Big Data Platforms (5 ECTS)
DATA14003
Lecture 2

Keijo Heljanko

Department of Computer Science
University of Helsinki
keijo.heljanko@helsinki.fi

9.9-2021

Apache Spark

Apache Spark

- Distributed programming framework for Big Data processing
- Based on functional programming, with functionality extending the Google MapReduce framework
- Implements distributed Scala collections like interfaces for Scala, Java, Python, and R
- Implemented on top of the Akka actor framework
- Original paper:
  Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012: 15-28

Google MapReduce

- A scalable batch processing framework developed at Google for computing the Web index that predates Apache Spark
- Still in very widespread production use
- When dealing with Big Data (a substantial portion of the Internet in the case of Google!), the only viable option is to use servers and hard disks in parallel
- Some of the challenges for storage is coming from Web services to store and distribute pictures and videos
- We need a system that can effectively utilize hard disk parallelism and both hide hard disk and computer failures from the programmer

Google MapReduce (cnt.)

- MapReduce is tailored for batch processing with hundreds to thousands of machines running in parallel, typical job runtimes are from minutes to hours
- As an added bonus we would like to get increased programmer productivity compared to each programmer developing their own tools for utilizing hard disk parallelism

Google MapReduce (cnt.)

- The MapReduce framework takes care of all issues related to parallelization, synchronization, load balancing, and fault tolerance. All these details are hidden from the programmer
- The system needs to be linearly scalable to thousands of computers working in parallel. The only way to do this is to have a very restricted programming model where the communication between computers happens in a carefully controlled fashion
- Apache Hadoop is an open source MapReduce implementation that has been used by Yahoo!, Facebook, and Twitter

MapReduce and Functional Programming

- Based on the functional programming in the large:
  - User is only allowed to write side-effect free functions "**Map**" and "**Reduce**"
  - Re-execution is used for fault tolerance. If a computer executing a Map or a Reduce task fails to produce a result due to hardware failure, the task will be re-executed on another computer
  - Side effects in functions would make this impossible, as one could not re-create the environment in which the original task executed
  - One just needs fault tolerant storage of task inputs
  - MapReduce uses a fault tolerant distributed filesystem to store compute job inputs and outputs
  - The map and reduce functions themselves are usually written in a standard imperative programming language, typically Java or Python

Why No Side-Effects?

- Side-effect free programs will produce the same output irregardless of the number of computers used by MapReduce
- Running the code on one machine for debugging purposes produces the same results as running the same code in parallel
- It is easy to introduce side-effect to MapReduce programs as the framework does not enforce a strict programming methodology. However, the behavior of such programs is undefined by the framework, and should therefore be avoided.

Yahoo! MapReduce Tutorial

- We use a Figures from a MapReduce tutorial of Yahoo! Developer [YDN-MR] available from:

  `https://web.archive.org/web/20181215123514/https://developer.yahoo.com/hadoop/tutorial/module4.html`

- In functional programming, two list processing concepts are used
  - Mapping a list with a function
  - Reducing a list with a function

Mapping a List

Mapping a list applies the mapping function to each list element (in parallel) and outputs the list of mapped list elements:

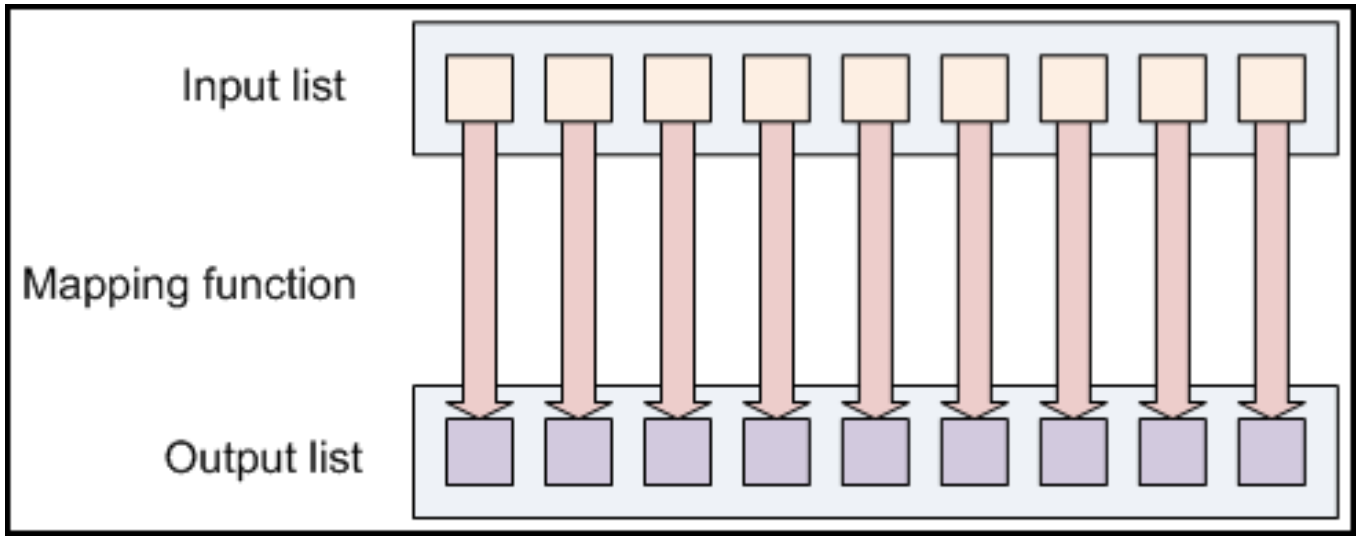


Figure: Mapping a List with a Map Function, Figure 4.1 from [YDN-MR]

