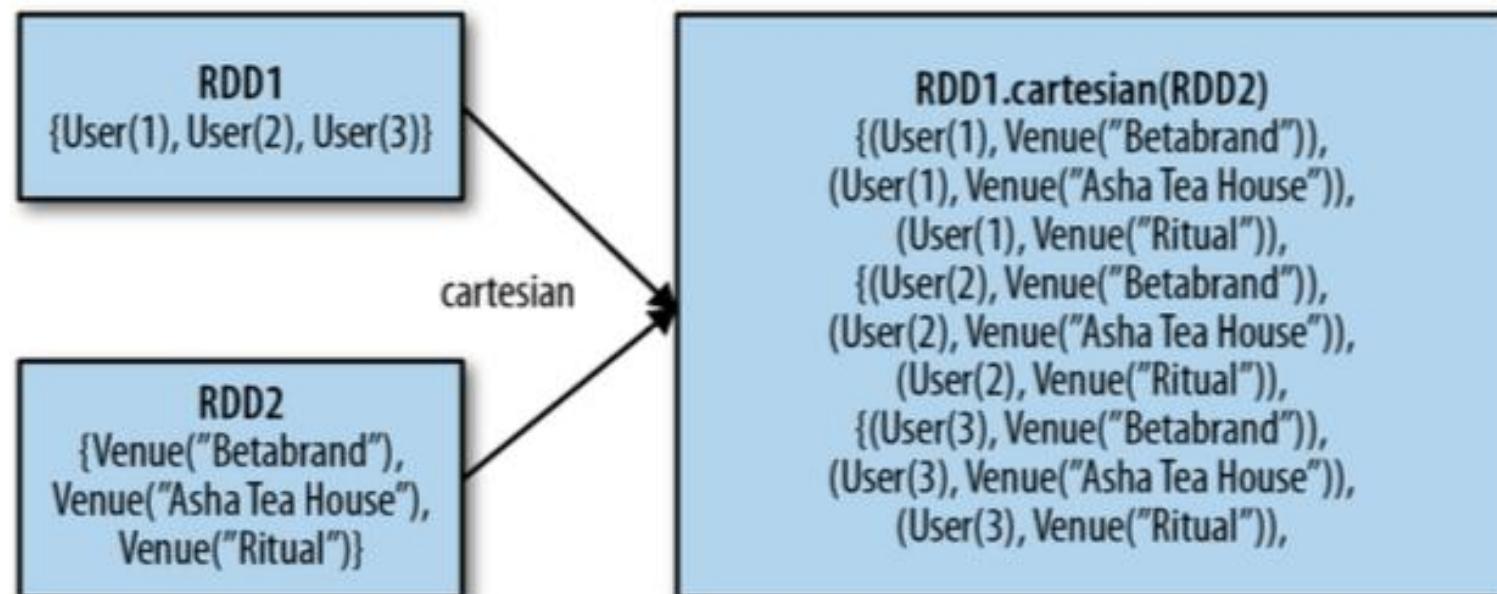
RDD1.union(RDD2)
{coffee, coffee,
coffee, panda,
monkey, monkey,
tea, kitty}

RDD1.intersection(RDD2)
{coffee, monkey}

RDD1.subtract(RDD2)
{panda, tea}

RDD Operations – Transformation

- `cartesian(other)` transformation:
 - returns **all possible** pairs of (a, b)
 - **a** is in the source RDD and **b** is in the other RDD.
 - similar to `cross join` in SQL
 - The Cartesian product can be **useful** but is **very expensive** for large RDDs.



Outline

- RDD Creation: `textFile` & `parallelize` methods
- RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- • RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- Partition for Paired RDDs
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- Loading and Saving for RDD programming

RDD Operations – Action

Common Actions (10+) supported by Spark

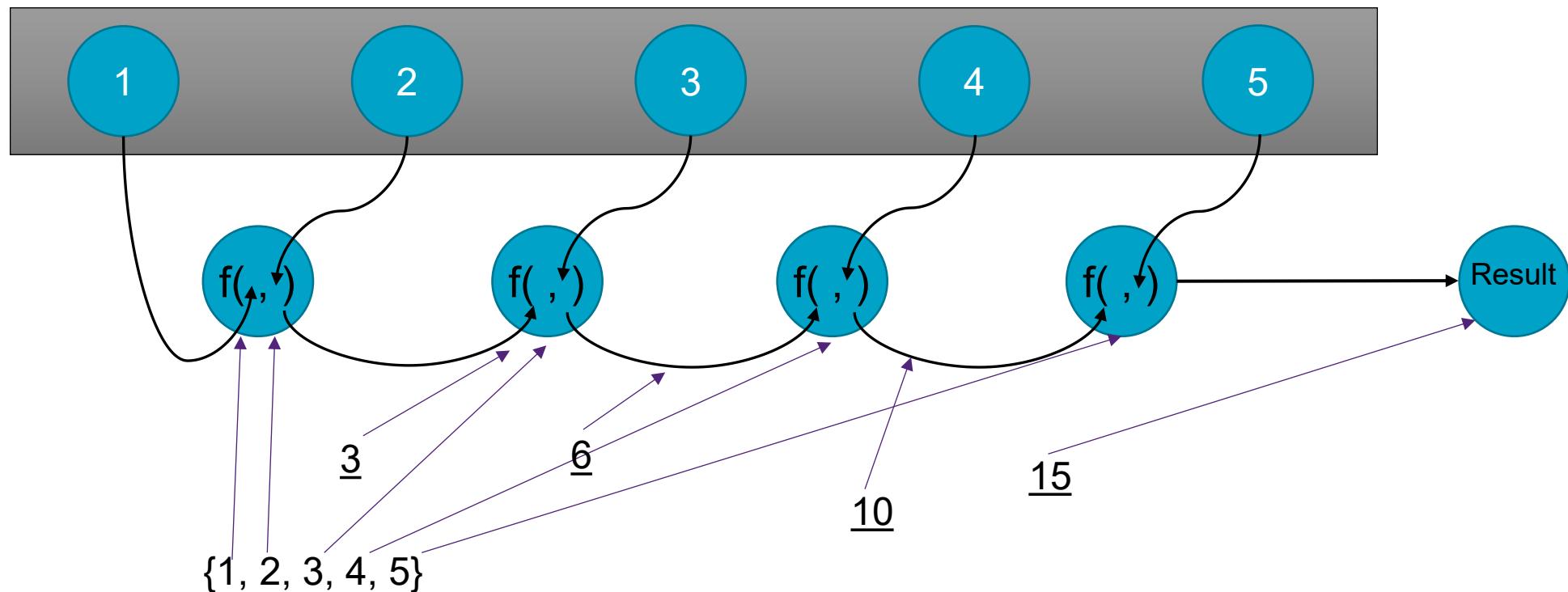
Action	Meaning
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
reduce(<i>func</i>)	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
first()	Return the first element of the dataset (similar to take(1)).
take(<i>n</i>)	Return an array with the first <i>n</i> elements of the dataset.
foreach(<i>func</i>)	Run a function <i>func</i> on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.

RDD Operations – Action

- `collect()` action:
 - returns the `entire` RDD's contents to our driver program
 - When RDDs are distributed over worker nodes, to display the correct result, `collect()` is used to gather all data into one node
- `reduce(func)` action (the most common action in RDD programming):
 - takes a function (`func`) that operates on `two elements` of the type in your RDD
 - returns `a new` element of the `same type`.
 - Example: `val sum = rdd.reduce((x, y) => x + y)`

RDD Operations – Action

- Given an RDD $rdd = \{1, 2, 3, 4, 5\}$
- `reduce(func): val sum = rdd.reduce((x, y) => x + y)`



RDD Operations – Action

- `take(n)`:
 - returns `n` elements from the RDD and attempts to minimize the number of accessed partitions
 - it may represent a `biased` collection (may not return the elements in the expected order).
 - `Useful` for unit tests and quick debugging.
- `top()`: extracts the `top` elements from an RDD (an ordering defined).
- `takeSample(withReplacement, num, seed)`: allows us to take a sample of our data.
- `foreach(func)`: performs computations on each element in the RDD.
- `count()`: returns a count of the elements.
- `countByValue()` returns a map of each unique value to its count.

RDD Operations – Action

Examples: given a rdd containing {1,2,3,3}

- `rdd.reduce((x,y) => x+y) ->`
- `rdd.collect ->`
- `rdd.count() ->`
- `rdd.take(2) ->`
- `rdd.top(2) ->`
- `rdd.countByValue() ->`
- `rdd.foreach(println) ->`



Outputs of the actions

- 9
- {1,2,3,3}
- 4
- {1,2}
- {3,3}
- {(1,1),(2,1),(3,2)}
- 1, 2, 3, 3

Outline

- RDD Creation: `textFile` & `parallelize` methods
- RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- • RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- Partition for Paired RDDs
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- Loading and Saving for RDD programming

RDD Persistence and Caching

- Spark will recompute the RDD and all of its dependencies each time.
 - Long chain of transformations
 - Specific expensive transformations in the chain
- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations.
- This in-memory re-usage allows future actions to be much faster (often by more than 10x).
 - Caching is a key tool for iterative algorithms and fast interactive use.
 - `persist(level)` or `cache()` methods
- If too much data to be cached to fit in memory, Spark will automatically evict old partitions using a Least Recently Used (LRU) cache policy.
- RDDs come with a method called `unpersist()` that lets you manually remove them from the cache

RDD Persistence and Caching cont'd

```
val numbersRDD = sc.parallelize(List(1, 2, 3, 4, 5))

val processedRDD = numbersRDD.map { x =>
  println(s"Processing element: $x")
  x * 10
}

processedRDD.cache()

println("--- Running FIRST action (reduce) ---")

val product = processedRDD.reduce((a, b) => a * b)
println(s"Product of elements is: $product")

println("\n--- Running SECOND action (count) ---")

val totalCount = processedRDD.count()
println(s"Total count of elements is: $totalCount")
```

- Create a simple RDD with some toy data.
- Define a transformation and add a `println` to see when this code is executed.
- Persist the transformed RDD in memory using `.cache()`, which is a shorthand for `.persist(StorageLevel.MEMORY_ONLY)`

Q1: Why printing "running first action" is displayed before "processing element"??

