More about Spark

Diego Rivera Correa

Slides owned by Mirek Riedewald

Key Learning Goals

- What is the purpose of Spark SQL, Spark Streaming, Spark MLlib, and Spark GraphX?
- Why do map and flatMap remove the Partitioner of a pair RDD? Why do mapValues and flatMapValues preserve the Partitioner of a pair RDD?
- When can the hash+shuffle join on pair RDDs avoid shuffling?
- Given a Spark program, determine how many jobs are going to be executed.
- Given a Spark program, determine which operations are executed by the master and which by the worker tasks.

Introduction

- This module surveys important components and aspects of Spark that have not been covered in detail yet:
  - Spark SQL
  - Spark Streaming
  - Spark GraphX
  - Partitioning and shuffling in Spark
  - The interplay between lineage, jobs and lazy execution.

Let us start with SQL operations in Spark.

Running Example

```
case class Student(
   id: Int,  name: String, major: String,
   age: Int,  gpa: Double )

val studentsDS = Seq(
        Student(1, "Amy", "CS", 20, 3.8),
        Student(2, "Bill", "Math", 22, 3.2),
        Student(3, "Carla", "CS", 21, 3.9),
        Student(4, "Dan", "History", 23, 2.9) ).toDS()
val studentsDF = studentsDS.toDF()
val studentsRDD = studentsDS.rdd

val enrollments = Seq(
   (1, "CS6240"), (2, "MATH2331"), (3, "CS6240")
).toDF("id", "course")
```

Running Example

- studentsDS/DF

id	name	major	age	gpa
1	Amy	CS	20	3.8
2	Bill	Math	22	3.2
3	Carla	CS	21	3.9
4	Dan	Hist.	23	2.9

- enrollments

id	course
1	CS6240
2	MATH2321
3	CS6240

SQL Basics

- SQL is based on the relational calculus. A calculus expression describes what we are looking for, not how to compute it.

- The computation steps and their order of execution are expressed in relational algebra. For a given SQL query, the logical query plan corresponds to an expression in relational algebra.

SQL Basics

- Relational algebra has only 5 primitive operators, which can be combined to compose complex queries: selection, projection, Cartesian product (a.k.a. cross product or cross join), set union, and set difference.
  - The renaming operator is needed for formal reasons but does not manipulate data.

- In addition, grouping and aggregation operators were introduced as well. We already encountered most of these operators before.

Reminder: Selection and Projection

DBMS (SQL): Assume the input relation R has schema (A1, A2, A3); P is a predicate returning true/false; F is a function

Selection:	SELECT * FROM R WHERE F(A1, A2, A3)
Projection (on A1):	SELECT A1 FROM R
Extended projection:	SELECT F(A1, A2, A3) FROM R

Spark:

```
myRDD.filter( P(x) )          // selection
myRDD.map( x => F(x) )        // extended projection

myDS.filter( P(x) )           // selection
myDS.select( "A1" )           // projection
myDS.map( x => F(x) )         // extended projection
```

Selection and Projection

```
studentsDF
    .filter($"age" >= 21)
    .select($"name", $"major")
    .show()
```

```
SELECT name, major
FROM students
WHERE age >= 21;
```

id	name	major	age	gpa
1	Amy	CS	20	3.8
2	Bill	Math	22	3.2
3	Carla	CS	21	3.9
4	Dan	Hist.	23	2.9

name	major
Bill	Math
Carla	CS
Dan	Hist.

Reminder: Grouping and Aggregation

DBMS (SQL): Assume the input relation R has schema (Key, Val)

SELECT Key, myAGG(Val) FROM R GROUP BY Key

Spark:

myPairRDD.aggregateByKey(aggFunction)

myDS.groupBy("Key").agg(aggFunction)

Grouping and Aggregation

```
studentsDF
    .groupBy($"major")
    .agg(
        round(avg($"gpa"), 2).as("avg_gpa")
     )
     .show()
```

```
SELECT major, AVG(gpa) as avg_gpa
FROM students
GROUP BY major
```

major	avg_gpa
CS	3.85
Math	3.20
Hist.	2.90

Cartesian Product

DBMS (SQL): Cartesian product of relations R and S

```
SELECT * FROM R, S
```

Spark:

```
myRDD.cartesian( otherRDD )

myDF.crossJoin( otherDF )

// The joinWith method of DataSet also supports a version where
join types, including cross, can be selected.
```

Cartesian Product

RDDs

```
val names = sc.parallelize(Seq("Amy", "Bill", "Carla"))

val slots = sc.parallelize(Seq("Morning", "Afternoon"))

val allAssignments = names.cartesian(slots)
```

DataFrame

```
val slotsDF = Seq("Morning", "Afternoon").toDF("slot")

StudentsDF
    .select($"name")
    .crossJoin(slotsDF)
    .show()
```

Result, ignoring ordering:

(Amy, Morning)
(Amy, Afternoon)
(Bill, Morning)
(Bill, Afternoon)
(Carla, Morning)
(Carla, Afternoon)

Set Difference

- Set functions generally require both inputs to be of the same type.

- rdd1.subtract(rdd2) transformation: returns all elements from rdd1 that are not in rdd2. This is for ordinary RDDs and hence compares the entire element.

- rdd1.subtractByKey(rdd2) transformation: This is for pair RDDs, returning all elements from rdd1 whose keys do not appear as key in rdd2.

Set Difference

- val lecture1 =
- sc.parallelize(Seq("Amy", "Bill", "Carla"))
-
- val lecture2 =
- sc.parallelize(Seq("Bill", "Dan"))
-
- val onlyLecture1 = lecture1.subtract(lecture2)

- Result: {Amy, Carla}

Union and Intersection in Spark

- rdd1.union(rdd2) transformation: returns all elements that occur in either input, but does not remove duplicates.
  - For DataSet, the corresponding functions are union and unionByName.
- rdd1.intersection(rdd2) transformation: returns all elements that occur in both.
  - For DataSet, the corresponding function is intersect.
- Intersection does not add new functionality—it can be expressed with union and set difference.

Union and Intersection in Spark

- val rdd1 = sc.parallelize(
- Seq("Amy", "Bill", "Bill")
-)
-
- val rdd2 = sc.parallelize(
- Seq("Bill", "Carla")
-)

- Union = rdd1.union(rdd2). Conceptual result:

  Amy, Bill, Bill, Bill, Carla

- It concatenates the RDDs. It does not perform deduplication.

Union and Intersection in Spark

- For set-union semantics: rdd1.union(rdd2).distinct()

- Result:

  Amy, Bill, Carla

- Intersection = rdd1.intersection(rdd2)

- Result: Bill
  - Even though Bill occurs multiple times, RDD intersection returns distinct common elements.

Reminder: Joins

- The join can be expressed as a selection applied to the Cartesian product. Then why was it introduced as a separate operator?
  - Joins are very common in practice, and they tend to be expensive to compute. Since most joins in databases are equi-joins, specialized techniques were proposed to reduce their computation cost.
- Refer to the module on joins for Spark join operations.