Reducing a List

Reducing a list iterates over a list and produces an output created by the reduce function:

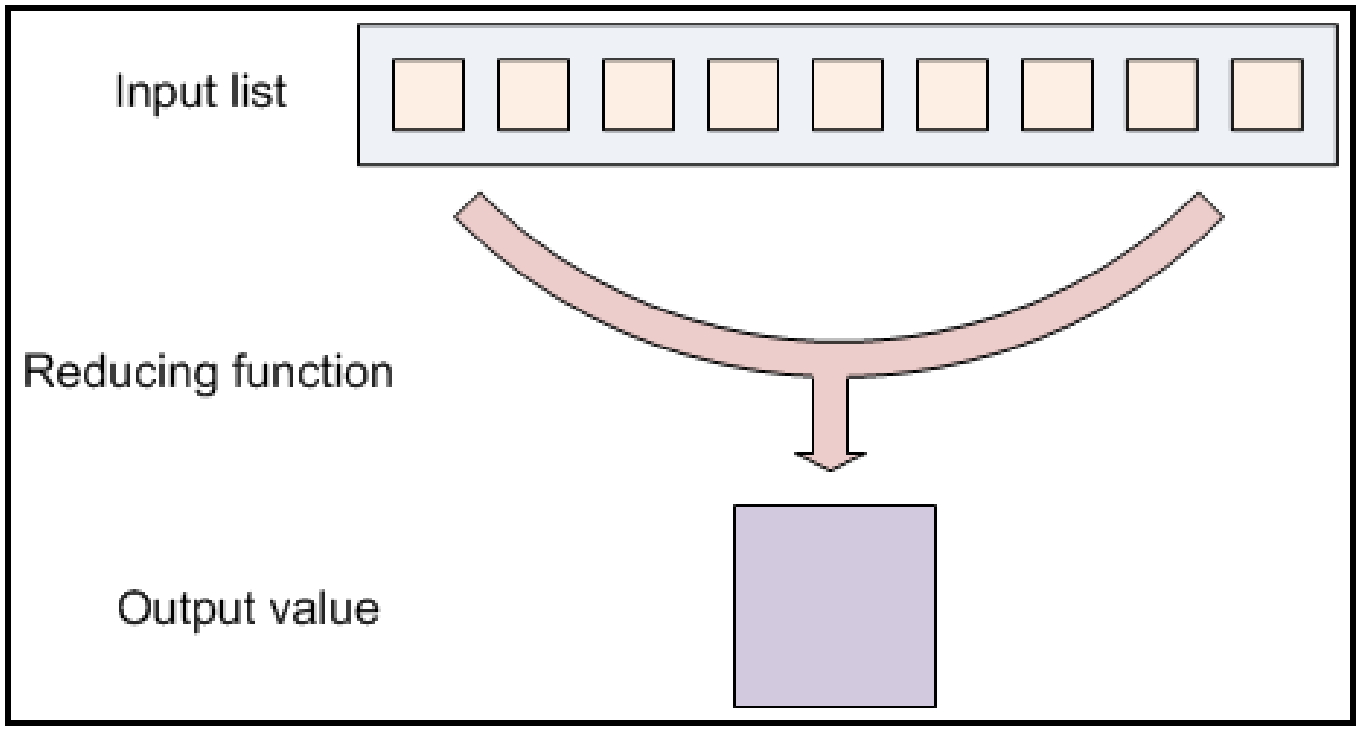


Figure: Reducing a List with a Reduce Function, Figure 4.2 from [YDN-MR]

Grouping Map Output by Key to Reducer

In MapReduce the map function outputs `(key, value)`-pairs. The MapReduce framework groups map outputs by key, and gives each reduce function instance `(key, (..., list of values, ...))` pair as input. Note: Each list of values having the same key can be processed in parallel:

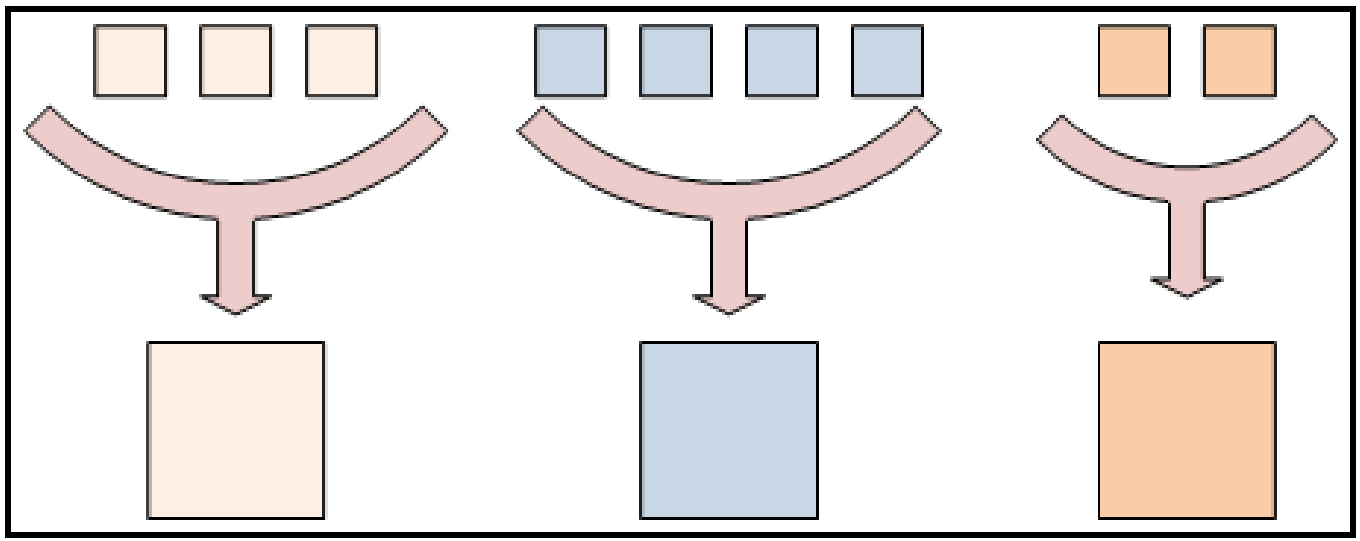


Figure: Keys Divide Map Output to Reducers, Figure 4.3 from [YDN-MR]

MapReduce Data Flow

Practical MapReduce systems split input data into large (64MB+) blocks fed to user defined map functions:

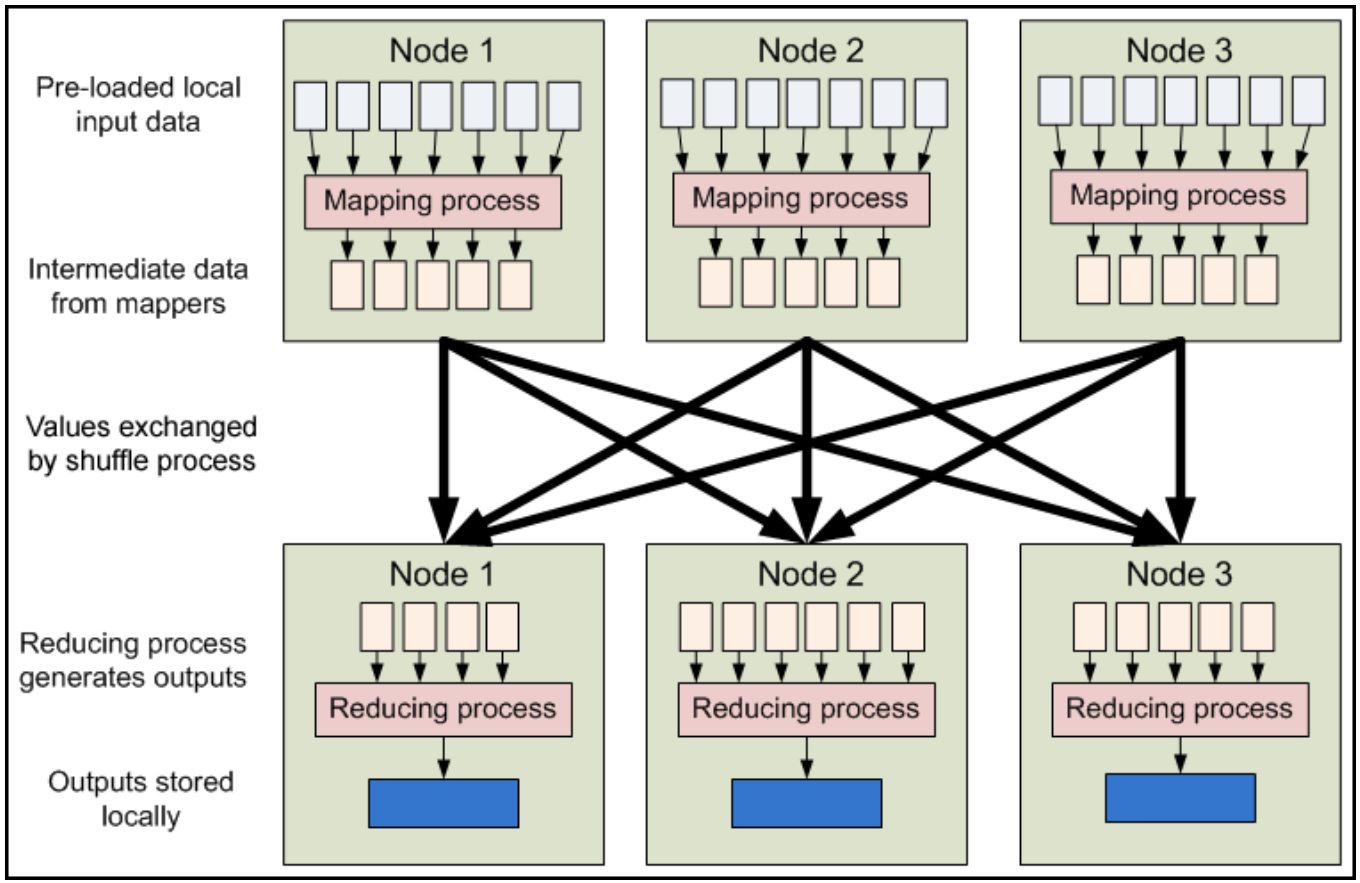


Figure: High Level MapReduce Dataflow, Figure 4.4 from [YDN-MR]

