DASCI

Instituto Andaluz Interuniversitario
en Ciencia de Datos e
Inteligencia Computacional

Ciencia de Datos a través del Big Data

Diego García (djgarcia@ugr.es)
Isaac Triguero (isaaktriguero@ugr.es)

Financiado por
la Unión Europea
NextGenerationEU

GOBIERNO
DE ESPAÑA

MINISTERIO
PARA LA TRANSFORMACIÓN DIGITAL
Y DE LA FUNCIÓN PÚBLICA

SECRETARÍA DE ESTADO
DE DIGITALIZACIÓN
E INTELIGENCIA ARTIFICIAL

Plan de
Recuperación,
Transformación
y Resiliencia

UNIVERSIDAD
DE GRANADA

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

REPSOL

CAMBRIDGE UNIVERSITY PRESS

Chapter 4

Spark II

Learning Outcomes

- The importance of caching for efficient Big Data processing [KU]
- Extending Map Reduce: The concept of distributed datasets (RDDs) for Big Data [KU]
- How to operate with RDDs: lazy transformation vs. actions [KU, IS, PPS]
- How a Spark application is distributed guaranteeing fault-tolerance and transparency [KU]
- How to use Spark in practice for problem solving [IS, PPS]

Objective for today

- Spark
  - Key-value operations with Spark RDDs
  - Caching RDDs
  - Advanced concepts: shared variables, partitions and numeric RDDs
  - Internal Working

RDDs – Key-value Transformations

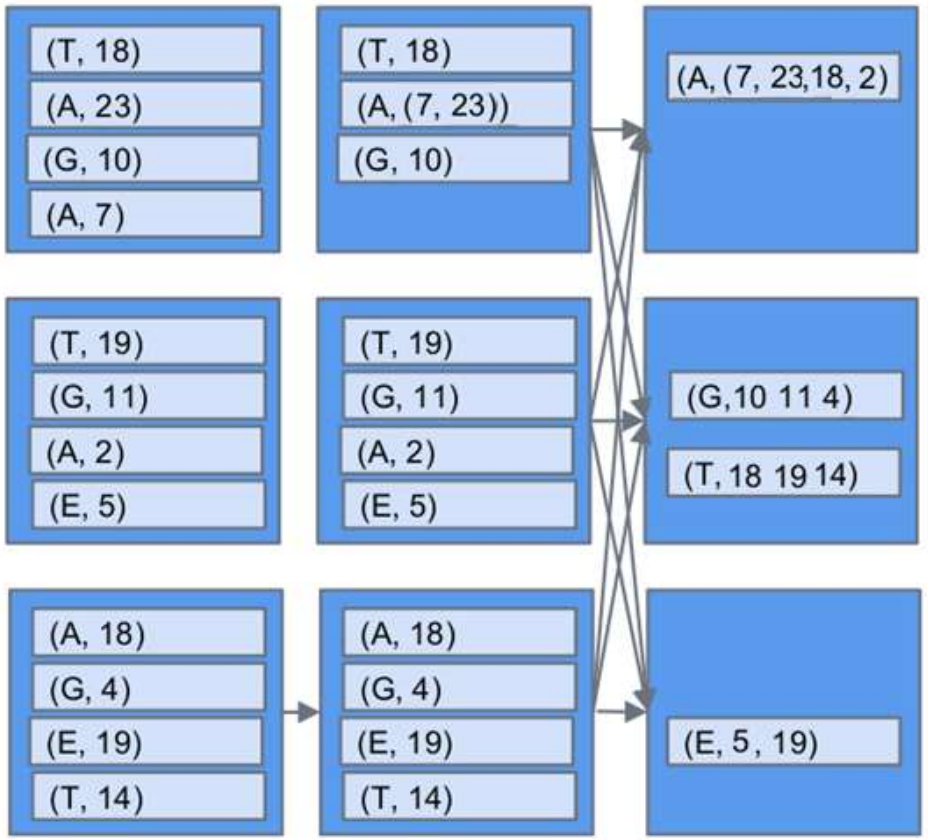
CAMBRIDGE UNIVERSITY PRESS

- **Key-value RDDs**
  - Like Hadoop MapReduce, Spark supports key-value pairs
  - Each element of an RDD need to be a tuple `(key, value)`