Spark SQL

- Spark SQL works with DataFrames.
  - Reminder: A DataFrame is like a database table. It consists of rows, each adhering to a fixed schema that defines the name and type of all columns. You can think of a DataFrame as an RDD with additional structure. In recent Spark versions, it is an alias for DataSet[Row].

Spark SQL

- DataFrames can be processed using SQL and SQL-inspired functions. Think of Spark SQL as a distributed Spark-powered SQL query processor. It can even be accessed like a database server using JDBC.

- Operations on DataFrames are translated into optimized low-level RDD operations. Like in a DBMS, the structure enables optimizations not possible for general RDDs.

Implementation Notes

- Spark's default file format for structured data is Parquet.

- Parquet is typically better suited for analytical workloads. It divides records into "row-groups", and each group has some metadata summarizing the values in their columns.
  - This is a common design tradeoff for DBMSs when leveraging a row-oriented vs column-oriented storage.

Implementation Notes

- DataFrame metadata is managed in a table catalog. Spark applications can then access a DataFrame by name.

- This information disappears when the Spark context is restarted. To make the catalog persistent, Spark needs to be built with Hive support.

Implementation Notes

- Suppose you load a Parquet dataset: val peopleDF = spark.read.parquet("/data/people")

- Here, peopleDF is a Scala variable in the driver program. You can use it like this:

```
peopleDF
  .filter($"age" >= 30)
  .show()
```

- But SQL does not automatically know that this DataFrame should be called peopleDF:

```
spark.sql("""
    SELECT *
    FROM peopleDF
 """)
```

Implementation Notes

- That will not work merely because the Scala variable is named peopleDF. The variable name belongs to the Scala program, not to Spark SQL's table namespace.

- If you
  use registpeopleDF.createOrReplaceTempView("people"), Spark's catalog now contains an entry conceptually resembling the table.

- You can now access it by name:

```
spark.sql("""
  SELECT name, income
  FROM people
  WHERE age >= 30
""").show()
```

Creating and Storing DataFrames

- DataFrames can be created as follows:
  - Convert an RDD.
  - Run an SQL query.
  - Load external data.

- Spark can easily load and import data from JDBC sources, and files in JSON and Parquet format.

Working with DataFrames

- show(n) returns the first n rows as formatted text.
- printSchema returns the schema of the DataFrame.
- columns returns a list of the column names.
- dtypes returns a list of the column names and their types.

SQL-Style Functions

- select: relational projection operator, choosing which columns to keep in the resulting DataFrame.
- drop: relational projection operator, choosing which columns to drop.
- where, filter: relational selection operator, choosing which rows to keep.
- withColumnRenamed: renames a column.
- orderBy, sort: sort by specified column(s). When sorting on multiple columns, this sorts in "row-major" order, i.e., compares based on one column, then in case of equality compares the next, etc.
- Spark SQL also allows user-defined functions (UDF) in expressions, e.g., to extract information from a string column.

SQL-Style Functions

- groupBy groups by a list of columns, returning a GroupedData object.
  - When calling an aggregate function such as count on a GroupedData object, the result is a DataFrame with an additional column holding the corresponding aggregate values for the groups.

- join performs an equi-join on the specified columns, offering the “outer” option.

Using SQL Directly

- sql(SQLstring): is a function of SparkSession that executes an SQL query using Spark on a DataFrame, returning a DataFrame.
  - The SQL dialect can be configured with spark.sql.dialect.

- For interactive query mode, Spark also offers an SQL shell.

Running Example

- StudentsDS

id	name	major	age	gpa
1	Amy	CS	20	3.8
2	Bill	Math	22	3.2
3	Carla	CS	21	3.9
4	Dan	Hist.	23	2.9

- Enrollments

id	course
1	CS6240
2	MATH2321
3	CS6240

Using SQL Directly

- Find the names and grades of CS majors enrolled in CS6240. DF and SQL solutions:

```
val result = studentsDF
      .join(enrollmentsDF, Seq("id"))
      .filter( $"major" === "CS"
               && $"course" === "CS6240" )
      .select($"name", $"grade")

result.show()
```

```
StudentsDF.createOr...View("Students")
 EnrollmentsDF.createOr...View("Enrollments")

val result = spark.sql("""
      SELECT S.major, AVG(E.grade) AS avg
      FROM Students S
      JOIN Enrollments E ON S.id = E.id
      GROUP BY S.major""")
```

major	avg
CS	93.5
Math	88.0

Query Optimization

- Like in a DBMS, Spark's Catalyst optimizer translates a query into Spark jobs. It goes through several steps:
  - The query is parsed and relation references are checked to produce the logical plan.
  - This plan is optimized, mostly by re-arranging (e.g., projection and filter before join) and combining (e.g., apply multiple filters on same relation together) operations. It currently is based on "safe" heuristics but could be extended to include cost estimation.
  - From the optimized logical plan, a physical plan is generated, which includes operator implementation choices. Here cost models should be used to decide between different join implementations.
  - Finally, the physical plan is translated into Spark jobs.
- To see the logical and physical plans, use DataFrame's explain method. They can also be seen through the Web UI.

Next, we take a closer look at partitioning of RDDs and DataSets in Spark.

Primer Slide: Transformations and Actions

- Transformations build a plan and actions run it

- **Transformation**
  - Creates a new RDD from an existing RDD.
  - Evaluated lazily: Spark records what should be computed.
  - map, filter, flatMap, mapValues, reduceByKey, join.

- **Action**
  - Requests a final result or writes output.
  - Triggers Spark to execute the required transformations.
  - Examples: count, collect, take, lookup, foreach, saveAsTextFile.

Primer Slide: Transformations and Actions

```
val numbers = sc.parallelize(Seq(1, 2, 3, 4), 2)

val even = numbers.filter(x => x % 2 == 0) // Transformation

val squared = even.map(x => x * x) // Transformation

val total = squared.sum() // Action: returns 20
```

Primer Slide: Transformations and Actions

- Nothing processes the records when filter and map are called. They create the recipe. sum() needs an actual value, so Spark executes the recipe. This is the central distinction in Spark's RDD model.

- For programs in this module, count one job for each executed action call. A shuffle may divide that job into multiple stages.

Partitioners in Spark

- In MapReduce, the Partitioner often plays an important role, and it is relatively easy to understand. In Spark, the situation is potentially confusing:

- Pair RDDs support custom Partitioner objects that behave like MapReduce Partitioners, mapping objects based on the key.
  - The Partitioner object is associated with the pair RDD and can be exploited to eliminate the need for shuffling for equi-joins between pair RDDs that have the same Partitioner.

Partitioners in Spark

- Plain RDDs do not support custom Partitioners. Transformations repartition and coalesce can achieve hash partitioning.

- DataSet also does not support Partitioners, but we can use the repartition transformation. In general, the idea is to leave partitioning decisions to the optimizer instead of the user.

Controlling Pair RDD Partitions

- The Partitioner object performs RDD partitioning. The default is HashPartitioner. It assigns an element e to partition hash(e) % numberOfPartitions.
  - hash(e) is the key's hash code for pair RDDs.
  - The default number of partitions is determined by Spark configuration parameter spark.default.parallelism.

