MapReduce

MapReduce

In the previous set of slides you learned about using Hadoop to store large volumes of data across a network of computers (also called a cluster of nodes).

Storing the data is one thing; being able to analyse it is another.

A problem, and a solution

Suppose you want to know the frequency of words in a certain file. If the file were small enough to fit on a single computer then you could write and run a program on that computer, to count the frequency of words in the file. In essence, we would be applying a function to the file and getting a result (a list of words and their frequencies in the file).

But what if the file is too big for a single computer, and is distributed across a Hadoop cluster?

If you could gather the parts of the file onto a single computer then there is no problem - you could do so and then just run the same program. But you can't, because the file is too big.

You could bring the parts to the computer one at a time, in bits small enough to fit on the computer. But that will be very slow because it requires moving a lot of data across the cluster.

It would be faster if you could leave the data on its node and send the program to it.

Basically, that's what we do.

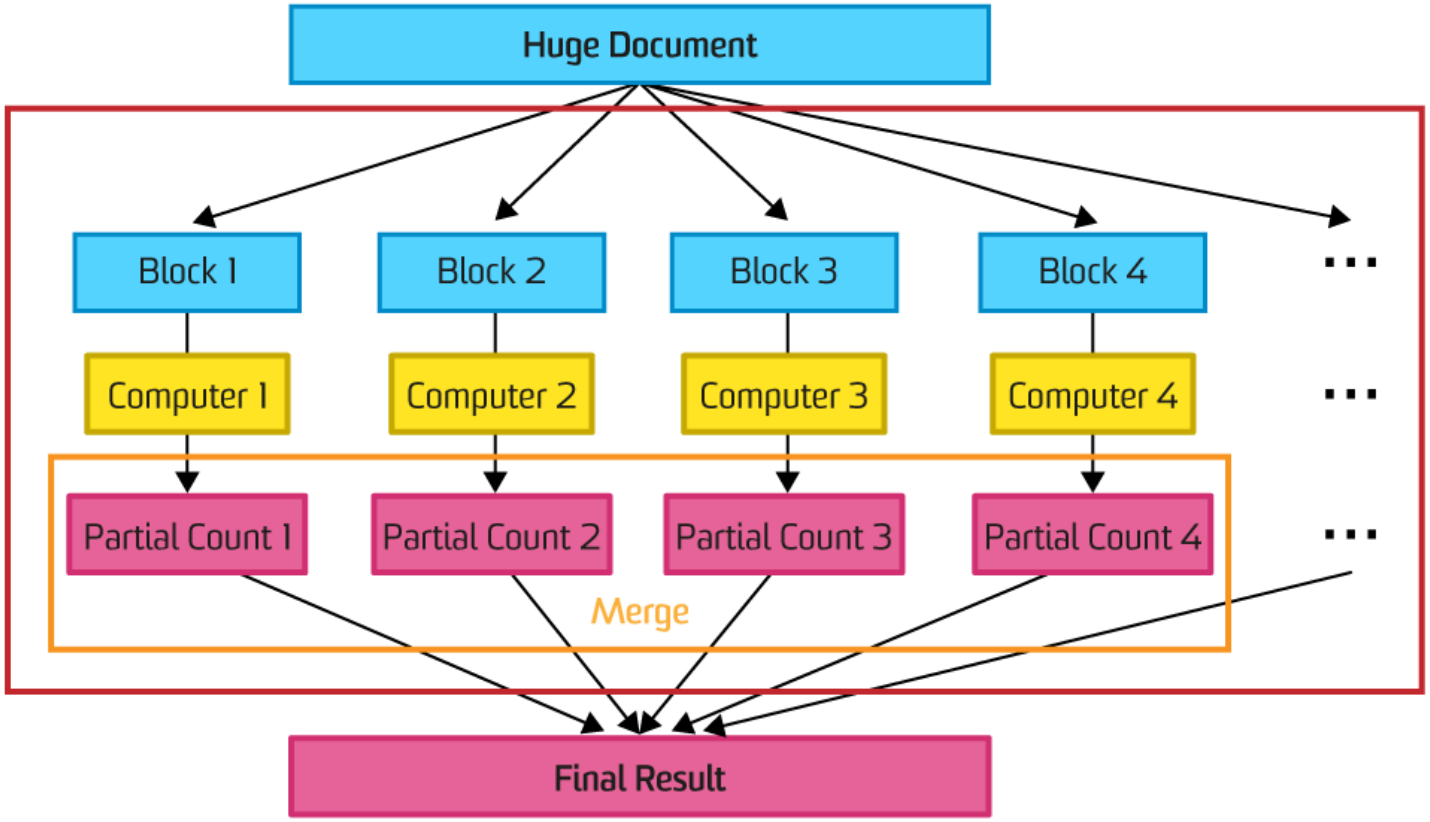
But it's not quite that simple. This gives us multiple results - one result for each node that has run the program. We also need to combine these results to get the final answer. In essence, we need to apply a function to the individual results to get the final result.

