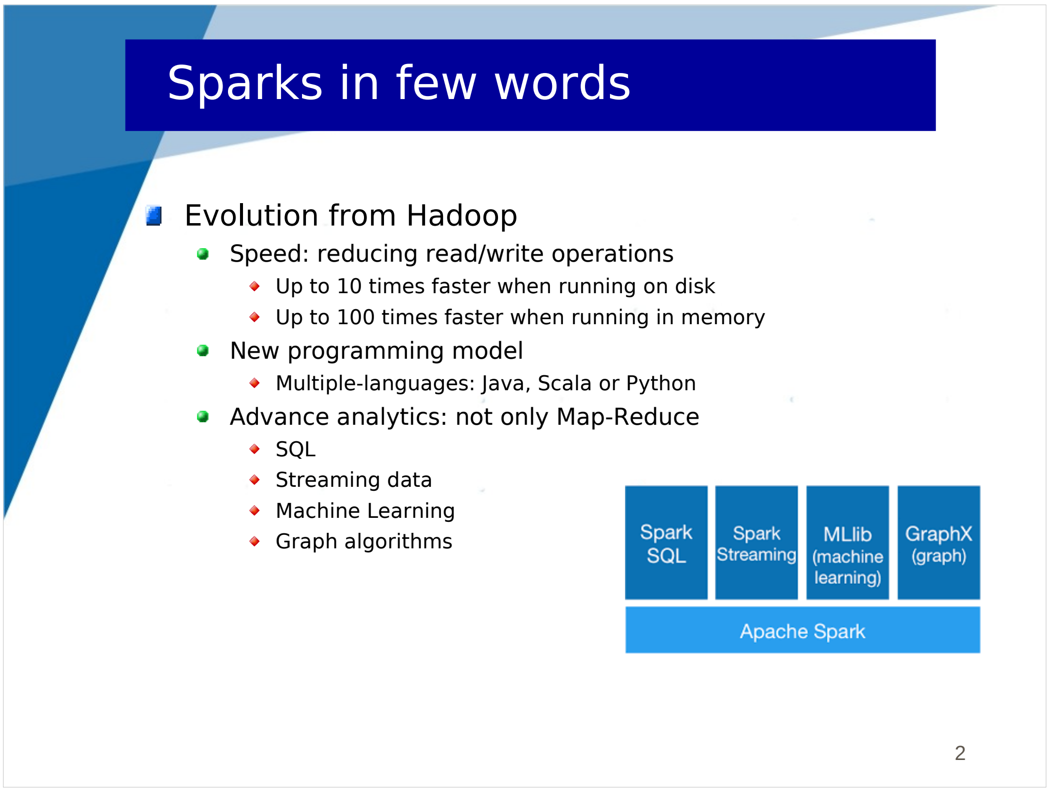
Spark

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

1

This lecture is about the Spark framework from Google.



Sparks in few words

- Evolution from Hadoop
  - Speed: reducing read/write operations
    - Up to 10 times faster when running on disk
    - Up to 100 times faster when running in memory
  - New programming model
    - Multiple-languages: Java, Scala or Python
  - Advance analytics: not only Map-Reduce
    - SQL
    - Streaming data
    - Machine Learning
    - Graph algorithms

Spark SQL | Spark Streaming | MLlib (machine learning) | GraphX (graph)

Apache Spark

Spark is an evolution from Hadoop.

One of the main goal was to improve performance, either when used for datasets that fit in memory, or for larger datasets that have to be stored on disk.

Another major improvement is to provide a new programming model (still implementing a map-reduce strategy) for the manipulation of large datasets. This programming model is available in multiple popular languages (Java, Scala or Python).

The Spark environment also provides additional services for

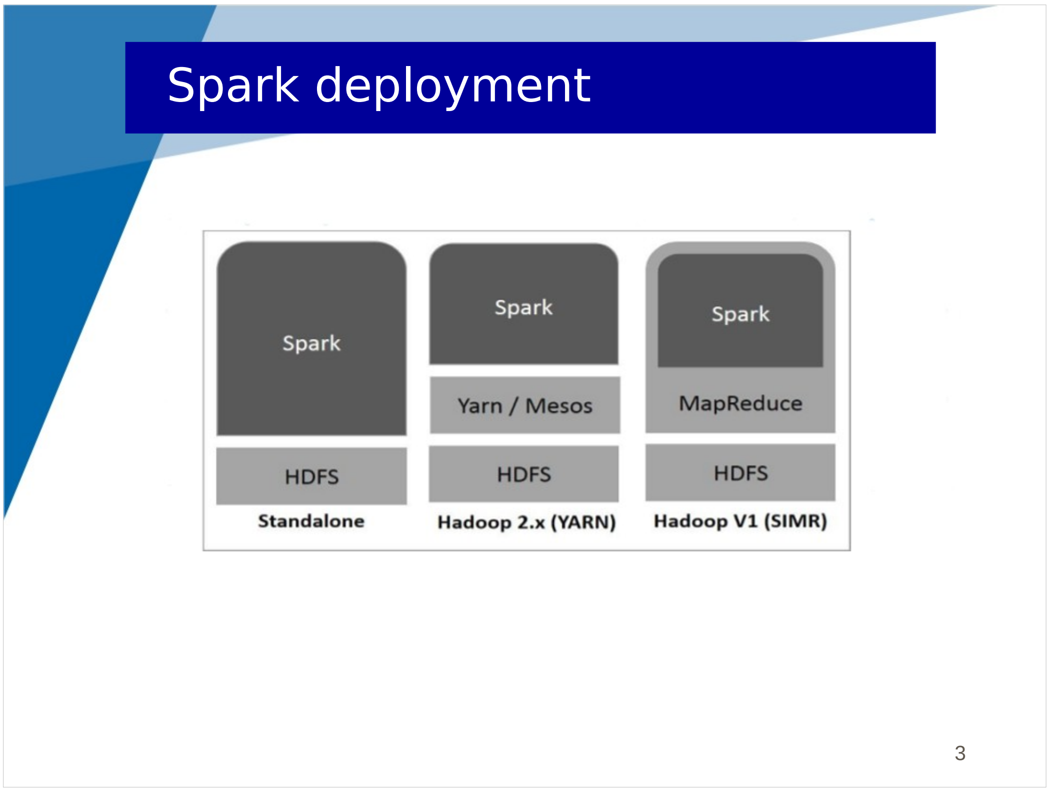
- expressing request in an SQL like syntax

- handling streams of data (arriving continuously)

- applying machine learning algorithms to large datasets

- handling datasets which include graphs

This lecture covers the basics of Spark and another lecture will present SparkStreaming.



Spark deployment

Spark
HDFS
Standalone

Spark
Yarn / Mesos
HDFS
Hadoop 2.x (YARN)

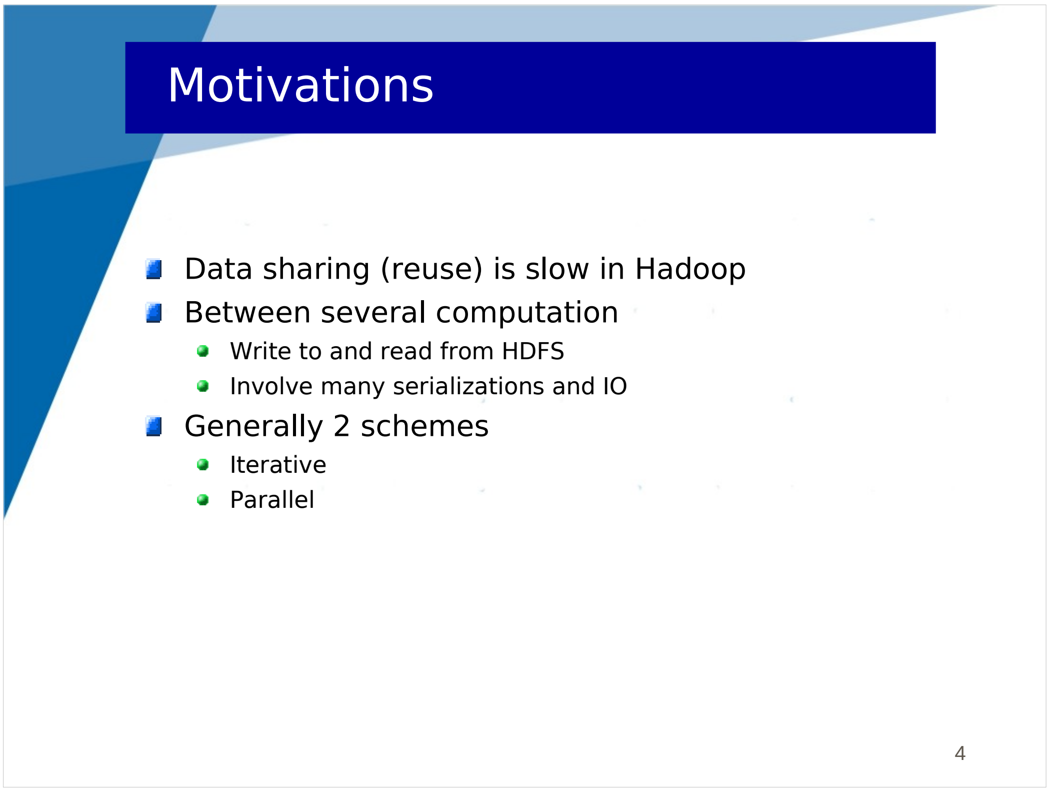
Spark
MapReduce
HDFS
Hadoop V1 (SIMR)

For testing, Spark can work centralized on your laptop relying on the local filesystem. When you run Spark in a cluster, you have to rely on a distributed filesystem which is generally HDFS.

Spark can work in standalone mode on top of HDFS. This means that Spark provides everything to distribute jobs over the nodes to benefit from parallelism. It also provides tools to monitor your applications.

You can also use Spark with Yarn, in order to benefit from its resource management. This is especially interesting when your cluster is used by several users running many applications at the same time.

SIMR is a way to run Spark applications on older Hadoop clusters (without having to reinstall everything).



Motivations

- Data sharing (reuse) is slow in Hadoop
- Between several computation
  - Write to and read from HDFS
  - Involve many serializations and IO
- Generally 2 schemes
  - Iterative
  - Parallel

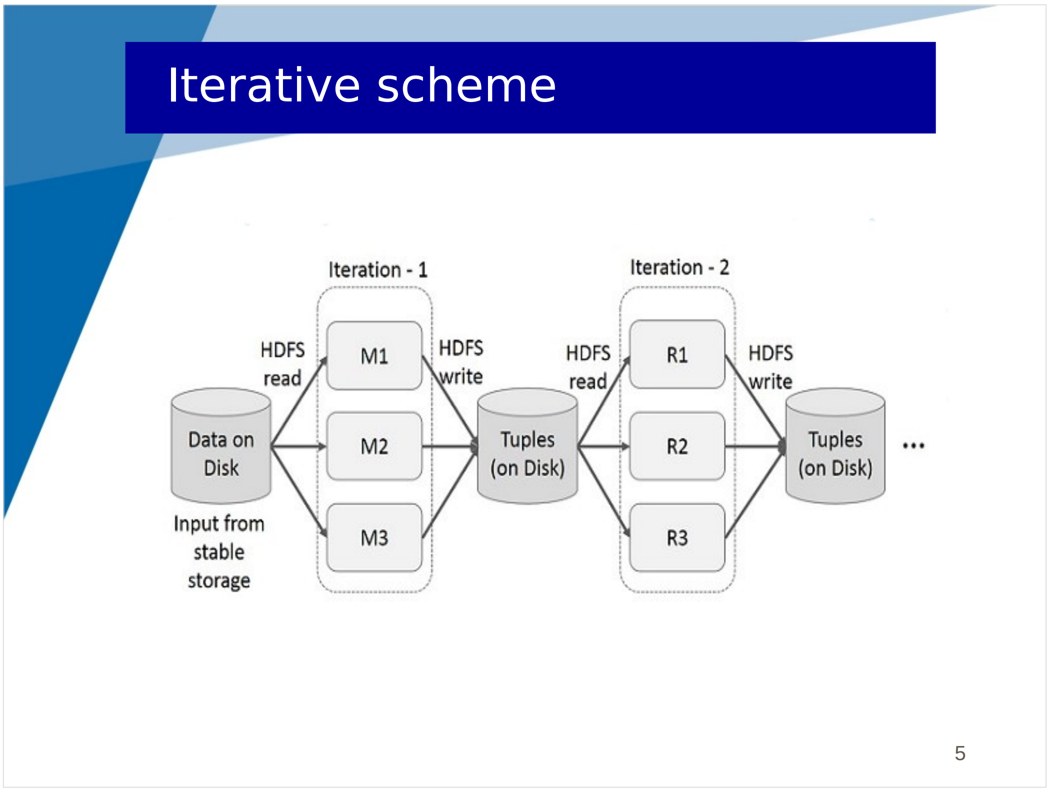
4

One important motivation for Spark was performance improvement.

It came from the observation that in Hadoop, data sharing or reuse was slow. When a Hadoop job is generating data (output) which is in turn reused (input) by another job, this data is written to disk by the first job and then read from disk by the second job. This involves many serialization and IO operations which are very costly. Also, if two jobs are using the same dataset as input, they both have to read the dataset from disk.

The targeted improvement is to be able to reuse data in memory without having to systematically read it from disk.

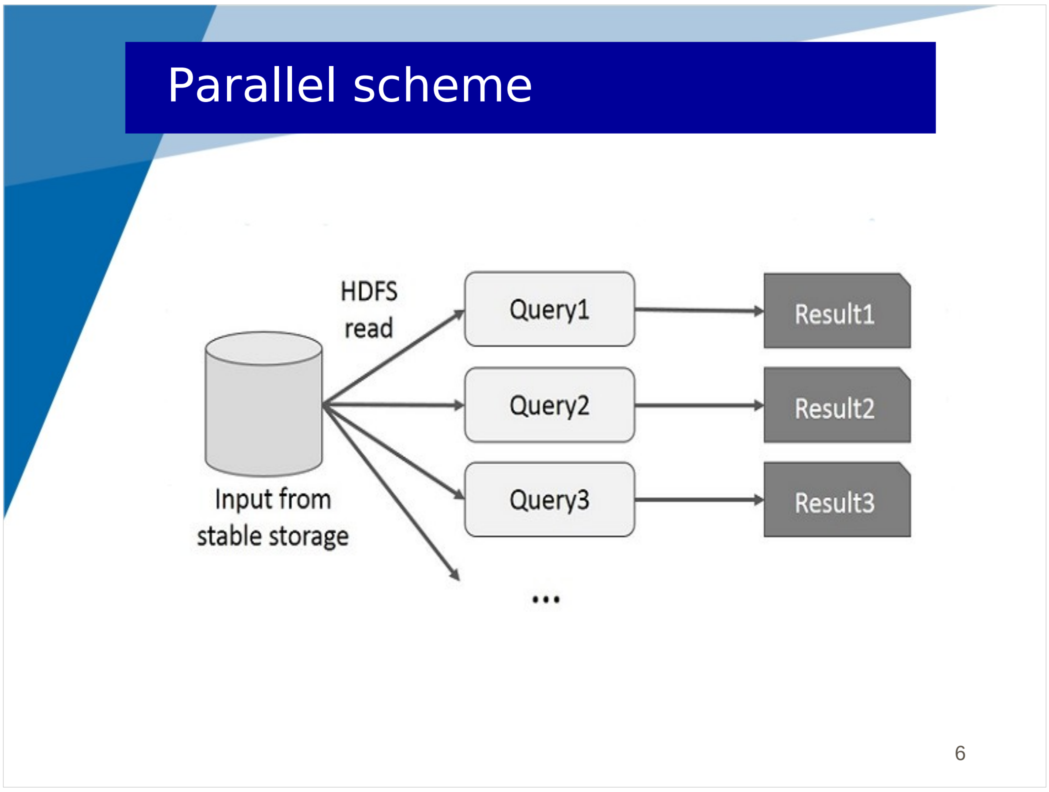
Two schemes are then considered for such improvements.



