Spark and RDD

Introduction

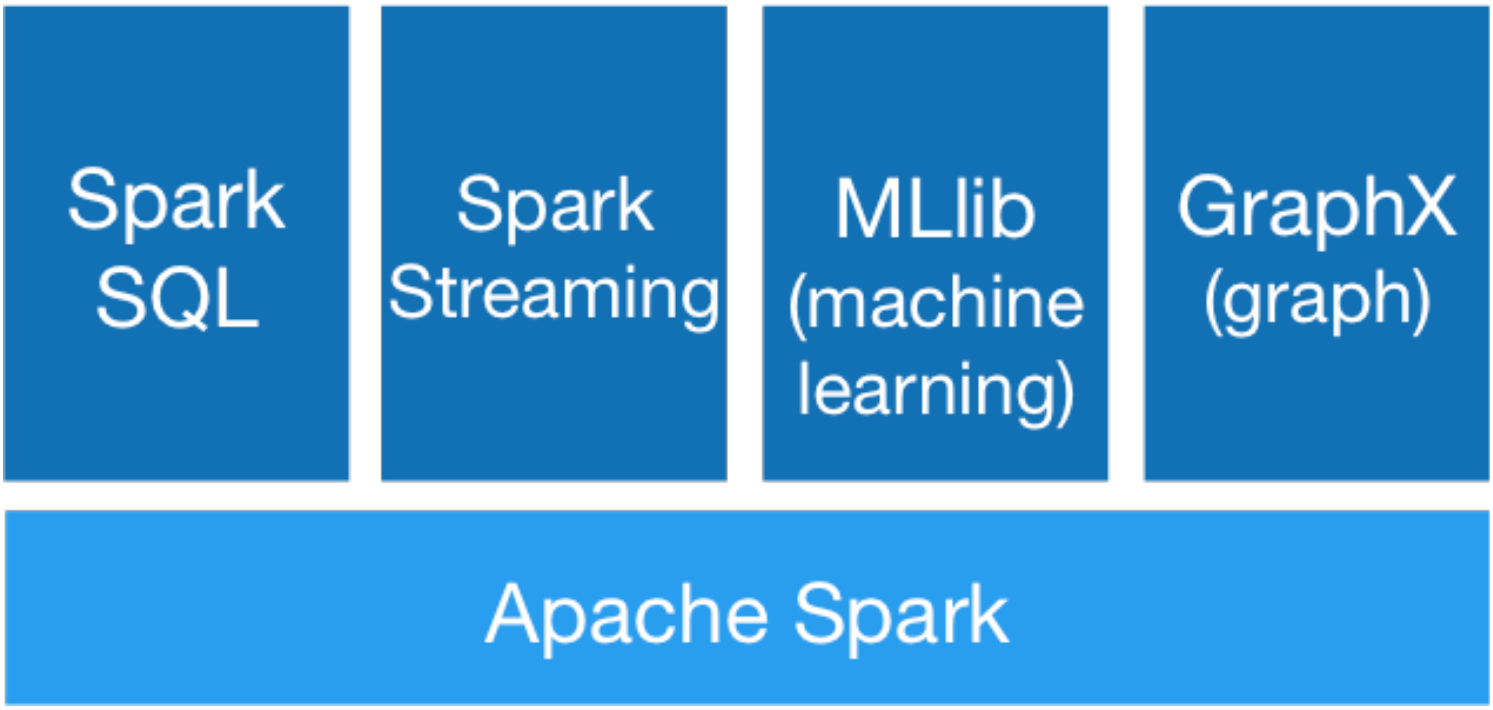
One drawback of MapReduce is that it relies on node computers frequently reading and writing data from their disks. In the map phase, data is read from disk, processed, and the results are written back to disk. In the reduce phase, these intermediate results are read from disk, aggregated, and the final result is written back to disk. This reading and writing is very slow. If your job requires two or more iterations of mapping and reducing then the problem is even worse.

To avoid this problem a piece of software called **Spark** has been developed. Spark still uses Hadoop and YARN to access data stored in a cluster (although it has its own versions of these if you'd prefer to use them). But it provides a way of working with that data that relies much less on disk reading and writing, by keeping data in node RAM instead. This makes Spark much faster than MapReduce, especially if the mapper and reducer functions need to be called iteratively. Spark is up to 100 times faster when all of the data is kept in memory, but still up to 10 times faster when it has to resort to disk.

You can execute Spark commands directly, or you can execute them by writing programs in Java, Python, or Scala. You will learn how to do so in Python, using Python's **pyspark** library.

Spark Components

Spark also has a rich set of higher-level tools for performing special kinds of tasks: querying data SQL-style (**Spark SQL**), processing streaming data (**Spark Streaming**), processing graph data (**GraphX**), and doing machine learning (**MLLib**).



Spark Core

The Spark Core is the underlying general execution engine for Spark that all other functionality is built upon.

Spark SQL

Spark SQL is a component on top of Spark Core for processing structured data. With Spark SQL, you can use your knowledge of SQL to work with big data.

Spark Streaming

Spark Streaming uses Spark Core's fast processing speed to work with live data streams. Many organisations (such as in Twitter) have to work with large volumes of streaming data.

MLlib

MLlib is Spark's machine learning (ML) library, whose goal is to make machine learning practical and easy. It provides APIs to support many machine learning tasks, such as classification, regression, clustering, feature extraction, dimensionality reduction, and so on.

GraphX

GraphX is a component of Spark for working with graphs.

History

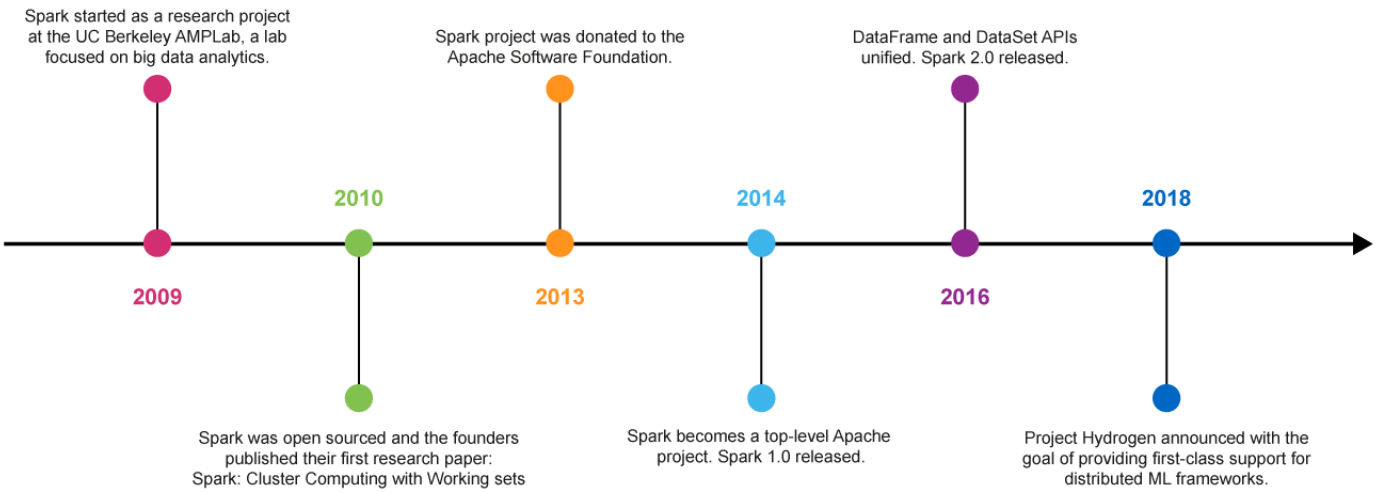
Spark was developed at the University of California, Berkeley campus AMPLab in 2009. It was open-

sourced in 2010 and then donated to the Apache Software Foundation in 2013.

In February 2014, Spark became a top-level Apache project and Spark 1.0 was released. In November 2014, Spark founder M. Zaharia's company Databricks set a new world record in large scale sorting using Spark.

In 2016, Spark 2.0 was released with a new Dataset object whose use is encouraged over the original RDD object.

Currently Spark is one of the most active projects in the Apache Software Foundation and one of the most active open source big data projects.