Q2: Will you see "processing" when run a second action on the SAME RDD??

Shared Variables - Broadcast Variables

- A broadcast variable is a **read-only** variable that is cached on each worker node instead of being shipped with every single task.
- When tasks across multiple stages need the **same** large dataset (e.g., **a lookup map**, a **machine learning model**), it's far more efficient to send it to each executor once. This reduces network overhead significantly.
- Broadcasting Steps:
 - The driver serialises the variable and sends it to each executor.
 - Executors store it in memory.
 - Tasks running on that executor can access the variable locally using the `.value` method.

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast@0

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

Shared Variables - Broadcast Variables

```
val salesRDD = sc.parallelize(Seq(  
    (101, 1, 89.99),  
    (102, 2, 14.50),  
    (103, 1, 89.99),  
    (104, 3, 200.10),  
    (105, 2, 14.50)  
)  
  
val productNames = Map(  
    1 -> "Premium Laptop",  
    2 -> "Wireless Mouse",  
    3 -> "Mechanical Keyboard"  
)  
  
val broadcastedProductNames = sc.broadcast(productNames)  
  
val salesWithNamesRDD = salesRDD.map { case (transactionId, productId, amount) =>  
  
    val productName = broadcastedProductNames.value.getOrElse(productId, "Unknown Product")  
    (transactionId, productName, amount)  
}  
  
salesWithNamesRDD.collect().foreach(println)
```

- Create a RDD to store your large, distributed dataset of sales records: (***transactionId, productId, amount***)
- Create a RDD to store your smaller, static lookup table: **Map[productId, productName]**
- Broadcast the lookup table to every worker node. This happens only once per executor.
- Use the broadcasted variable inside a map transformation.
- Each task on each worker can now access the full map locally.
- Use **.value** to get the map from the broadcast variable
- Action to trigger the job and display results

Shared Variables - Accumulators

- An accumulator is a variable that can only be "**added**" to using an associative and commutative operation. They are used to perform **counters** or **sums** in parallel.
- Workers can **increment** an accumulator, but they **cannot read** its value.
- Only the **driver program can read** the final aggregated value of the accumulator.
- This ensures that updates are performed **efficiently and safely** without race conditions.
- Accumulators are perfect for tracking metrics during a job.
 - For example, you can count how many records were malformed or how many times a specific event occurred.

```
1 // Create a long accumulator, initializing it to 0
2 val corruptedLines = sc.longAccumulator("CorruptedLinesCounter")
3
4 val data = sc.textFile("path/to/logs.txt")
5
6 // Increment the accumulator inside a transformation
7 val cleanedData = data.filter { line =>
8   if (isCorrupted(line)) {
9     corruptedLines.add(1) // Increment the counter
10    false
11  } else {
12    true
13  }
14 }
15
16 // An action is needed to trigger the calculation
17 cleanedData.count()
18
19 // Get the final value on the driver
20 println(s"Number of corrupted lines: ${corruptedLines.value}")
```

Outline

- RDD Creation: `textFile` & `parallelize` methods
- RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- • Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- RDD Partition
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- Loading and Saving for RDD programming

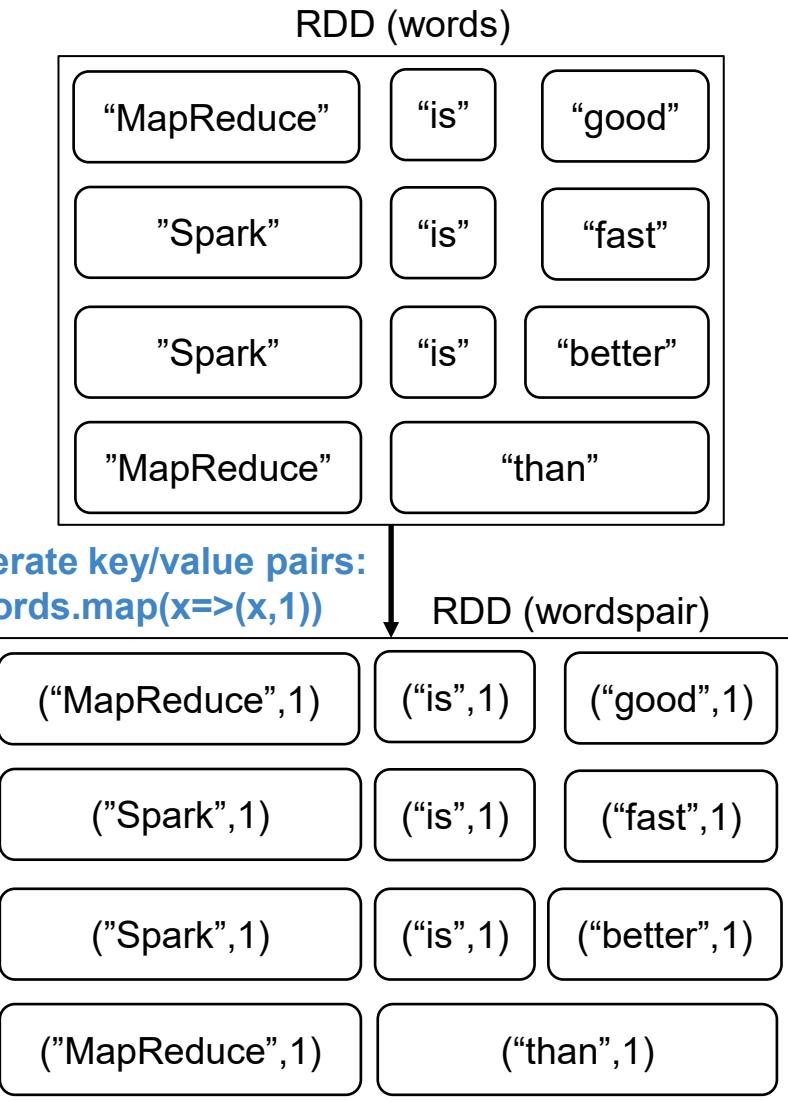
Working with Key/Value Pairs

- RDDs of key/value pairs are a [common data type](#) required for many operations in Spark.
 - (“Spark”, 2), (“is”, 3), (“is”, (1,1,1)), etc.
- Key/value RDDs are commonly used to perform [aggregations](#).
 - counting up reviews for each product,
 - grouping together data with the same key,
 - grouping together two different RDDs.
- Spark provides special operations on RDDs containing key/value pairs.
 - [reduceByKey\(\)](#) method can aggregate data separately for each key
 - [join\(\)](#) method can merge two RDDs together by grouping elements with the same key
- ._1 & ._2 stand for key and value, respectively.
 - E.g. ._1 stands for “Spark” or “is”, while ._2 stands for 2 or 3.

Creating Pair RDDs

Use map() method:

- `map(x => (x, 1))`
 - `x` is each element in RDD words
 - `=>` (right arrow or fat arrow): separate parameters of a function and function body
 - E.g. `(x,y) => (x+y)` is equivalent to
`func(x, y) { return x+y }`
 - `x => (x, 1)` is thus equivalent to
`func(x) {return (x, 1)}` a pair of key/value
 - In this way, you can conveniently write anonymous functions

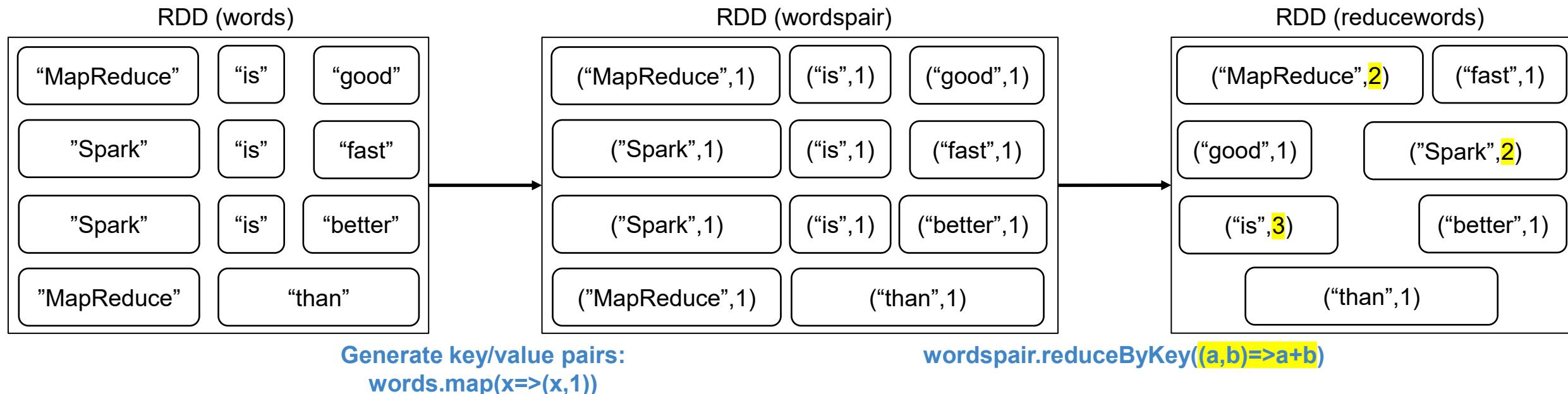


Transformations on Pair RDDs

- `reduceByKey(func, [numTasks])` transformation:
 - is quite similar to `reduce()`: both take a function and use it to combine values.
 - runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key.
 - different from `reduce()`, `reduceByKey()` is not implemented as an action that returns a value to the user program.
 - returns a new RDD consisting of each `key` and the `reduced value` for that key.
- Example:
 - Given grades of all the courses in EECS, calculate the averaged GPA for each student;
 - Given a text, count each word's occurrence.