Iterative scheme

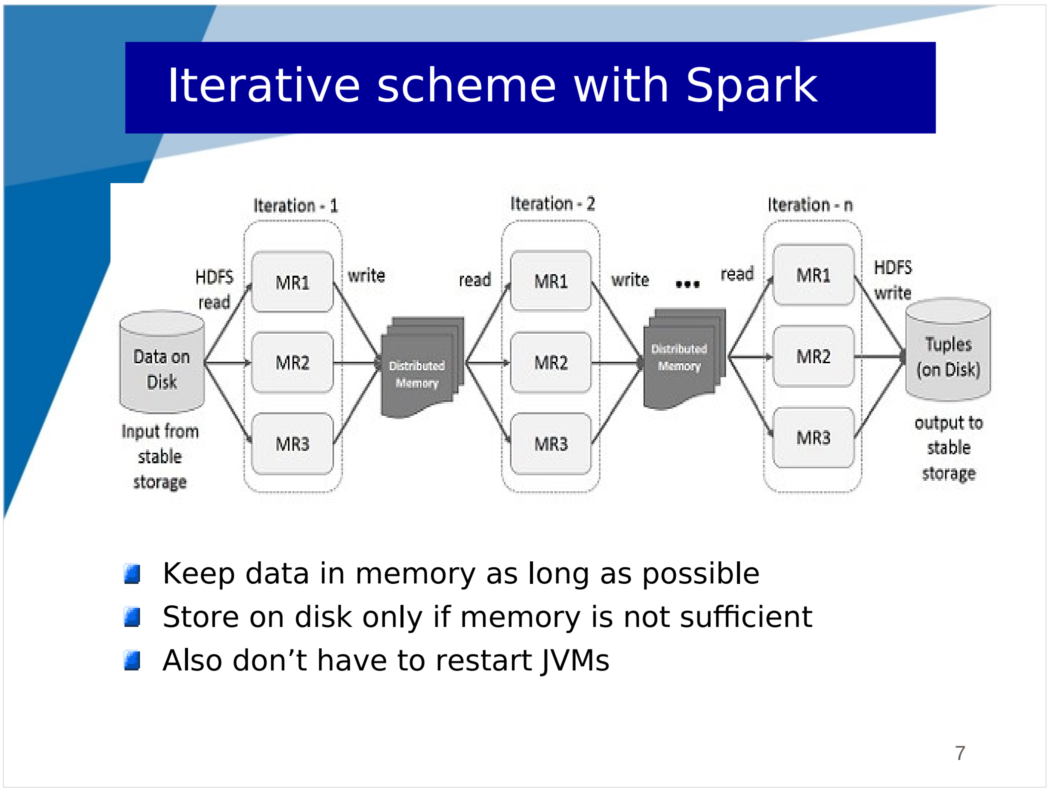
5

In the iterative scheme, when 2 jobs are run iteratively, data are read from disk by the first job and results are written to disk. Then the results from the first job are read from disk and results are written to disk.



Parallel scheme

In the parallel scheme, several parallel jobs are using the same input data. Each job is reading the same input data from disk. Even with caching at the filesystem level, each job is using its own memory for hosting data.



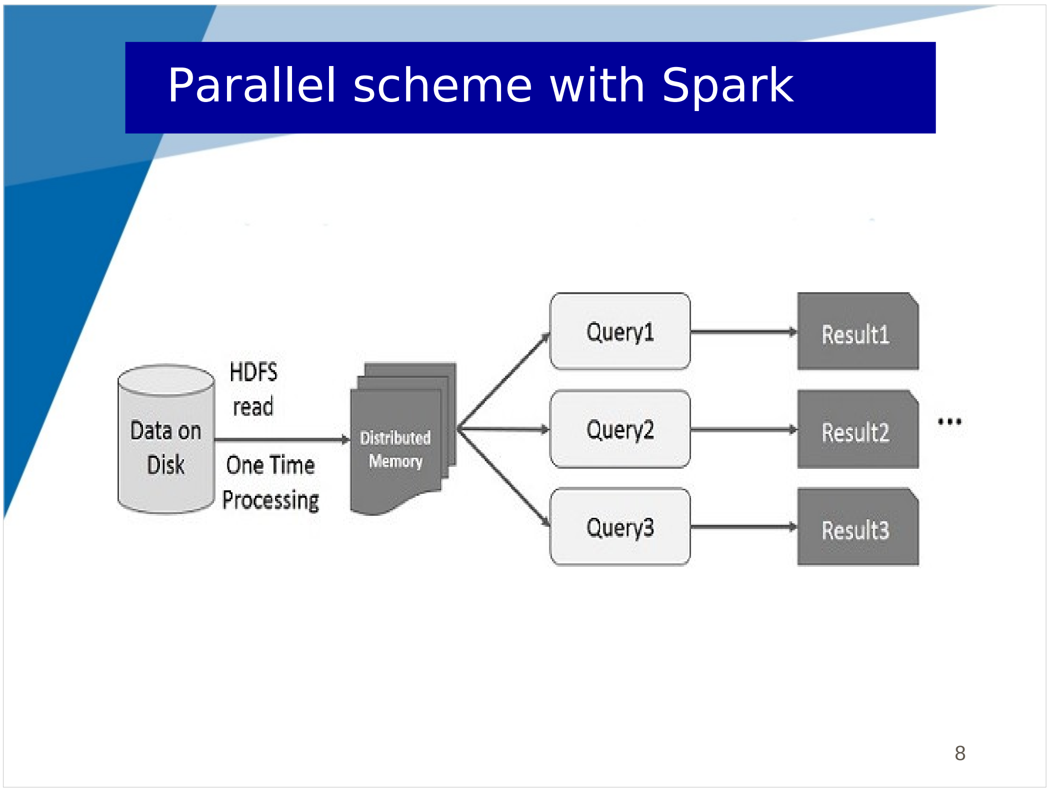
Iterative scheme with Spark

- Keep data in memory as long as possible
- Store on disk only if memory is not sufficient
- Also don’t have to restart JVMs

7

Results from the first job could simply be kept in memory to be reused by the second job without having to be stored on disk.

Also, with Hadoop, new Java Virtual Machines are started for each job (for maps and reduces). JVMs could be resused between jobs, allowing intermediate results (between jobs) to be kept in memory.

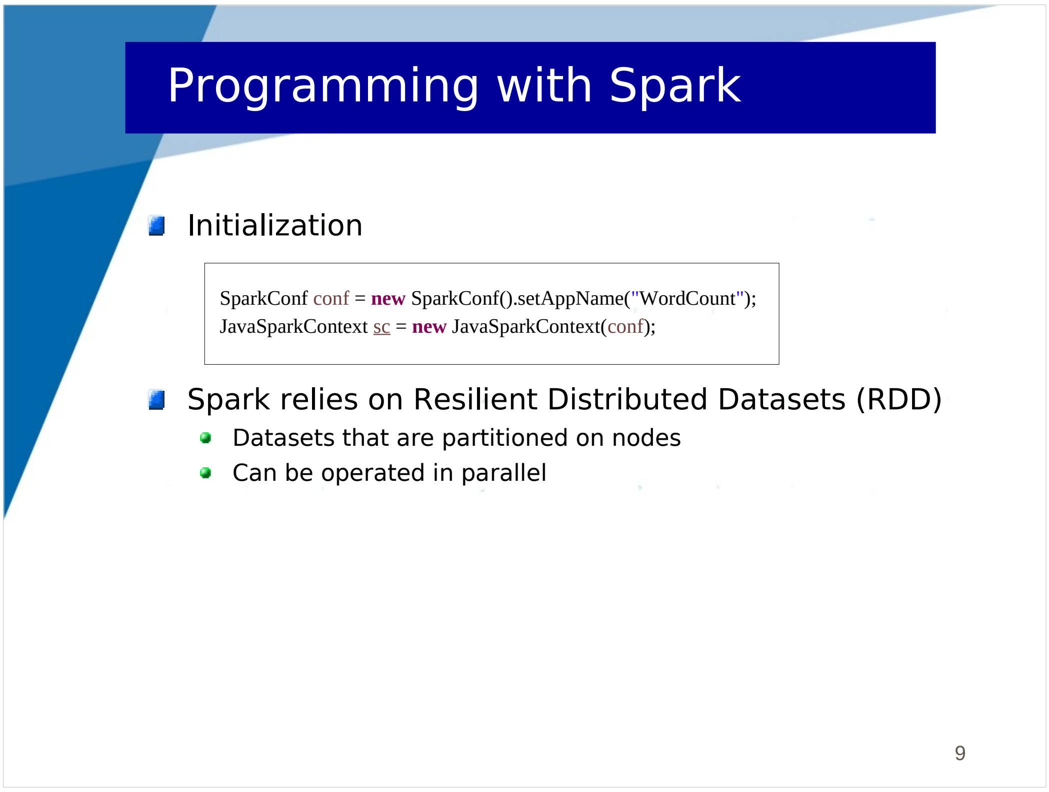


Parallel scheme with Spark

Here also, JVMs could be reused and input data loaded only once and shared between queries.

Spark enables such improvements for iterative and parallel tasks.

To go in this direction and also to ease the development of applications, it introduces a new programming model as described in the following.



Programming with Spark

- Initialization

```
SparkConf conf = new SparkConf().setAppName("WordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
```

- Spark relies on Resilient Distributed Datasets (RDD)
  - Datasets that are partitioned on nodes
  - Can be operated in parallel

Here is a simple tutorial about programming in Spark. I use Java for the lecture and also for labworks, although I also have a Python version of both.

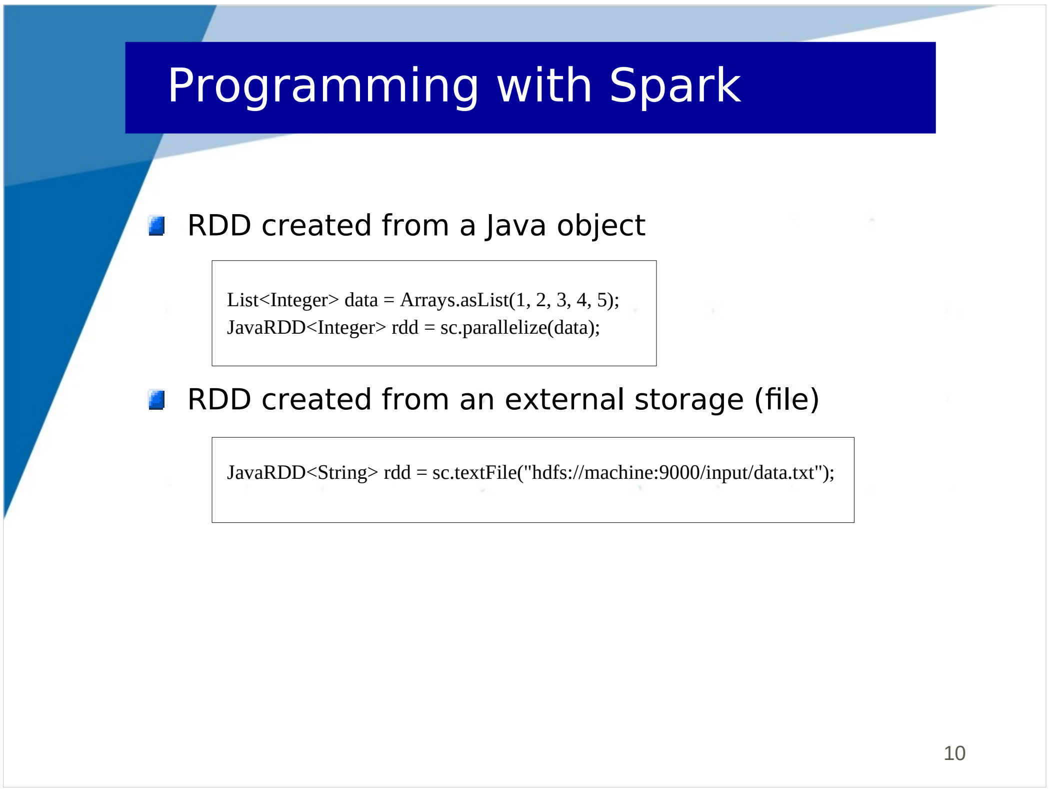
The first step is to initialize Spark.

You must create a configuration object and then create a spark context object.

This spark context object is then used mainly for accessing (generally from HDFS) the data to be processed.

The Spark programming model relies on Resilient Distributed Datasets or RDD. A RDD is a very large vector of values (any Java object). This vector is partitioned, which means that it is split into fragments (called partitions) that are distributed on different nodes.

This partitioning on nodes allows computations on a RDD to be performed in parallel. For instance, if you want to apply the same operation on each element of one RDD, It can be done in parallel on each node where a partition is stored, each node iterating on the elements it hosts.