So, there are two things we need to do:

First, we need to send a function to each node which has part of the file, and apply the function to that piece of the file. This is called the **mapping** phase. Second, we need to gather the results and combine them into a final answer. This is called the **reducing** phase.

This technique is called **MapReduce**. It is a fundamentally important technique for working with large volumes of data that is stored across a network of computers.

Here is a simple diagram of the technique:


Huge Document
Block 1
Block 2
Block 3
Block 4
...
Computer 1
Computer 2
Computer 3
Computer 4
...
Partial Count 1
Partial Count 2
Partial Count 3
Partial Count 4
...
Merge
Final Result

Why the name?

The name "MapReduce" is inspired by two techniques from functional programming: mapping a list, and reducing a list.

Suppose we want to write a Python program that takes a list of words and calculates their total length. There are various ways to do this, but one way is to use Python's `map()` and `reduce()` functions.

Here is a Python program that does so:

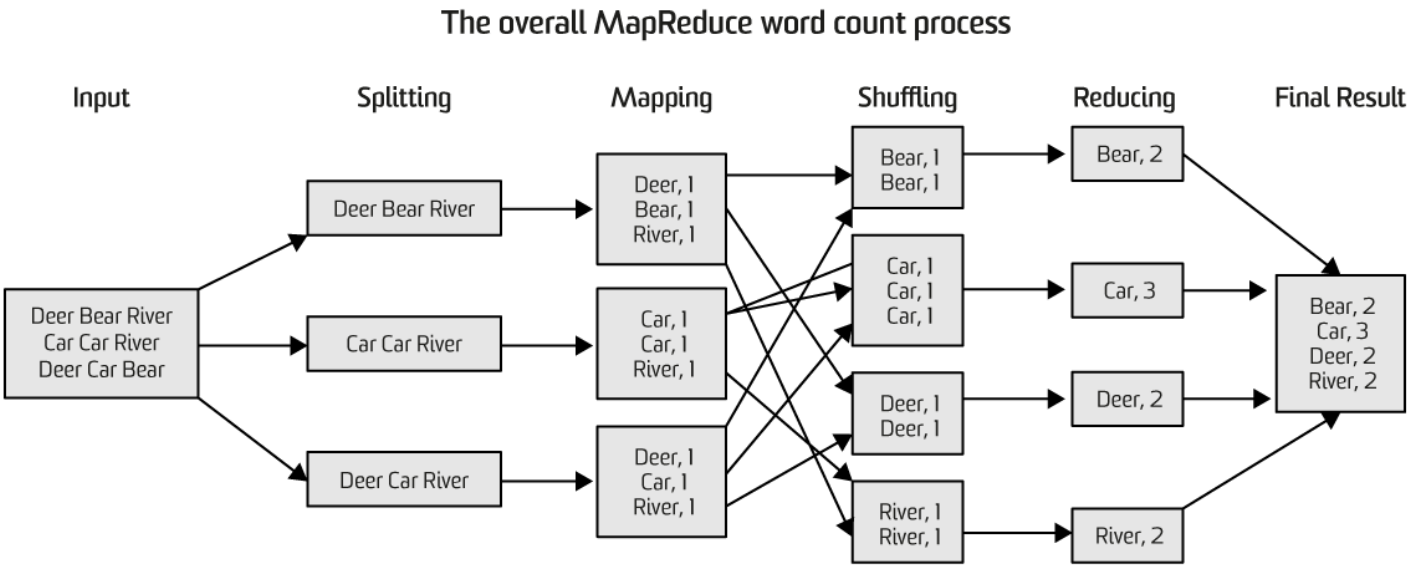
```
# map() is part of core Python
# but reduce() needs to be imported from the functools library:
from functools import reduce

# Here is a list of words:
words = ['The','quick','brown','fox','jumped','over','the','lazy','dog']

# First, we apply the map function to the list
# The first argument is a function
# map applies the function to each item in the list
# and returns the results as a list
word_lengths = map(lambda item: len(item), words)

# Next, we apply the reduce function to the new list
# The first argument is a function that builds a result from the list
# The third argument is the initial value of the result
total_length = reduce(lambda result, item: result + item, word_lengths, 0)

# Finally, we print the result:
print("The words have a total length of", total_length, "characters.")
```