RDD Operations – Transformation

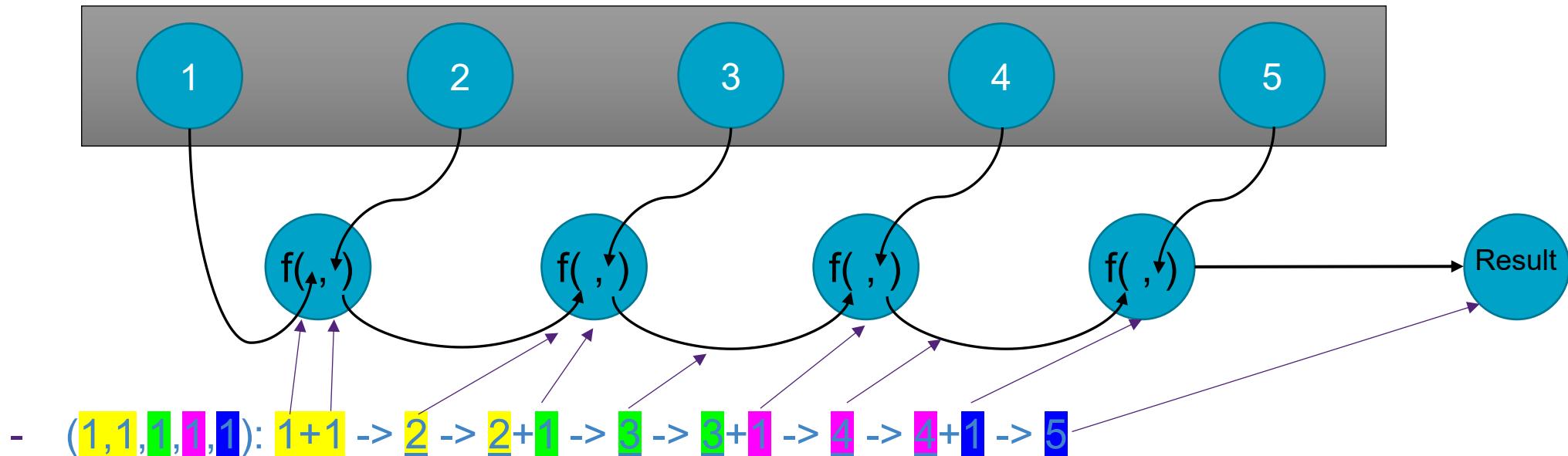
- **reduceByKey(func)**: called on a dataset of (K, V) pairs,
- returns a dataset of (K, **V**) pairs where the values for each key are **aggregated** using the given reduce function **func**,
- func must be of type **(V,V) => V**, e.g. **(a,b) => a+b**.



RDD Operations – Transformation

- `reduceByKey(func):`
 - `wordspair.reduceByKey((a,b)=>a+b) on {("is", 1), ("is", 1), ("is", 1), ("is", 1), ("is", 1)}`

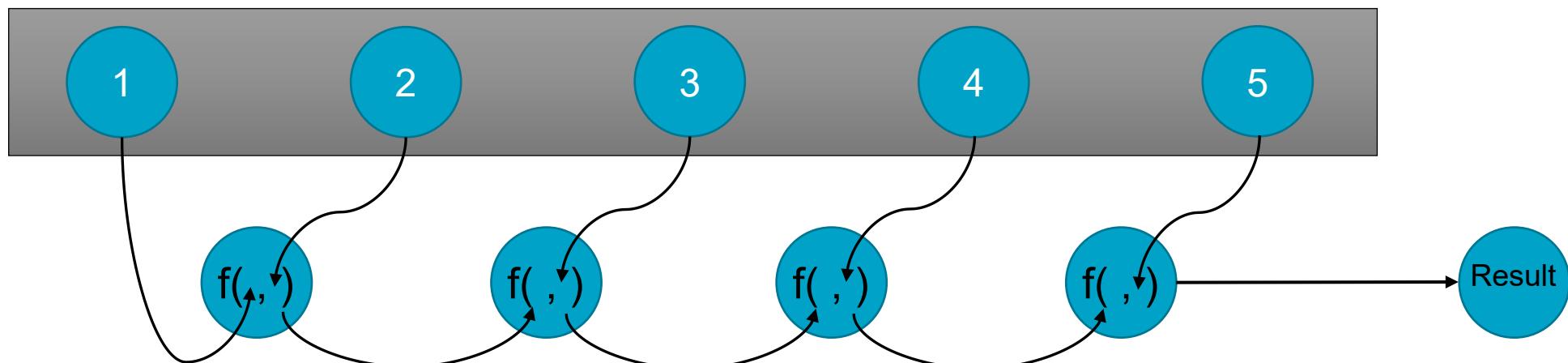
$f(a, b) \{ \text{return } a + b \}$



Transformations on Pair RDDs

- Elements 1, 2, 3, 4, and 5 should have the same key so that the value can be aggregated with the func:
 - {("s123456", 78), ("s123456", 80), ("s123456", 65), ("s123456", 90),
 - ("s654321", 80), ("s654321", 40) , ("s654321", 50) , ("s654321", 90) , ("s654321", 80) }
 - For the key "s123456", 78, 80, 65, 90 will be aggregated with func
 - For the key "s654321", 80, 40, 50, 90, 80 will be aggregated with func

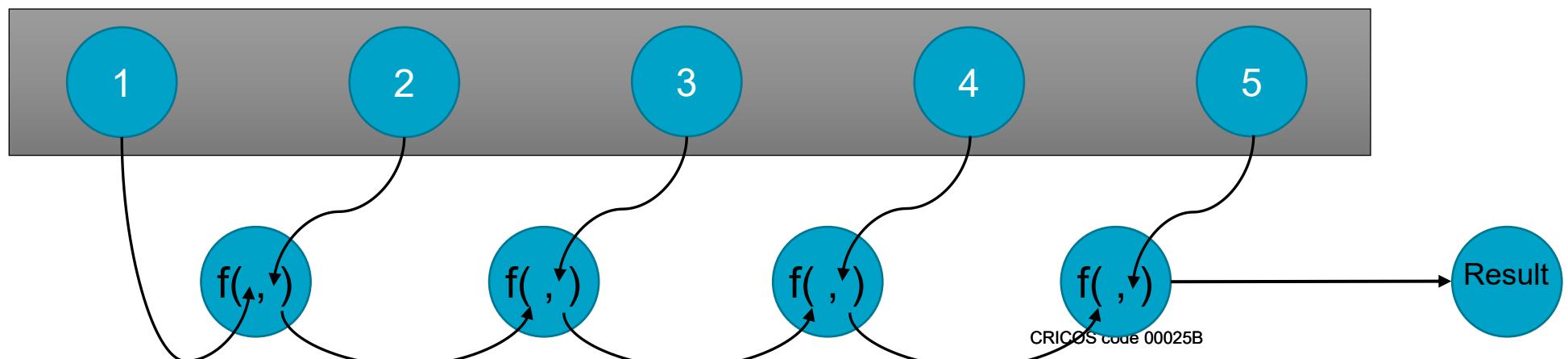
$f(a, b) \{ \text{return } a + b \}$



Transformations on Pair RDDs

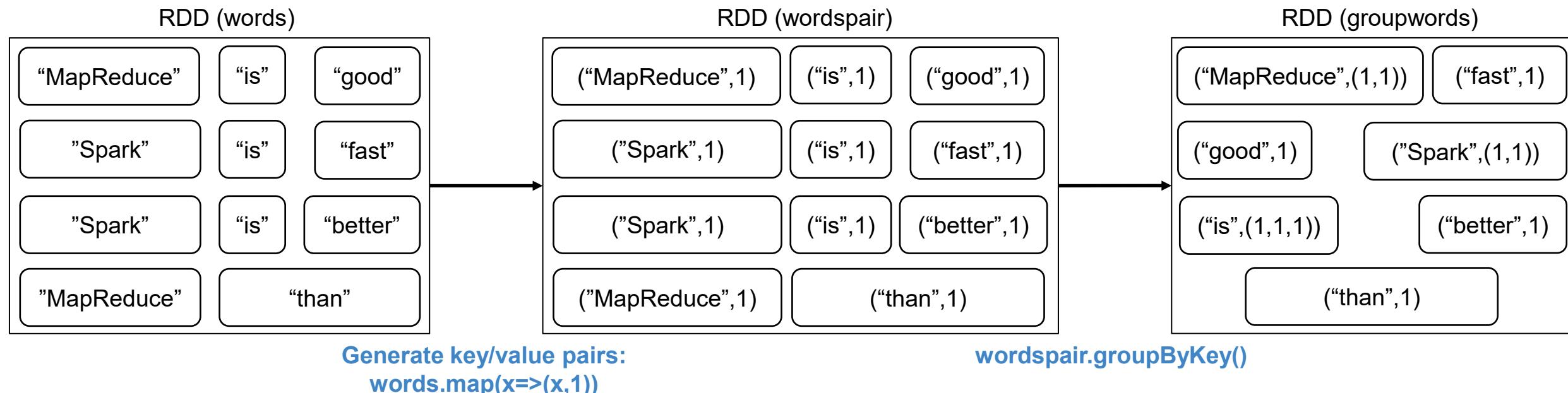
`reduceByKey(func)` vs `reduce(func)`:

- `reduce(func)` and `reduceByKey(func)` are aggregating operations
- `reduceByKey(func)` is a transformation on paired RDD while `reduce(func)` is an action on regular RDD.
- `reduce(func)` can perform on non-pair RDDs
- `reduceByKey(func)` needs to match key first
- The aggregation function (`func`) works very similar:



RDD Operations – Transformation

- `groupByKey()`: called on a dataset of (K, V) pairs,
- returns a dataset of (K, `Iterable<V>`) pairs
- E.g. (“is”,(1,1,1)) pair, (1,1,1) is `Iterable`.



Transformations on Pair RDDs

Differences between `reduceByKey(func)` and `groupByKey()`

- `groupByKey()` has no function, while `reduceByKey(func)` has a reduce function
- Returns are different:
 - `reduceByKey(func)` returns a pair RDD of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*.
 - E.g. {("MapReduce",2), ("Spark",2), ("is",3), ("better",1), ("than",1), ("fast",1), ("good",1)}
 - `groupByKey()` returns a pair RDD of (K, Iterable<V>) (still key/value pairs)
 - E.g. {("MapReduce", (1,1)), ("Spark", (1,1)), ("is", (1,1,1)), ("better",1), ("than",1), ("fast",1), ("good",1)}
- `groupByKey()` aims to group RDD according to a key and results in data shuffling when RDD is not already partitioned.
- `reduceByKey(func)` aims to perform grouping + aggregation. `reduceByKey()` is equivalent to `groupByKey().reduce(func)`.

Transformations on Pair RDDs

.keys and .values transformations

- Sometimes, you may want to return `keys` in a Key/Value RDD
- `.keys` will return a new RDD that contains all keys in the previous RDD.
- Example: `rdd.keys`
- Sometimes, you may want to return `values` in a Key/Value RDD
- `.values` will return a new RDD that contains all values in the previous RDD.
- Example: `rdd.values`

```
rddPairS.keys.collect.foreach(println)
```

MapReduce
is
good
Spark
is
fast
Spark
is
better
than
MapReduce

```
rddPairS.values.collect.foreach(println)
```

1
1
1
1
1
1
1
1
1
1
1

Transformations on Pair RDDs

sortByKey(bool):

- called on a dataset of (K, V) pairs where K implements Ordered,
- returns a dataset of (K, V) pairs sorted by keys in ascending (default) or descending order
- specify false for descending order.
- Example:
 - {("MapReduce",2), ("Spark",2), ("is",3), ("better",1), ("than",1), ("fast",1), ("good",1)}
 - `sortByKey()`:
 { ("MapReduce",2), ("Spark",2), ("better",1), ("fast",1), ("good",1), ("is",3), ("than",1) }
 - `sortByKey(false)`:
 { ("than",1), ("is",3), ("good",1), ("fast",1), ("better",1), ("Spark",2), ("MapReduce", 2) }

Transformations on Pair RDDs

mapValues(func):

- apply func to each values without changing keys
- Example:

```
{("MapReduce",2), ("Spark",2), ("is",3), ("better",1), ("than",1), ("fast",1), ("good",1)}
```

```
pairRDD.mapValues(x => x+1).foreach(println)
```

```
(“MapReduce”,3)
```

```
(“Spark”, 3)
```

```
(“is”,4)
```

```
(“better”, 2)
```

```
(“than”, 2)
```

```
(“fast”,2)
```

```
(“good”,2)}
```

Transformations on Pair RDDs

join()

- called on datasets of type (K, V) and (K, W) ,
- returns a dataset of $(K, (V, W))$ pairs with all pairs of elements for each key.
- Example:
- Given datasets
 - student(sNumber: String, prgName: String), infs3208(sNumber:String, fMark: Double)
 - Join code: student.join(infs3208).

Paired RDDs Transformation Examples

Given one pair RDD: { (1,2), (3,4), (3,6) }

Function Name	Purpose	Example	Result
reduceByKey(func)	Combine values with the same key.	rdd.reduceByKey($(x,y) \Rightarrow x+y$)	{(1,2), (3,10)}
groupByKey()	Group values with the same key.	rdd.groupByKey()	{(1,[2]),(3,[4,6])}
keys()	Return an RDD of just the keys.	rdd.keys()	{1, 3, 3}
values()	Return an RDD of just the values.	rdd.values()	{2, 4, 6}
mapValues(func)	Apply a function to each value of a pair RDD without changing the key.	rdd.mapValues($x \Rightarrow x + 1$)	{(1,3), (3,5), (3,7)}
sortByKey()	Return an RDD sorted by the key.	rdd.sortByKey()	{(1,2), (3,4), (3,6)}

Paired RDDs Action Examples

Given one pair RDD: { (1,2), (3,4), (3,6) }

Function Name	Purpose	Example	Result
countByKey()	Count the number of elements for each key.	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	Collect the result as a map to provide easy lookup.	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	Return all values associated with the provided key.	rdd.lookup(3)	[4, 6]

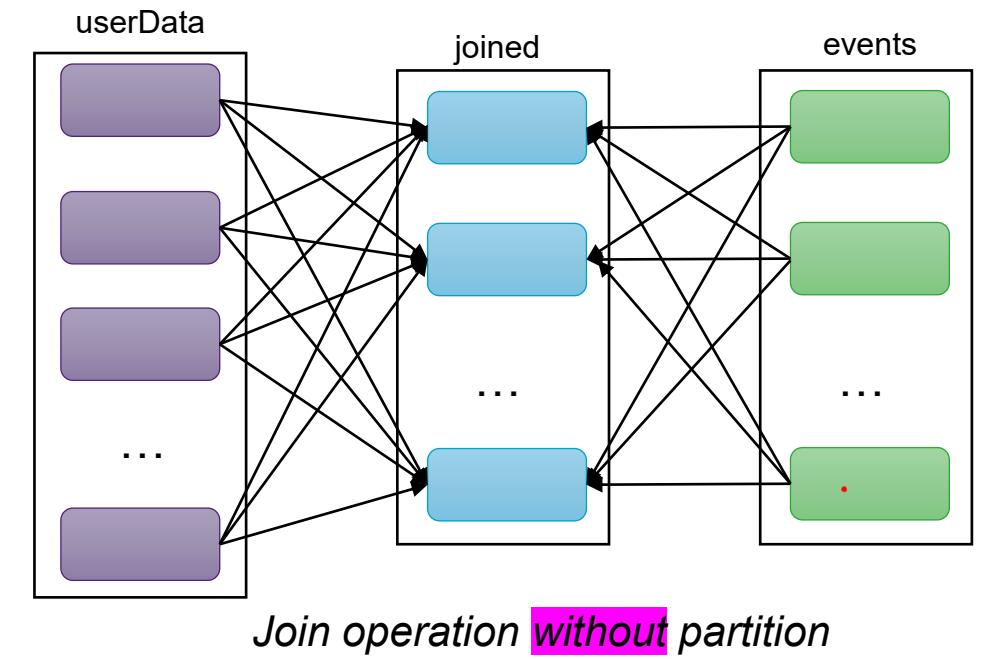
Outline

- RDD Creation: `textFile` & `parallelize` methods
- RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- • Partition for Paired RDDs
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- Loading and Saving for RDD programming

PartitionBy for Paired RDD

- Normally, in a distributed program, RDDs are very big and need to be partitioned across worker nodes, which result in very expensive communications.
- Given UserData (UserId, UserInfo) and Events(UserId, LinkInfo), to join two RDDs (UserId, UserInfo, LinkInfo)

```
def processNewLogs(logFileName: String) {  
    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)  
    val joined = userData.join(events)  
    // RDD of (UserID, (UserInfo, LinkInfo)) pairs  
    val offTopicVisits = joined.filter {  
        case (userId, (userInfo, linkInfo)) => // Expand  
            the tuple into its components  
            !userInfo.topics.contains(linkInfo.topic)  
        }.count()  
    println("Number of visits to non-subscribed  
topics: " + offTopicVisits)  
}
```



(UserID, UserInfo)

(UserID, LinkInfo)