Programming with Spark

- RDD created from a Java object

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> rdd = sc.parallelize(data);
```

- RDD created from an external storage (file)

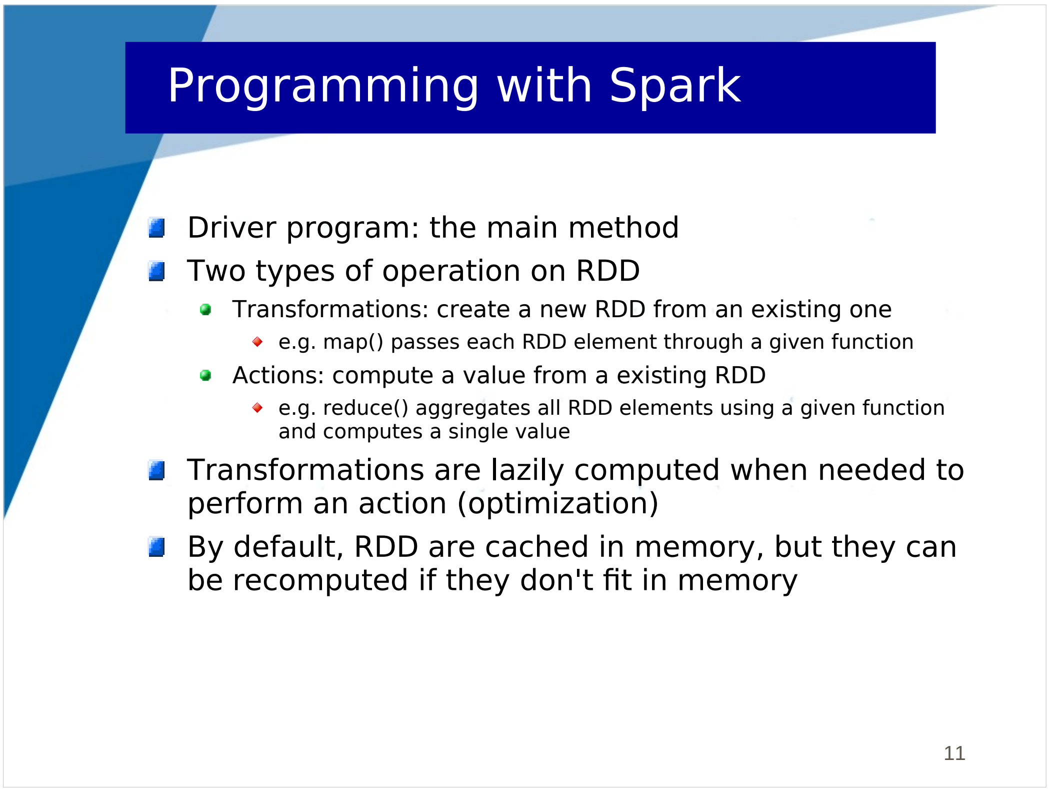
```
JavaRDD<String> rdd = sc.textFile("hdfs://machine:9000/input/data.txt");
```

10

You can create a RDD from a Java object (here a list of Integer) with the parallelize() method. Notice that a RDD is typed: here it is a JavaRDD<Integer>

Generally a RDD is large (we are in the Big Data world). It is read from an external storage such as HDFS. In the second example, the textFile() method on the sparkcontext allows to create a RDD from a text file, thus resulting in a JavaRDD<String>. The elements in this RDD are the lines from the text file. This RDD is composed of distributed partitions, each partition corresponding to a block in HDFS.

It is important to note that a RDD is an abstract entity. When you execute textFile(), data are not loaded in memory. It registers that data will be loaded from a text file in the future. The data are effectively loaded when operations are performed on the data.



Programming with Spark

- Driver program: the main method
- Two types of operation on RDD
  - Transformations: create a new RDD from an existing one
    - e.g. map() passes each RDD element through a given function
  - Actions: compute a value from a existing RDD
    - e.g. reduce() aggregates all RDD elements using a given function and computes a single value
- Transformations are lazily computed when needed to perform an action (optimization)
- By default, RDD are cached in memory, but they can be recomputed if they don't fit in memory

11

The execution of a Spark application first executes the main method in a single process. This process is called the Driver program. Then, the operations on RDDs as distributed, i.e. executed on all the nodes where partitions are located. For instance, if a RDD is initialized with a HDFS file distributed on several nodes (blocks in HDFS), an operation on that RDD will be distributed and executed in parallel on these nodes, where partitions will be loaded from blocks in HDFS.

Two types of operation can be performed on RDD:

- Transformations which create a new RDD from an existing one. An example is a map(f) operation which applies the f function on all elements within the RDD. The resulting RDD is distributed on all the nodes where the existing RDD is.

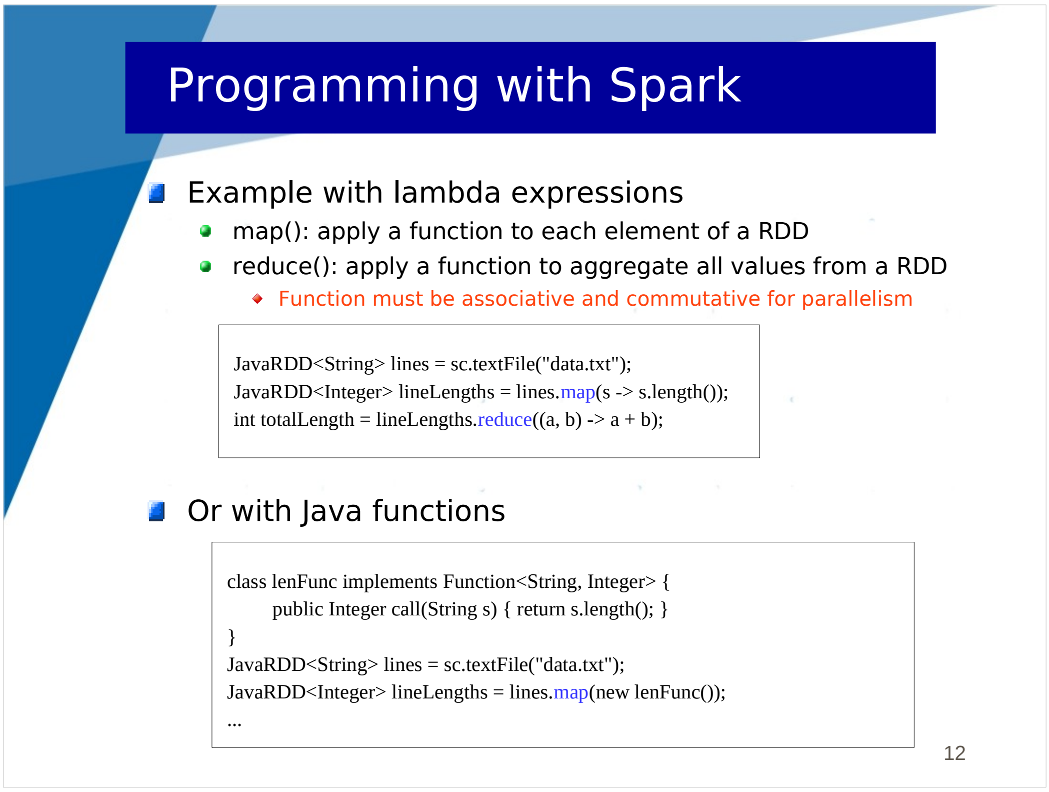
- Actions which compute a final value from an existing RDD. The result is stored in the Driver program. An example is a reduce(f) operation which aggregates all elements within the existing RDD. The f function takes a pair of elements as parameter and computes the aggregation into one element. The same function f is used to aggregate all the elements into a single value.

Transformations are computed lazily, generally when an action has to be performed. For instance :

```
rdd=(1,2,3,4)

rdd.map(increment()) // means each element should be incremented

rdd.map(double()) // means each element should be doubled

sum=rdd.reduce(add()) // compute a final value (sum), add() being an additioner
                                   // which aggregates 2 elements into one
```

We don't have to parse the RDD for each map. We can parse the RDD only once and apply both the increment() and double() functions only when the element is considered for aggregating it with another element.

When the result of a transformation is reused for different computations, it is cached in memory. But if memory is lacking, it may be evicted from cache and later recomputed (remember that RDD are immutable, only new RDDs are generated).



Programming with Spark

- Example with lambda expressions
  - map(): apply a function to each element of a RDD
  - reduce(): apply a function to aggregate all values from a RDD
    - Function must be associative and commutative for parallelism

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

- Or with Java functions

```
class lenFunc implements Function<String, Integer> {
        public Integer call(String s) { return s.length(); }
}
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new lenFunc());
...
```

12

To program functions (f or increment in the previous slide) which can be used in operations (like map() or reduce()), we can use lambda expressions or Java functions. Map and reduce operations were presented in the previous slide.

In this example, lines is a RDD of String, initialized from a data.txt file (may be in HDFS). It contains the lines from the document. The map() operation replaces each line by its length, resulting into a RDD of Integer. The reduce() operation aggregates all the elements from that RDD, using an additioner : (a,b) → a+b

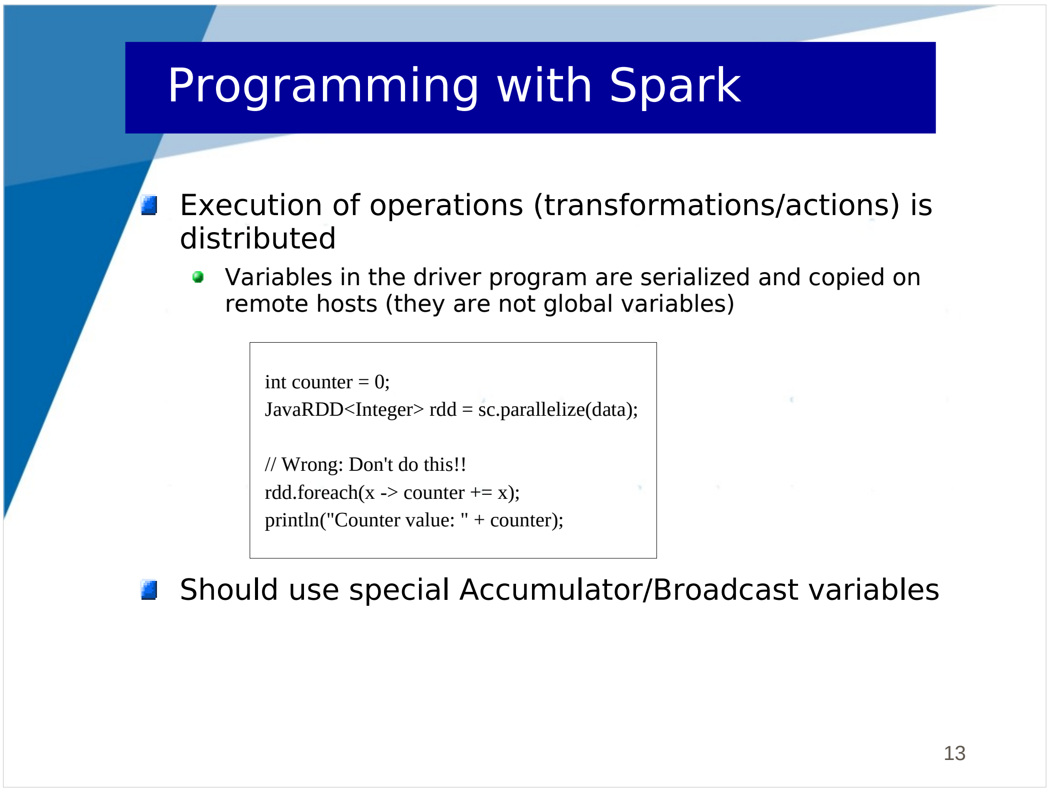
This way of writing a function is a lambda expression.

In a reduce(), the aggregation function must be associative and commutative, which means that the order the elements are aggregated does not matter. For instance, the aggregation of (1,2,3,4) may be performed in these orders:

(1+2) + (3+4) or ((1+2)+3)+4

You provide an aggregation function (commutative and associative) and Spark is responsible for applying it.

In the bottom, you have the equivalent with Java functions. A Java function is a parametric class : Function <param_1, param_2 …, result> where param_i is the type of the ith param, and result the returned type. The class must implement a call() method which is the method invoked when the function is called. Notice in the example that the parameters of the call() method are consistent with those of the class definition.



Programming with Spark

- Execution of operations (transformations/actions) is distributed
  - Variables in the driver program are serialized and copied on remote hosts (they are not global variables)

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!
rdd.foreach(x -> counter += x);
println("Counter value: " + counter);
```

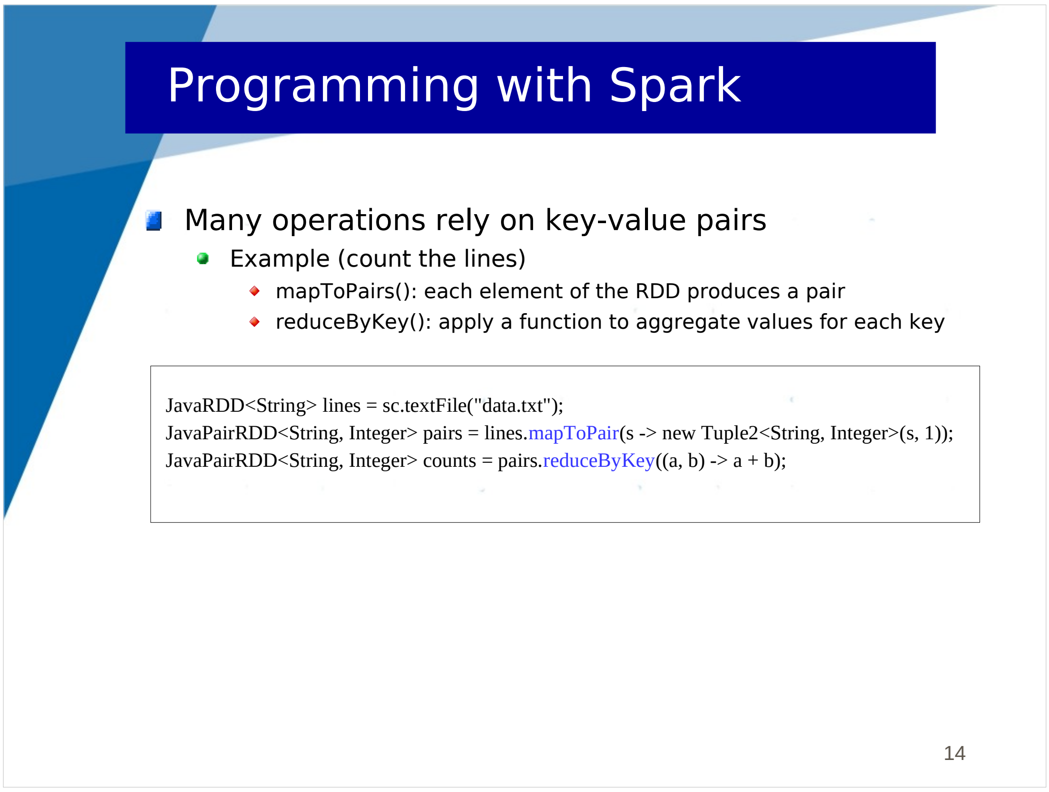
- Should use special Accumulator/Broadcast variables

You must remember that the executions of transformations and actions are distributed on the nodes of the cluster. This implies that variables in the Driver program (the main method) cannot be used in operations on RDDs. Actually, they are serialized and copied on all the nodes, but their values may be inconsistent.

In the given example, we try to use a variable (counter) in the Driver program to sum values in a RDD (foreach() applies a function on each element of the RDD, but without generating a new RDD).

DON'T DO THAT

Global variables exist in Spark. You should use special variables : Accumulator and Broadcast (described later).