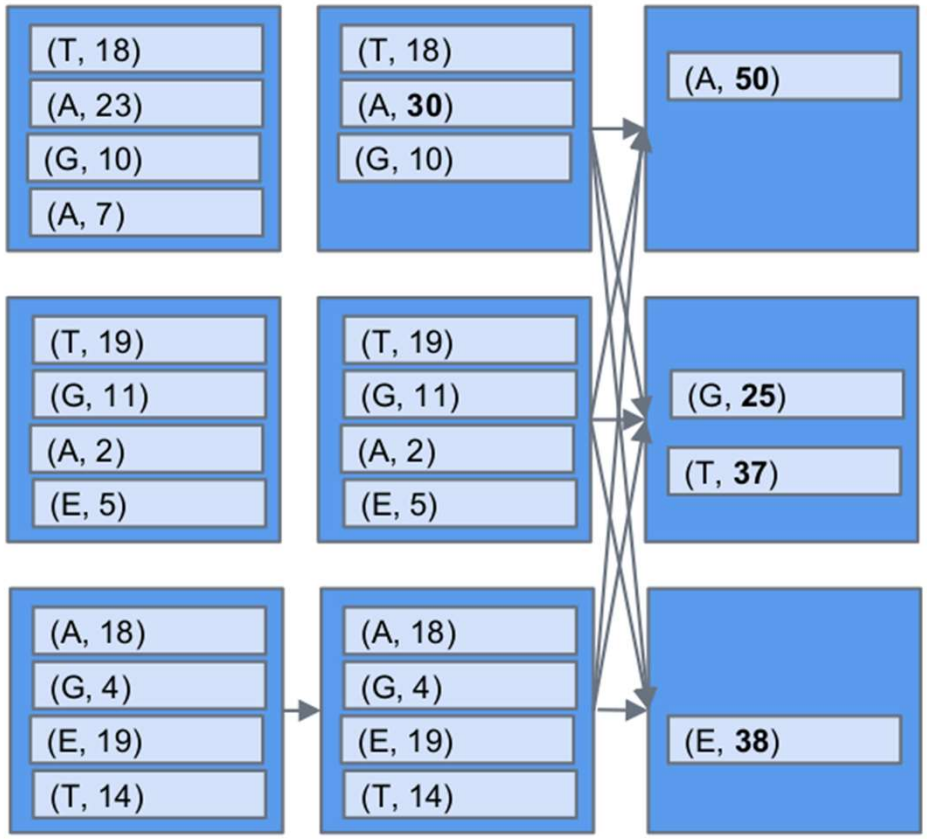
```
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```

Transformation	Meaning
`groupByKey()`	Returns a new RDD of tuples `(k, iterable(v))`. **Warning: This requires a Shuffle!**
`reduceByKey(`*`func`*`)`	Returns a new RDD of tuples `(k, v)` where the values of each key *k* are aggregated using using the function `func`. This function should take two elements of type *v* and return the same type.
`sortByKey([`*`ascending`*`])`	Returns a new RDD of tuples `(k, v)` that has been sorted (by default, in ascending order).

RDDs – Key-value Transformations: `groupByKey()`



RDDs – Key-value Transformations: `reduceByKey(x+y)`



RDDs – Key-value Transformations

CAMBRIDGE UNIVERSITY PRESS

- **Key-value transformations**

```
>>> rdd = sc.parallelize([(1, 2), (3, 4), (3, 6)])

>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1, 2), (3, 4), (3, 6)] → [(1, 2), (3, 10)]


>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])

>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] → [(1,'a'), (1,'b'), (2,'c')]
```

RDDs – Join-like SQL Transformations

CAMBRIDGE UNIVERSITY PRESS

- **Join-like SQL Transformations**
  - SQL joins allow one to combine tow tables based on a related column
    - In RDDs, keys are used for joining
    - Differences reside in the elements not having matching keys

Transformation	Meaning
`join(rdd)`	Inner join between RDDs, where the key must be present in both RDDs.
`leftOuterJoin(rdd)`	Joins the elements of two RDDs where the key must be present in the second RDD.
`rightOuterJoin(rdd)`	Joins the elements of two RDDs where the key must be present in the first RDD.
`fullOuterJoin(rdd)`	Joins the elements of two RDDs where the key must be present in any of the two RDDs.

RDDs – Join-like SQL Transformations

CAMBRIDGE UNIVERSITY PRESS

- **Join-like SQL Transformations**

```
>>> people = sc.parallelize([("Lam", 30), ("Direnc", 32),\
    ("Rebecca", 25), ("Edwina", 24)])
>>> hobbies = sc.parallelize([
    ("Lam", ["Triathlon", "Running", "Cycling"]),
    ("Direnc", ["Lifting", "Running", "Reading"]),
    ("Rebecca", ["Singing", "Dancing"]),
    ("Edwina", ["Running", "Music"])])
```