Recap: Map and Reduce Functions

- The framework only allows a user to write two functions: a "**Map**" function and a "**Reduce**" function
- The **Map**-function is fed blocks of data (block size 64-128MB), and it produces `(key, value)` -pairs
- The framework groups all values with the same key to a `(key, (..., list of values, ...))` format, and these are then fed to the **Reduce** function
- The grouping can be achieved by hashing by the key by the so called Shuffle phase
- Master process takes care of the scheduling and fault tolerance by re-executing Mappers or Reducers

MapReduce Diagram

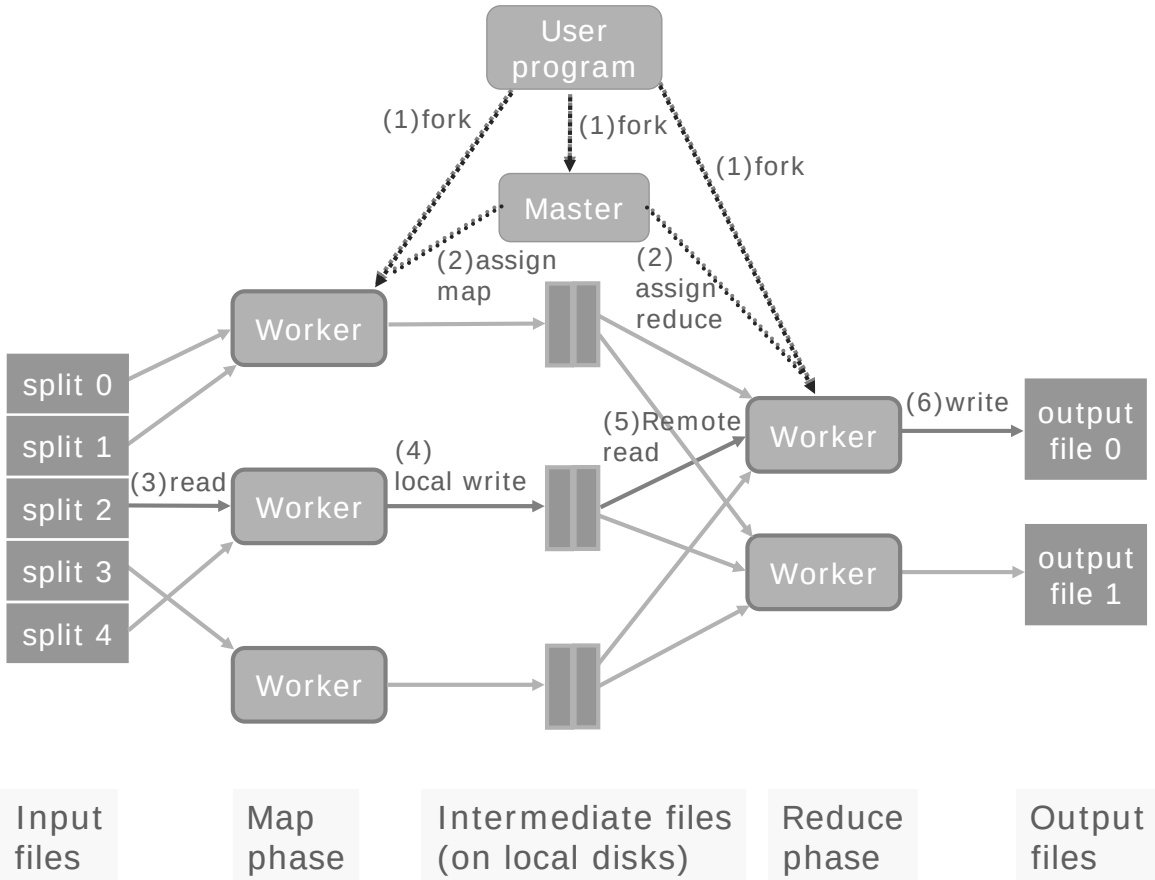


Figure: J. Dean and S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004

Example: Word Count

- Classic word count example from the Hadoop MapReduce tutorial:
  `http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`
- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Example: Word Count (cnt.)

- Consider a Map function that reads in words one a time, and outputs `(word, 1)` for each parsed input word
- The Map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Example: Word Count (cnt.)

- The Shuffle phase between Map and Reduce phases creates a list of values associated with each key
- The Shuffle phase creates the following input for the Reduce phase:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```

Example: Word Count (cnt.)