Programming with Spark

- Many operations rely on key-value pairs
  - Example (count the lines)
    - mapToPairs(): each element of the RDD produces a pair
    - reduceByKey(): apply a function to aggregate values for each key

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs = lines.mapToPair(s -> new Tuple2<String, Integer>(s, 1));
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b);
```

In Spark (as in Hadoop), many operations rely on key-value pairs.

While a RDD of type T is : JavaRDD<T>

A RDD of pairs of types K and V is : JavaPairRDD<K,V>

Such a pair RDD can be generated by a mapToPair(f) operation, f being a function which transforms an element into a pair (an instance of the Tuple2 class in Java).

Another form of reduce operation allows to do what we were doing with Hadoop : sorting/grouping keys and aggregating values for each unique key.

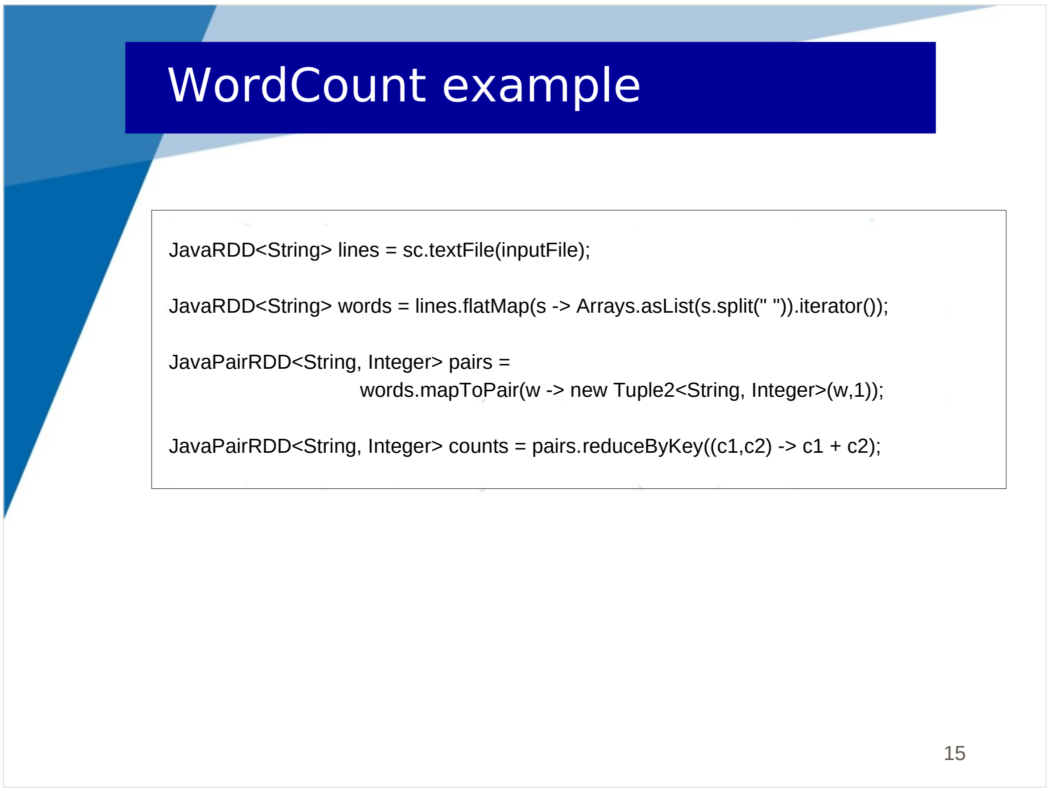
reduceByKey(f) on a pair RDD groups keys and aggregates values with the f function (which should still be associative and commutative)

In the example,

- the mapToPair() operation generates for each element (line) a pair <line,1>. Notice that it returns a JavaPairRDD<String, Integer>

- the reduceByKey() operation groups identical lines and additions all the 1 behind a unique line.

Globally the result gives for each unique line the number of times the line appears in the text file (it counts redundant lines).



WordCount example

```
JavaRDD<String> lines = sc.textFile(inputFile);

JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(s.split(" ")).iterator());

JavaPairRDD<String, Integer> pairs =
                         words.mapToPair(w -> new Tuple2<String, Integer>(w,1));

JavaPairRDD<String, Integer> counts = pairs.reduceByKey((c1,c2) -> c1 + c2);
```

15

Here is the implementation of the WordCount example that we saw in Hadoop.

- the split() method returns a String[]

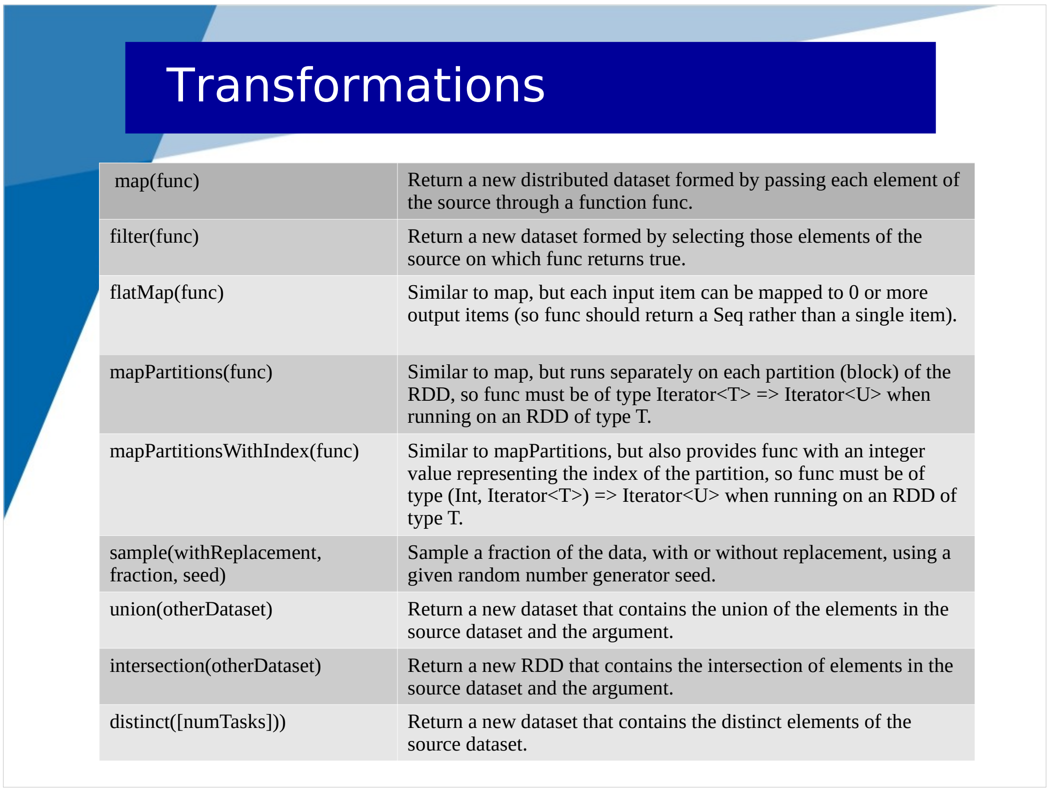
- asList() returns a List<String>

- like map(f), flatMap(f) applies the f function on each element of the RDD. The difference is that map() generates for each element one element in the result RDD, while flatMap() may generate 0 or n elements. The function f in flatMap(f) should return an iterator used by Spark to insert the generated elements in the result RDD. In this example, the function in flatMap() splits each line into words and returns an iterator on the list of words. The flatMap() returns a JavaRDD<String> includeing all the words.

- mapToPair() generates for each word w a pair <w, 1>

- reduceByKey() groups words and additions all the 1 behind each unique word

```
JavaRDD<String> lines = sc.textFile(inputFile);
// returns ("toto titi tutu", "tata titi toto", "tata toto tutu")
JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(s.split(" ")).iterator());
// returns ("toto", "titi", "tutu", "tata", "titi", "toto", "tata", "toto", "tutu")
JavaPairRDD<String, Integer> pairs =
                    words.mapToPair(w -> new Tuple2<String, Integer>(w,1));
// returns (("toto",1), ("titi",1), ("tutu",1), ("tata",1), ("titi",1), ("toto",1), ("tata",1),
("toto",1), ("tutu",1))
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((c1,c2) -> c1 + c2);
// group words (("toto",{1,1,1}), ("titi",{1,1}), ("tutu",{1,1}), ("tata",{1,1}))
// and then addition (("toto",3), ("titi",2), ("tutu",2), ("tata",2))
```



Transformations

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).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(func)	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	Sample a fraction of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.

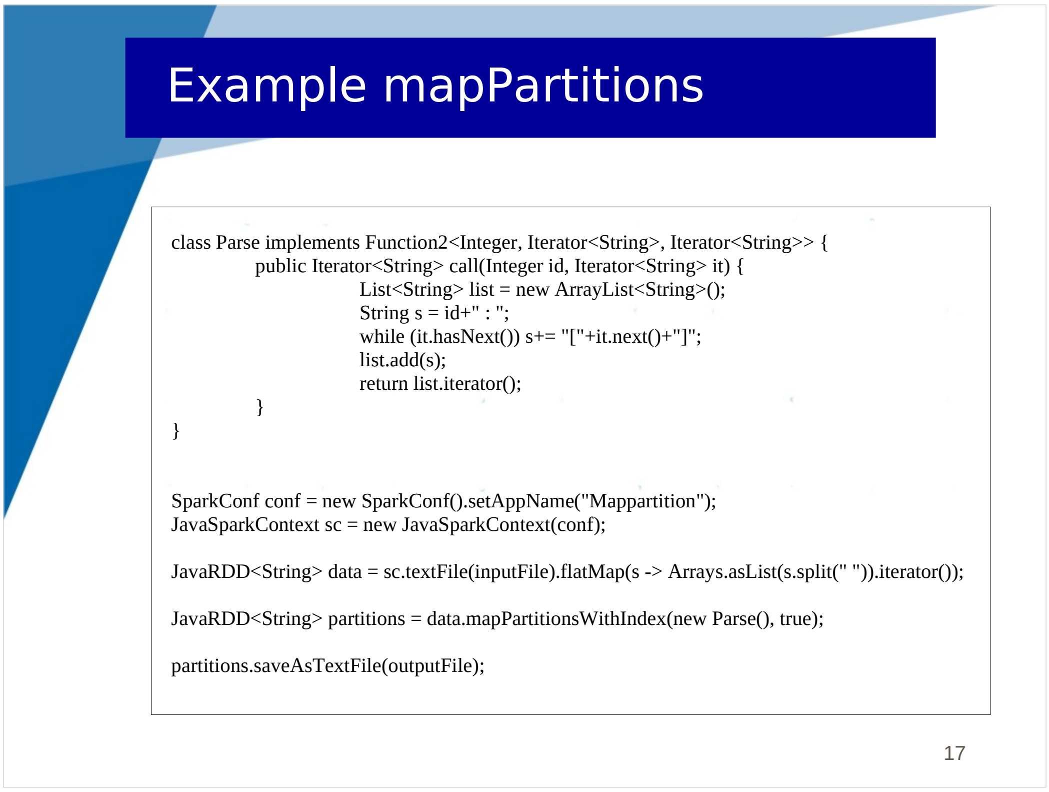
The documentation is huge and it's out of scope to present it exhaustively.

We have seen map() and flatMap().

Notice filter() which allows removing elements in a RDD or distinct() for removing redundant elements.

mapPartitionsxxx() operations allow executing an iteration function at the level of each partition. An example is given in the next slide.

Also notice operations for doing the union or intersection of 2 datasets (RDDs).



Example mapPartitions

```
class Parse implements Function2<Integer, Iterator<String>, Iterator<String>> {
        public Iterator<String> call(Integer id, Iterator<String> it) {
                List<String> list = new ArrayList<String>();
                String s = id+" : ";
                while (it.hasNext()) s+= "["+it.next()+"]";
                list.add(s);
                return list.iterator();
        }
}


SparkConf conf = new SparkConf().setAppName("Mappartition");
JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> data = sc.textFile(inputFile).flatMap(s -> Arrays.asList(s.split(" ")).iterator());

JavaRDD<String> partitions = data.mapPartitionsWithIndex(new Parse(), true);