Controlling DataSet Partitions

- Function repartition(numPartitions: Int, partitionExprs: Column*) returns a new DataSet that is hash partitioned based on the partition expression.
  - The expression can select existing columns or a new column created from them. E.g., given myDS(A, B) one can hash on (A+B) or some other function of A and B.

Controlling DataSet Partitions

- Partition using an existing column (e.g., val partitioned = myDS.repartition(3, $"A"))

- Spark computes conceptually: partition = hash(A) mod 3. Therefore, all records with the same A value go to the same partition:

- Partition ?:
- (1, 4)
- (1, 7)
-
- Partition ?:
- (2, 3)
- (2, 8)
-
- Partition ?:
- (3, 2)

A	B
1	4
2	3
1	7
3	2
2	8

Controlling DataSet Partitions

- If we do val partitioned = myDS.repartition(3, $"A" + $"B")

- Partition of A+B = 5:
- (1, 4)
  (3, 2)
  (2, 3)

  Partition of A+B = 8:
  (1, 7)

  Partition A+B = 10:
  (2, 8)

A	B	A + B
1	4	5
2	3	5
1	7	8
3	2	5
2	8	10

Controlling DataSet Partitions

- Function repartitionByRange similarly performs range partitioning.
  - When no explicit sort order is given, Spark uses a default ordering.
  - The data in a range is not sorted on the partitioning columns!

Data Shuffling

- Some pair-RDD transformations *preserve partitioning*, e.g., mapValues and flatMapValues.

- Changes to the Partitioner, including changing the number of partitions, require shuffling.

Data Shuffling

- map and flatMap remove the pair RDD's Partitioner, but do not result in shuffling. But any transformation that adds a Partitioner, e.g., reduceByKey, results in shuffling. Transformations causing a shuffle after map and flatMap:
  - aggregateByKey, foldByKey, reduceByKey, groupByKey, join, leftOuterJoin, rightOuterJoin, fullOuterJoin, subtractByKey
  - sortByKey (always causes shuffle)
  - partitionBy and coalesce with shuffle=true setting.

Reminder: Joins

- For pair RDDs, there are join, leftOuterJoin,rightOuterJoin, and fullOuterJoin.

- These operations have versions that take a Partitioner object or a number of partitions as parameter.
  - If no Partitioner or number of partitions is specified, Spark takes the Partitioner of the first RDD. This way only the second RDD needs to be shuffled.

Shuffle Implementation

- Sorting (default) or hashing.
- Parameter spark.shuffle.manager specifies which is used. Sort is the default, because it uses less memory and creates fewer files. However, sorting has higher asymptotic complexity than hashing.
- Several other parameters control options such as consolidation of intermediate files, shuffle memory size, and if spilled data are compressed.

Next, we explore an important but somewhat more subtle issue: how does Spark pass data and functions to tasks (executors)?

What Is the Problem?

- Assume a function that is executed in a task (in an executor) accesses an object that lives in the master. Then the object needs to be passed to the task.
  - What if that object contains another object that may be accessed in the task? Then that object also needs to be passed to the task. And so on.

What Is the Problem?

- What if the function itself is a member function of an object?
  - Then the function may depend on the object (and it usually will—otherwise a good programmer would have created a static function) and hence the entire object needs to be passed to the task.

- What about other functions, e.g., anonymous functions like (x,y)=>(x+y)?

Let's Make This Concrete

- We explore the behavior of Spark with some example programs. These are from or inspired by the Spark Programming Guide.

- For the following, pay attention to where the data lives and how it is communicated between application master (driver) and tasks (executors).

Object References

- What happens if an RDD operation references an object that has not been broadcast?
- The tasks need to have access to the object so Spark will automatically send it to them.

```
// This object is created by the master
val obj = new Object( … )

// RDD map calls are executed on RDD partitions
// in a task. Spark recognizes that the map call
// references the object and hence automatically
// sends it to the task.
myRdd.map(x => x + obj.field)
```

Big Object Reference

- If the object is big, this will create a lot of data traffic. What if we only need a small part of the object?

```
class BigObject(val useful: Int, val dummy:List[Double]) extends Serializable
```

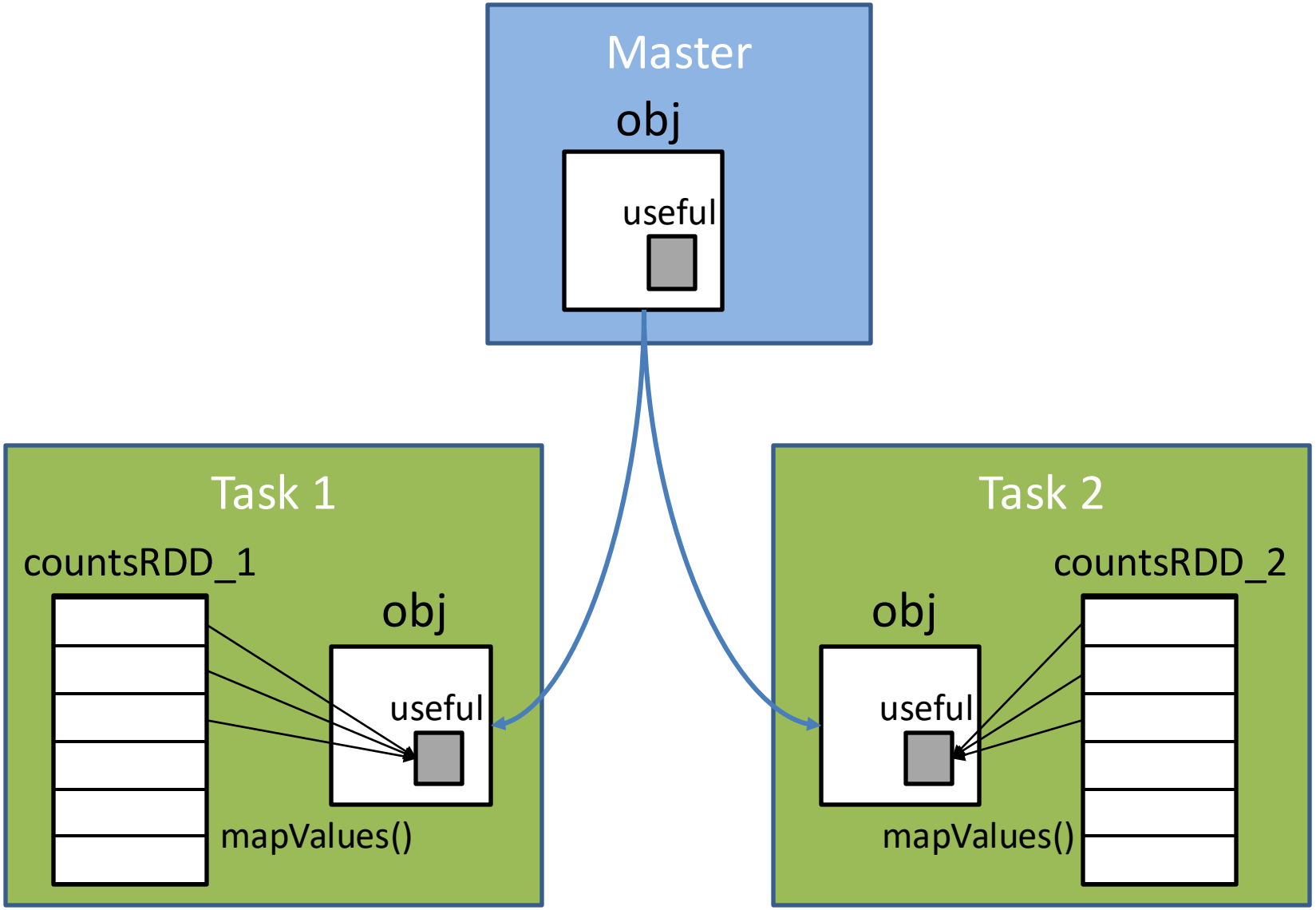
```
// Initialize object
val obj = new BigObject(usefulValue, List.fill(2000000)(Random.nextDouble))
```

```
val countsRDD = textFile.flatMap(line => line.split(" "))
                        .map(word => (word, 1))
                        .reduceByKey(_ + _)
                        .mapValues(x => x + obj.useful)
```