- Consider a reduce function that sums the numbers in the list for each key and outputs `(word, count)` pairs. The output of the Reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

Spark Motivation

Spark Motivation

- The MapReduce framework is still a widely used Big Data processing framework
- Apache Spark is a newer framework that extends the MapReduce framework with many additional features
- MapReduce has several problems that Spark is able to address:
  - Reading and storing data from main memory instead of hard disks - Main memory is much faster than the hard drives
  - Running iterative algorithms much faster - Especially important for many machine learning algorithms

Spark Benchmarking

Spark Benchmarking

- The original papers claim Spark to be upto 100x faster when data fits into RAM vs MapReduce, or upto 10x faster when data is on disks
- Large speedups can be observed in the RAM vs HD case
- However, independent benchmarks show when both use HDs, Spark is upto 2.5-5x faster for CPU bound workloads, however MapReduce could still sort faster with hard drives:
  Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, Fatma Özcan: Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. PVLDB 8(13): 2110-2121 (2015)
  `http://www.vldb.org/pvldb/vol8/p2110-shi.pdf`
  `http://www.slideshare.net/ssuser6bb12d/a-comparative-performance-evaluation-of-apache-flink`

Resilient Distributed Datasets

Resilient Distributed Datasets

- Resilient Distributed Datasets (RDDs) are distributed datastructures that allow processed data to be distributed over several computers
- The framework stores the RDDs in data partitions
- Each computer can have several data partitions, and one thread can process each partition on its own
- To implement fault tolerance, the RDD partitions record lineage: A recipe to recompute the RDD partition based on the parent RDDs and the operation used to generate the partition
- RDDs are immutable: Once created the RDD data contents does not change.

Creating RDDs

RDDs can be created from:

- Local datastructures or local files
- From other RDDs using RDD transformations
- Usually with Big Data: Files stored in distributed storage systems: Hadoop Distributed Filesystem (HDFS), Amazon S3, HBase, ...

RDD Partitions: Performance Tuning

If you need to start to performance tune applications:

- When an RDD is created, it is initially split to a number of partitions, which should be large enough (e.g., at least 2-10 x the number of total number of cores in the used computers) to allow for efficient load balancing between cores
- Each partition should still be large enough to take more than 100 ms to process, so not to waste too much time in starting and finishing the task processing a partition

PySpark Wordcount Example

PySpark Wordcount Example

- PySpark is the Python interface to the Apache Spark framework
- As an example we will create a wordcount application with PySpark and RDDs
- The system is using JupyterHub based PySpark environment which automatically imports the needed libraries and defines the SparkContext object `sc`
- The input file used is the project Gutenberg version of the Finnish national epic Kalevala, available from:
  `http://www.gutenberg.org/ebooks/7000`

PySpark Wordcount: Reading Input

PySpark Wordcount: Reading Input

- We create an RDD from local filesystem using `textFile` function, which returns an RDD where each line is its own data item.
- The `take(10)` function returns an array that contains the 10 first data items in the RDD, as “debug printout”.

```
In [1]: kalevala_lines_rdd=sc.textFile("pg7000.txt")
        kalevala_lines_rdd.take(10)
```

```
Out[1]: ['The Project Gutenberg EBook of The Kalevala, by Elias Lönnrot',
         '',
         'This eBook is for the use of anyone anywhere at no cost and with',
         'almost no restrictions whatsoever.  You may copy it, give it away or',
         're-use it under the terms of the Project Gutenberg License included',
         'with this eBook or online at www.gutenberg.net',
         '',
         '',
         'Title: The Kalevala',
         '']
```

PySpark Wordcount: Splitting Lines

PySpark Wordcount: Splitting Lines

- Each line is split into words by running the `flatMap` routine, which maps each element to potentially several output elements.
- Here the Python `split` function is by `flatMap` for each line, generating as its output the sequence of words inside each line.

```
In [2]: kalevala_words_rdd=kalevala_lines_rdd.flatMap(lambda line: line.split(" "))
        kalevala_words_rdd.take(10)
```

```
Out[2]: ['The',
         'Project',
         'Gutenberg',
         'EBook',
         'of',
         'The',
         'Kalevala,',
         'by',
         'Elias',
         'Lönnrot']
```

PySpark Wordcount: Mapping Word

PySpark Wordcount: Mapping Words

- We call the `map` function which creates an RDD that has pairs with the word instances as keys and the constant `1` as value.
- Note that the `map` function (as well as other RDD transformations such as e.g., `flatMap`) can be run in parallel for different partitions of the RDD.

```
In [3]: kalevala_count_rdd=kalevala_words_rdd.map(lambda word:(word,1))
        kalevala_count_rdd.take(10)
```

```
Out[3]: [('The', 1),
         ('Project', 1),
         ('Gutenberg', 1),
         ('EBook', 1),
         ('of', 1),
         ('The', 1),
         ('Kalevala,', 1),
         ('by', 1),
         ('Elias', 1),
         ('Lönnrot', 1)]
```