partitions.saveAsTextFile(outputFile);
```

This is an example using mapPartitionsWithIndex operation.

The beginning is the same as in the WordCount example.

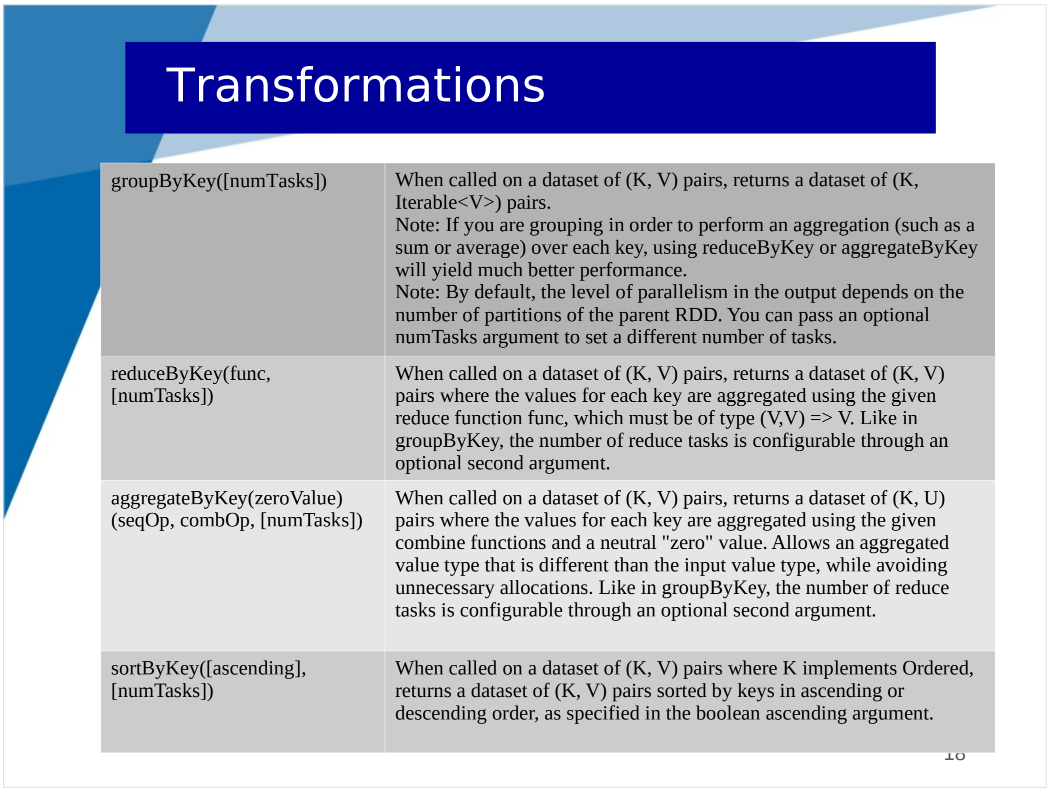
On the data RDD (which includes a list of words), we call the mapPartitionsWithIndex() operation. Its allows executing the Parse Java function on each partition of the RDD. The Parse Java function :

- has a first parameter : the partition number

- has a second parameter : the Iterator object which allows iterating on the elements of the partition

- has a result : the Iterator object which allows Spark to iterate on the generated elements to insert them in the resulting partition of the RDD.

In this example, the Parse function concatenates all the elements in the partition and generates a single element in the partition.



Transformations

groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(func, [numTasks])	When 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, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([ascending], [numTasks])	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.

18

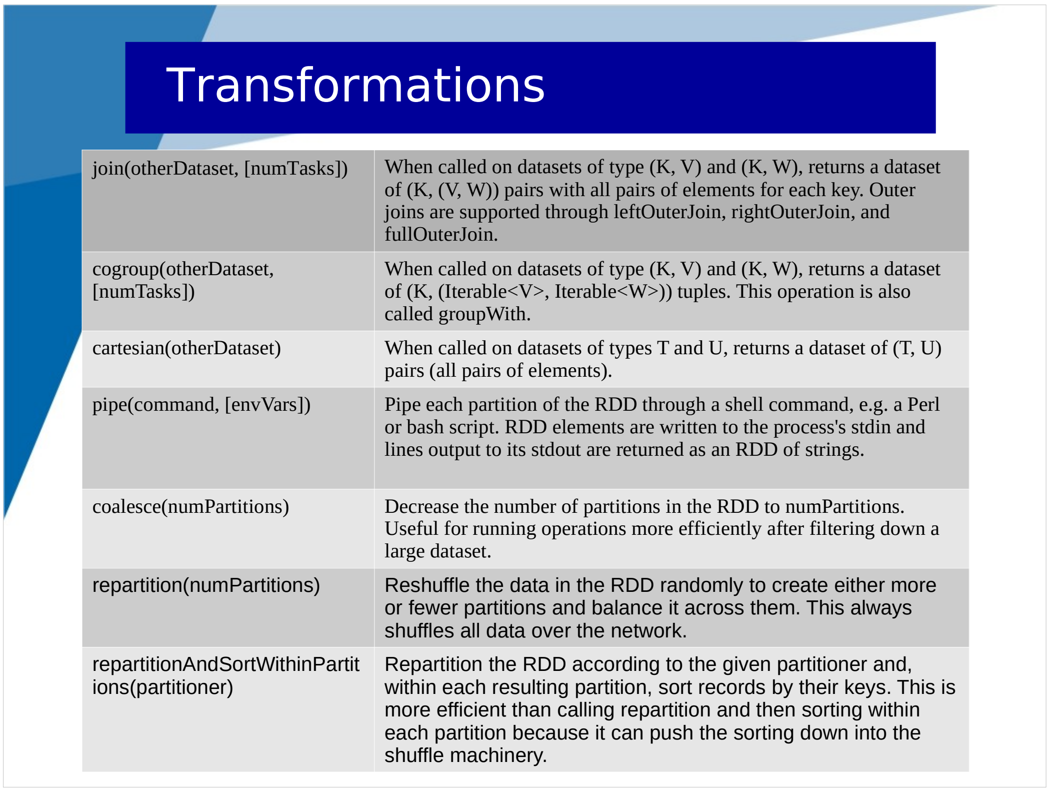
We have seen reduceByKey().

There are many other forms of grouping/aggregation by key. Here only few of them :

groupByKey() does the grouping, but does not aggregate.

aggregateByKey() does the same as reduceByKey() but the result of the aggregation of values can be of a different type.

sortByKey() is only sorting KV by keys.



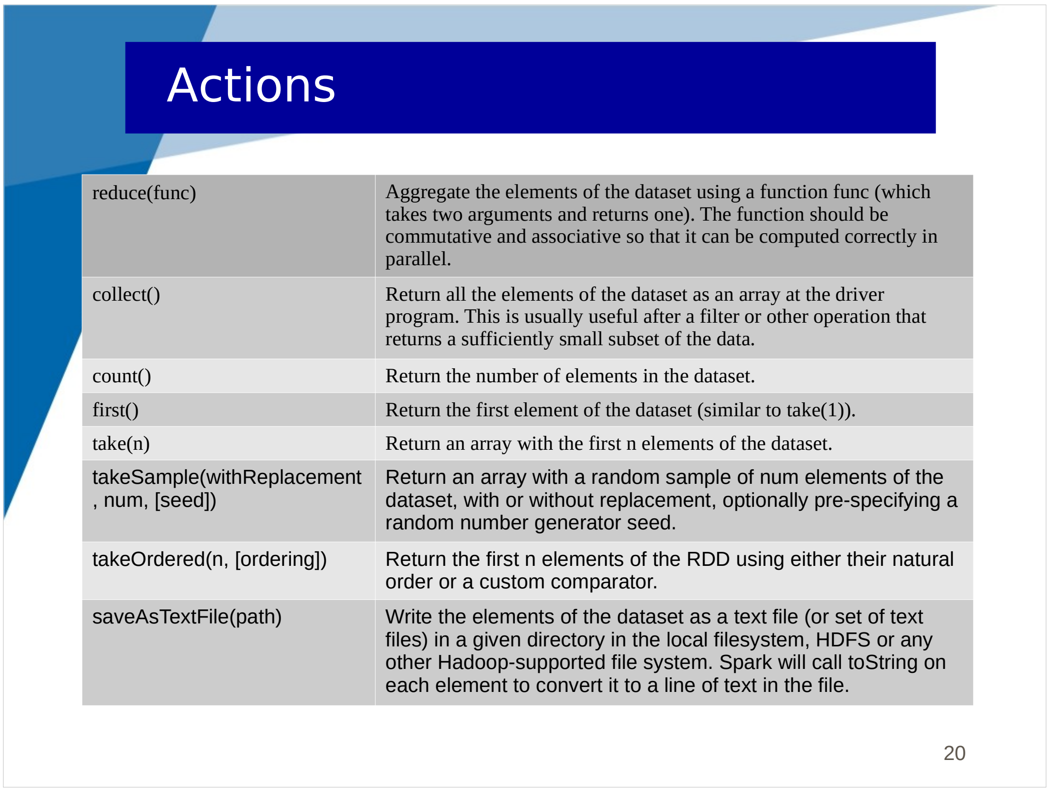
Transformations

join(otherDataset, [numTasks])	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.
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(partitioner)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Some other interesting operations. Notice :

- join() : a join of dataset1(K,V) and dataset2(K,W) becomes dataset3(K,(V,W)) for identical keys

All the previous operations where transformations, which generate new RDDs.



Actions

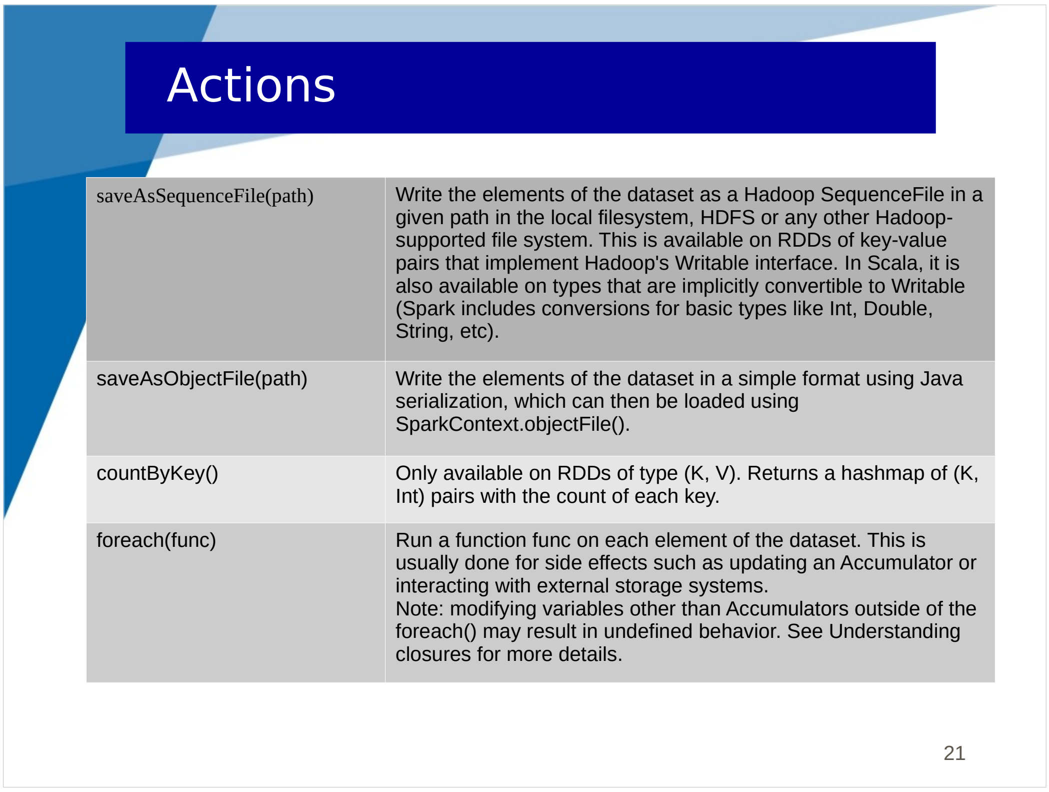
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. 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.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

20

Here are Action operations which don't generate RDDs, but compute a final result.

We have already seen reduce().

- collect() gathers/downloads all the elements of the RDD on the Driver program (to be used for small datasets only).
- count()
- first()
- take()
- saveAsTextFile()



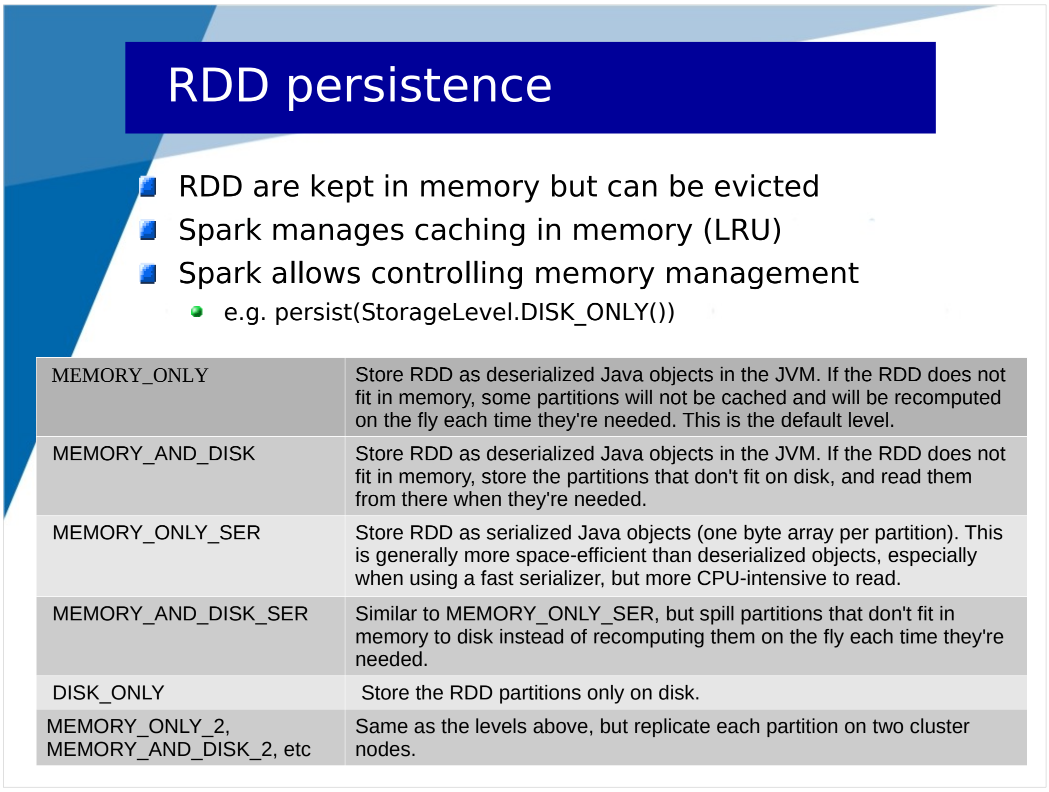
Actions

saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Notice here:

- foreach() that we already saw in a previous example

- countByKey() which does a sort of reduceByKey() as each key is unique in the result, but the value for each key is the number of KV with that key in the initial RDD. Notice that this is an Action, so the result is not an RDD, but a Java HashMap in the Driver program.



RDD persistence

- RDD are kept in memory but can be evicted
- Spark manages caching in memory (LRU)
- Spark allows controlling memory management
  - e.g. persist(StorageLevel.DISK_ONLY())

MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

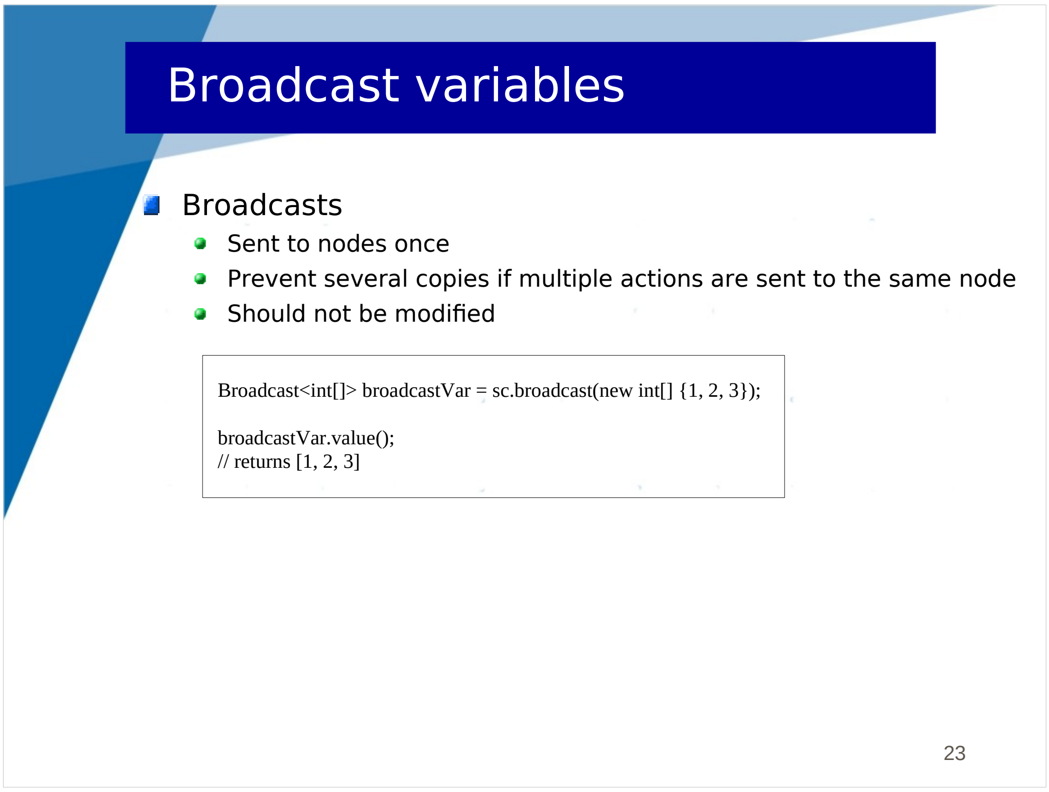
RDDs are kept in memory as much as possible. When memory is lacking, partitions can be evicted from memory. Spark manages caching of partitions in memory with a LRU eviction scheme. It allows to control (with the persist() method on a RDD) how the RDD's partitions are handled.

The default behavior is MEMORY_ONLY, which means that an evicted partition is removed and will be recomputed if needed.

MEMORY_AND_DISK means evicted partitions are stored on disk (like for swap).

MEMORY_ONLY_SER means the RDD's partitions are stored in serialized format to occupy less space. They can still be removed and recomputed. Serialization incurs a deserialization cost on read.

DISK_ONLY means the RDD's partitions have to be stored on disk.



Broadcast variables

- Broadcasts
  - Sent to nodes once
  - Prevent several copies if multiple actions are sent to the same node
  - Should not be modified

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});

broadcastVar.value();
// returns [1, 2, 3]
```