WordCount

Add a value to the counts

Data Movement



The Program in Action

- Local run with 5 threads, 5 tasks:

```
MemoryStore:54 - Block broadcast_2_piece0 stored as bytes in memory (estimated size 4.0 MB, free 335.1 MB)
BlockManagerInfo:54 - Added broadcast_2_piece0 in memory on 129.10.113.120:41270 (size: 4.0 MB, free: 362.3 MB)
MemoryStore:54 - Block broadcast_2_piece1 stored as bytes in memory (estimated size 4.0 MB, free 331.1 MB)
BlockManagerInfo:54 - Added broadcast_2_piece1 in memory on 129.10.113.120:41270 (size: 4.0 MB, free: 358.3 MB)
MemoryStore:54 - Block broadcast_2_piece2 stored as bytes in memory (estimated size 4.0 MB, free 327.1 MB)
BlockManagerInfo:54 - Added broadcast_2_piece2 in memory on 129.10.113.120:41270 (size: 4.0 MB, free: 354.3 MB)
MemoryStore:54 - Block broadcast_2_piece3 stored as bytes in memory (estimated size 4.0 MB, free 323.1 MB)
BlockManagerInfo:54 - Added broadcast_2_piece3 in memory on 129.10.113.120:41270 (size: 4.0 MB, free: 350.3 MB)
MemoryStore:54 - Block broadcast_2_piece4 stored as bytes in memory (estimated size 1673.9 KB, free 321.4 MB)
BlockManagerInfo:54 - Added broadcast_2_piece4 in memory on 129.10.113.120:41270 (size: 1673.9 KB, free: 348.6 MB)
```

```
DAGScheduler:54 - Job 0 finished: runJob at SparkHadoopWriter.scala:78, took 10.256529 s
```

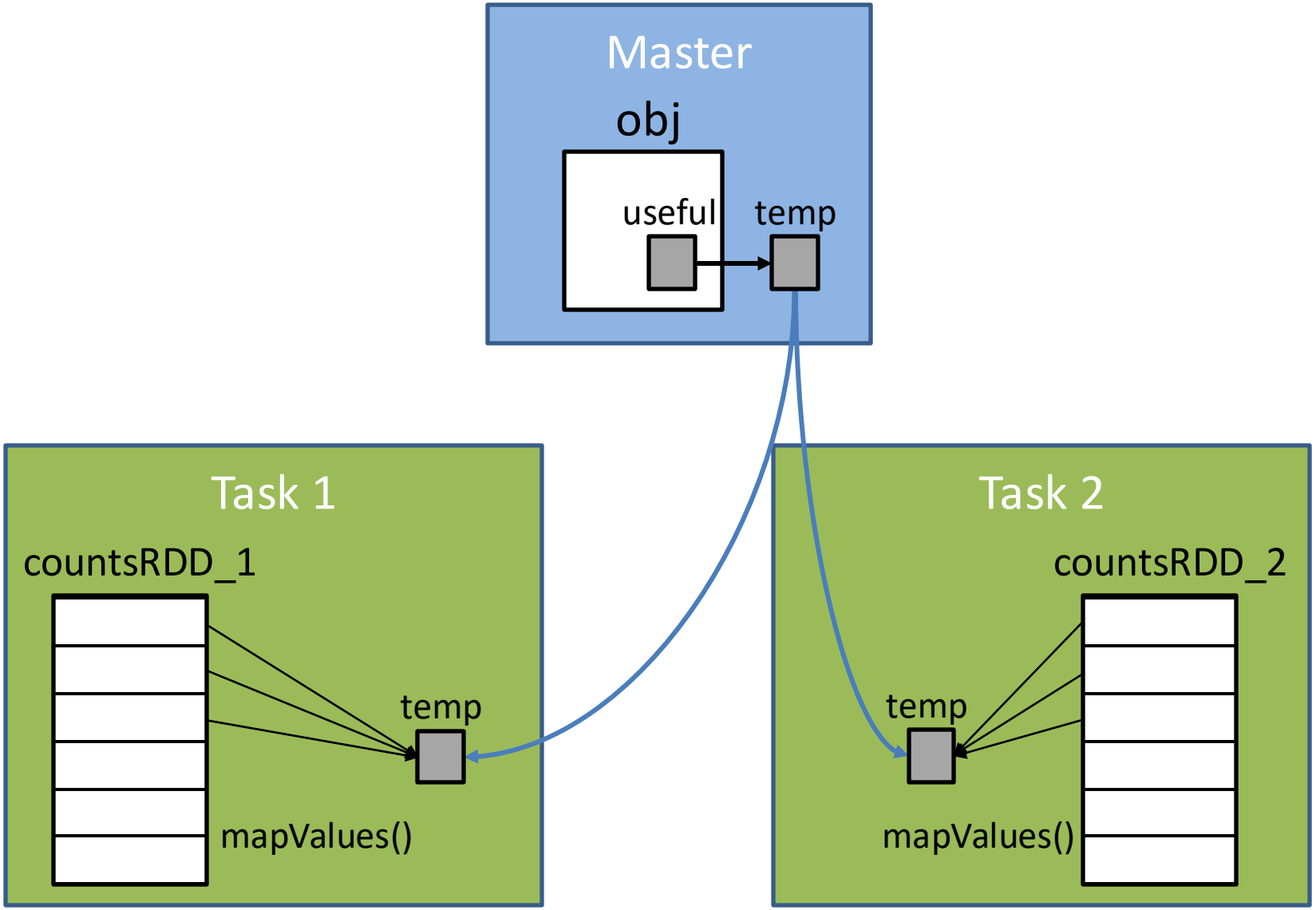
Workaround

- How do we avoid copying the big object unnecessarily?
- We can simply reference a temporary variable that contains only the part of the object we need.

```
val temp = obj.useful
```

```
val countsRDD = textFile.flatMap(line => line.split(" "))
                        .map(word => (word, 1))
                        .reduceByKey(_ + _)
                        //.mapValues(x => x + obj.useful)
                        .mapValues(x => x + temp)
```