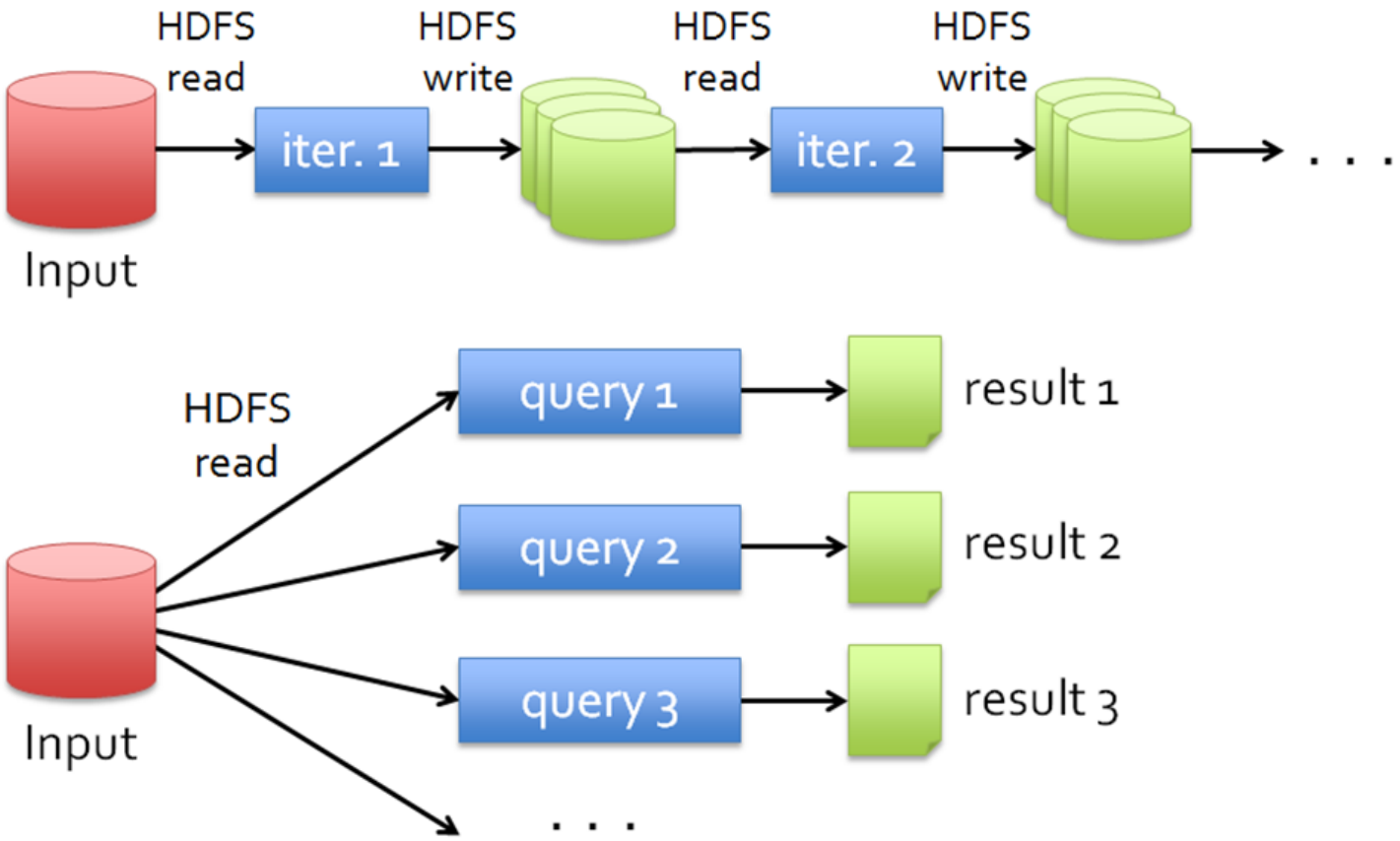
Why Spark?

In weeks 4 and 5, we have learned Hadoop MapReduce. Then, why we need to learn a new tool in this week?

The reason is: MapReduce has some limitations. MapReduce greatly simplified big data analysis on large, unreliable clusters, but it is only great at one-pass computation. Think about the following tasks:

- More **complex jobs** that require multi-pass analytics. For example, machine learning algorithms (such as K-Means clustering) or graph algorithms (such as shortest path computation)
- More **interactive** ad-hoc queries (such as SQL-like queries)
- More **real-time** stream processing (such as finding the most frequent items in the transactions of the past week).

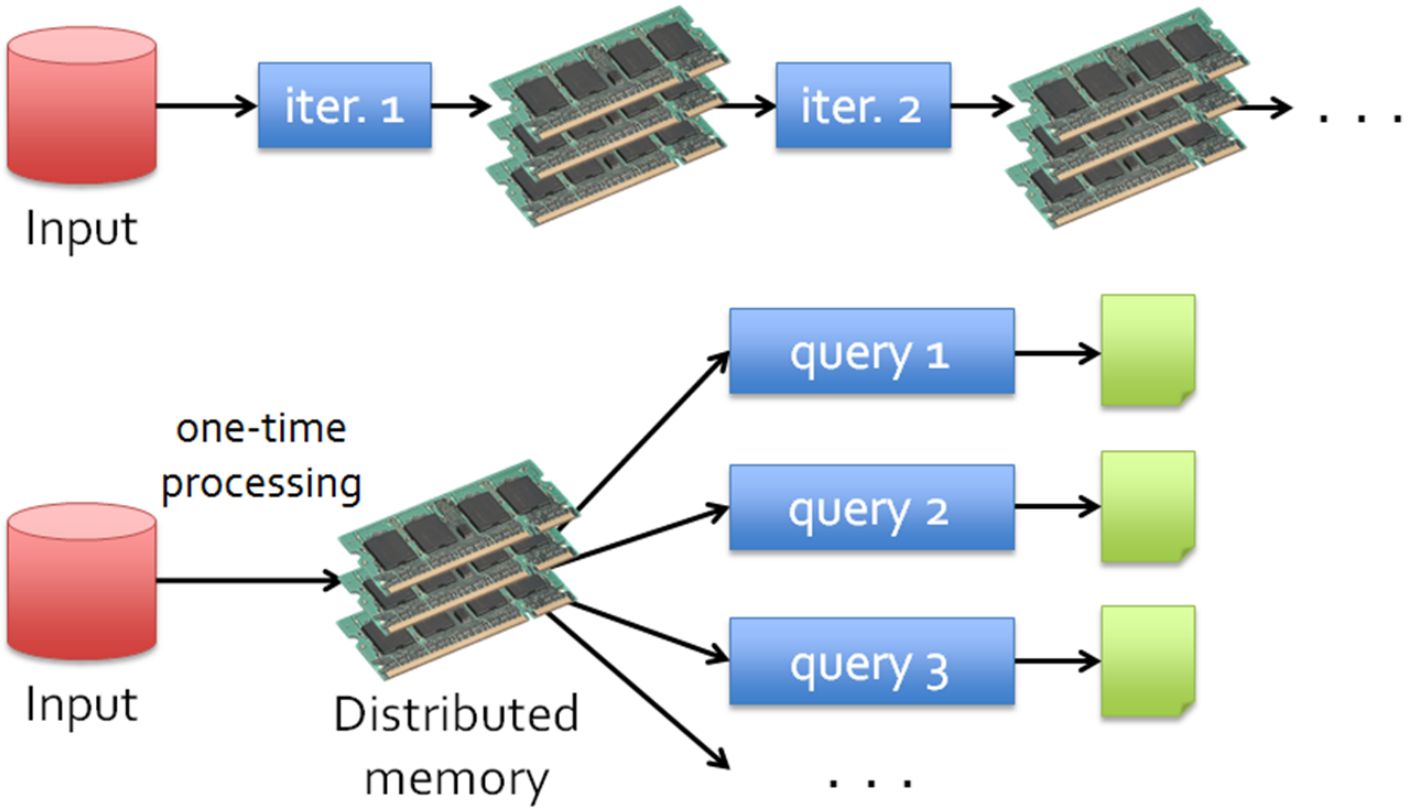
All such tasks require huge data shared across different nodes in a cluster in multiple round jobs. However, data sharing is very inefficient in MapReduce, since it is based on HDFS.



As shown in the Figure, multiple round MapReduce jobs need to frequently read data from and write data to HDFS. For processing different queries, each time MapReduce has to read the data from HDFS once and scan the data from the beginning to the end.

MapReduce has utilized the CPUs and hard drives in the cluster for distributed computation. What has not been utilized so far? The answer is: **memory**!

Hence, Spark aims to keep more data in-memory to improve the performance! In Spark, the data sharing is through memory rather than HDFS, and thus is much more efficient, as shown below:



Note that Spark is not a modified version of Hadoop. It dependent on Hadoop because it has its own cluster management. Spark uses Hadoop for storage purpose only (Spark can also use the local disk for storing data).

RDD and RDD operations

In Spark you work with **Resilient Distributed Datasets** (**RDDs**).