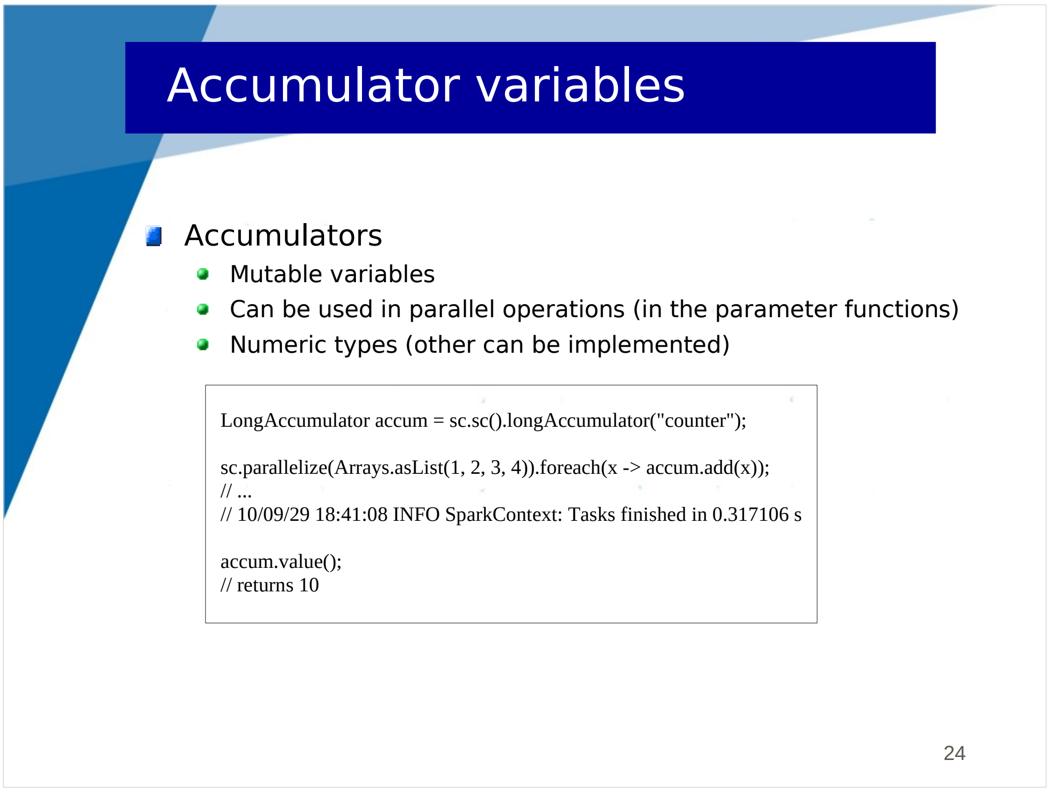
23

As previously explained, we cannot manage global variables in the Driver program. To allows managing variables which are global to all nodes, Spark provides :

- broadcast variables : they are read-only

- accumulator variables : they are read-write

Broadcast variables are read-only and sent to all nodes. They can be used in transformations and actions which are distributed. They are sent only once to each node even if several operations are sent to the nodes.

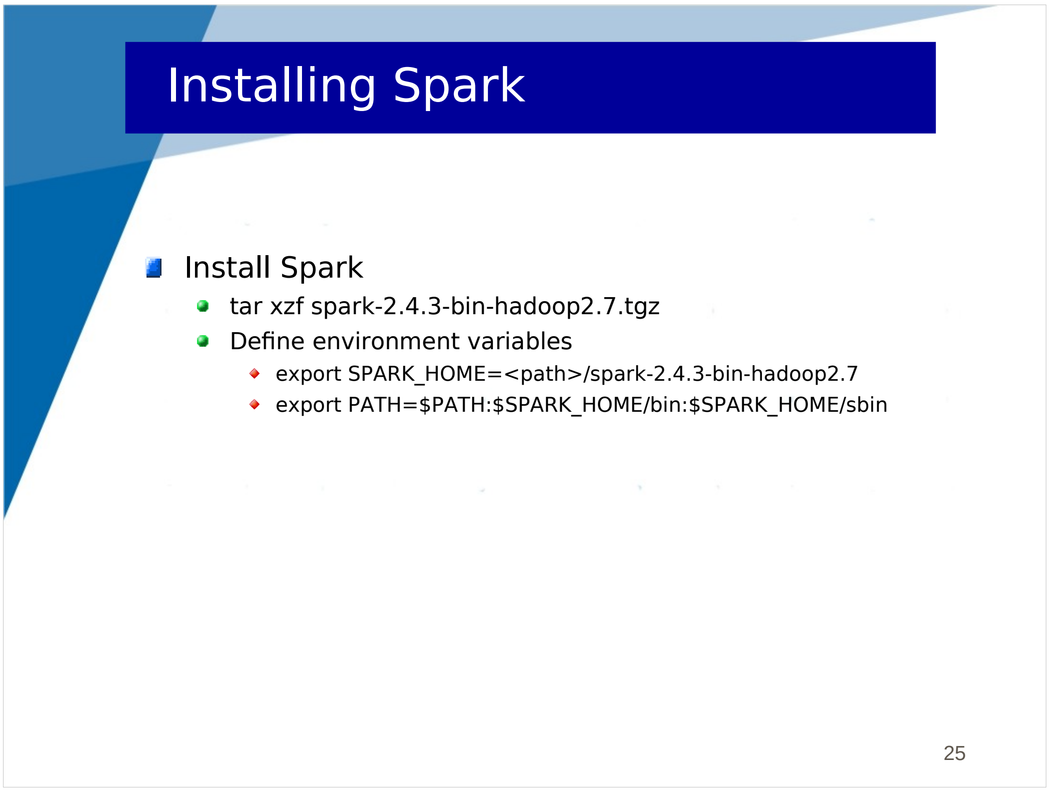


Accumulator variables

- Accumulators
  - Mutable variables
  - Can be used in parallel operations (in the parameter functions)
  - Numeric types (other can be implemented)

```
LongAccumulator accum = sc.sc().longAccumulator("counter");

sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x -> accum.add(x));
// ...
// 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

accum.value();
// returns 10
```

Accumulators can be modified. There are accumulators for numeric types (e.g. LongAccumulator). They can be used in functions used in parallel operations (e.g. the foreach operation in the example).



Installing Spark

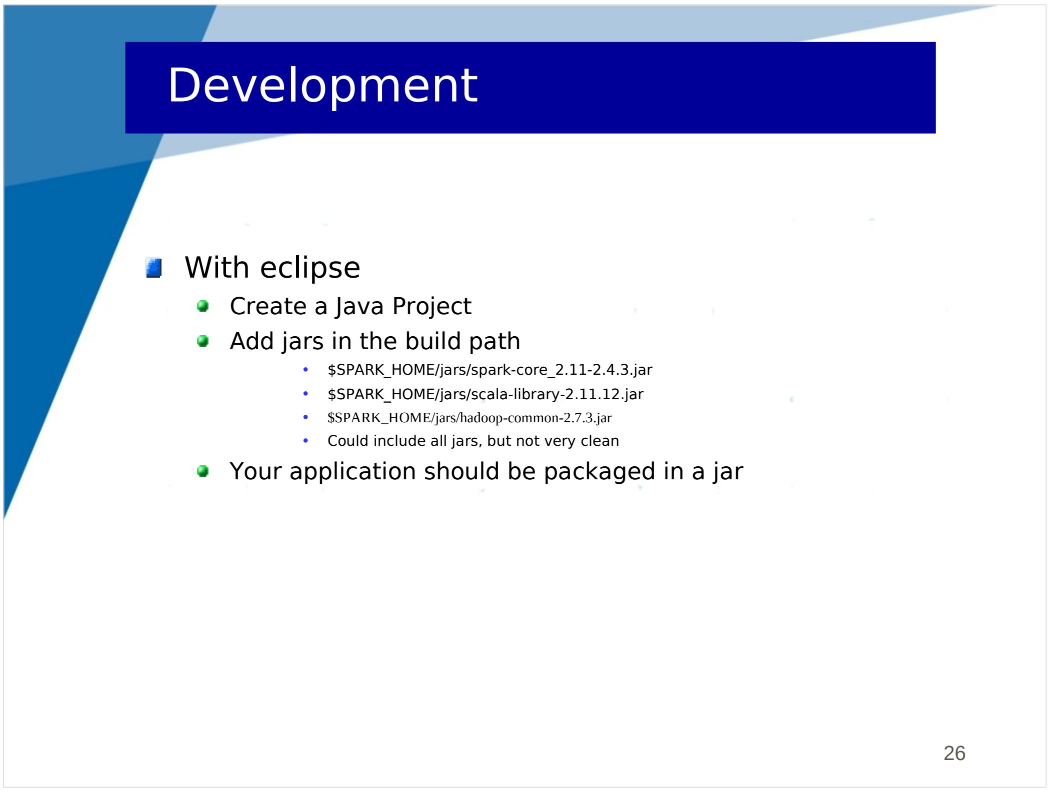
- Install Spark
  - tar xzf spark-2.4.3-bin-hadoop2.7.tgz
  - Define environment variables
    - export SPARK_HOME=<path>/spark-2.4.3-bin-hadoop2.7
    - export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin

25

The following slides describe the main instructions for using Spark.

Installing Spark is simply expanding an archive. Notice that in a cluster, the binaries should be accessible at the same path on any node. This is obvious with proper NFS mounts.

You have to define environment variables (store that in your bashrc).



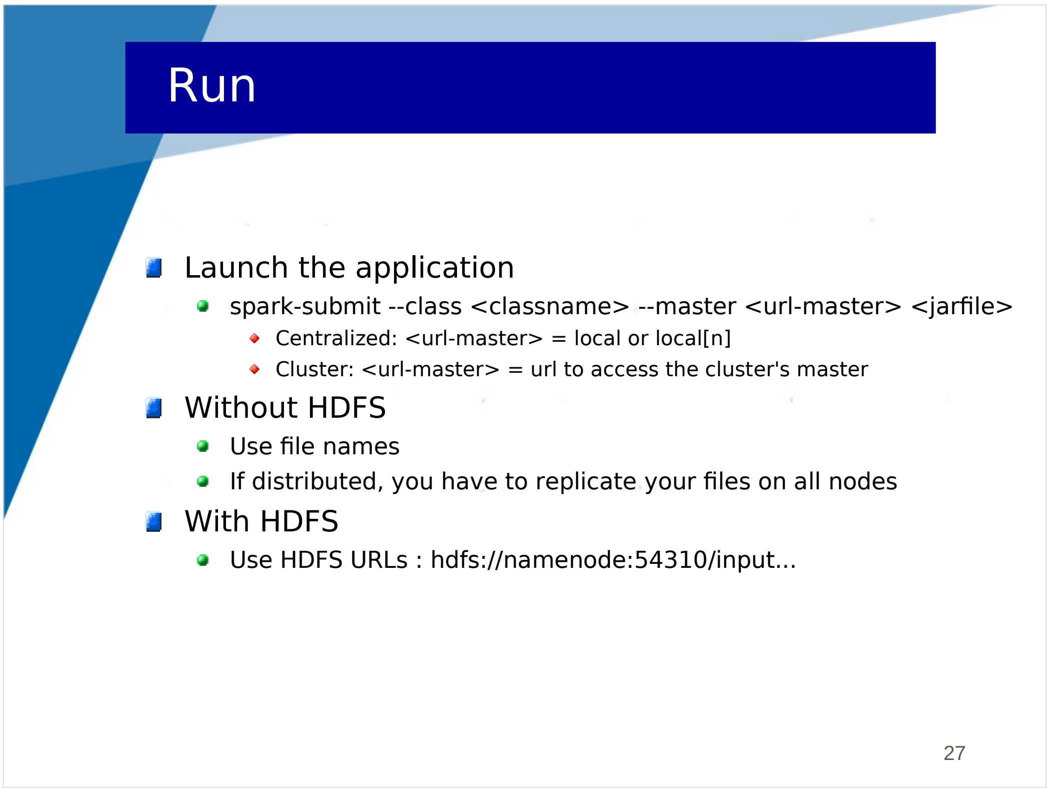
Development

- With eclipse
  - Create a Java Project
  - Add jars in the build path
    - $SPARK_HOME/jars/spark-core_2.11-2.4.3.jar
    - $SPARK_HOME/jars/scala-library-2.11.12.jar
    - $SPARK_HOME/jars/hadoop-common-2.7.3.jar
    - Could include all jars, but not very clean
  - Your application should be packaged in a jar

As usually, I prefer using eclipse for edition of code only and not for running programs (I run programs with shell commands only).

With eclipse, you just need to create a Java project and add the given jars in the build-path.

When your application is developed, it has to be packaged in a jar which includes the compiled Java classes. This can be done with eclipse or using a shell command (jar).



Run

- Launch the application
  - spark-submit --class <classname> --master <url-master> <jarfile>
    - Centralized: <url-master> = local or local[n]
    - Cluster: <url-master> = url to access the cluster's master
- Without HDFS
  - Use file names
  - If distributed, you have to replicate your files on all nodes
- With HDFS
  - Use HDFS URLs : hdfs://namenode:54310/input...