Data Movement



The Workaround in Action

- Local run with 5 threads, 5 tasks:

```
MemoryStore:54 - Block broadcast_2_piece0 stored as bytes in memory (estimated size 38.9 KB, free 365.7 MB)
BlockManagerInfo:54 - Added broadcast_2_piece0 in memory on 129.10.113.120:34382 (size: 38.9 KB, free: 366.2 MB)
```

```
DAGScheduler:54 - Job 0 finished: runJob at SparkHadoopWriter.scala:78, took 1.036705 s
```

~10 times faster

Referencing an Object Method

- The same problem occurs when we reference a method of an object, even if it does not access the object's internal data.
  - To guarantee correct execution, the entire object must be passed to the task.

```
class BigObject(val useful: Int, val dummy:List[Double]) extends Serializable
{
    def sum(x: Int, y: Int): Int = {
      return x + y
    }
}
```

Adds up 2 numbers

```
val countsRDD = textFile.flatMap(line => line.split(" "))
                        .map(word => (word, 1))
                        .reduceByKey(_ + _)
                        //.mapValues(x => x + obj.useful)
                        //.mapValues(x => x + temp)
                        .mapValues(x => obj.sum(x, x))
```

```
DAGScheduler:54 - Job 0 finished: runJob at SparkHadoopWriter.scala:78, took 10.548821 s
```

Solution: Static Methods

- If the function does not depend on data in the object, make it a static method.
  - In Spark Scala, such static methods are defined using **object** instead of **class**. They do not appear inside the definition of a class like BigObject.
- A static function is sent to the tasks without including irrelevant object data.

```
object MyFunctions {
  def func1(s: String): String = { ... }
}

myRdd.map(MyFunctions.func1)
```

Anonymous Functions

- Recall examples such as myRDD.reduce((x,y) => x+y) and myRDD.map(line => line.split(" ")). Here "=>" defines an anonymous function.

- Anonymous functions are automatically serialized and passed to each task that applies it to the corresponding partition of the RDD.
  - They behave like simple static functions.

General Concept: Closures

- Closure variables are task-local copies. Updating such a copy does not update the driver's original variable.

- Broadcast variables are also read-only from the tasks' perspective, but they are explicitly distributed and cached for reuse. They are not simply ordinary closures

General Concept: Closures

- When writing Spark programs, we must pay attention where the data lives and how it is communicated between application master (driver) and tasks (executors).
  - As the examples demonstrated, understanding what is included in the closure is key to writing efficient Spark programs.

Closure vs Global Variables

- The closure is not the same as a general global variable. It creates a “1-way street” to share data from master to tasks, but it does not provide a “back channel” to let tasks communicate updates to the master or to other tasks.
  - It behaves like broadcast variables.

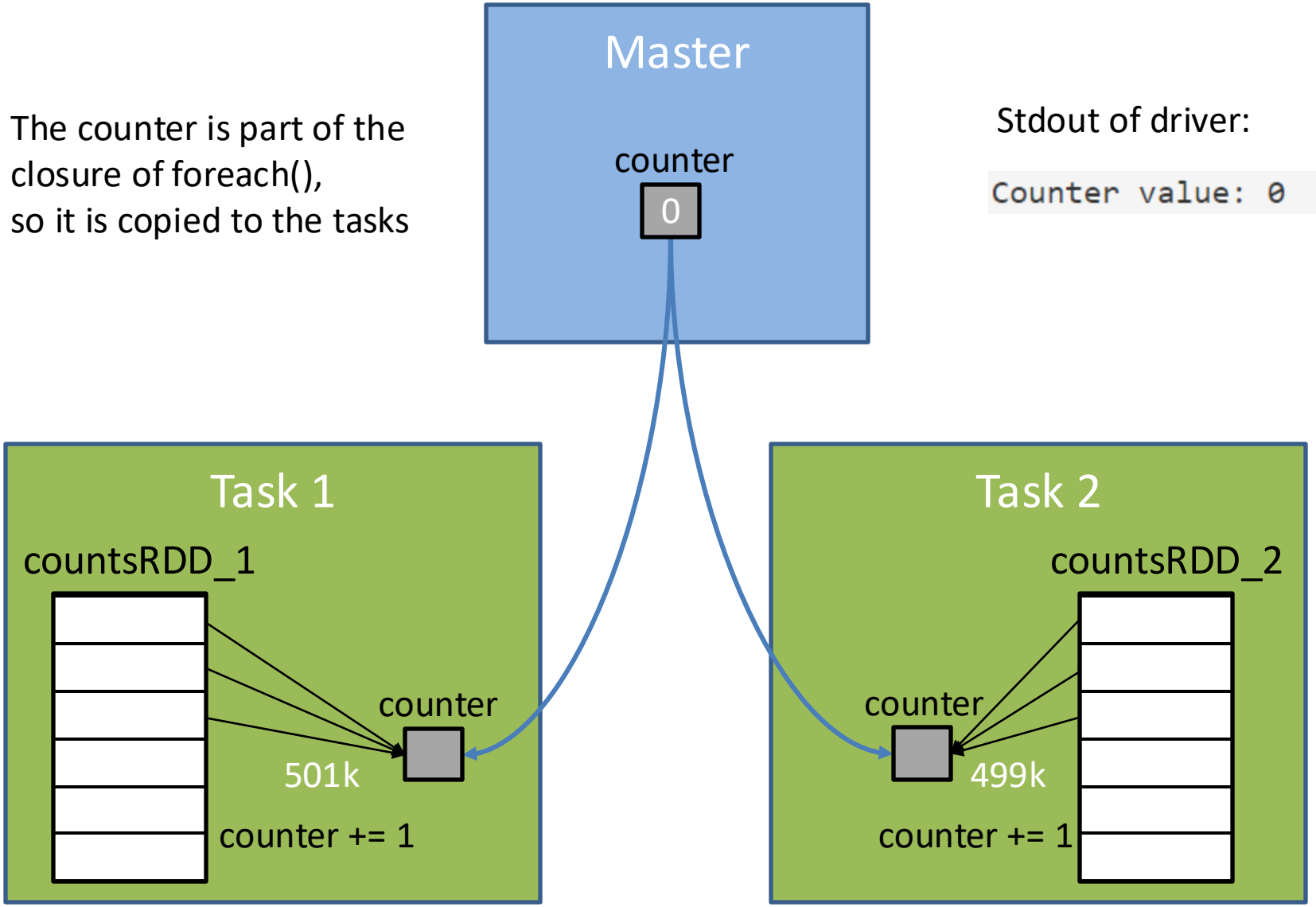
Incrementing a Counter

- Goal: count the number of rows in an RDD.
- What does the following program print?
  - Assume countsRDD has two partitions with sizes 501k and 499k records respectively.

```
var counter = 0
countsRDD.foreach(x => counter += 1)
println("Counter value: " + counter)
```

Incrementing a Counter

The counter is part of the closure of foreach(), so it is copied to the tasks



Stdout of driver:

```
Counter value: 0
```

Workaround

- Closures provide local copies to a task and hence cannot be used to update global state.

- To globally process updates from the tasks, use an Accumulator instead. It is designed to automatically pass its local updates back to the shared global copy in the master.

- Let us look at one more example.