An RDD is a collection of data items. Each data item can be any type of Python, Java, or Scala object, including user-defined classes. It is **resilient**, because it is fault-tolerant - Spark automatically recomputes missing or damaged data when a node fails. It is **distributed**, because this data is kept on multiple nodes across a cluster. As much as possible the data in an RDD is stored in memory.

RDD has the following traits:

- **In-Memory**: data inside an RDD is stored in memory as much (size) and as long (time) as possible.
- **Immutable or Read-Only**: it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**: the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**: you can hold all the data in a persistent *storage*, like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**: process data in parallel.
- **Typed**: values in an RDD have types, for example, RDD[Long] or RDD[(Int, String)].
- **Partitioned**: the data inside an RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

Operating on an RDD

Spark uses RDD to manage the data. There are two types of operations you can perform on an RDD: **transformations** and **actions**. A transformation creates a new RDD from an existing RDD. An action performs a computation on the RDD and returns the result. The idea can be shown as the below figure.

The best way to see how the transformations and actions work is to see them in action, and we will be going through a number of examples. But here is a list of the most important ones:

Transformations

A transformation creates a new RDD from an existing one. Here are some commonly used ones:

`rdd.distinct()`

Returns a new RDD that contains the distinct elements of `rdd`.

`rdd.filter(func)`

Returns a new RDD obtained by applying `func` to each element of `rdd`, keeping elements that return true and discarding elements that return false.

`rdd.map(func)`

Returns a new RDD obtained by applying `func` to each element of `rdd`.

`rdd.flatMap(func)`

Similar to `map()`, but `func` can return multiple items.

`rdd.sortByKey()`

Only for pair RDDs. Returns a new RDD with the same <key, value> pairs but sorted with the keys in ascending order (use `sortByKey(false)` for descending order).

`rdd.groupByKey()`

Only for pair RDDs. Groups all values for the same key into a collection `values` and returns a new RDD with of <key, values> pairs.

`rdd.reduceByKey(func)`

Only for pair RDDs. Uses `func` to reduce all values for the same key to a single value `total` and

returns a new RDD of <key, total> pairs.

More transformation operations can be found at: https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations.

Actions

An action performs a computation on the RDD and returns the result. Here are some commonly used ones:

`rdd.count()`

Returns the number of items in `rdd`.

`rdd.countByValue()`

Only for pair RDDs. Returns the number of each value in `rdd`.

`rdd.countByKey()`

Only for pair RDDs. Returns the number of each key in `rdd`.

`rdd.first()`

Returns the first element of `rdd`.

`rdd.take(n)`

Returns the first `n` elements of `rdd` as an array. `take(1)` is equivalent to `first()`.

`rdd.collect()`

Returns all the elements of `rdd` as an array.

`rdd.saveAsTextFile(path)`

Saves `rdd` as a text file to HDFS at the location `path`.

`rdd.reduce(func)`

Aggregates the elements of `rdd` using `func`. Because the computations occur in parallel, `func` must be commutative and associative.

More action operations can be found at: https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions.

We will practice the transformation and action operations in some examples later.

Further reading:

1. RDDs were first proposed in the following paper:

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley,...Stoica, I. (2012). *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA.

2. More detailed RDD programming guide can be found from the Spark's official website: https://spark.apache.org/docs/latest/rdd-programming-guide.html

Creating an RDD

Spark provides a special kind of object, a **SparkContext**, which has the properties and methods that you need to work with RDDs. Later you will see how to create a SparkContext in Python. In the rest of this slide we will assume that a SparkContext has been created, called `sc`. SparkContext is the entry point to Spark for a Spark application. Once a SparkContext instance is created, you can use it to

- Create RDDs
- Create accumulators
- Create broadcast variables
- Access Spark services and run jobs

Spark provides two ways to create an RDD.

Using parallelize()

You can use this to create an RDD from a Python object. Here's an example in Python:

```
data = [1, 2, 3, 4]
numbers = sc.parallelize(data)
```

In this example we create a Python list called "data", and then we pass it to the `parallelize()` method of `sc`, the SparkContext object.

This is very useful when you're learning Spark, since you can quickly create your own RDDs. However, it's not widely used in real tasks since it requires that you have your entire dataset in memory on one machine (for most big data problems your memory is not enough to hold all the data).

Using textFile()

This is a much more common way to create an RDD. Suppose that you have a text file stored in the home folder of your local file system, called "text.txt". You can, using Python, create an RDD as follows:

```
lines = sc.textFile("file:///home/text.txt")
```

You can also create an RDD from a file stored in HDFS. We will practice creating RDDs in later slides.

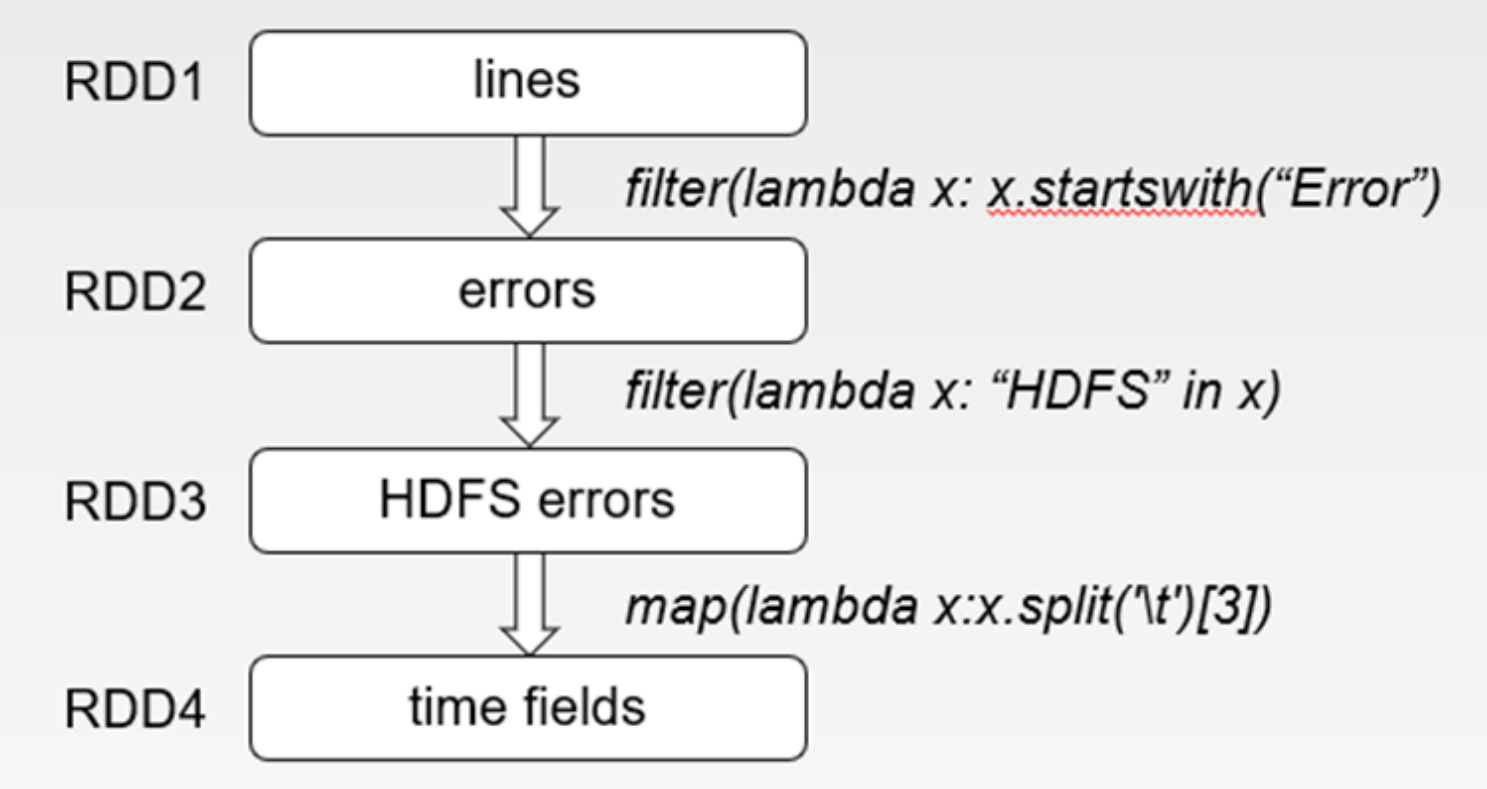
Lazy evaluation

One of the features that makes Spark fast is that transformations on RDDs are **lazily evaluated**, which means that Spark will not execute any of them until it sees an action. When we call a transformation on an RDD the operation is not immediately performed. Instead, Spark remembers that this transformation has been requested. So, rather than thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations. When an action operation is called on an RDD object, all the transformations are then performed and the resulting value is returned.