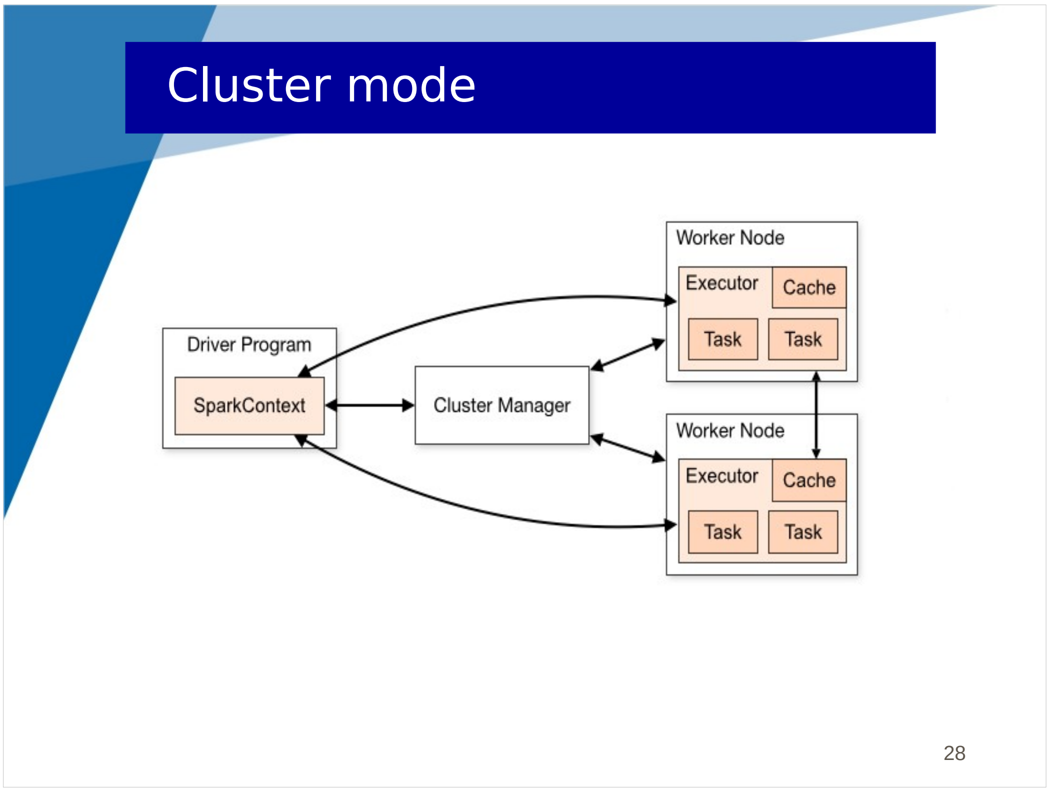
To run an application, you have to use the spark-submit command (accessible in your PATH), giving the name of the main class, the name of the jar archive and a url-master which can be :

- "local" if you run the application locally (one node) for testing

- "local[n]" if you want to run it locally with n cores (processors)

- the URL of the master daemon if you want to run it in a cluster (explained later)

You can run Spark without HDFS, but if distributed, you have to replicate files on all nodes. With HDFS, file name becomes HDFS URLs.



Cluster mode

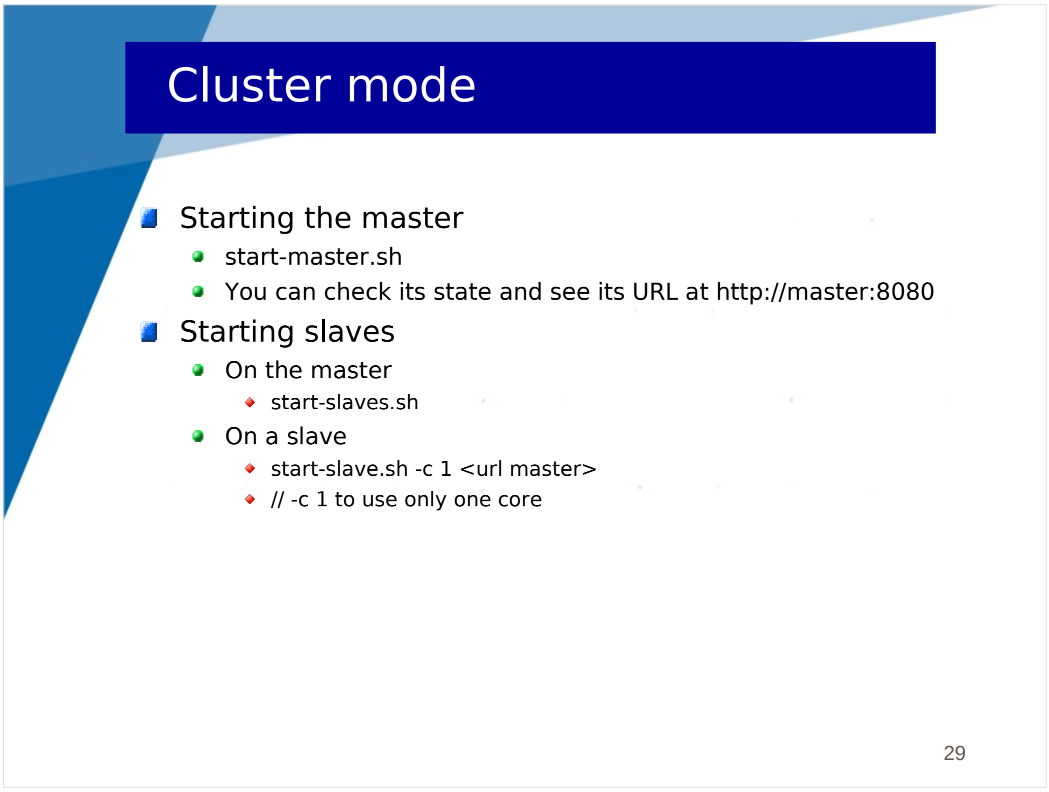
28

When running in cluster mode, you have :

- a master daemon running on one node (called master node)

- one worker daemon running on each compute node (called slave nodes)

When launching an application, the Driver program is run in the master and all operations on partitions are run in slaves.



Cluster mode

- Starting the master
  - start-master.sh
  - You can check its state and see its URL at http://master:8080
- Starting slaves
  - On the master
    - start-slaves.sh
  - On a slave
    - start-slave.sh -c 1 <url master>
    - // -c 1 to use only one core

To deploy a cluster, you have to launch :

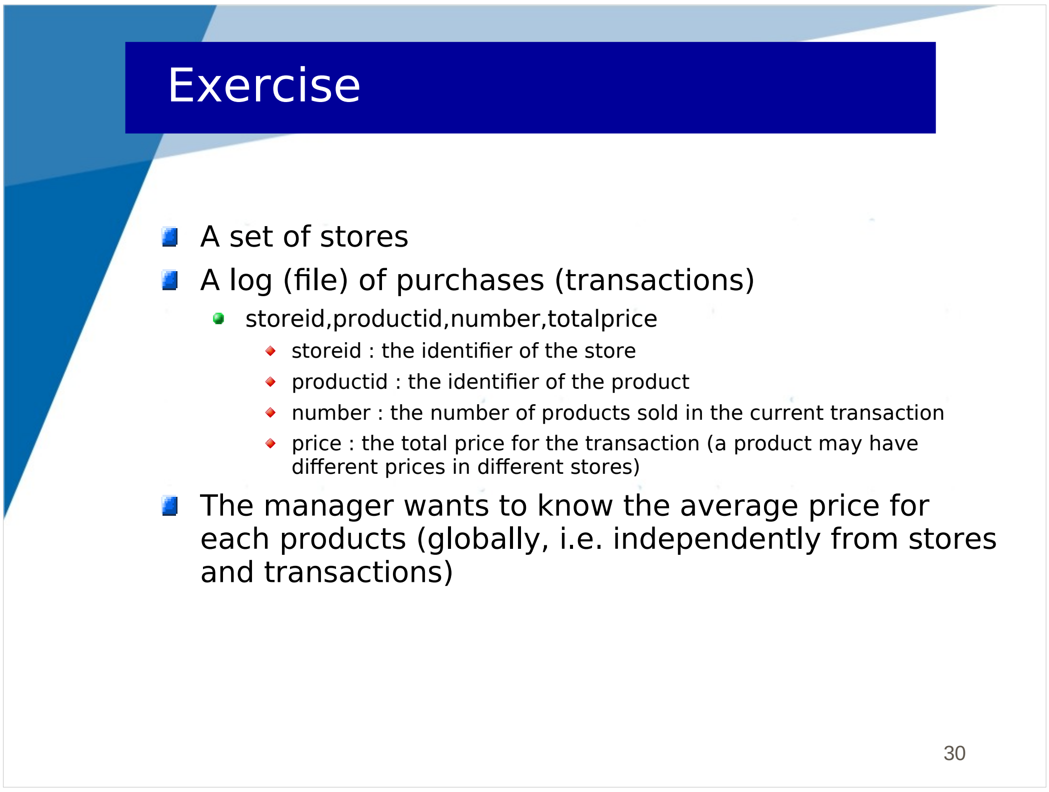
- start-master.sh on the master node to start the master daemon

You can check its state with a web console (and also copy the URL of the master, spark://masternode:7077)

- start-slaves.sh on the master node to start the worker daemons on the slave nodes (assuming that you have configured a "slaves" file describing the nodes where workers should be started)

Alternatively, you can start slaves (by hand) on each node with start-slave.sh giving as parameter the URL of the master.

The labworks will demonstrate the deployment in local and cluster mode.



Exercise

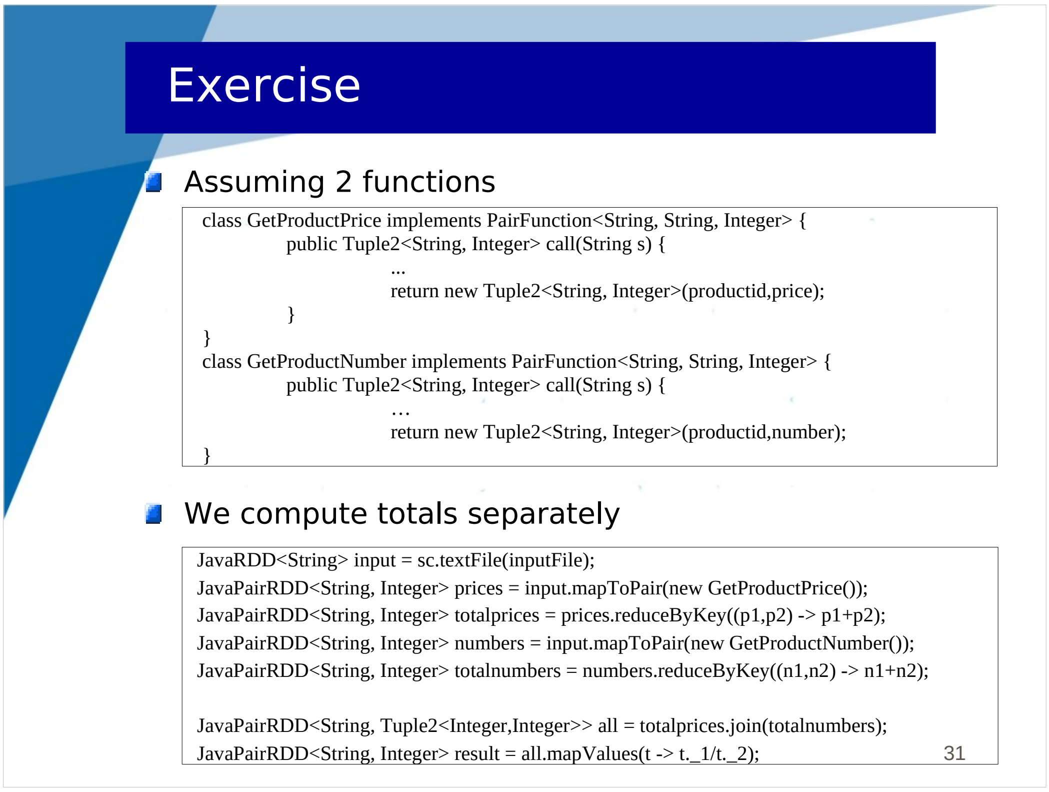
- A set of stores
- A log (file) of purchases (transactions)
  - storeid,productid,number,totalprice
    - storeid : the identifier of the store
    - productid : the identifier of the product
    - number : the number of products sold in the current transaction
    - price : the total price for the transaction (a product may have different prices in different stores)
- The manager wants to know the average price for each products (globally, i.e. independently from stores and transactions)

We here consider an programming exercise with Spark.

We assume a set of stores which record all their sales in the unique large file.

In this file, each line corresponds to a sale and gives the store identifier, product identifier, number of sold items and total price.

First question : compute the average price for each product



Exercise

- Assuming 2 functions

```
class GetProductPrice implements PairFunction<String, String, Integer> {
        public Tuple2<String, Integer> call(String s) {
                ...
                return new Tuple2<String, Integer>(productid,price);
        }
}
class GetProductNumber implements PairFunction<String, String, Integer> {
        public Tuple2<String, Integer> call(String s) {
                …
                return new Tuple2<String, Integer>(productid,number);
}
```

- We compute totals separately

```
JavaRDD<String> input = sc.textFile(inputFile);
JavaPairRDD<String, Integer> prices = input.mapToPair(new GetProductPrice());
JavaPairRDD<String, Integer> totalprices = prices.reduceByKey((p1,p2) -> p1+p2);
JavaPairRDD<String, Integer> numbers = input.mapToPair(new GetProductNumber());
JavaPairRDD<String, Integer> totalnumbers = numbers.reduceByKey((n1,n2) -> n1+n2);

JavaPairRDD<String, Tuple2<Integer,Integer>> all = totalprices.join(totalnumbers);
JavaPairRDD<String, Integer> result = all.mapValues(t -> t._1/t._2);
```

31

We assume that 2 Java functions are available :

- GetProductPrice is a PairFunction (a function which returns a pair, so the 2 last parameters (String, Integer) are the types of the pair). The first parameter (String) is a line. This function takes a line and extracts the productid and price fields in a pair. You can see that the parameters of PairFunction (types), the parameters of call() and the return statement are consistent.

- GetProductNumber is very similar, but it returns in a pair the productId and number of items.