```
>>> people.join(hobbies).collect()
[('Direnc', (32, ['Lifting', 'Running', 'Reading'])),
('Lam', (30, ['Triathlon', 'Running', 'Cycling'])),
('Edwina', (24, ['Running', 'Music'])),
('Rebecca', (25, ['Singing', 'Dancing']))]
```

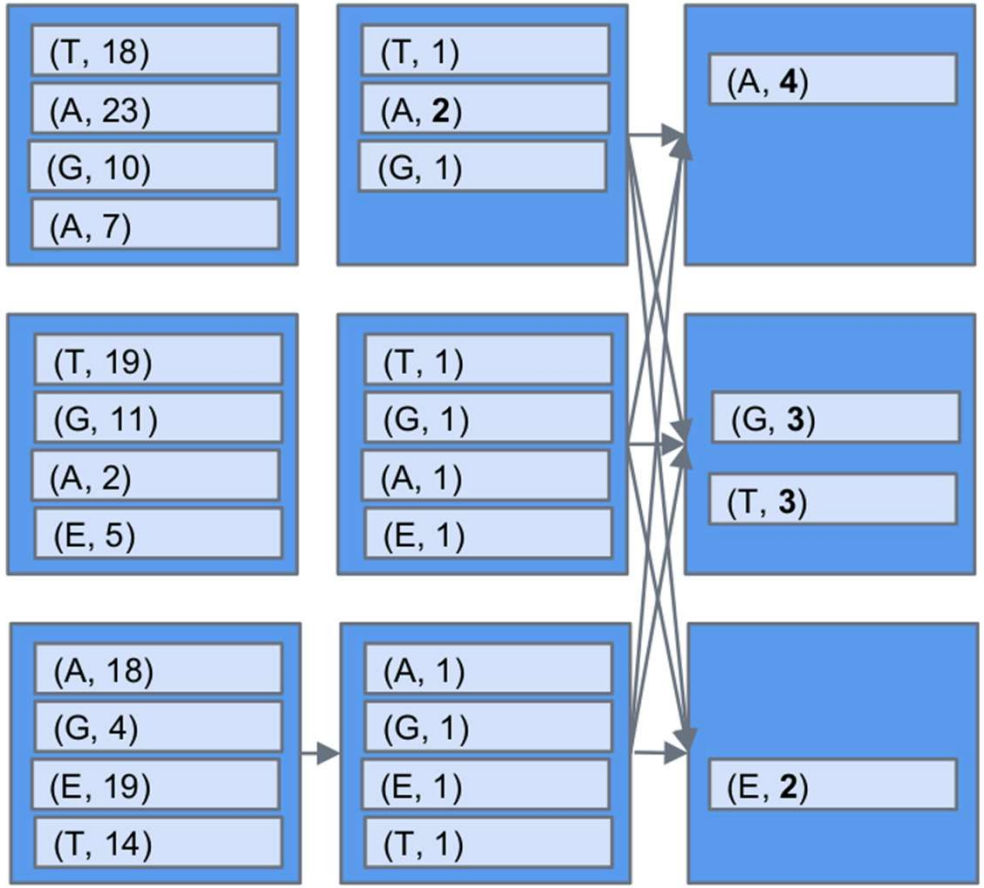
RDDs – Key-value Actions

CAMBRIDGE UNIVERSITY PRESS

- **Key-value Actions**
  - Additional actions for key-value RDDs

Transformation	Meaning
`countByKey()`	Counts the number of elements for each key. Returns a dictionary.
`collectAsMap()`	Collect the RDD as a dictionary, but it only provides one of the values.
`lookup(key)`	Return the value associated to a given key.

RDDs – Key-value Actions: countByKey()



Source: Dirk Van den Poel. Spark: The new kid on the block (2014)

Caution! This is an action!
The result must fit into the driver

RDDs – Key-value Actions

CAMBRIDGE UNIVERSITY PRESS

- **Key-value actions**

```
>>> rdd = sc.parallelize([(1, 2), (3, 4), (3, 6)])

>>> rdd.countByKey()
Value: {1: 1, 3: 2}

>>> rdd.collectAsMap()
Value: {1: 2, 3: 6}

>>> rdd.lookup(3)
Value: [4, 6]
```

RDDs – Caching

- **Caching RDDs doesn't happen by default!!**

```
lines = sc.textFile("...", 4)
comments = lines.filter(condition)
print(lines.count(), comments.count())
```

Spark recompute the `lines` RDD:

- Reading the data **AGAIN**
- Perform the addition for each partition
- Combine intermediate results in the driver

- **Solution**: Cache the `lines` RDD