For example, assume that a web service is experiencing errors and operators want to search terabytes of logs in the Hadoop file system to find the cause. Consider the following Spark command:

```
lines = sc.textFile("hdfs://…") //base RDD, obtained from a file on HDFS
errors = lines.filter(lambda x: x.startswith("Error")) //get messages that start with "Error", "err
errors.persist() //persist the data in memory
errors.count()
errors.filter(lambda x: "HDFS" in x).map(lambda x:x.split('\t')[3]).collect()
```

The code would generate four RDDs, as shown below:



Since `filter()` and `map()` are transformation operations, Spark does not perform any computations until it sees the action operation `collect()`, and only then are the executions are started. Loading data into an RDD is also lazily evaluated, since it is a transformation operation as well. So, when we call `sc.textFile("hdfs://...")`, the data is not loaded from HDFS until it is

necessary (i.e. an action operation is applied to the data).

To explain the code line by line:

- Line 1: loads RDD from an HDFS file (base RDD “lines” not loaded in memory at this stage).
- Line 2: filter out lines that are not about error messages, and only keep lines starting with "Error".
- Line 3: asks for “errors” to persist in memory (“errors” are in RAM, will explain later)
- Line 4: counts the number of error messages in the error RDD.
- Line 5: assume that the operators first are going to find out the time when HDFS errors happened, which is recorded in the fourth field of the error message. This line of code first get all the error messages relevant to "HDFS", and then split the text message and get the error time information field. All the execution will happen at this line, when the "collect()" operation is seen by Spark.

Lazy evaluation allows Spark to reduce the number of passes it has to take over the data by grouping operations together. In systems like Hadoop MapReduce, developers often have to spend a lot of time considering how to group together operations to minimize the number of MapReduce passes. In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations. Thus, users are free to organize their program into smaller, more manageable operations.

Persisting an RDD

One of the most important capabilities in Spark is persisting (or caching) an RDD in memory across operations. When you persist an RDD, each node stores its part of the RDD. You can reuse the RDD in other actions on that dataset.

You can persist an RDD by calling the persist() method on it:

```
rdd.persist()
```

By default, the RDD is stored in memory only - if the RDD does not fit into memory then some parts of it will not be persisted and will be recomputed when they're needed. You can also specify that you'd like Spark to use memory and disk - If the RDD does not fit in memory then some parts of it will be stored on disk and read them from there when they're needed. You do this as follows:

```
rdd.persist(StorageLevel.MEMORY_ONLY)
```

Why persist an RDD?

Consider the example in the previous slide again, if we do like:

```
lines = sc.textFile("hdfs://…")
errors = lines.filter(lambda x: x.startswith("Error"))
errors.count()
errors.filter(lambda x: "HDFS" in x).map(lambda x:x.split('\t')[3]).collect()
```

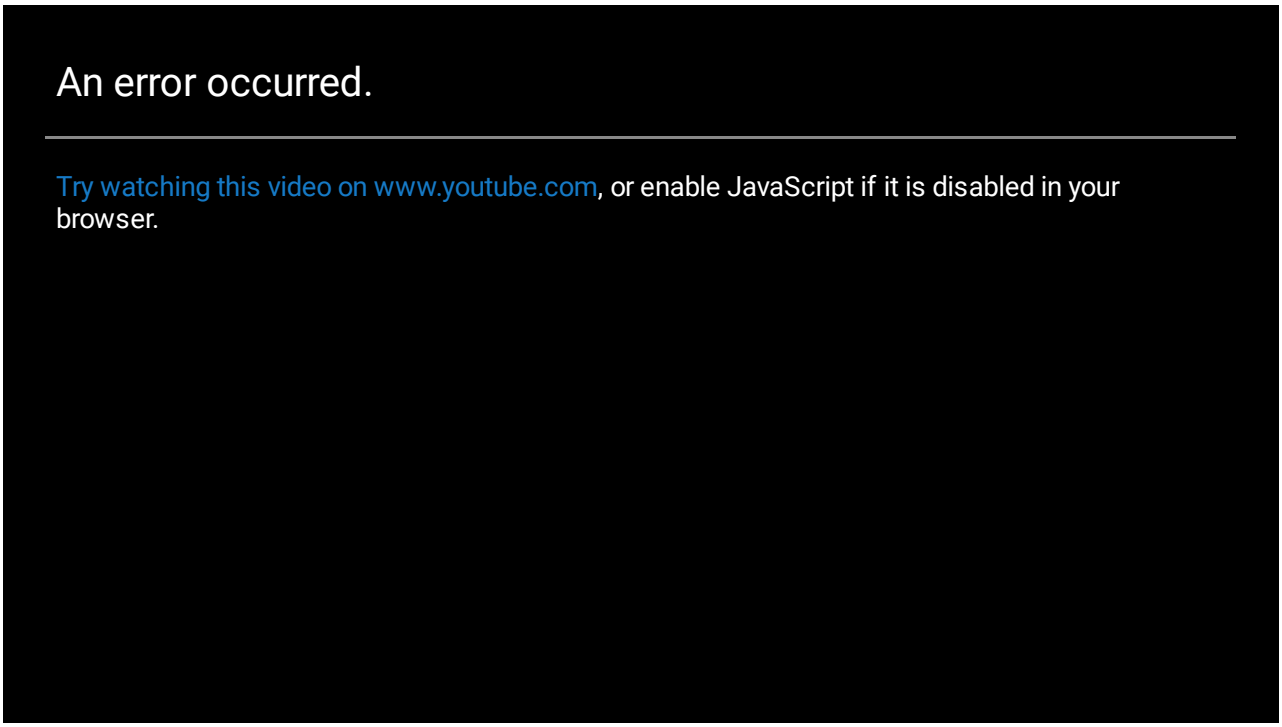
When you call `errors.count()`, the file is loaded, and the transformations and actions are performed, and then `errors` is released in memory. When you next call errors.filter(...), the whole process happens again in order to obtain the RDD `errors`.

Contrast this with the following code:

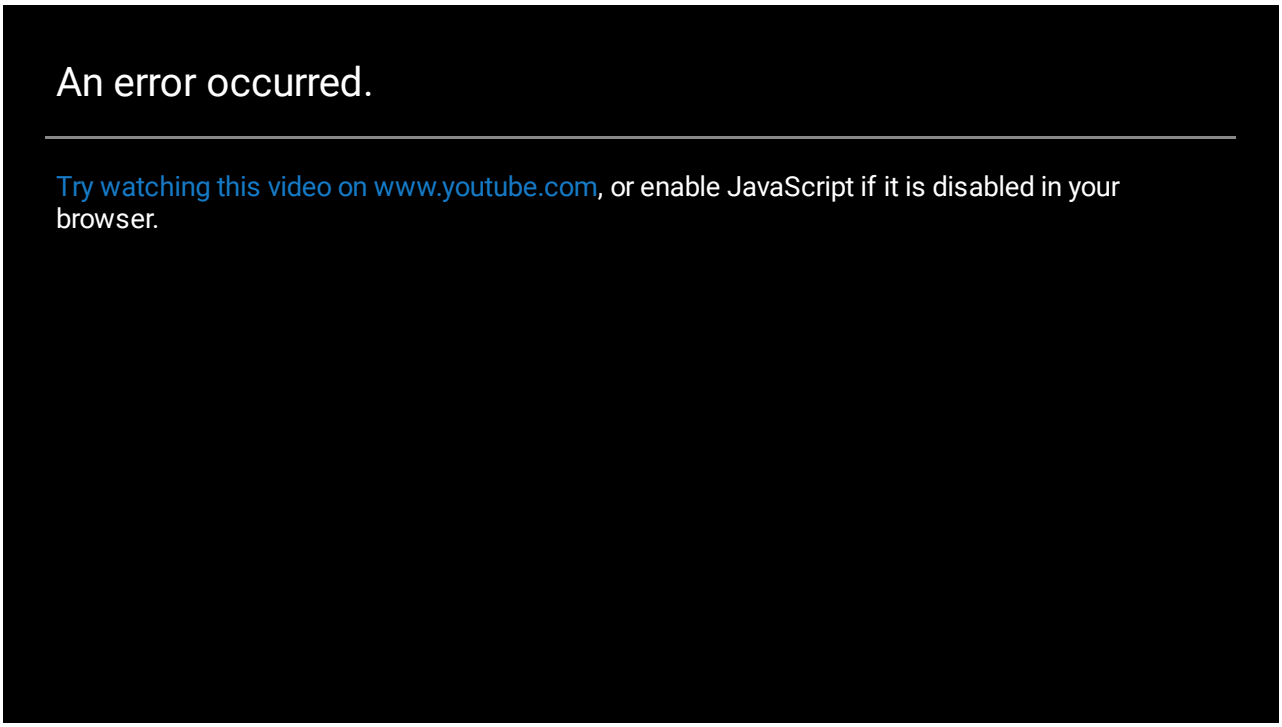
```
lines = sc.textFile("hdfs://…")
errors = lines.filter(lambda x: x.startswith("Error"))
errors.persist()
errors.count()
errors.filter(lambda x: "HDFS" in x).map(lambda x:x.split('\t')[3]).collect()
```

As in the first example, when you call `errors.count()`, the file is loaded, and the transformations and actions are performed. But when next you call errors.filter(...), since `errors` has been kept in memory already, it is not needed to load the file and compute for it again.