Where Is The RDD Printed?

```
countsRDD.foreach(println)

countsRDD.collect().foreach(println)
```

Where Is The RDD Printed?

- The first foreach is called in the tasks (executors). Hence the println writes to the executor's stdout and the RDD content will not show up in the master's stdout.
- The second foreach collects the RDD on the master, then prints it there.

```
countsRDD.foreach(println)

countsRDD.collect().foreach(println)
```

Lessons Learned

- Make local copies of object members needed by a function, especially when the object is large and the relevant member is small.
- Use anonymous functions and static methods instead of methods in a class instance.
- Use Accumulators to implement global counters.
- An RDD must be collected if we want one machine to have access to all its data (be careful about memory constraints).

Next, we explore a bit more the lazy execution of Spark.

RDD Dependencies

- RDD lineage forms a DAG: RDDs are vertices, dependencies are edges. Each transformation appends a new vertex.

- **Narrow dependency**: one-to-one relationship between input and output partitions. No shuffle.
  - Examples: map, filter, flatMap, mapValues.

- **Wide dependency**: output partitions depend on multiple input partitions. Requires a shuffle.
  - Examples: reduceByKey, groupByKey, join.

- Wide dependencies define **stage boundaries** — Spark executes all narrow transformations within a stage as a single pipeline, then shuffles before the next stage.

- Debugging tip: use `println(myRDD.toDebugString)` to see the DAG. "ShuffleRDD" indicates a wide dependency.

Program Execution

- A Spark job is divided into stages, defined by points where shuffles occur.
- For each stage, the driver creates tasks and sends them to the executors.
- The executors write their intermediate results to local disk. Shuffling is performed by special *shuffle-map* tasks.
- The last stage returns its results to the driver, using special *results* tasks.

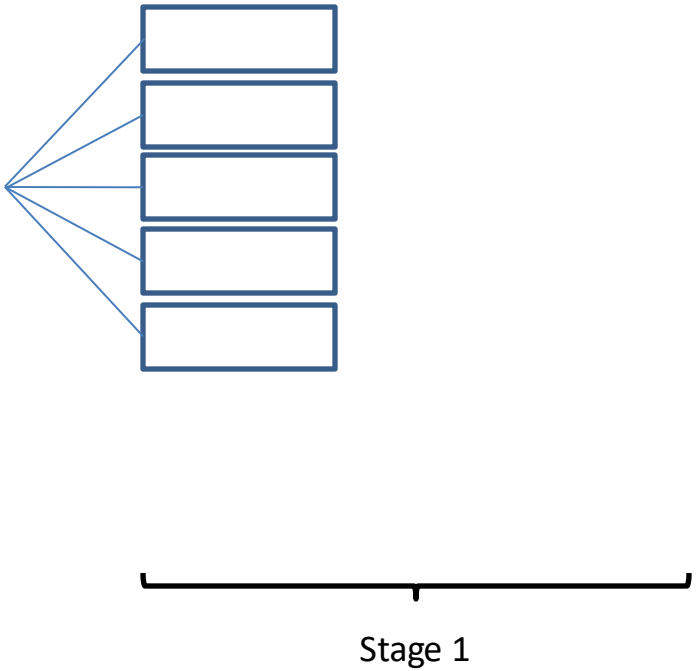
```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
```

RDD lineage:

parallelize

→ listRDD

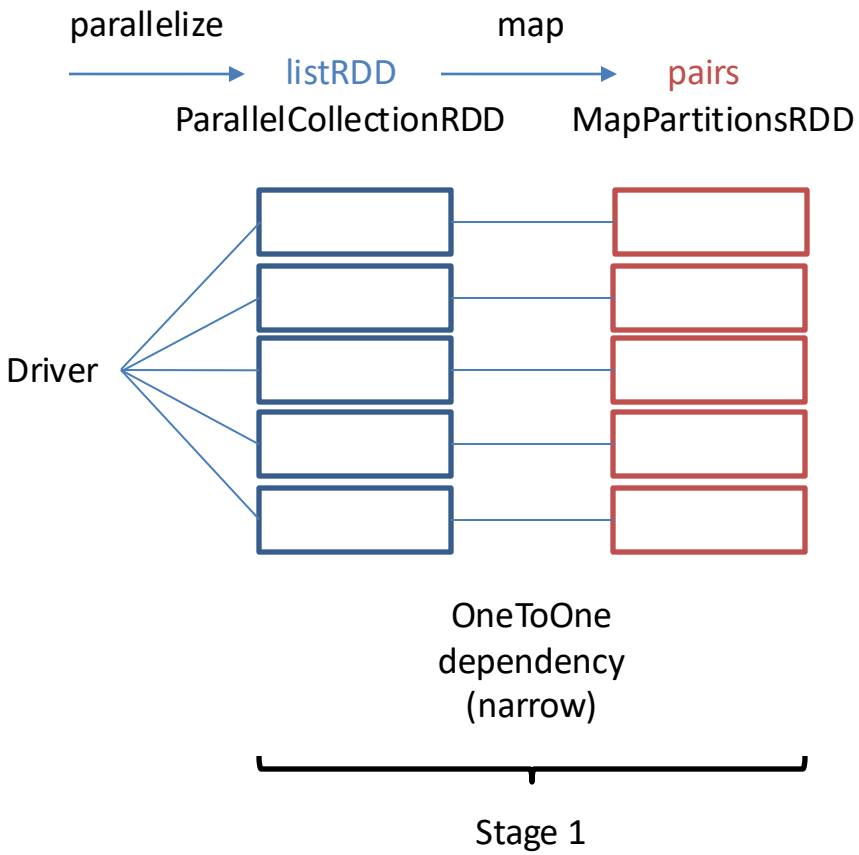
ParallelCollectionRDD



Example from on Zecevic/Bonaci book

```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
val pairs = listRDD.map(x => (x, x*x) )
```