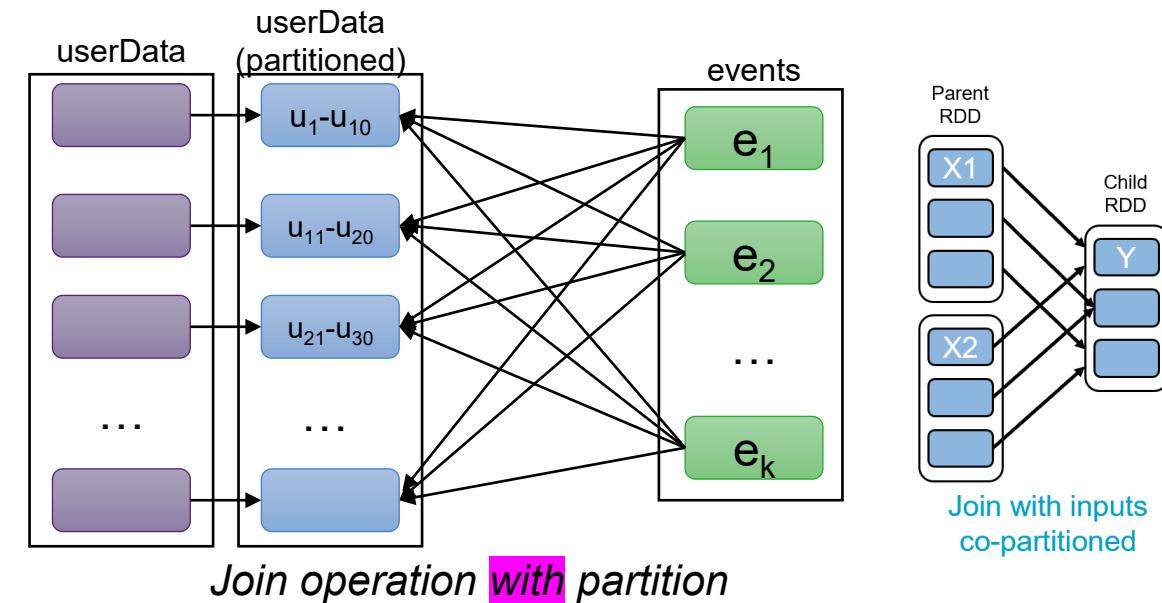
PartitionBy for Paired RDD

- To reduce the cost of communication:
 - use the `partitionBy()` transformation on `userData` to hash-partition it at the start of the program.
 - `spark.HashPartitioner` object to `partitionBy`

```
val sc = new SparkContext(...)  
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://  
...").partitionBy(new HashPartitioner(100))  
// Create 100 partitions .persist()
```

- The result is that a **lot less** data is communicated over the network, and the program runs **significantly faster**.
- Thus, laying out data to minimize network traffic can **greatly improve** performance.

Note: Repartition() vs PartitionBy()



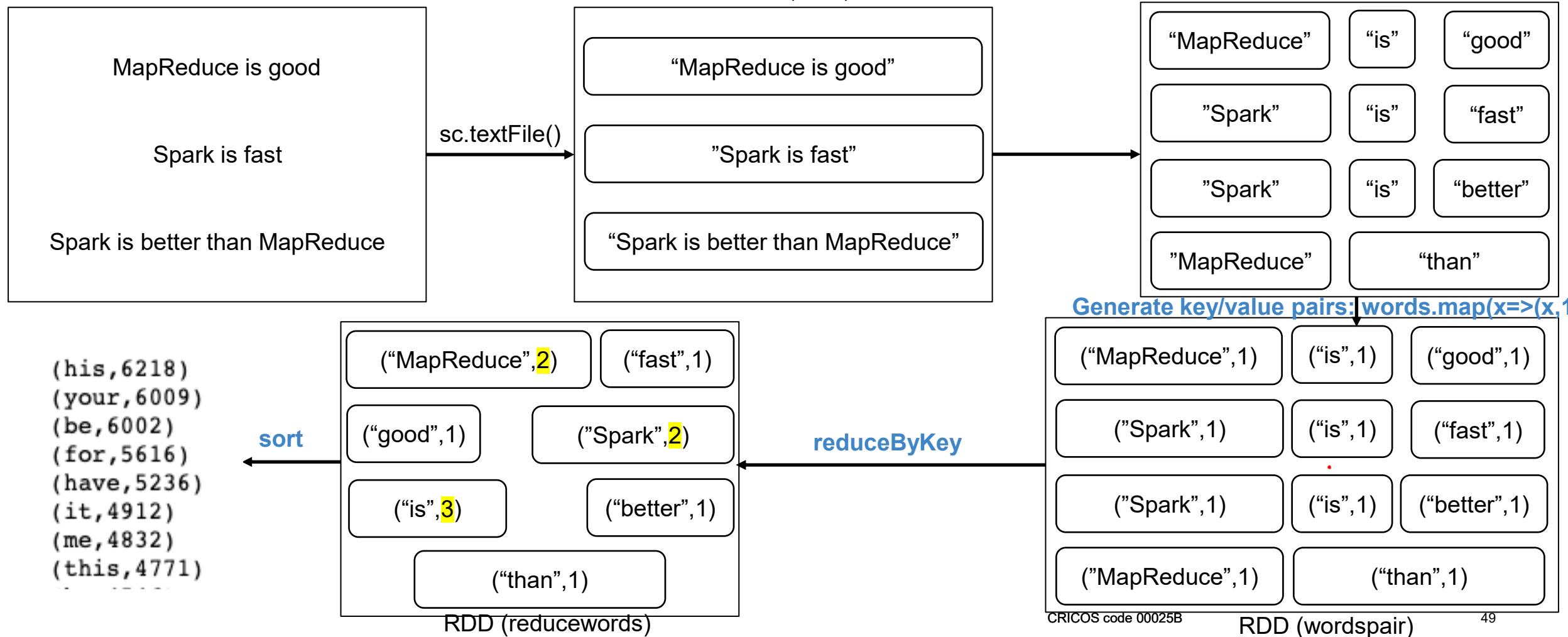
Outline

- RDD Creation: `textFile` & `parallelize` methods
- RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- Partition for Paired RDDs
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- Loading and Saving for RDD programming

RDD Programming Example I: Count words in Shakespeare



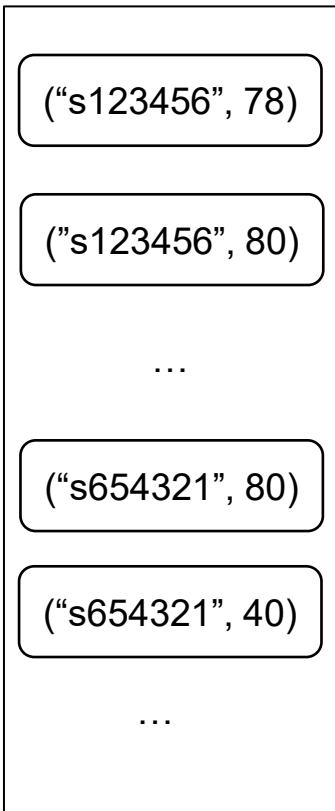
Solution: data.txt



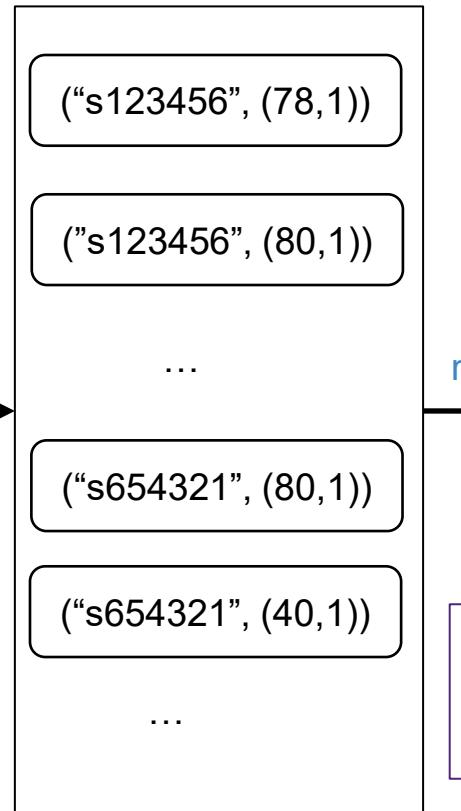
RDD Programming Example II: Calculate averaged marks

Solution:

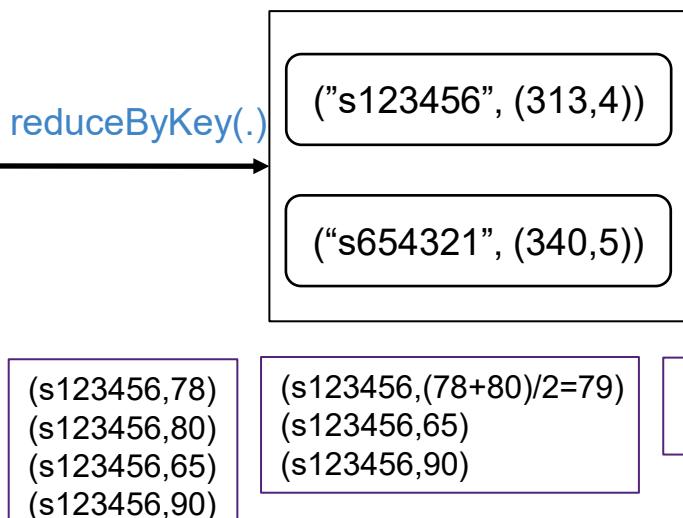
RDD (scores)



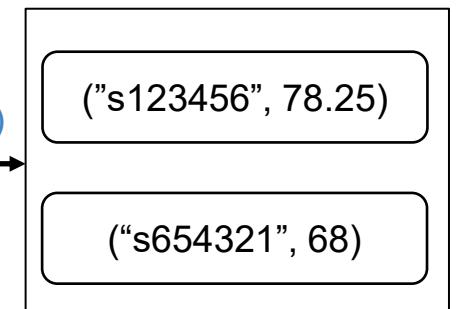
RDD (rdd1)