About Spark (38:19)



Hadoop vs Spark (15:28)



Spark in a Python shell

Let's see how to use Spark in a Python shell. We'll use it to explore a text file, including our usual example - the frequency of words in the file.

When you click the panel on the right you'll get a connection to a server with Hadoop and YARN installed and running.

Upload a file to HDFS

In your home directory on the server there is the file "text.txt". To work with this file using Spark we need to put it into HDFS. Use the following commands, with which you should be familiar by now:

```
$ hdfs dfs -mkdir -p /user/user
$ hdfs dfs -put text.txt
```

Start a Python shell

You can get a Python shell, with Spark loaded for you, using the following command:

```
$ pyspark
```

Exploring the file

Python is now running with a SparkContext object `sc` created for you. You can run Spark commands directly using at the command line.

First, create an RDD:

```
>>> text = sc.textFile("text.txt")
```

You now have an RDD called "text" which contains the contents of the file "text.txt". To see the number of items in the RDD you can use the `count()` action:

```
>>> text.count()
```

To see the first item of the RDD you can use the `first()` action:

```
>>> text.first()
```

To see the first three items of the RDD you can use the `take()` action:

```
>>> text.take(3)
```

To see the contents of the RDD you can use the `collect()` action:

```
>>> text.collect()
```

To see the lines that contain "war" you first filter the RDD, then collect it:

```
>>> text.filter(lambda line: "war" in line).collect()
```

The argument of `filter()` is an anonymous function, which takes a line of the file as input and returns true if this line contains “war” and false otherwise. As a result, only lines containing “war” are included in the resulting RDD.

You can use the `map()` function to map each line of the file to the number of words the line contains. Let's save it as a new RDD called "wordNums":

```
>>> wordNums = text.map(lambda line: len(line.split()))
```

The argument of `map()` is an anonymous function, which takes a line as the input and returns the number of words in the line (we're assuming that words are separated by a space). You can view the items in `wordNums` using the `collect()` action:

```
>>> wordNums.collect()
```

To the largest number of words in a line we can apply the `reduce()` action to `wordNums`:

```
>>> wordNums.reduce(lambda result, value: max(result, value))
```

You can try to cache the data in memory:

```
>>> wordNums.cache()
```

Word frequency

Now let's use Spark to get **the frequency of words in the file**.

First, let's create an RDD that contains the words in the file. We can do this by applying the `flatMap()` transformation to `text`, as follows:

```
>>> words = text.flatMap(lambda line: line.split())
```

We are using `flatMap()` here, rather than `map()`, because `line.split()` returns a list of words for each line, and we want to "flatten" all of those lists of words into a single list of words.

We can get the total number of words using `count()`:

```
>>> words.count()
```

We can get the number of *distinct* words by first transforming `words` using `distinct()`:

```
>>> words.distinct().count()
```

Compare the results with `words.count()`. You should see that duplicate words have not been counted twice.

Now count the frequency of each word.

First, use `map()` to transform `words` into a new RDD that contains each of the words in `words` paired with a 1 (i.e <word, 1>):

```
>>> pairs = words.map(lambda word: (word, 1))
```

Next, transform this RDD into a new RDD that contains the word frequencies:

```
>>> frequencies = pairs.reduceByKey(lambda total, value: total + value)
```

Finally, show the results:

```
>>> frequencies.collect()
```

You should get the same results as we did using MapReduce, MRJob and Hive.

You could also combine these commands into a single command:

```
>>> text.flatMap(lambda line: line.split()).map(lambda word: (word, 1)).reduceByKey(lambda total, va
```

Laziness

Take a look at that last command. It's only once it gets to the last part, `collect()`, that Spark actually works with the data. The previous commands have all been *transformations*, and since Spark is lazy it just remembers the transformations without actually performing them. But `collect()` is an *action*, which requires Spark to return some results. When it gets to this action Spark performs all of the transformations and then the action (after first working out the best way to do so).

Saving the results

If at any stage you would like to save an RDD, you can do so by using `saveAsTextFile()`, and providing a path for the file:

```
>>> frequencies.saveAsTextFile("results")
```

This command will save `frequencies` as a text file in HDFS, in a sub-directory of your default directory called "results".

You can check that this has happened.

First, you need to quit the Python shell. Use the following command:

```
>>> quit()
```

Now you can check for the file in HDFS. You could use the following command:

```
$ hdfs dfs -ls results
```