PySpark Wordcount: Calculating Totals

PySpark Wordcount: Calculating Totals

- `reduceByKey` is used to reduce the list of values associated with each key to a single value using the function given as parameter.
- The reduce function takes as input two values and combines them into one value. The function must be associative and commutative!

```
In [4]: kalevala_count_totals_rdd=kalevala_count_rdd.reduceByKey(lambda x,y:(x+y))
        kalevala_count_totals_rdd.take(10)
```

```
Out[4]: [('The', 17),
         ('Project', 81),
         ('EBook', 2),
         ('of', 108),
         ('Kalevala,', 2),
         ('Elias', 5),
         ('Lönnrot', 6),
         ('', 46379),
         ('is', 26),
         ('use', 8)]
```

Spark Reduce: Associativity and Commutativity

Spark Reduce: Associativity and Commutativity

- Associativity and commutativity of "+" allows the reduceByKey to freely schedule the reductions, and to even do parts of the reductions in parallel on different machines!
- This is actually happening inside Spark, each worker first computes local word counts inside each partition, and then sends only the sum of the partition word counts to the final reduce routine computed on potentially a different machine, saving a lot of network bandwidth.
- The final reduce routine sums up the local word counts from each partition.
- This only works because of associativity and commutativity!

RDD Transformations an Actions

RDD Transformations:

- RDD Transformations build a DAG of dependencies between RDD partitions but do not yet start processing
- The actual data processing is done lazily
- The RDD Actions are needed to start the computation

RDD Transformations

The following RDD transformations are available:

- map(func)
- filter(func)
- flatMap(func)
- mapPartitions(func)
- mapPartitionsWithIndex(func)
- sample(withReplacement, fraction, seed)
- union(otherDataset)
- intersection(otherDataset)
- distinct([numTasks]))

RDD Transformations (cnt.)

The following RDD transformations are available:

- groupByKey([numTasks])
- reduceByKey(func, [numTasks])
- aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])
- sortByKey([ascending], [numTasks])
- join(otherDataset, [numTasks])
- cogroup(otherDataset, [numTasks])
- cartesian(otherDataset)
- pipe(command, [envVars])
- coalesce(numPartitions)

RDD Transformations (cnt.)

The following RDD transformations are available:

- repartition(numPartitions)
- repartitionAndSortWithinPartitions(partitioner)

For more details, see:

`https://spark.apache.org/docs/latest/rdd-programming-guide.html`

RDD Actions

Available RDD Actions:

- reduce(func)
- collect()
- count()
- first()
- take(n)
- takeSample(withReplacement, num, [seed])
- takeOrdered(n, [ordering])
- saveAsTextFile(path)
- saveAsSequenceFile(path)

RDD Actions (cnt.)

Available RDD Actions:

- saveAsObjectFile(path)
- countByKey()
- foreach(func)

Again more details, see:

`https://spark.apache.org/docs/latest/rdd-programming-guide.html`

RDD Operations and Actions

RDD Operations and Actions:

- Many well known functional programming constructs such as `map` (or `flatMap`) and `reduce` (`reduceByKey`) are available as operations
- One can implement relational database like functionality easily on top of RDD operations, if needed
- This is how Spark SQL, a Big Data analytics framework, was originally implemented
- Many operations take a user function to be called back as argument
- Freely mixing user code with RDD operations limits the optimization capabilities of Spark RDDs - You get the operations you write, not many automatic optimization can be done

Implementing RDD Operations and Actions

Implementing RDD Operations and Actions

- As the RDD operations are run in several computers, there is no shared memory to use to share state between operations
- The functions run in all of the distributed computers need to copy all of the data they need to all of the Spark workers using so called closures
- To minimize the amount of data that needs to be copied to all computers, the functions should refer to as little data as possible to keep the closures small
- Note that variables of the closure modified in a Worker are lost and thus can not be used to communicate back to the driver program
- Communication is done through RDDs