RDD (rdd2)



RDD (rdd3)

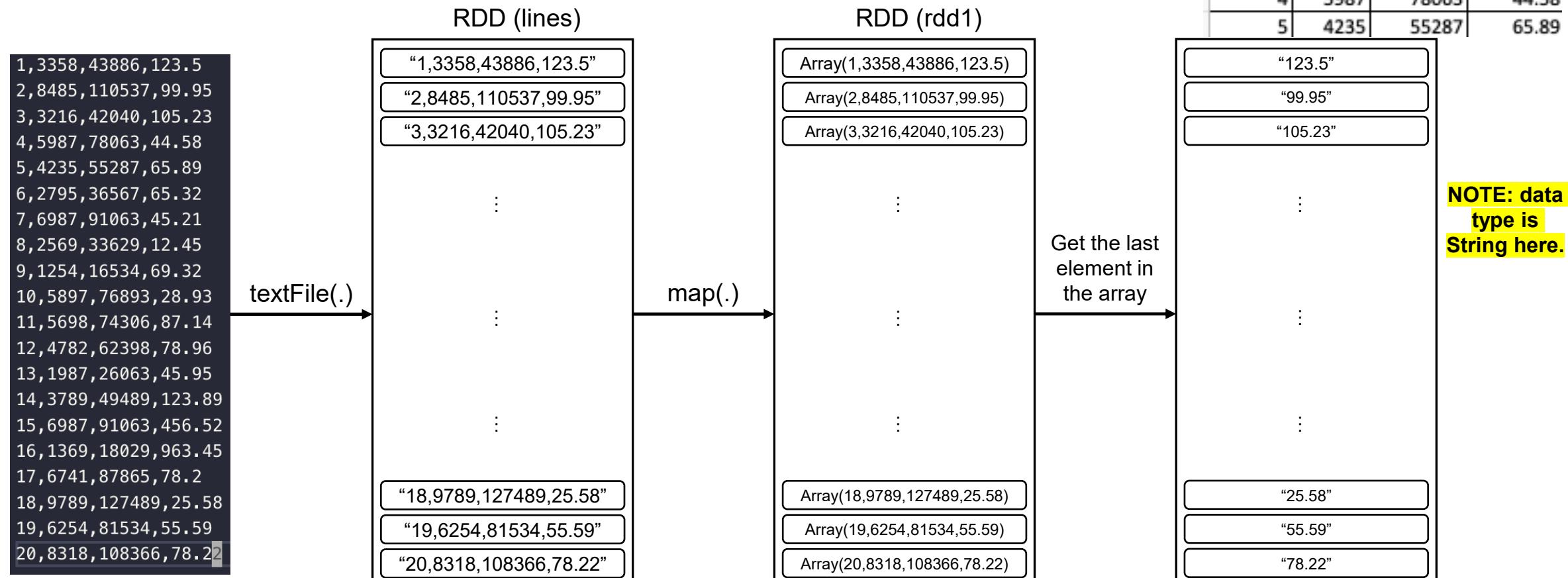


$$(78+80+65+90)/4=78.25$$

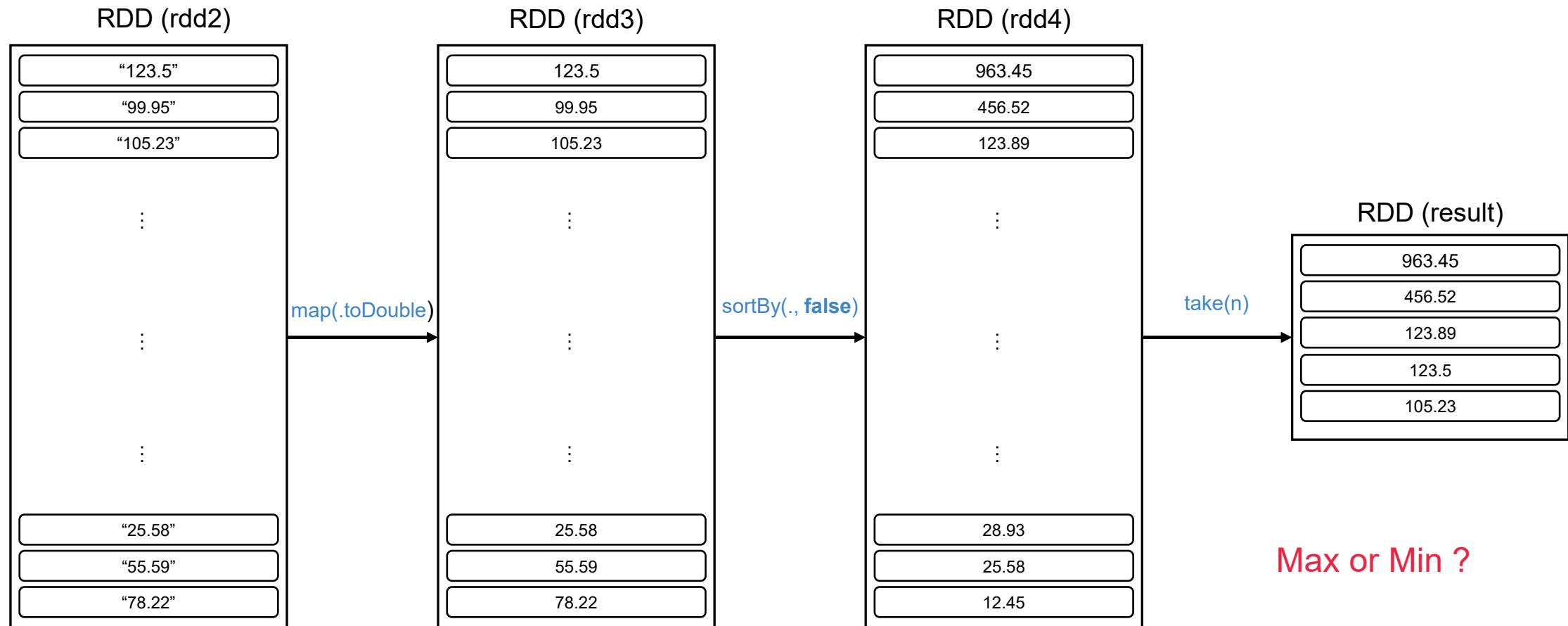
RDD Programming Example III: Get Top-5 Values

Get the Top value of RDD

Given a dataset of sale records: **sales(orderId, userId, productId, payment)**



RDD Programming Example: Get Top-5 Values



RDD Programming Example IV: File Sorting

Given multiple files (local file system or distributed file system), sort contents in the files.

File 1		File 2		File 3		File 4		sorted results	
1	589	1	548	1	963	1	639	9357	-----
2	3654	2	97	2	321	2	8528	9014	
3	8741	3	60	3	654	3	791	8741	
4	5674	4	3	4	963	4	317	8528	
5	581	5	588	5	85	5	91	5674	
6	30	6	498	6	714	6	63	3654	
7	2036	7	145	7	951	7	20	2036	
8	201	8	963	8	9357	8	60	986	
9	546	9	21	9	657	9	620	963	
10	9014	10	212	10	986	10	801	963	
								951	
								801	

RDD Programming Example V: Movie Rating

MovieLens

- GroupLens Research has collected and made available rating data sets: MovieLens 20M
 - 20 million ratings and 465,000 tag applications applied to 27,000 movies by 138,000 users.
 - Includes tag genome data with 12 million relevance scores across 1,100 tags.
 - Released 4/2015; updated 10/2016 to update links.csv and add tag genome data.

Goal:

- Given user ratings (more than 20,000,000 ratings by 138,000 users), calculate the average rating score for each movie (27,000).
- Display the titles and averaged rating scores for top 100 movies

movies(movielid, title, genres)

movielid	title	genres
1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
2	Jumanji (1995)	Adventure Children Fantasy
3	Grumpier Old Men (1995)	Comedy Romance
4	Waiting to Exhale (1995)	Comedy Drama Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action Crime Thriller
7	Sabrina (1995)	Comedy Romance
8	Tom and Huck (1995)	Adventure Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action Adventure Thriller