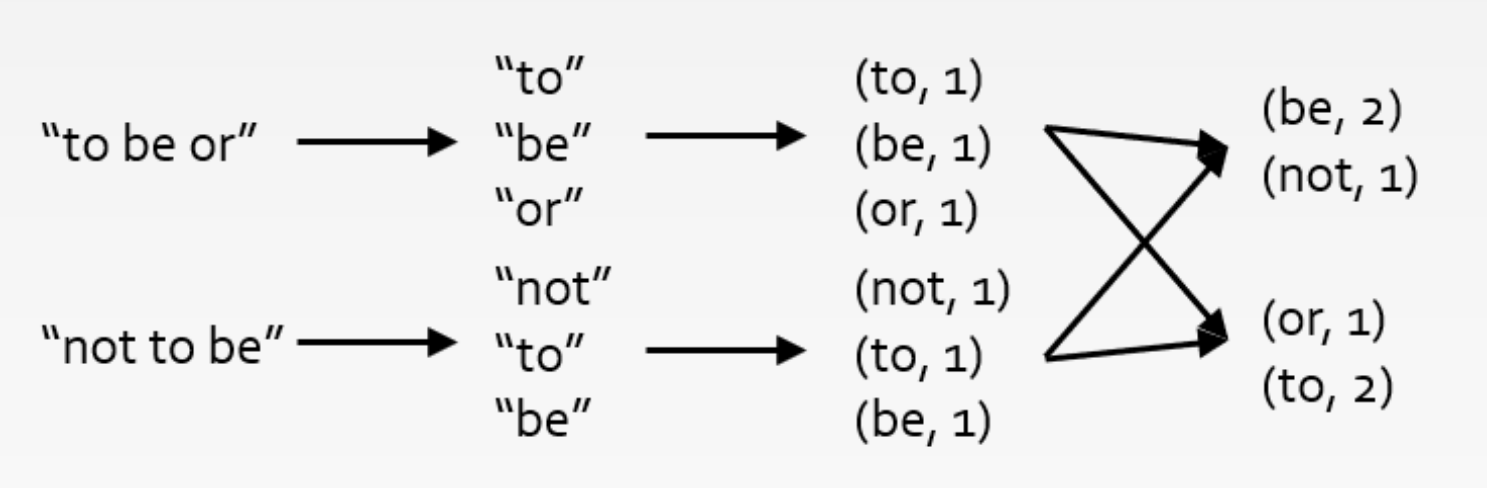
How Spark does word count

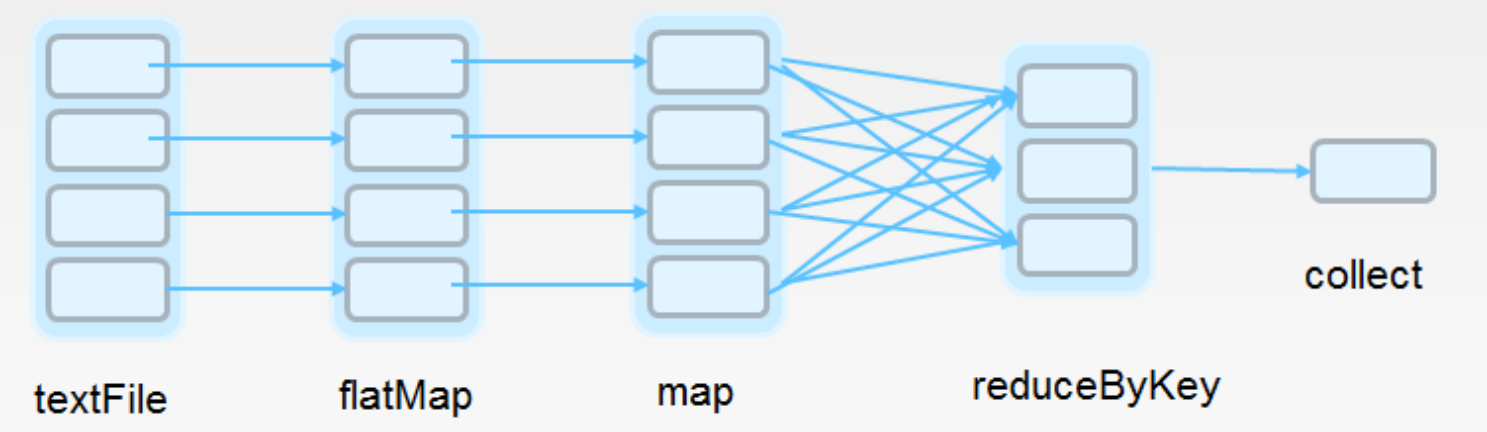
We further demonstrate how Spark solves big data problems using a Word Count example. The Spark code is like below:

```
textfile = sc.textFile("hdfs://...", 4)

words = textfile.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
count = pairs.reduceByKey(lambda a, b: a + b)
count.collect()
```

Here is a diagram of the transformations and actions we have performed to get the word frequencies:





Next, let's look at the code line by line.

1. textfile = sc.textFile("hdfs://...", 4)

We have learned that textFile() is a transformation operation to load file from disk to memory. Here we see one more parameter, the number of *partitions,* which tells Spark how many for parallel collections to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of

partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to `parallelize` (like shown in the word count example).

2. words = text.flatMap(lambda line: line.split())

`flatMap` is a transformation operation. `map(`*func*`)` returns a new distributed dataset formed by passing each element of the source through a function *func*. `flatMap(`*func*`)` is similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a sequence rather than a single item).

In one sentence, this line converts the RDD containing the "lines" into an RDD containing the "words". We can see the difference between them in the next slide.

3. pairs = words.map(lambda word: (word, 1))

This is similar to what we have done using MapReduce. We transform each word into a pair.

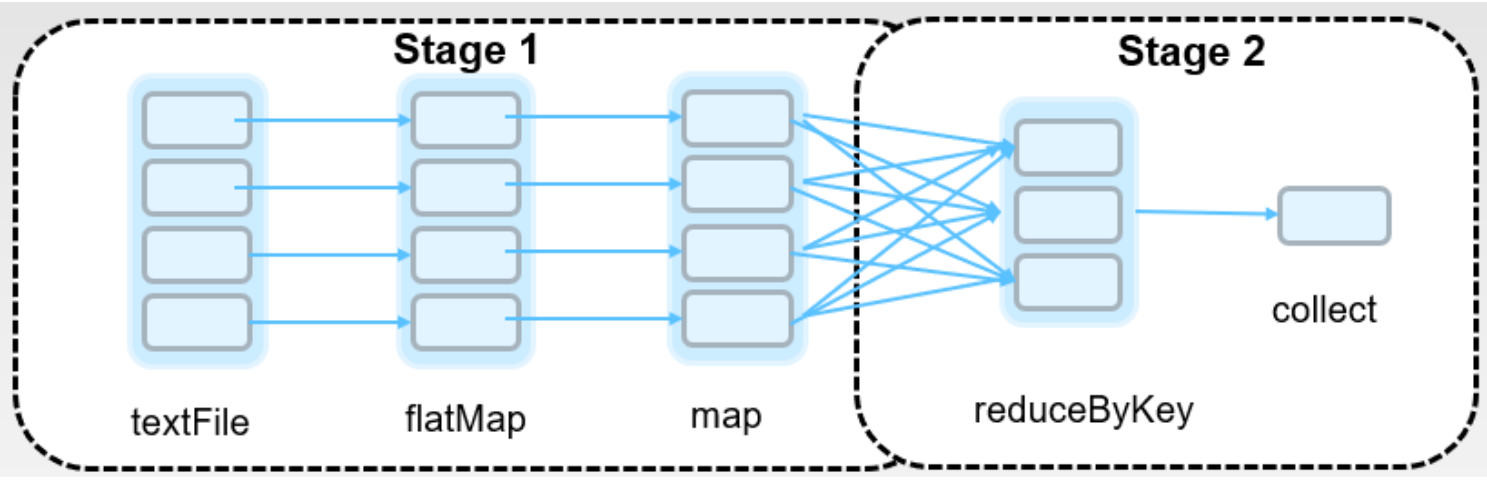
4. count = pairs.reduceByKey(lambda a, b: a+b)

`pairs` stores the key-value pairs for each word appearing in the file. We call such an RDD the "**pair RDD**", and we will practice more on pair RDDs in a later slide. `reduceByKey(`*func*`)` is a transformation operation which is similar to we have done using MapReduce. This would cause data shuffling across different partitions. Spark will group the key-value pairs according to the keys, and then apply *func* over the values to compute a result for each key.

5. count.collect()

This is an action operation, and this incurs all previous operations (such as loading the file).

Spark will create an execution plan based on your code. The scheduler examines build a DAG (directed acyclic graph) of stages. Stages are sequences of RDDs, that don't have a Shuffle in between. The boundaries are the shuffle stages. For the word count example, two stages would be created like below:



In each stage, the operations will run in parallel for different RDD partitions.

map() vs flatmap()

Assume we have a file containing two lines:

> This is the first sentence.
>
> This is the second sentence.

First, enter the pyspark shell using the command "pyspark". Next, we load the file into an RDD. This time we load the file directly from your local file system, rather than HDFS

```
>>> file = sc.textFile("file:///home/text.txt")
>>> file.collect()
```

Next, we use map and flatmap to operate the file.

```
>>> file.map(lambda x: x.split(" ")).collect()
```

You can see the result as:

[['This', 'is', 'the', 'first', 'sentence.'], ['This', 'is', 'the', 'second', 'sentence.']].

```
>>> file.flatMap(lambda x: x.split(" ")).collect()
```

You can see the result as:

['This', 'is', 'the', 'first', 'sentence.', 'This', 'is', 'the', 'second', 'sentence.'].

From this example, you could understand why we use flatMap rather than map in the word count problem.

Pair RDDs

The items in an RDD can be of any type, but one type are especially important and useful: <key, value> pairs. When the items in an RDD are <key, value> pairs then the RDD is said to be a "pair RDD". This slide covers how to work with RDDs of key/value pairs. Key/value RDDs are commonly used to perform aggregations, and often we will do some initial ETL (extract, transform, and load) to get our data into a key/value format. Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with
the same key, and grouping together two different RDDs).

Creating Pair RDDs

Normally, we have a regular RDD that we want to turn into a pair RDD. We can do this by running a map() function that returns key/value pairs. Open pyspark shell, and let's see an example as below.

```
>>> lines = sc.parallelize(["hello world", "this is a python program", "to create a pair RDD", "in s
>>> lines.collect()
```

We can see that an RDD containing four strings have been created. Next, if we want to create a pair RDD, in which the key is the first term of the string, and the value is the string itself, we can do like:

```
>>> pairRDD = lines.map(lambda x: (x.split(" ")[0], x))
```

You can try to check what are stored in the created pair RDD.

If you want to keep only the lines whose first term containing less than three characters, you can do:

```
>>> pairRDD.filter(lambda x: len(x[0])<3).collect()
```

Transformations on Pair RDDs

Pair RDDs are allowed to use all the transformations available to standard RDDs. Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.

Let's first create an pair RDD from the list [(1, 2), (3, 4), (3, 6)].

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

Next, try each operation as below to see the results and understand what the operation can do (you can use collect() to see the result of each operation, like `rdd.keys().collect()`).