```
lines = sc.textFile("...", 4)
lines.cache()
comments = lines.filter(condition)
print(lines.count(), comments.count())
```

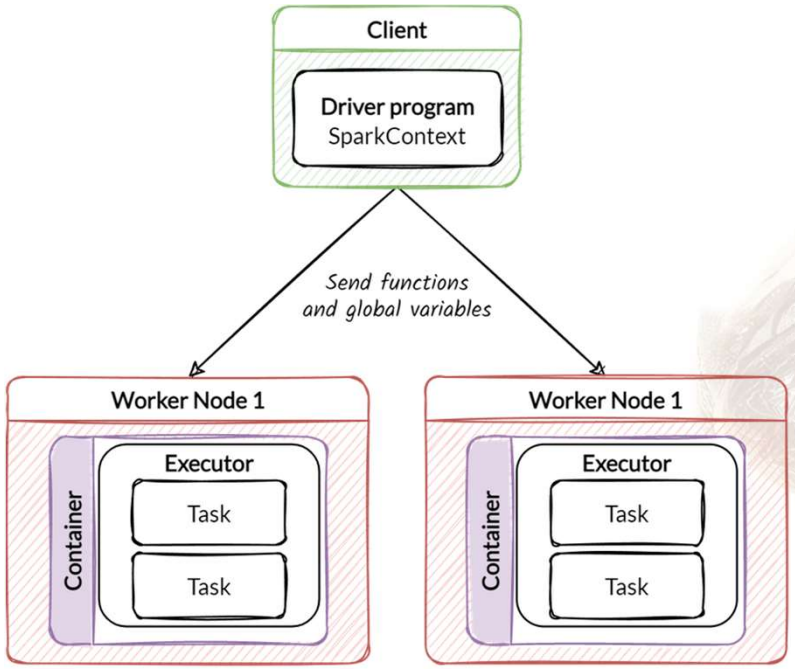
RDDs – Caching

- **Caching RDDs**
  - There are different persistence/caching levels. E.g.:
    - Memory Only
    - Memory and Disk
    - Disk Only
    - …
- **In Python, all objects are serialized with the pickle library**

Advanced Concepts – Shared Variables

CAMBRIDGE UNIVERSITY PRESS

- **Shared variables in Spark**
  - Spark sends **functions and global variables** to all executors for each task, and there isn't communication among executors
  - Any changes made in global variables by executors are not visible in the driver!
- This might be a problem for iterative jobs that share big data structures
  - Inefficient to send large sized collections to each worker



Spark provides two advanced variables

- **Broadcast variables**:
  - **Read-only** variables that are sent efficiently to all executors (cached)
  - Stored in the workers, so that, they can be used by one or more Spark operations. It is only sent once, not for each task.
- **Accumulators**:
  - Aggregate values from the executors in the driver
  - Only the driver can access the values of these variables
  - For the tasks, accumulators are **written-only**
  - You can only add values; typically used to implement efficient counters and parallel additions

Advanced Concepts – Broadcast Example

CAMBRIDGE UNIVERSITY PRESS

- Imagine you have a big (yet possible to store in main memory without parallelization) look-up table

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> look_up_table = {1: "a", 2: "b", 3: "c", 4: "d"}
```

Assume that this is VERY BIG!

```
# As a global variable, it is sent to the workers for each task!
>>> rdd.map(lambda v: look_up_table[v]).collect()
['a', 'b', 'c', 'd']
```

Without broadcast
Inefficient

```
# As a broadcast variable, it is only sent once!
>>> look_up_table_bc = sc.broadcast(look_up_table)
>>> rdd.map(lambda v: look_up_table_bc.value[v]).collect()
['a', 'b', 'c', 'd']
```

Broadcast variable

Using the broadcast variable
Same result, but **efficient** distribution

Advanced Concepts – Accumulators Example

- Using accumulators to count how many times a function is run

```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])

def f(x):
    global accum
    accum += 1

>>> rdd.foreach(f)

>>> accum.value
4
```

Accumulators work well with actions

Advanced Concepts – Accumulators Example

- Counting empty rows in a file

```
>>> quixote_rdd = sc.textFile("data/quixote.txt")
>>> blank_lines = sc.accumulator(0)

def extract_words_blanklines(line):
    global blank_lines
    if line == "":
        blank_lines += 1
    return line.split(" ")

>>> words_quixote = quixote_rdd.flatMap(extract_words_blanklines)

>>> words_quixote.count()
437863

>>> blank_lines.value
6820
```

Will the value of `blank_lines` always correct?

Accumulators on Transformations might not be ideal!