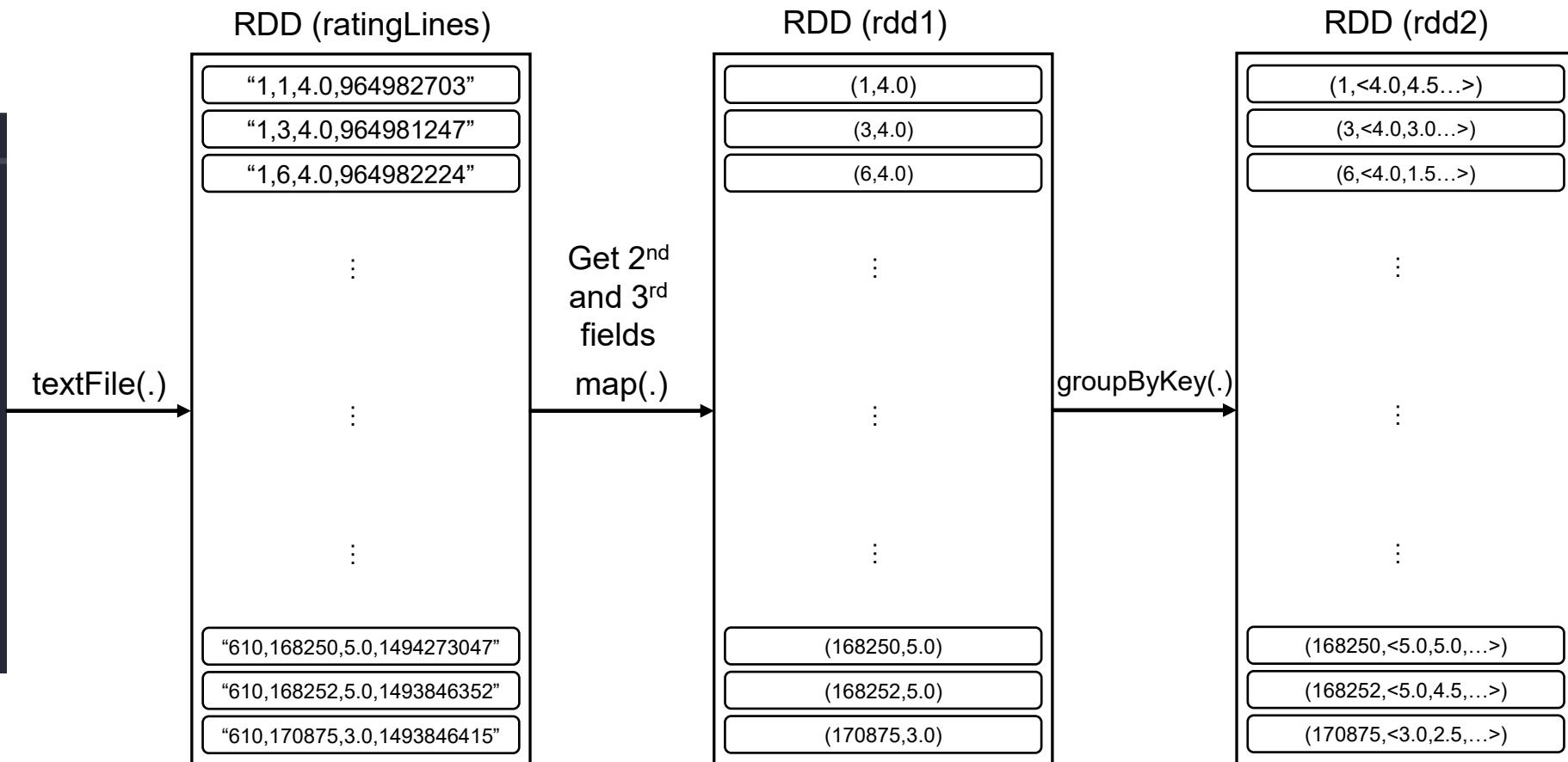
ratings(userId, movielid, rating, timestamp)

1	userId	movielid	rating	timestamp
2	1	1	4	964982703
3	1	3	4	964981247
4	1	6	4	964982224
5	1	47	5	964983815
6	1	50	5	964982931
7	1	70	3	964982400
8	1	101	5	964980868
9	1	110	4	964982176
10	1	151	5	964984041

RDD Programming Example: Movie Rating

Solution:

```
1  1,1,4.0,964982703
2  1,3,4.0,964981247
3  1,6,4.0,964982224
4  1,47,5.0,964983815
5  1,50,5.0,964982931
6  1,70,3.0,964982400
7  1,101,5.0,964980868
8  1,110,4.0,964982176
9  1,151,5.0,964984041
10 1,157,5.0,964984100
```

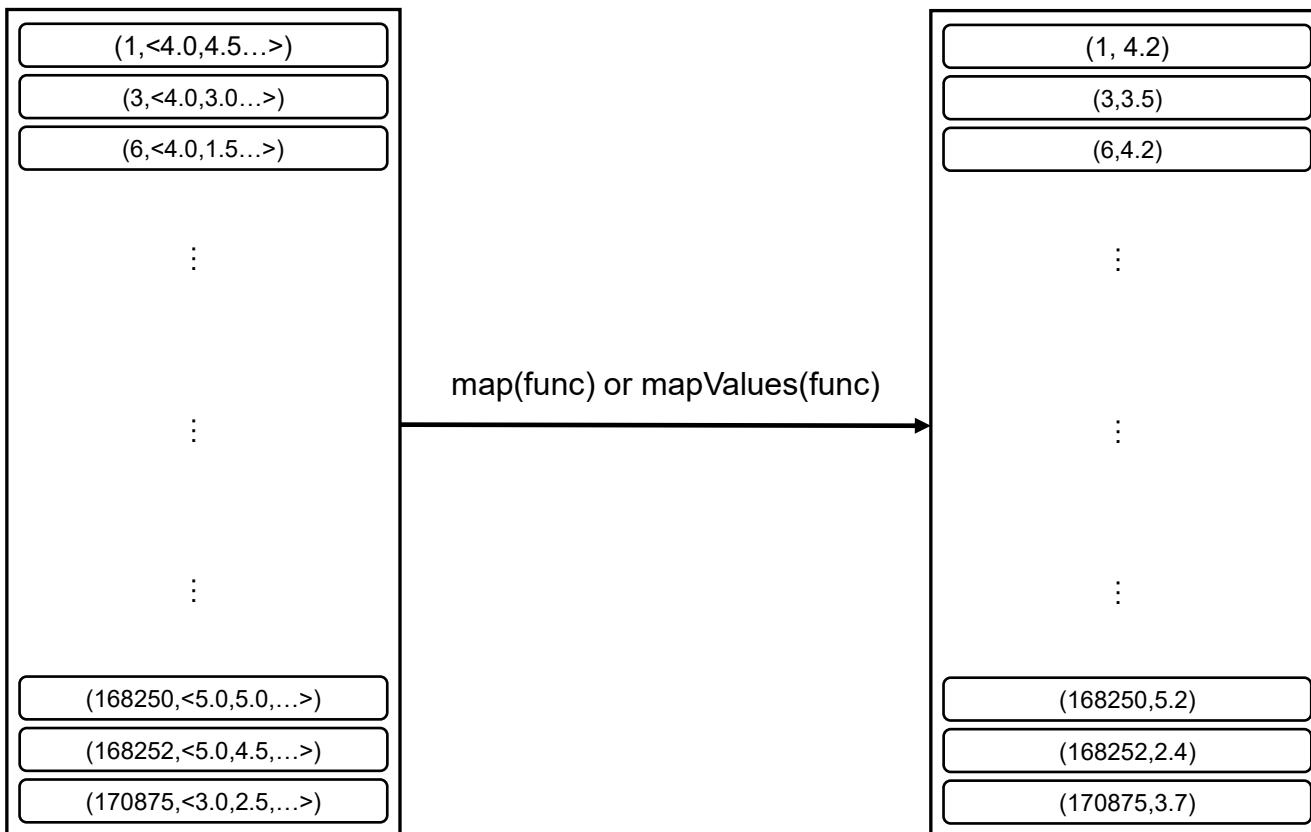


RDD Programming Example: Movie Rating

Solution:

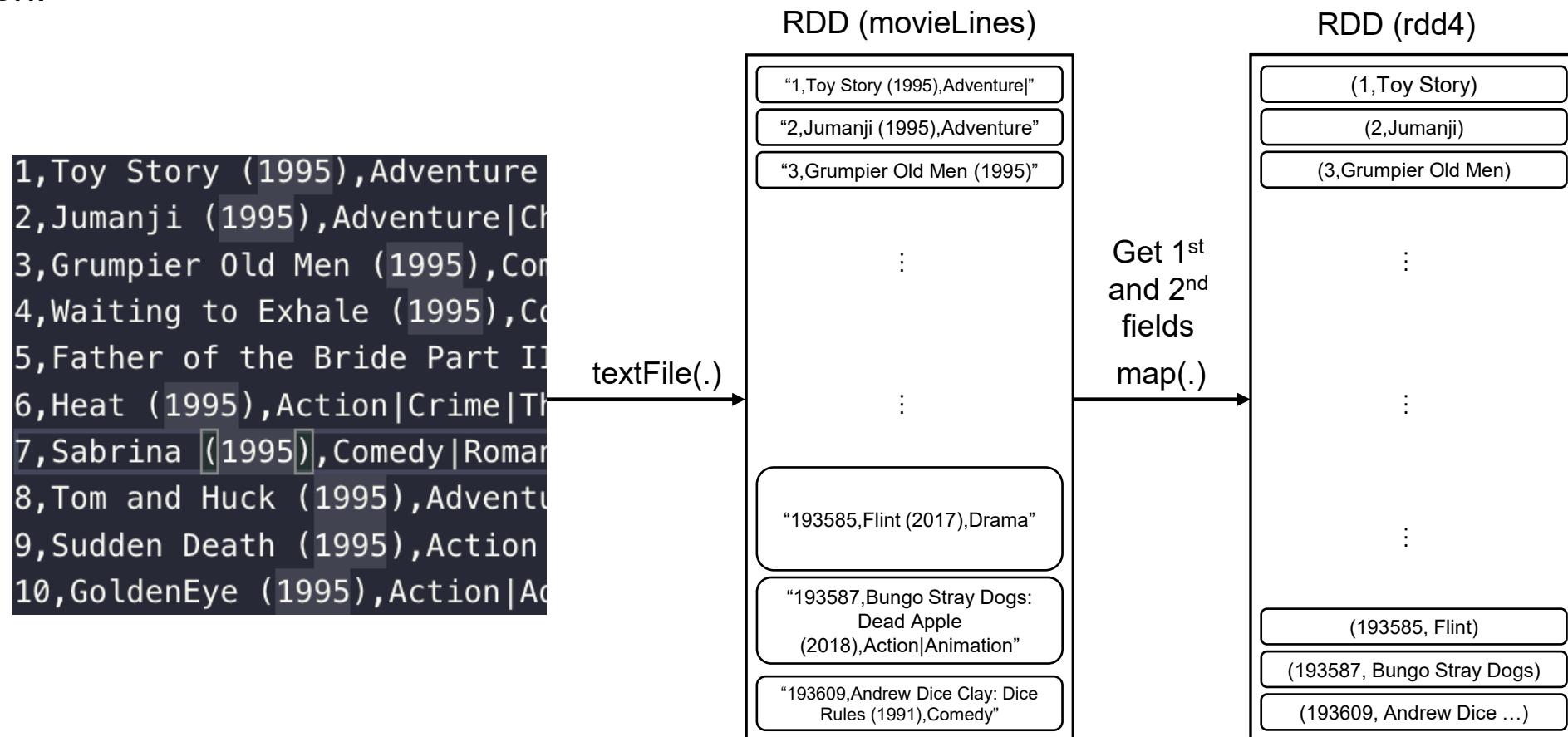
$$A = \langle 4.0, 4.5, 4.0, 4.0, 4.0, 4.5 \rangle$$