- **Operation**: reduceByKey(*func*). **Purpose**: Combine values with the same key. **Example Command**: `rdd.reduceByKey(lambda x, y: x + y)`

- **Operation**: groupByKey(). **Purpose**: Group values with the same key. **Example Command**: `rdd.groupByKey()`
- **Operation**: mapValues(*func*). **Purpose**: Apply a function to each value of a pair RDD without changing the key. **Example Command**: `rdd.mapValues(lambda x: x+1)`
- **Operation**: keys(). **Purpose**: Return an RDD of just the keys. **Example Command**: `rdd.keys()`
- **Operation**: values(). **Purpose**: Return an RDD of just the values. **Example Command**: `rdd.values()`
- **Operation**: sortByKey(). **Purpose**: Return an RDD sorted by the key. **Example Command**: `rdd.sortByKey()`
- **Operation**: flatMapValues(*func*). **Purpose**: Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization. **Example Command**: `rdd.flatMapValues(lambda x: list(range (x, 6)))`

Next, let's create another pair RDD, and practice some transformation operations over two pair RDDs.

```
>>> other = sc.parallelize([(3, 9)])
```

- **Operation**: subtractByKey. **Purpose**: Remove elements with a key present in the other RDD. **Example Command**: `rdd.subtractByKey(other)`
- **Operation**: join. **Purpose**: Perform an inner join between two RDDs. **Example Command**: `rdd.join(other)`
- **Operation**: cogroup. **Purpose**: Group data from both RDDs sharing the same key. **Example Command**: `rdd.cogroup(other)`

Actions Available on Pair RDDs

As with the transformations, all of the traditional actions available on the base RDD are also available on pair RDDs. Some additional actions are available on pair RDDs to take advantage of the key/value nature of the data.

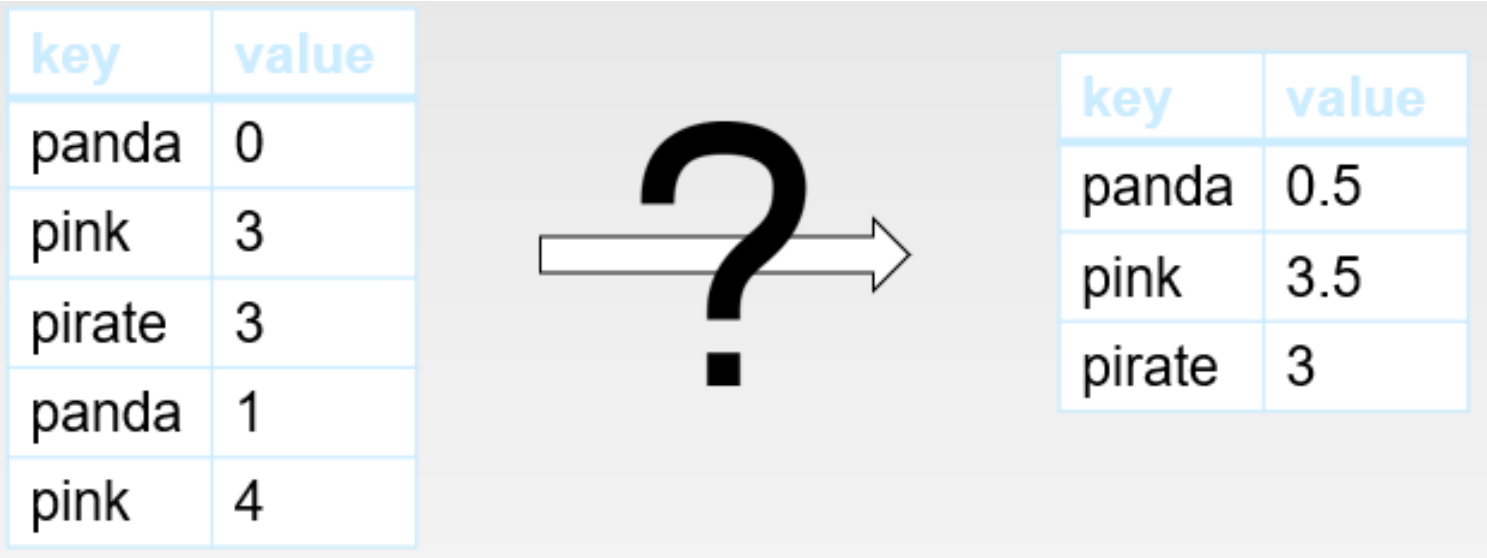
Let's still use the previous RDD [(1, 2), (3, 4), (3, 6)], and practice some action operations over this RDD.

- **Operation**: countByKey(). **Purpose**: Count the number of elements for each key. **Example Command**: `rdd.countByKey()`
- **Operation**: collectAsMap(). **Purpose**: Collect the result as a map to provide easy lookup. **Example Command**: `rdd.collectAsMap()` (**Warning**: this doesn't return a multimap. So if you have multiple values to the same key, only one value per key is preserved in the map returned.)
- **Operation**: lookup(key). **Purpose**: Return all values associated with the provided key. **Example Command**: `rdd.lookup(3)`

Try it: per-key average (pair RDD)

When datasets are described in terms of key/value pairs, it is common to want to aggregate statistics across all elements with the same key. Spark has a similar set of operations that combines values that have the same key. These operations return RDDs and thus are transformations rather than actions. Let's use an example of computing per-key average to practice pair RDD operations.

Given a dataset in which each key is associated with a value, your task is to compute the average value for each key. That is, the total value with this key divided by the occurrence of the key.



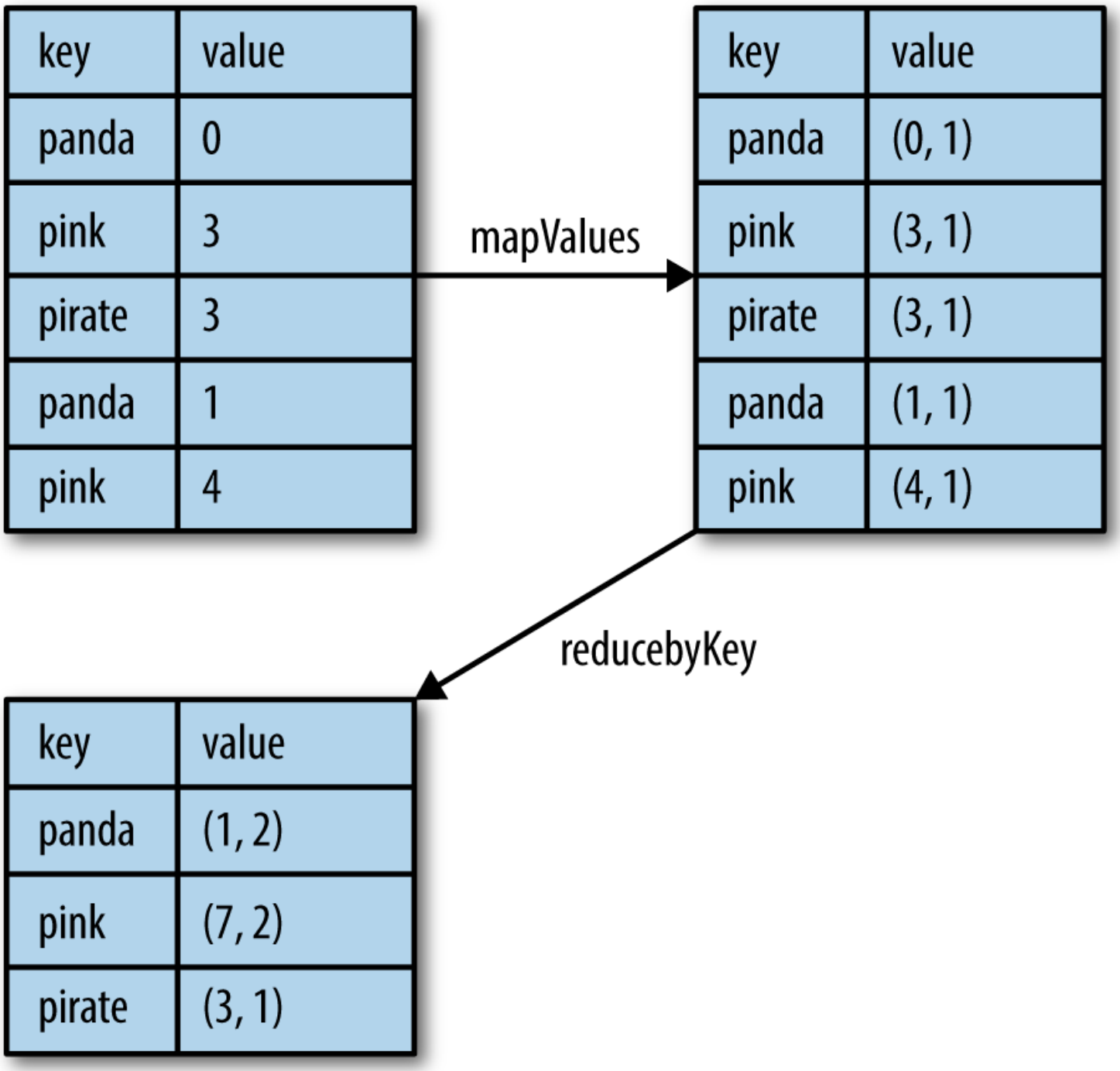
How can you use Spark to obtain the output for the given input as shown in the above figure?