Advanced Concepts – Partitions

CAMBRIDGE UNIVERSITY PRESS

- **Working with partitions**
  - Sometimes we may want to apply an operations to all elements of a partition at once (e.g., divide-and-conquer for ML or for efficiency)

Transformation	Meaning
`mapPartitions(func)`	Applies the function `func` to each partition of the RDD. `func` receives an iterator and returns another iterator that can be of a different type.
`mapPartitionsWithIndex(func)`	Applies the function `func` to each partition of the RDD. `func` receives a tuple (integer, iterator), where the integer represents the index of the partition, and iterator contains all the elements of the partitions.
`foreachPartition(func)`	Applies the function `func` to each partition of the RDD but it doesn't return anything. `func` receives an iterator and returns nothing.

Advanced Concepts – Partitions

- **Task**: Computing the average with `map()` vs. `mapPartitions()`
  - With `map()`

```
>>> rdd = sc.parallelize([1, 2, 3, 4, 5, 6, 7])

>>> sum_count = rdd.map(lambda num: (num, 1))\
                   .reduce(lambda x, y: (x[0] + y[0], x[1] + y[1]))

>>> sum_count
(28, 7)

>>> sum_count[0] / sum_count[1]
4.0
```

Advanced Concepts – Partitions

- **Task**: Computing the average with `map()` vs. `mapPartitions()`
  - With `mapPartitions()`

```
def partition_counter(nums):
    sum_count = [0, 0]
    for num in nums:
        sum_count[0] += num
        sum_count[1] += 1
    return [sum_count]
```

Pre-computing may save a lot in some cases!

```
>>> rdd.mapPartitions(partition_counter)\
       .reduce(lambda x, y: (x[0] + y[0], x[1] + y[1]))
(28, 7)
```

Advanced Concepts – Numeric RDDs

- Spark provides some built-in methods to generate some descriptive statistics of a numeric RDDs.
  - E.g., `stats()`, `count()`, `mean()`, `max()`, etc.

```
>>> a = sc.parallelize([1, 2, 3, 4, 5, 6, 7])
>>> stat = a.stats()
>>> stat

Value: (count: 7, mean: 4.0, stdev: 2.0, max: 7.0, min: 1.0)

>>> a.count()
Value: 7

>>> a.mean()
Value: 4.0
```

Using **`stats()` is more efficient** than doing `mean()` and `std()` separately
Requires a single **pass through the RDD**

Advanced Concepts – Read/Write from/to files

- **Load and save data in Spark**
  - We use **`sc.textFile`** to read from a filesystem, a single file or and entire directory:

```
input = sc.textFile("file:///home/pszit/spark/")
```

WARNING: All nodes should be able to access the file!

  - Using **`wholeTextFiles()`** we get the filenames as key, and content as values:

```
input = sc.wholeTextFiles("file:///home/pszit/spark/")
```