RDD (rdd2) Average = $A.sum / A.size = 4.2$ RDD (rdd3)



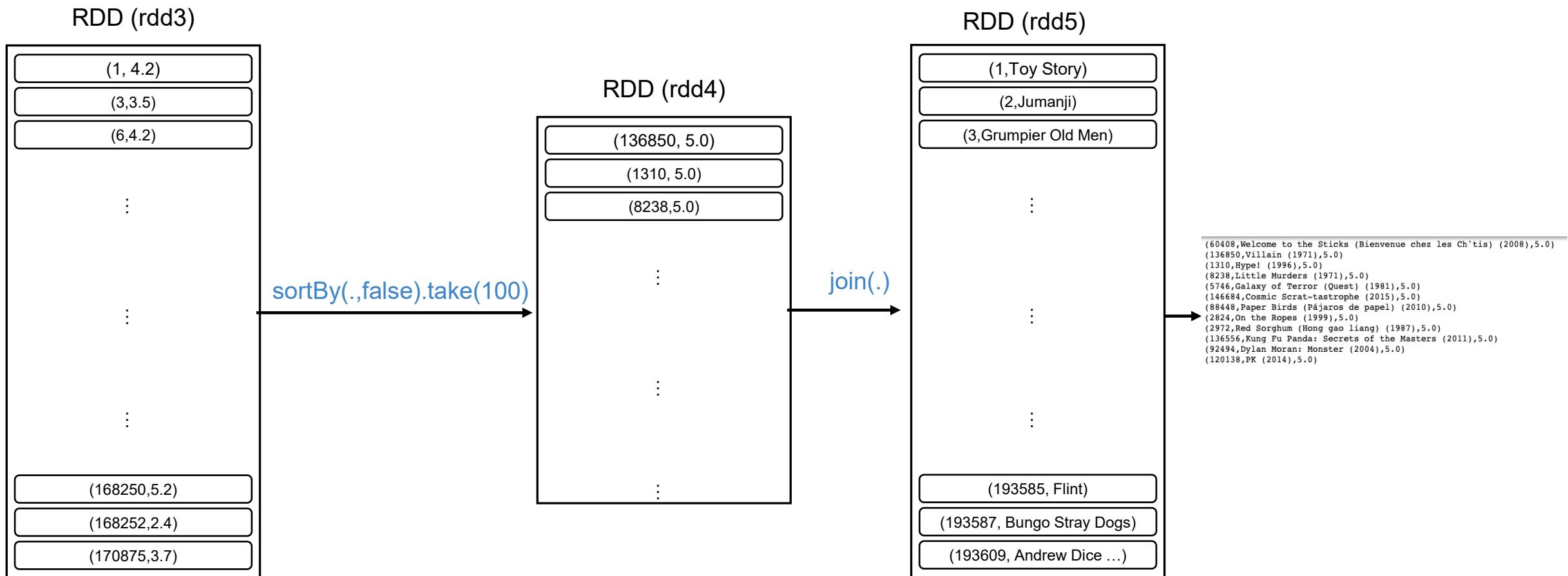
RDD Programming Example: Movie Rating

Solution:



RDD Programming Example: Movie Rating

Solution:



Outline

- RDD Creation: `textFile` & `parallelize` methods
- RDD Transformation: `Map`, `filter`, `flatMap`, `distinct`, `union`, `intersection`, `subtract`, and `cartesian`
- RDD Action: `Collect`, `count`, `reduce`, `first`, `take`, `foreach`
- RDD Persistence, Caching, Shared Variables (Broadcast & Accumulator)
- Key/Value Pairs: Creating KV Pairs
- Transformations and actions on Key/Value Pairs
 - `reduceByKey`, `groupByKey`, `keys`, `values`, `sortByKey`, `mapValues`, and `join`
- Partition for Paired RDDs
- RDD Programming Examples:
 - I. Word count
 - II. Average marks calculation
 - III. Get top-5 values
 - IV. File sorting
 - V. Movie Rating
- • Loading and Saving for RDD programming

Loading and Saving Data for RDD Programming

Spark supports a wide range of input and output sources and options are:

- File formats and filesystems
 - [Text](#), [JSON](#), [SequenceFiles](#), etc.
 - Local or distributed filesystems: Network File System ([NFS](#)), [HDFS](#), Amazon [S3](#)
- Structured data sources through [Spark SQL](#)
 - The Spark SQL module provides a nicer and often more efficient API for structured data sources, including JSON and Apache Hive.
- Databases and key/values stores
 - built-in and third-party libraries for connecting to [Cassandra](#), [HBase](#), Elasticsearch, and JDBC databases

Loading and Saving Data for RDD Programming

Loading text files

- simply call `textFile()` method of `SparkContext`:
 - `val textFile = sc.textFile(path, minPartitions)`
 - e.g. `val textFile = sc.textFile("file:///home/sen/mysparkcode/wordcount/word.txt")`
- specify the `path` to the file and the `number of partitions` with `minPartitions`.
- As lazy evaluation in Spark, errors won't be displayed and reported until an action is encountered (even "word123.txt" does not exist):
 - e.g. `val textFile = sc.textFile("file:///home/sen/mysparkcode/wordcount/word123.txt")`

Saving text files

- Simply call `saveAsTextFile()` method
 - `textFile.saveAsTextFile(path)`
 - e.g. `textFile.saveAsTextFile("file:///home/sen/mysparkcode/wordcount/writeback")`
- The `path` is treated as a directory and Spark will output multiple files underneath that directory. This allows Spark to write the output from multiple nodes.

Loading and Saving Data for RDD Programming

Loading JSON (JavaScript Object Notation)

- There are a wide variety of JSON libraries available for the three languages.
- Scala has a built-in JSON library:
 - `scala.util.parsing.json.JSON`
 - `parseFull(jsonString:String)` method in `scala.util.parsing.json.JSON` can help you load a JSON file:
 - Return class `Some(map: Map[String, Any])`, if successful.
 - Return class `None`, if failed.

```
1 {  
2   "firstName": "John",  
3   "lastName": "Smith",  
4   "sex": "male",  
5   "age": 25,  
6   "address":  
7     {  
8       "streetAddress": "21 2nd Street",  
9       "city": "New York",  
10      "state": "NY",  
11      "postalCode": "10021"  
12    },  
13    "phoneNumber":  
14    [  
15      {  
16        "type": "home",  
17        "number": "212 555-1234"  
18      },  
19      {  
20        "type": "fax",  
21        "number": "646 555-4567"  
22      }  
23    ]  
24 }
```

What's Next for Spark Programming?

In Apache Spark, both **RDD** and **DataFrame** are fundamental data structures, but ...

- **RDD (Build foundation for Spark Programming):**
 - It's a **low-level** data structure in Spark, providing an object-oriented interface.
 - Requires more extensive coding for operations and transformations
 - Gives you a fine-grained control over data and its distribution across nodes.
 - RDDs are type-safe: the compiler will validate types while compiling (early error detection).
 - When you need fine-grained control over your dataset and its partitioning, or when you're dealing with data that doesn't fit into a structured model.
- **DataFrame (Will be introduced with Spark SQL&ML in Tutorial):**
 - It's a **higher-level** abstraction representing a distributed collection of data organized into **named columns**. It's conceptually **equivalent to a table** in a relational database.
 - Allows users to perform operations using SQL-like syntax, making it more **user-friendly**.
 - Can be easily integrated with various data sources (like Hive, ORC, JSON, JDBC, and others).
 - Not type-safe (returned as Array of rows) --> type-related errors can only be caught at runtime.
 - When you are working with structured data, or when you need to leverage Spark's in-built optimizations and want to perform SQL-like operations.

Reading Materials

1. Hamstra, Mark, and Matei Zaharia. Learning Spark: lightning-fast big data analytics. O'Reilly & Associates, 2013.
2. <https://spark.apache.org/docs/latest/>
3. <https://www.tutorialspoint.com/scala/>
4. <https://data-flair.training/blogs/spark-tutorial/>

Next (Week 10) Topic:

Distributed File Systems – GFS and HDFS