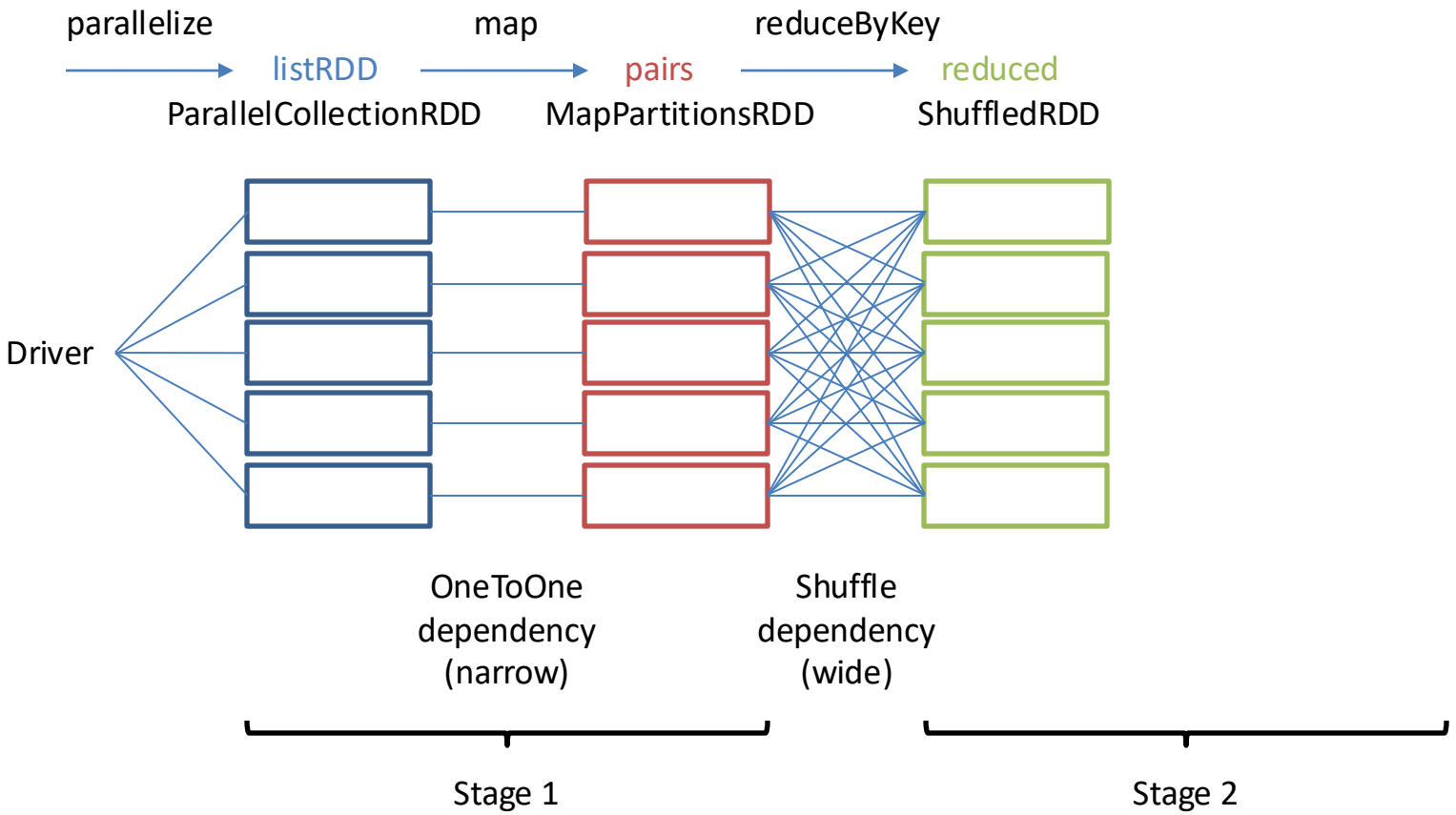
RDD lineage:



Example from on Zecevic/Bonaci book

```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
val pairs = listRDD.map(x => (x, x*x) )
val reduced = pairs.reduceByKey( (v1, v2) => (v1+v2) )
```

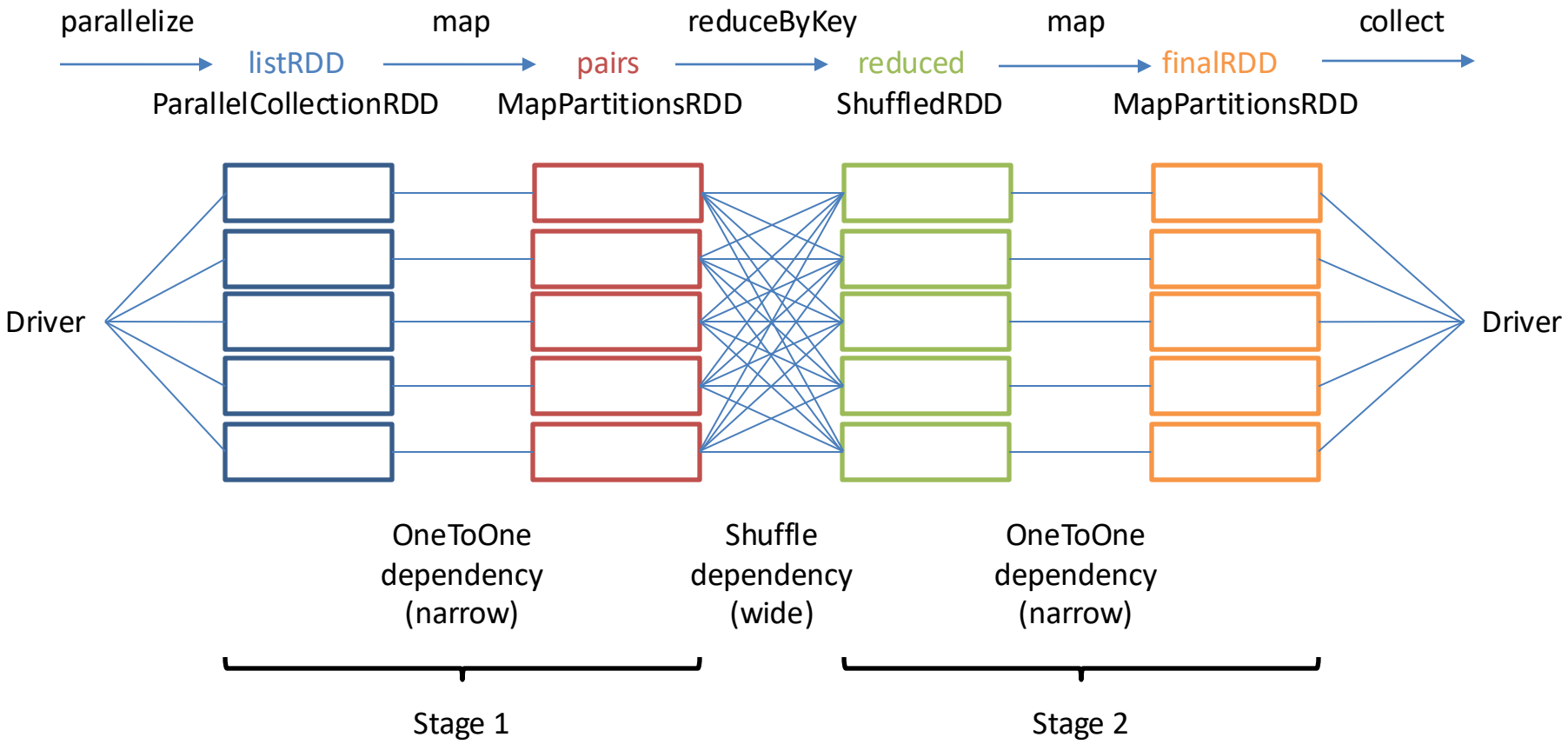
RDD lineage:



Example from on Zecevic/Bonaci book

```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
val pairs = listRDD.map(x => (x, x*x) )
val reduced = pairs.reduceByKey( (v1, v2) => (v1+v2) )
val finalRDD = reduced.mapPartitions( iter => iter.map({case(k, v) => "K="+k+", V="+v}) ).collect()
```

RDD lineage:



Example from on Zecevic/Bonaci book

Lineage

- Lineage Is the Recipe for an RDD

```
val numbers =
  sc.parallelize(Seq(1, 2, 3, 4), 2)

val even =
  numbers.filter(x => x % 2 == 0)

val squared =
  even.map(x => x * x)
```

Lineage

- Conceptual values:
  - numbers: [1, 2, 3, 4]
  - even: [2, 4]
  - squared: [4, 16]

- But before an action, even and squared are "recipes", not necessarily materialized collections.