First, use the following command to create a pair RDD from the file "KVP.txt":

```
>>> textfile = sc.textFile("file:///home/KVP.txt")

>>> pair = textfile.map(lambda s: (s.split()[0],  s.split()[1]))
```

Please open the pyspark shell and complete the remaining steps to compute the average value for each key. If you get stuck, please see the hints as shown in the below figure:


key
value
panda
0
pink
3
pirate
3
panda
1
pink
4
mapValues
key
value
panda
(0, 1)
pink
(3, 1)
pirate
(3, 1)
panda
(1, 1)
pink
(4, 1)
reducebyKey
key
value
panda
(1, 2)
pink
(7, 2)
pirate
(3, 1)

Spark in a Python program

Using Spark in a Python shell is a good way to learn about Spark, as a good way to analyse data interactively. In most tasks, however, we cannot simply use the shell to perform data analytics. Instead, we need to write programs in a separate Python file and then submit it to Spark to run. These are called **self-contained applications**.

Let's now learn how to do this. Let's use **the word frequency task** as an example.

When you click the panel on the right you will get a connection to a server that has Hadoop and YARN installed and running. It has a Python program called "frequencies.py", and the usual text file, "text.txt".

You first need to create your default directory in Hadoop, and copy text.txt to it:

```
$ hdfs dfs -mkdir -p /user/user
$ hdfs dfs -put text.txt
```

Now inspect the Python program.

When using Spark in a Python shell, the SparkContext object `sc` is automatically created for us to use. In a self-contained application we need to do this ourselves. That's the first few lines of the code. The rest of the code counts the word frequencies and store the result to disk.

To run this program we use the `spark-submit` command, as follows:

```
$ spark-submit frequencies.py
```

You should see Spark begin to execute and produce the output.

After Spark completes running the job, you can see the results by listing the contents of the "results" folder:

```
$ hdfs dfs -ls results
```

You should see two files: "_SUCCESS" and "part-00000". The first file shows the status of the job, the second file contains the result. You can see the results by showing the contents of the second file, using the following command:

```
$ hdfs dfs -cat results/part-00000
```

You might find it convenient to move this file to your home directory on the server's local file system. You might like to rename it in the process - let's call it "result". Here's the command:

```
$ hdfs dfs -get results/part-00000 result
```

Now you can view the contents of the file using the much simpler command:

```
$ cat result
```

Try it: the longest word

Now try it yourself - try writing a Python program using Spark to **find the length of the longest word** in a text file.

When you click the panel on the right you will get a connection to a server that has Hadoop and YARN installed and running. Our text.txt file is in your home directory, and a Python program has been created for you, called "longest.py". You just need to fill in the details of the program.

Rather than getting your program to save the result to a text file, you might find it more convenient to get it to print the result instead. To do this, don't use the "saveAsTextFile()" command - just use an ordinary Python "print()" command instead. Note that the result will be printed among the many lines of program output, and it might be hard to find!

Remember that you will need to create your default directory in Hadoop, and copy text.txt to it:

```
$ hdfs dfs -mkdir -p /user/user
$ hdfs dfs -put text.txt
```

To run your program you can use the following command:

```
$ spark-submit longest.py
```

Try it: word average length

Given a text file, compute the average length of words starting with each alphabetical letter "a" to "z". This means that for every letter, you need to compute: the **total length** of all words that start with the letter **divided by** the **total number** of words that start with the letter.

- Ignore the letter case, i.e., first convert all words to lower case (using the `lower()` function).
- Ignore terms starting with non-alphabetical characters, i.e., only consider words starting with “a” to “z”.
- You can use the space character to split the documents into words (like `line.split(" ")`).
- Sort the final result based on the alphabetical order (use the `sortBy()` operation or the `sortByKey()` operation).

For example, given "This is a test", we have one word starting with "a", one word starting with "i", and two words starting with "t". Thus, the result is like (a, 1), (i, 2), and (t, 4).

Your program could take two arguments receiving from the command line, the first one is the input text file and the second one is the output folder. For example, to run your program you can use the following command:

```
$ spark-submit word_average.py "file:///home/text.txt" "file:///home/output"
```

Next, you can enter the output directory in your home folder to check the results using the "cat" command.

The code template and the text file are provided to you. The final result should be like below (the saveAsTextFile() operation by default includes a pair of parentheses in each line, and thus you need to use Python to change the output format):

```
a, 3.0
b, 4.8
c, 6.285714285714286
d, 7.133333333333334
e, 6.5
f, 4.7368421052631575
g, 5.5
h, 4.9375
i, 2.6153846153846154
l, 5.5
m, 4.4
n, 4.411764705882353
o, 2.1818181818181817
p, 6.5
r, 7.333333333333333
s, 4.583333333333333
t, 3.558139534883721
```

```
u, 5.2
v, 9.0
w, 3.380952380952381
y, 5.0
```

Passing functions to Spark

As shown in the previous examples, you can see that Spark's API relies heavily on passing functions in the driver program to run on the cluster. There are three recommended ways to do this:

- Lambda expressions, for simple functions that can be written as an expression. (Lambdas do not support multi-statement functions or statements that do not return a value.) For example, `reduceByKey(lambda a, b: a+b)`
- Local `def`s inside the function calling into Spark, for longer code. For example, in pyspark:

```
>>> def containsError(s):

...     return "HDFS" in s

...

>>> msg = errors.filter(containsError)
```

- Top-level functions in a module. For example, as you can see in the file "passfunc.py", we define a function myFunc(s) which takes a string as input, and it splits the string using the space character, and returns the number of words obtained from this string. This function can be passed to the `map()` operation. Run the code by running `spark-submit passfunc.py`. Then, check the results by `cat output/part-00000`.

Try it: top-k most frequent pairs

In this challenge, you will practice how to pass a long function to Spark.

The task is that, given a text file, find out the top-k most frequent co-occurring term pairs. The co-occurrence of (w, u) is defined as: u and w appear in the same line (this means that (w, u) and (u, w) are treated as the same pair, and the first term has a lower alphabetical order). Your Spark program should generate a list of k key-value pairs ranked in descending order according to their frequencies, where the keys are the pair of terms (in format of (w, u)) and the values are the co-occurring frequencies. If two pairs have the same frequency, sort them by the first term and then by the second term alphabetically.

Your program should read "text.txt" as the input, and store the result in the directory "output" in your home folder. The parameter k is received from the command line. The code template has been provided, and please fill your code to complete this task.

Assume that k=5, you can run your program by `spark-submit topk_pair.py 5`, and the result is like:

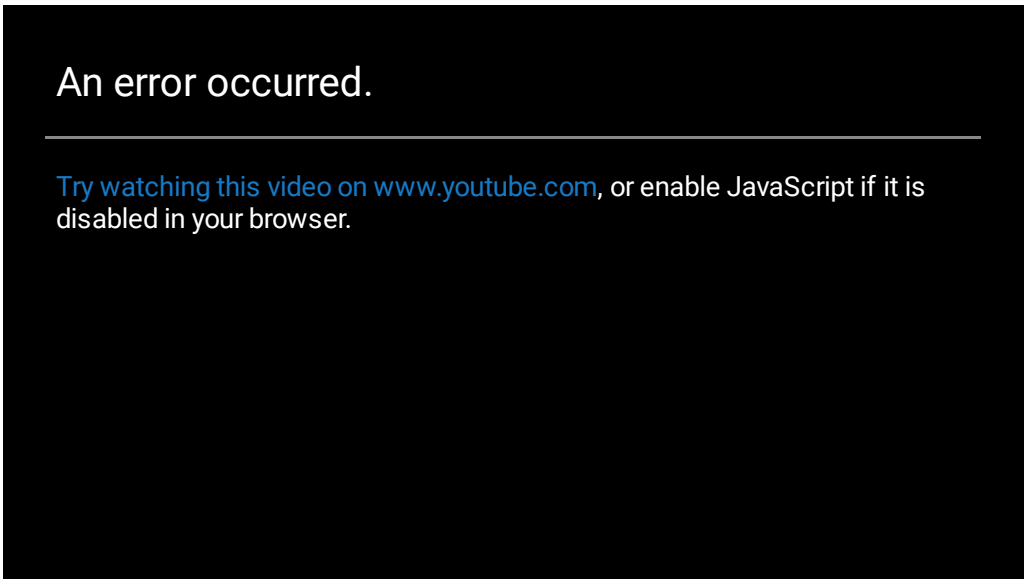
```
('the,to', 61)
('not,the', 50)
('the,the', 45)
('can,the', 40)
('for,the', 40)
```

Further resources

1. The official Spark website is a good source of further information about Spark:

Official Spark website

2. Hadoop vs Spark



3. More examples of using Spark RDDs