  - **Writing text files**

```
result.saveAsTextFile(outfile)
```

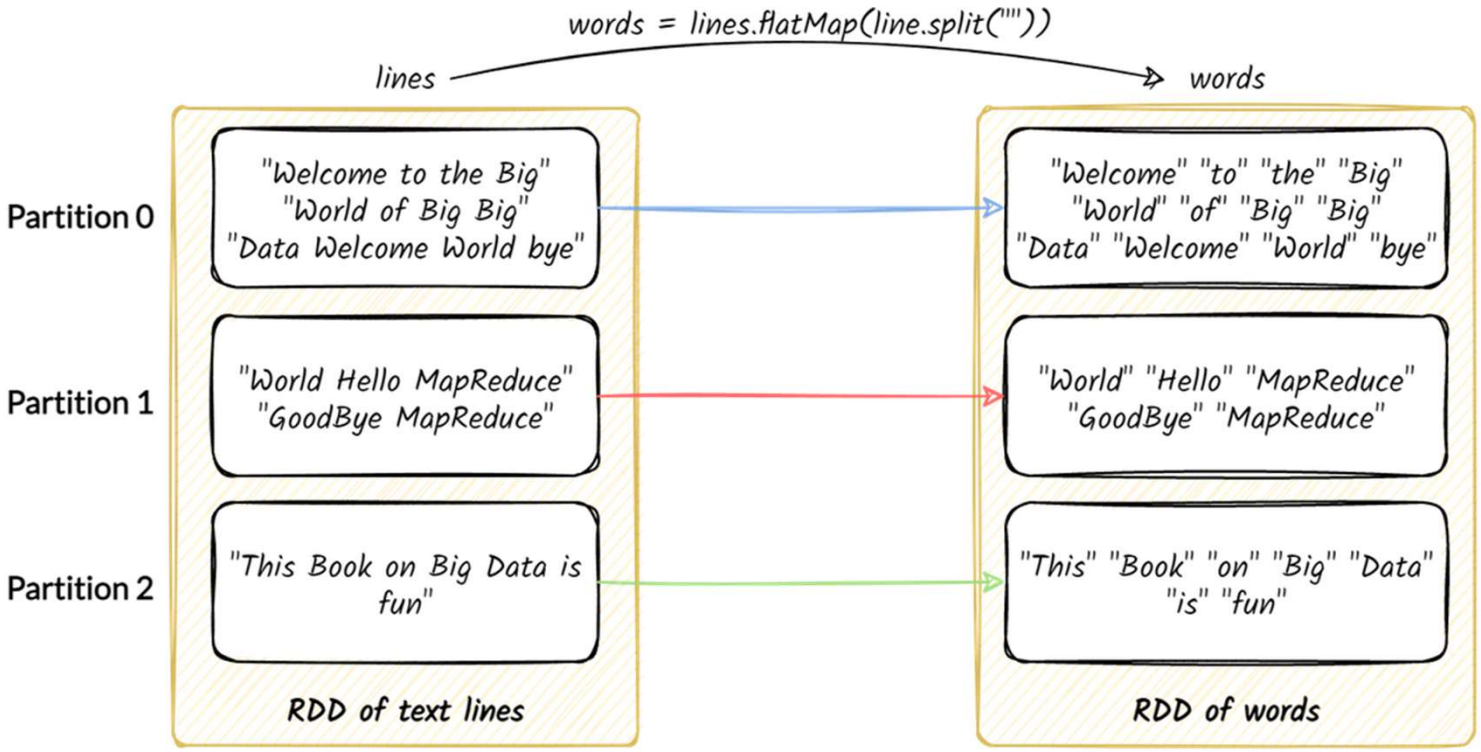
  - The output is not a file, but a directory!

Internal Working

- **Two types of transformations**
  - **Narrow** dependencies
    - Each partition will contribute to only one output partition
  - **Wide** dependencies
    - Input partitions contribute to many output partitions
    - There will be a shuffle, exchanging partitions/data across a cluster
    - Define Spark Stages
    - E.g., `sort`, `reduceByKey`, `groupByKey`, `join`