- Lineage is the chain of transformations and dependencies Spark can use to compute or reconstruct an RDD.
  - RDD transformations are lazy, and Spark remembers the operations needed to derive each transformed RDD from its source.

Lineage

- An Action Executes the Lineage. If we write val count = squared.count()
- Spark follows: numbers -> filter -> map -> count
  - Result: 2
- A second action: val total = squared.sum(). It follows the lineage again: numbers -> filter -> map -> sum
  - Result: 20

Lineage

- The transformations form the lineage. The action consumes that lineage to produce a result.

- The action is not another transformed RDD in the chain.

PageRank Iterations

- Let us revisit the PageRank program.
- To solve the dangling-page issue, we needed a lookup() to find the rank of a dummy node.
- This is an action that triggers a job execution in Spark – one for each iteration.

```
for (i <- 1 to iters)
{
    // ...

    // Look up the weight of the dummy node in order to redistribute it
    val dummy_weight = ranksRDD.lookup(0).head

    // ...
}
```

PageRank Lineage

- The lineage of our RDD keeps growing with all the transformations of each new iteration.

```
- (4) MapPartitionsRDD[8] at map at PageRank.scala:71 []
 |  ShuffledRDD[7] at reduceByKey at PageRank.scala:67 []
 +-(4) MapPartitionsRDD[6] at flatMap at PageRank.scala:58 []
    |  MapPartitionsRDD[5] at join at PageRank.scala:57 []
    |  MapPartitionsRDD[4] at join at PageRank.scala:57 []
    |  CoGroupedRDD[3] at join at PageRank.scala:57 []
    |  ShuffledRDD[2] at groupByKey at PageRank.scala:46 []
    |      CachedPartitions: 4; MemorySize: 896.0 B; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
    +-(4) ParallelCollectionRDD[1] at parallelize at PageRank.scala:46 []
    +-(4) ParallelCollectionRDD[0] at parallelize at PageRank.scala:43 []
```

Iteration 1

```
(4) MapPartitionsRDD[14] at map at PageRank.scala:71 []
 |  ShuffledRDD[13] at reduceByKey at PageRank.scala:67 []
 +-(4) MapPartitionsRDD[12] at flatMap at PageRank.scala:58 []
    |  MapPartitionsRDD[11] at join at PageRank.scala:57 []
    |  MapPartitionsRDD[10] at join at PageRank.scala:57 []
    |  CoGroupedRDD[9] at join at PageRank.scala:57 []
    |  ShuffledRDD[2] at groupByKey at PageRank.scala:46 []
    |      CachedPartitions: 4; MemorySize: 896.0 B; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
    +-(4) ParallelCollectionRDD[1] at parallelize at PageRank.scala:46 []
    +-(4) MapPartitionsRDD[8] at map at PageRank.scala:71 []
       |  ShuffledRDD[7] at reduceByKey at PageRank.scala:67 []
       +-(4) MapPartitionsRDD[6] at flatMap at PageRank.scala:58 []
          |  MapPartitionsRDD[5] at join at PageRank.scala:57 []
          |  MapPartitionsRDD[4] at join at PageRank.scala:57 []
          |  CoGroupedRDD[3] at join at PageRank.scala:57 []
          |  ShuffledRDD[2] at groupByKey at PageRank.scala:46 []
          |      CachedPartitions: 4; MemorySize: 896.0 B; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
          +-(4) ParallelCollectionRDD[1] at parallelize at PageRank.scala:46 []
          +-(4) ParallelCollectionRDD[0] at parallelize at PageRank.scala:43 []
```

Iteration 2

PageRank Lineage Use

- Fault tolerance: if any RDD partition is lost, it can be recomputed using its lineage.

- Optimization: Ideally, the RDD of the previous iteration is stored so that the next iteration does not need to recompute it.
  - This in general is not guaranteed and is up to the Spark optimizer.

Use of cache()/persist()

- We can give a hint to Spark that certain RDDs should be kept in memory/disk.
- In this case, the log files explicitly tell us if the RDDs are reused.

```
graphRDD.cache()
ranksRDD.cache()
```

Logs

```
MemoryStore:54 - Block rdd_2_3 stored as values in memory
MemoryStore:54 - Block rdd_2_2 stored as values in memory
MemoryStore:54 - Block rdd_2_1 stored as values in memory
MemoryStore:54 - Block rdd_2_1 stored as values in memory
```

…

```
BlockManager:54 - Found block rdd_2_2 locally
BlockManager:54 - Found block rdd_2_3 locally
BlockManager:54 - Found block rdd_2_0 locally
BlockManager:54 - Found block rdd_2_1 locally
```

Detecting Lineage Recomputation

- What if we do not use cache() or persist()? Then the log files do not explicitly tell us if recomputation occurs. How can we tell if Spark reuses RDDs across iterations?
- One idea is to use a print statement inside the for-loop. The hope is that recomputation should print the same iteration number multiple times.
  - However, this idea will not work because printing is not a transformation, thus not part of the lineage.

```
for (i <- 1 to iters)
{
    println(s"Iteration ${i}")
    // ...

    // Look up the weight of the dummy node in order to redistribute it
    val dummy_weight = ranksRDD.lookup(0).head

    // ...
}
```

Will always be executed once per iteration.

Detecting Lineage Recomputation

- To detect if the lineage is recomputed, we can look for:
  - a growing number of tasks per iteration
  - increasing running time for jobs

Example run for 1M nodes, no cache()

```
DAGScheduler:54 - Job 0 finished: lookup at PageRank.scala:89, took 3.963390 s
DAGScheduler:54 - Job 1 finished: lookup at PageRank.scala:89, took 1.650899 s
DAGScheduler:54 - Job 2 finished: lookup at PageRank.scala:89, took 1.374739 s
DAGScheduler:54 - Job 3 finished: lookup at PageRank.scala:89, took 1.415387 s
```

The running time of the jobs stays more or less the same, which indicates that Spark does reuse the RDDs.

Detecting Lineage Recomputation

- Will Spark always be smart enough to reuse the previous RDDs? For PageRank most probably yes.

- Since our PageRank program performs shuffling in each iteration, the RDDs will be stored on disk even if there is not enough memory to cache them.

Summary

- Spark provides powerful functionality and makes it easy to write programs for distributed data-intensive computations in the cloud. In addition to core functionality, a variety of libraries have been added, e.g., for SQL, machine learning, graph and stream processing.
- Writing efficient Spark programs and identifying the root cause of poor performance or memory errors requires deeper understanding of Spark's lineage, lazy evaluation, and closure.
  - In addition, one must be aware where data lives and where functions are executed (master vs. tasks/executors).

References

- Biswanath Panda and Joshua S. Herbach and Sugato Basu and Roberto J. Bayardo. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. Proc. Int. Conf. on Very Large Data Bases (VLDB), 2009
  - https://scholar.google.com/scholar?cluster=11753975382054642310&hl=en&as_sdt=0,22