Then:

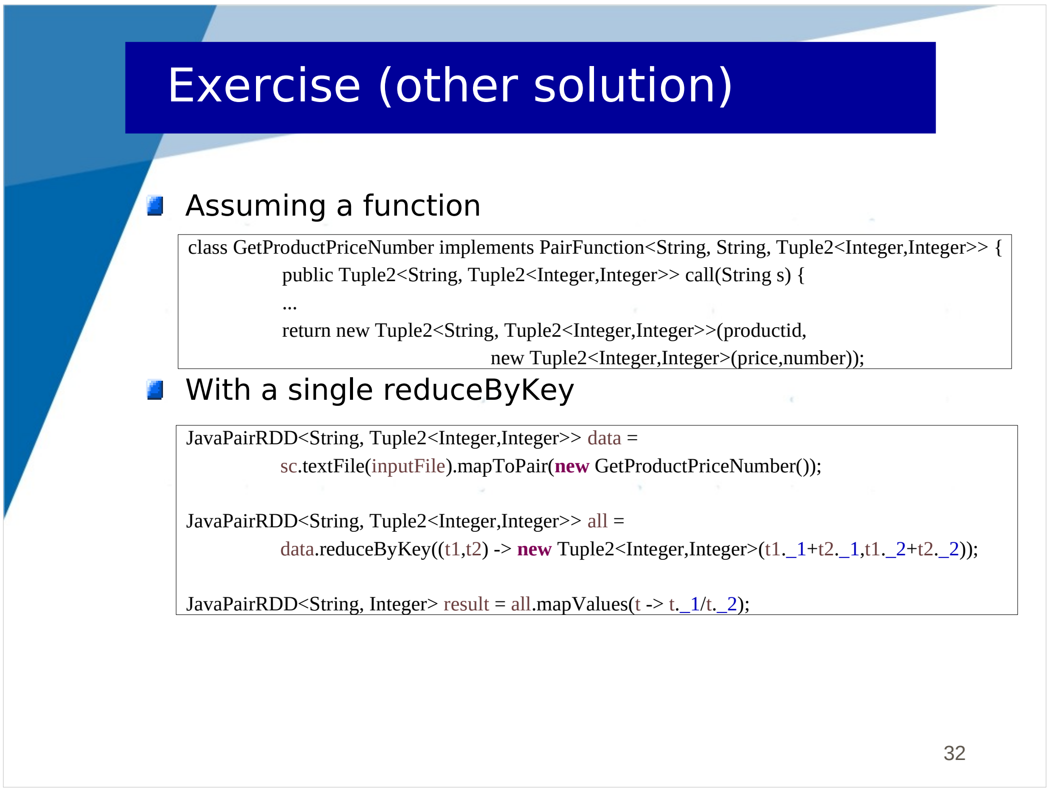
- prices is a JavaPairRDD which gives for each sale a pair <productid,price>

- with reduceByKey(), we obtain a JavaPairRDD which gives for each productid a pair <productid, totalprice>

- we do the same to obtain a JavaPairRDD which gives for each productid a pair <productid, totalnumber>

- with join(), we obtain a JavaPairRDD which gives for each productid a pair <productid, <totalprice, totalnumber>>

- mapValues() allows to apply the same function to all values in a JavaPairRDD



Exercise (other solution)

- Assuming a function

```
class GetProductPriceNumber implements PairFunction<String, String, Tuple2<Integer,Integer>> {
        public Tuple2<String, Tuple2<Integer,Integer>> call(String s) {
        ...
        return new Tuple2<String, Tuple2<Integer,Integer>>(productid,
                                new Tuple2<Integer,Integer>(price,number));
```

- With a single reduceByKey

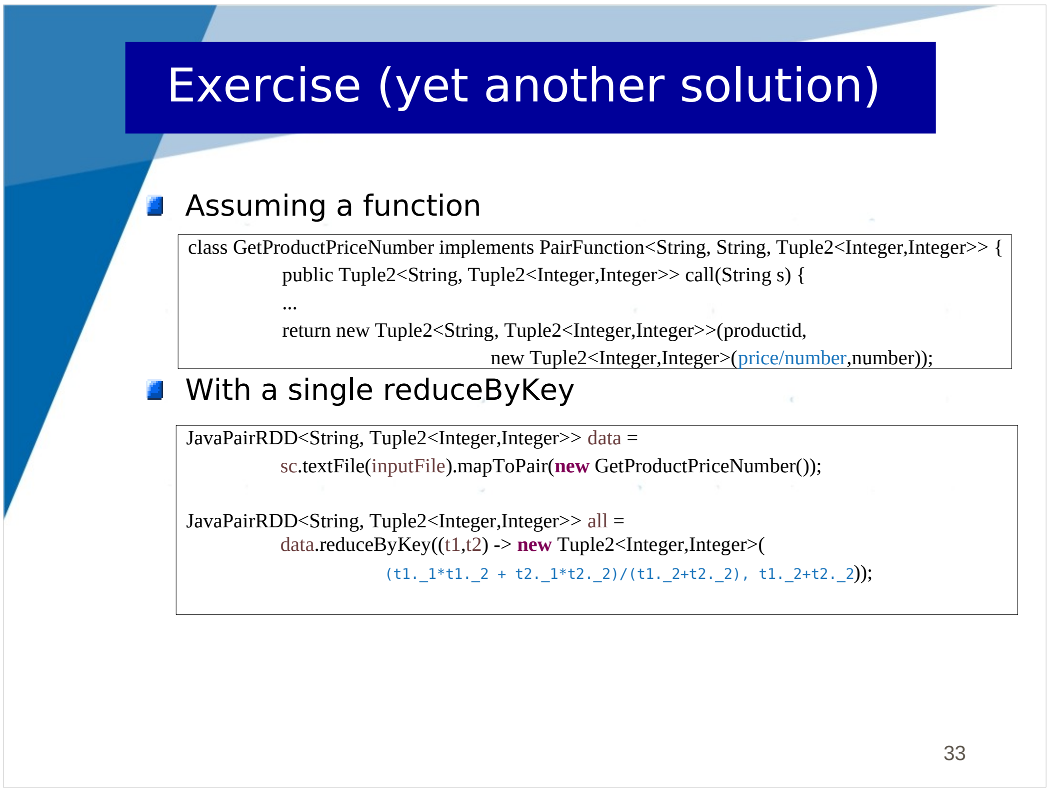
```
JavaPairRDD<String, Tuple2<Integer,Integer>> data =
        sc.textFile(inputFile).mapToPair(new GetProductPriceNumber());

JavaPairRDD<String, Tuple2<Integer,Integer>> all =
        data.reduceByKey((t1,t2) -> new Tuple2<Integer,Integer>(t1._1+t2._1,t1._2+t2._2));

JavaPairRDD<String, Integer> result = all.mapValues(t -> t._1/t._2);
```

32

In the previous solution, we were doing many passes on the RDD.

In this other solution, we assume a function which returns for a line a pair : <productid, <price, number>>

Then, we can do it with a single reduceByKey(), where the function describes how two pairs (tuples) can be aggregated.



Exercise (yet another solution)

- Assuming a function

```
class GetProductPriceNumber implements PairFunction<String, String, Tuple2<Integer,Integer>> {
        public Tuple2<String, Tuple2<Integer,Integer>> call(String s) {
        ...
        return new Tuple2<String, Tuple2<Integer,Integer>>(productid,
                        new Tuple2<Integer,Integer>(price/number,number));
```

- With a single reduceByKey

```
JavaPairRDD<String, Tuple2<Integer,Integer>> data =
        sc.textFile(inputFile).mapToPair(new GetProductPriceNumber());

JavaPairRDD<String, Tuple2<Integer,Integer>> all =
        data.reduceByKey((t1,t2) -> new Tuple2<Integer,Integer>(
                (t1._1*t1._2 + t2._1*t2._2)/(t1._2+t2._2), t1._2+t2._2));
```

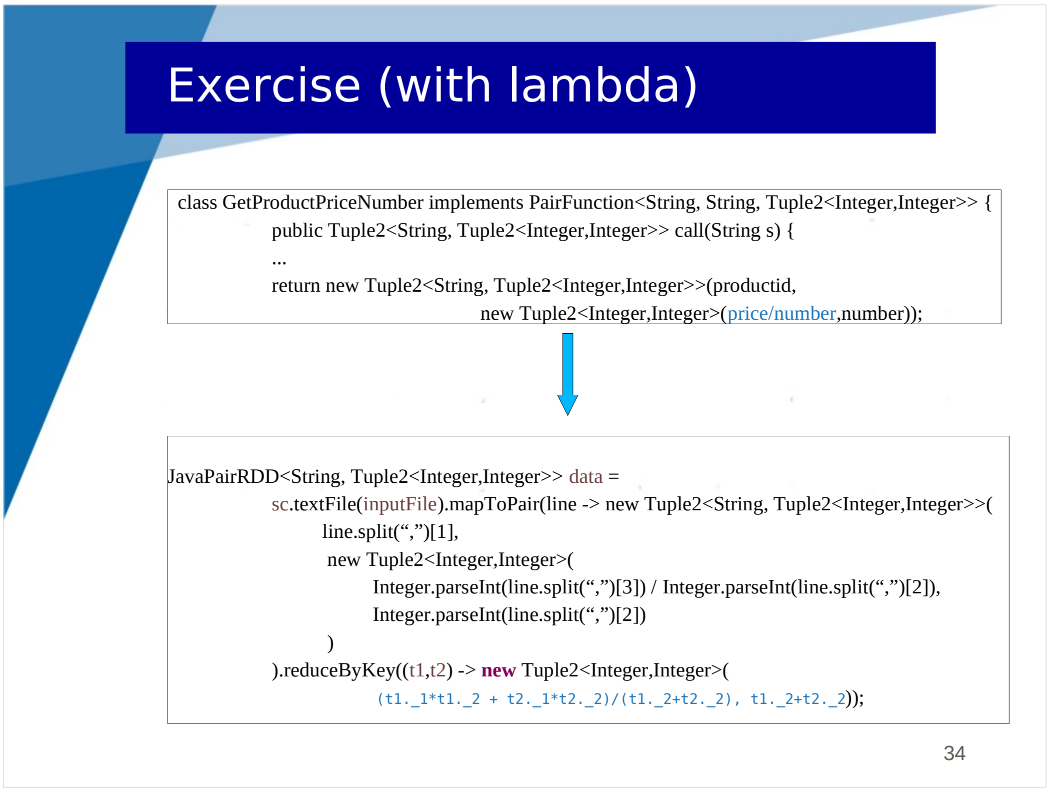
33

Here, in order to avoid to do another pass at the end to compute averages:

- the function GetProductPriceNumber returns for each product the average price and the number of products on which the average was computed

- the aggregator in reduceByKey compute for 2 averages the aggregated average

We don't need to do another pass with mapValues().



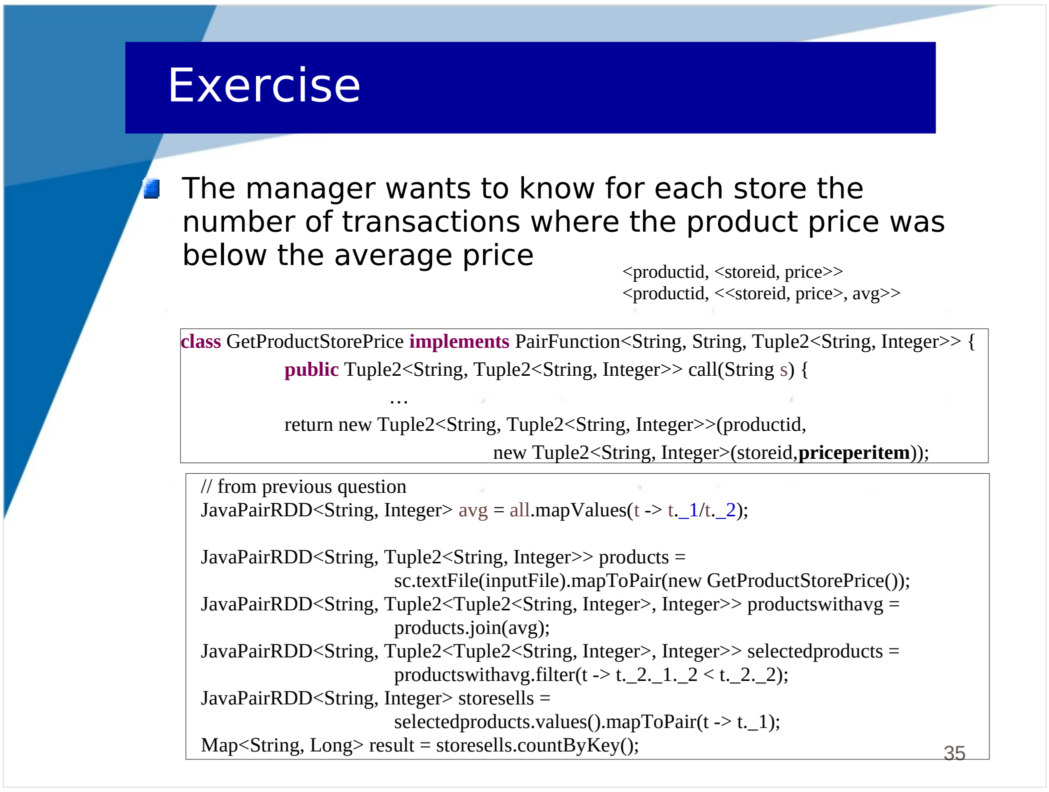
Exercise (with lambda)

```
class GetProductPriceNumber implements PairFunction<String, String, Tuple2<Integer,Integer>> {
        public Tuple2<String, Tuple2<Integer,Integer>> call(String s) {
        ...
        return new Tuple2<String, Tuple2<Integer,Integer>>(productid,
                                        new Tuple2<Integer,Integer>(price/number,number));
```

```
JavaPairRDD<String, Tuple2<Integer,Integer>> data =
        sc.textFile(inputFile).mapToPair(line -> new Tuple2<String, Tuple2<Integer,Integer>>(
            line.split(“,”)[1],
            new Tuple2<Integer,Integer>(
                Integer.parseInt(line.split(“,”)[3]) / Integer.parseInt(line.split(“,”)[2]),
                Integer.parseInt(line.split(“,”)[2])
            )
        ).reduceByKey((t1,t2) -> new Tuple2<Integer,Integer>(
                (t1._1*t1._2 + t2._1*t2._2)/(t1._2+t2._2), t1._2+t2._2));
```

34

We can also write it with lambda functions.



Exercise

- The manager wants to know for each store the number of transactions where the product price was below the average price

<productid, <storeid, price>>
<productid, <<storeid, price>, avg>>

```
class GetProductStorePrice implements PairFunction<String, String, Tuple2<String, Integer>> {
                public Tuple2<String, Tuple2<String, Integer>> call(String s) {
                                …
                return new Tuple2<String, Tuple2<String, Integer>>(productid,
                                                new Tuple2<String, Integer>(storeid,priceperitem));
```

```
// from previous question
JavaPairRDD<String, Integer> avg = all.mapValues(t -> t._1/t._2);

JavaPairRDD<String, Tuple2<String, Integer>> products =
                        sc.textFile(inputFile).mapToPair(new GetProductStorePrice());
JavaPairRDD<String, Tuple2<Tuple2<String, Integer>, Integer>> productswithavg =
                        products.join(avg);
JavaPairRDD<String, Tuple2<Tuple2<String, Integer>, Integer>> selectedproducts =
                        productswithavg.filter(t -> t._2._1._2 < t._2._2);
JavaPairRDD<String, Integer> storesells =
                        selectedproducts.values().mapToPair(t -> t._1);
Map<String, Long> result = storesells.countByKey();
```

In this last question, we want to compute for each store the number of sales where the product price was below the average.

We assume a function which returns for a line a pair <productid, <storeid, priceperitem>>. The priceperitem car be easily computed in that function.

- avg is the JavaPairRDD result from the previous question

- products is a JavaPairRDD which gives for each sale a pair <productid, <storeid, priceperitem>>

- with join(), productswithavg is a JavaPairRDD which gives for each sale a pair <productid, <<storeid, priceperitem>, average>>

- with filter(), selectedproducts is a JavaPairRDD which keeps only sales where the priceperitem is below the average

- storesells is a JavaPairRDD <storeid, priceperitem> for the sales where the priceperitem is below the average.

I would have liked to write :

JavaPairRDD<String, Integer> storesells = selectedproducts.values().keys();

But, it doesn't work because values() and keys() on JavaPairRDD both return a JavaRDD (which may include a pair, but it's not a JavaPairRDD), so we need to use a mapToPair() to obtain a JavaPairRDD.

- the last line counts the number of sales (below the average) for each store. The Map result is in the Driver program.