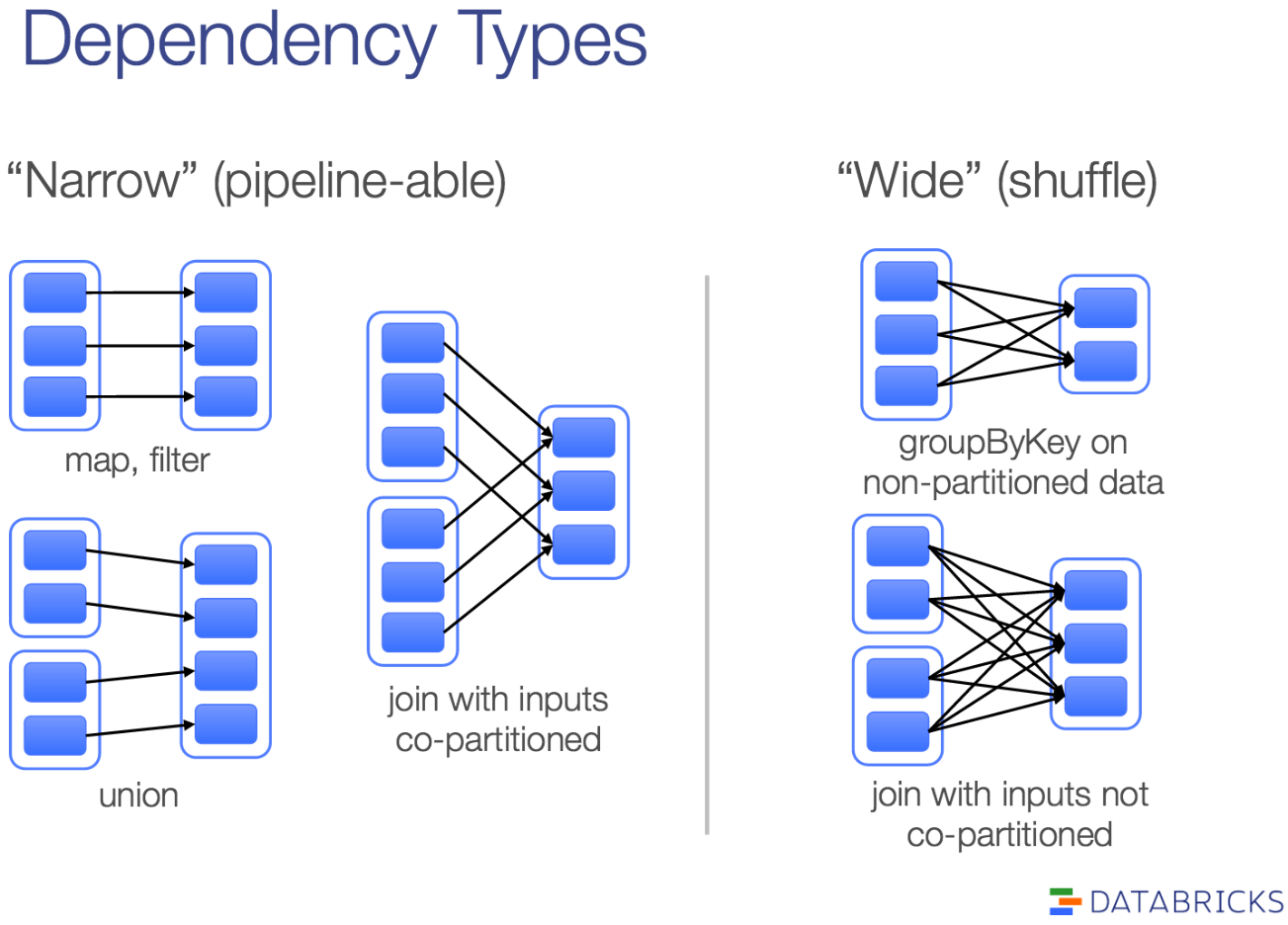
Lineage

Lineage can contain both narrow and wide dependencies:

- Figures in the following from “Databrics Advanced Spark Training by Reynold Xin”

Lineage (cnt.)

Lineage can contain both narrow and wide dependencies:



Narrow and Wide Dependencies

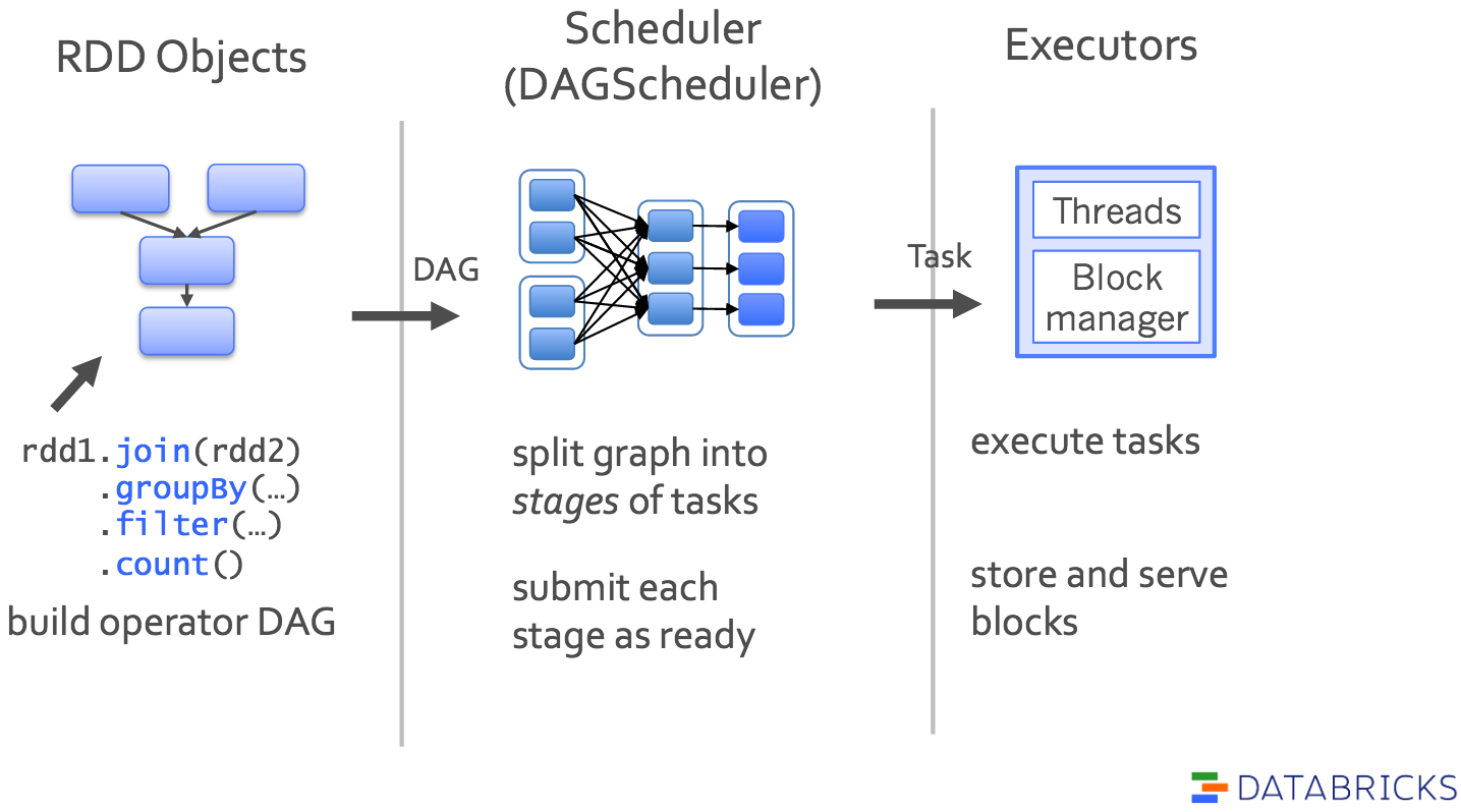
Narrow and Wide Dependencies

- In narrow dependencies, the data partition from an input RDD can be scheduled on the same physical machine as the out RDD. This is called “pipelining”
- Thus RDD operations with only narrow dependencies can scheduled to be done without any network traffic
- Wide dependencies require all RDD partitions at the previous level of the lineage to be inputs to computing an RDD partition
- This requires sending the RDD data over the network
- This operation is called the “shuffle” in Spark (and MapReduce) terminology
- Shuffles are unavoidable for applications needing e.g., global sorting of the output data

DAG Scheduling

DAG scheduler is used to schedule RDD operations:

Job Scheduling Process



Pipelining into Stages

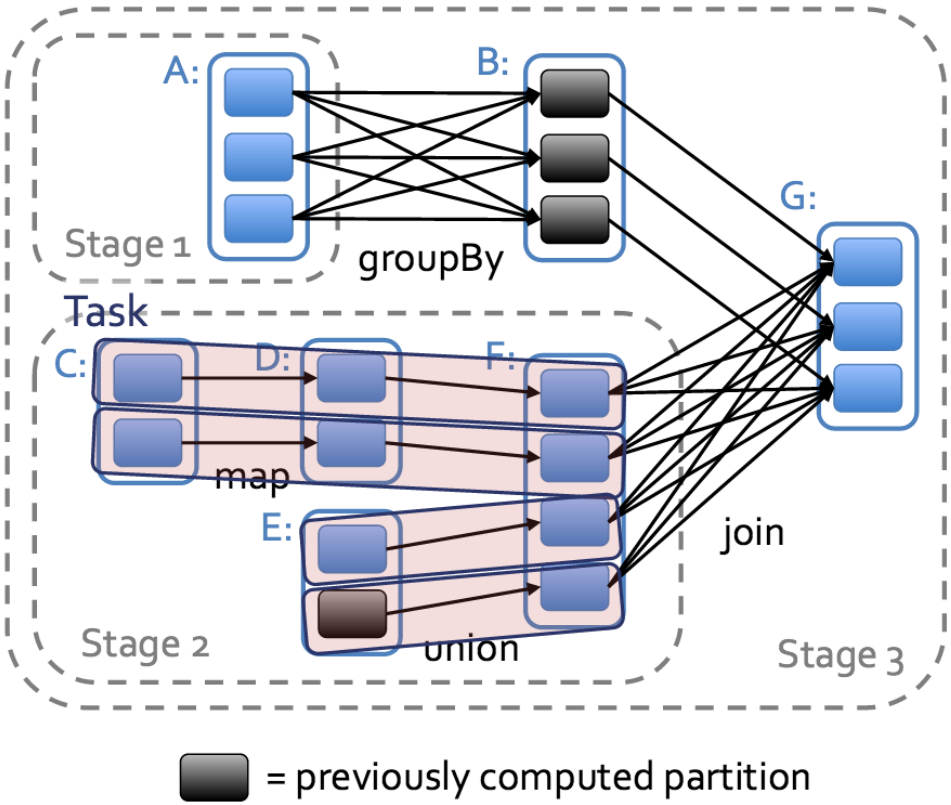
Scheduler pipelines work into stages separated by wide dependencies:

Scheduler Optimizations

Pipelines operations within a stage

Picks join algorithms based on partitioning (minimize shuffles)

Reuses previously cached data



Spark Extensions

Spark Extensions

- MLlib - Distributed Machine learning Library
- Spark SQL - Distributed Analytics SQL Database - Can be tightly integrated with Apache Hive Data Warehouse, uses HQL (Hive SQL variant)
- Spark Streaming and Structured Streaming - Streaming Data Processing Frameworks
- GraphX - A Graph processing system