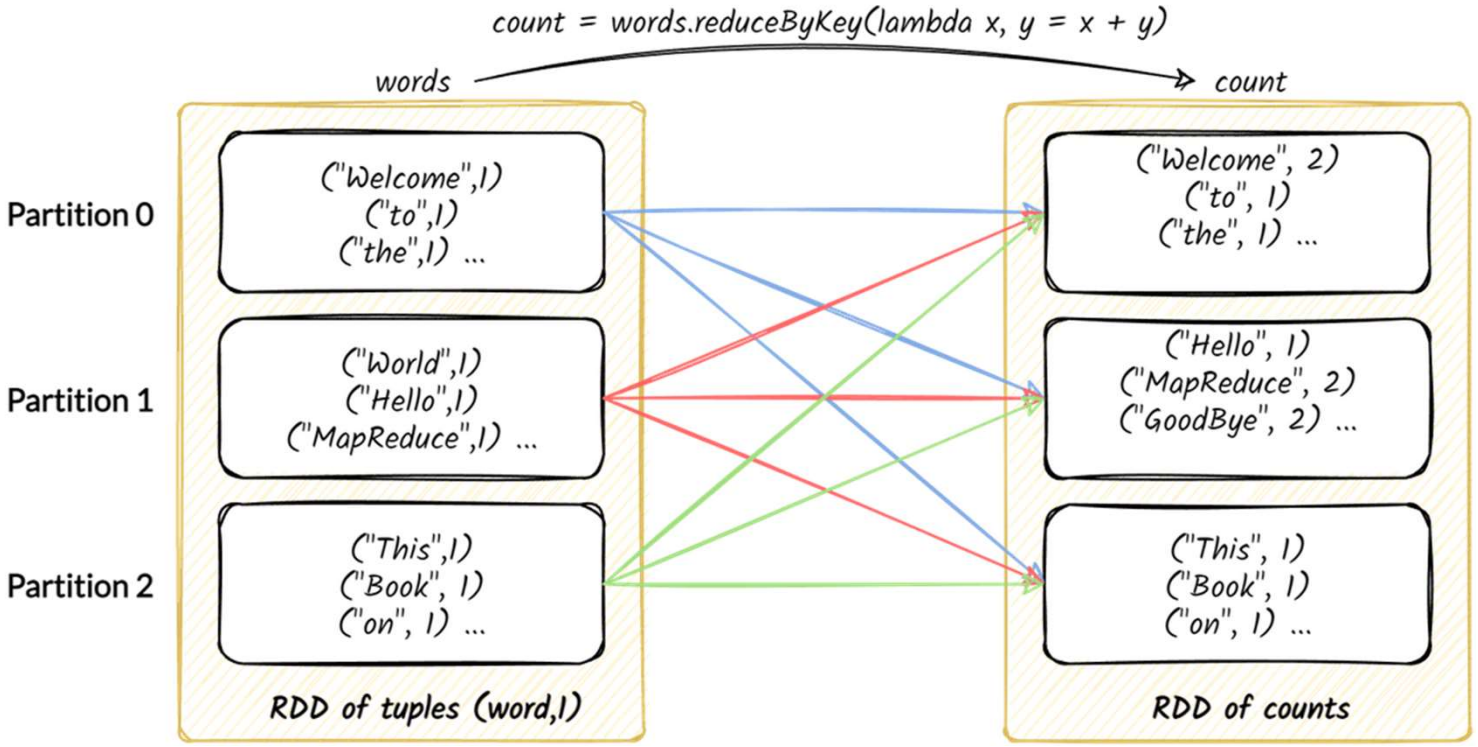
Internal Working

- **Two types of transformations**
  - **Narrow** dependencies



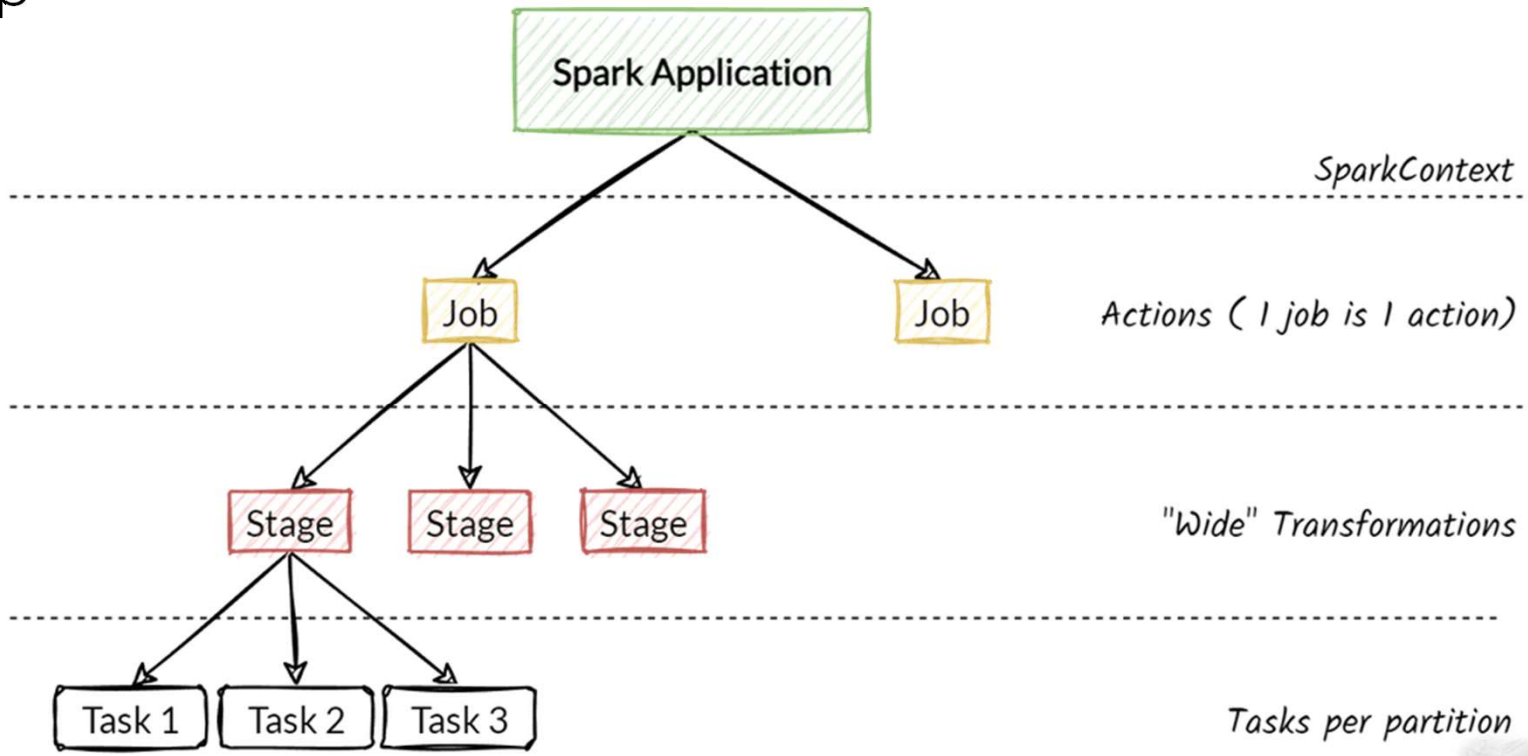
Internal Working

- **Two types of transformations**
  - **Wide** dependencies



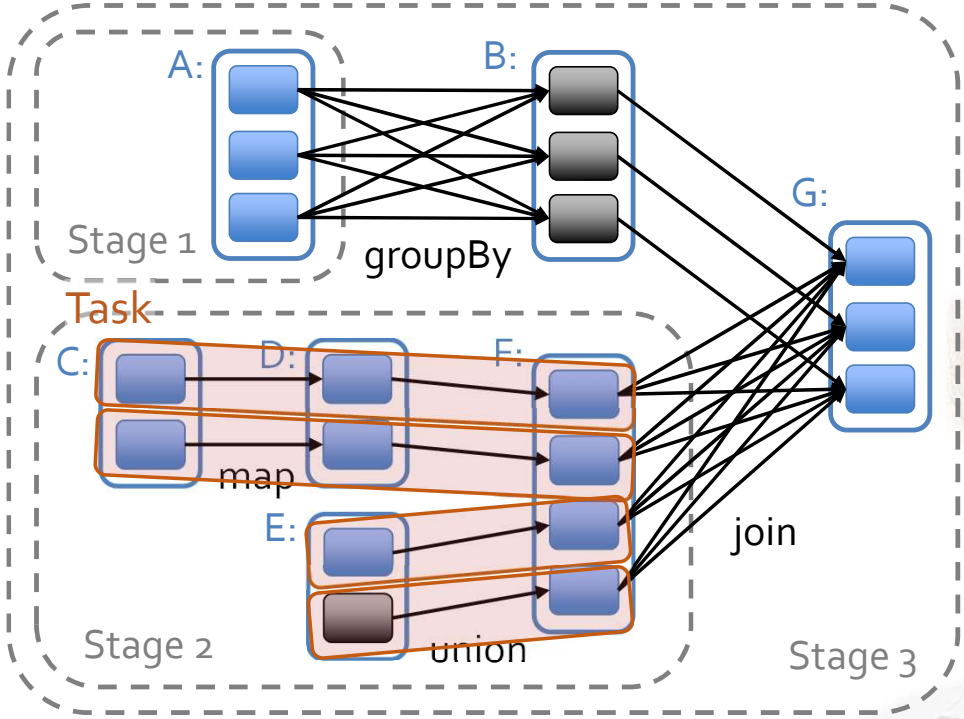
Anatomy of a Spark Application

- Sparks creates a Directed Acyclic Graph to optimise the execution of an App



- **Anatomy of a Spark Application**
  - Spark joins operations of one stage as a single task

```
input = sc.parallelize([1, 2, 3, 4, 5,
6, 7, 8, 9])
input\
    # Stage 1
    .filter(lambda x: x < 2)\
    .map(lambda x: (x, x))\
    # Stage 2
    .groupByKey() \
    .map(lambda (x, y): (sum(y), x))\
    # Stage 3
    .sortByKey()\
    .count()
```



Source: Spark Dev Meetup 2012

Challenge

- Write the WordCount using Apache Spark
  - Read a big text file
  - First simply using `map` and `reduceByKey`, then alternatively try:
    - `groupByKey`
    - `mapPartitions`
    - Which one you like the most?
  - Store the output as a file

Take-home message

- **Key-value transformations** and actions
- Differentiate properly
  - `map()` vs. `flatMap()`
  - `groupByKey()` vs. `reduceByKey()`
- Remember to **cache RDDs** that are reused
- Use **broadcast** variables to share large variables
- Internal working: jobs, stages and tasks
  - Differentiate between **narrow** and **wide** transformations

What's next?

- Spark SQL!

To Learn More

- **Spark and RDDs**
  - Learning Spark (Damji et al., 2020)
  - Spark RDD documentaiton
  - Apache Spark website
- **Other Big Data frameworks**
  - Apache Flink – particularly for data streaming
  - Dask – parallel computing with Python

CAMBRIDGE UNIVERSITY PRESS

Chapter 4

Spark II