What we've done in this program is **mapped** a list of words to a list of lengths, and then **reduced** this list of lengths to a number - the total length of the words.

In MapReduce we are working with a file in HDFS - it is distributed across many nodes. In effect, we are working with a list of file parts. We first **map** this list of file parts to a list of intermediate results, by applying our mapper function, then we **reduce** this list of intermediate results to a final result, by applying our reducer function. Thus the name of the technique, "MapReduce".

The process

Let's consider how we use MapReduce to count the frequency of words in a file in HDFS. Here is a diagram of the process - we explain each of the steps below:



Splitting

The input to a MapReduce job is a file that is split into parts. This splitting is done when the file is saved to HDFS.

Mapping

Each node in HDFS which has part of the file applies the mapper function to that part of the file. The mapper function returns a list of <key, value> pairs as output. These pairs can be anything we like - whatever it takes to get the final result we want. The programmer needs to specify this mapper function.

In our example, the mapper function breaks its part of the file into words, and for each word outputs a <key, value> pair of the form <word, 1>.

The mapper nodes work in parallel to produce their outputs.

Shuffling

MapReduce gathers the outputs of the mapping nodes and sends them to the reducing nodes. All <key, value> pairs with the same key are sent to the same reducer. There could be just one reducer, or many reducers. When using many reducers, all <key, value> pairs with different keys might be

sent to different reducers.

In our example, all pairs with the same word as key are grouped together and sent to the same reducer node.

It should be clear from the diagram above why this phase is called “the shuffle”.

Reducing

Each reducer gets a list of <key, value> pairs as input and produces a list of <key, value> pairs as output. The results are written to HDFS. The programmer needs to specify the reducer function.

In our example, each reducer calculates the total number of occurrences of each word that it receives, by adding together all of the 1s.

What the programmer does

The programmer of a MapReduce job Just provides the mapper and reducer functions, in her language of choice (we'll be using Python).

What MapReduce does

The MapReduce software handles everything else, including:

- Deciding which nodes should be mappers and which should be reducers
- Sending the mapping and reducing functions to the relevant nodes
- Gathering and shuffling the results from the mappers and sending them to the appropriate reducers
- Detecting node failures, and dealing with them accordingly

Versatility

The idea of using <key, value> pairs is very versatile - with a bit of ingenuity, keys can be combined in a wide variety of ways to answer questions about one's data (we will see various examples of how this is done). You might enjoy the challenge of working out how to answer questions about some data by using a mapper and reducer and <key, value> pairs.

More about MapReduce

In the series of slides that follow are some videos that explain the MapReduce technique in various ways - you might find them helpful.

Explain MapReduce by the WordCount Example

Explain MapReduce by the WordCount Example

ZZEN9313_MapReduce_Transcript.pdf

Hadoop Streaming

The native programming language for Hadoop MapReduce is Java. However, we can also write MapReduce programs using other programming languages such as Python based on Hadoop streaming.

Hadoop streaming is a utility that comes with the Hadoop distribution. The utility allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. For example:

```
mapred streaming \
  -input myInputDirs \
  -output myOutputDir \
  -mapper /bin/cat \
  -reducer /usr/bin/wc
```

In the above example, both the mapper and the reducer are executables that read the input from stdin (line by line) and emit the output to stdout. The utility will create a Map/Reduce job, submit the job to an appropriate cluster, and monitor the progress of the job until it completes.

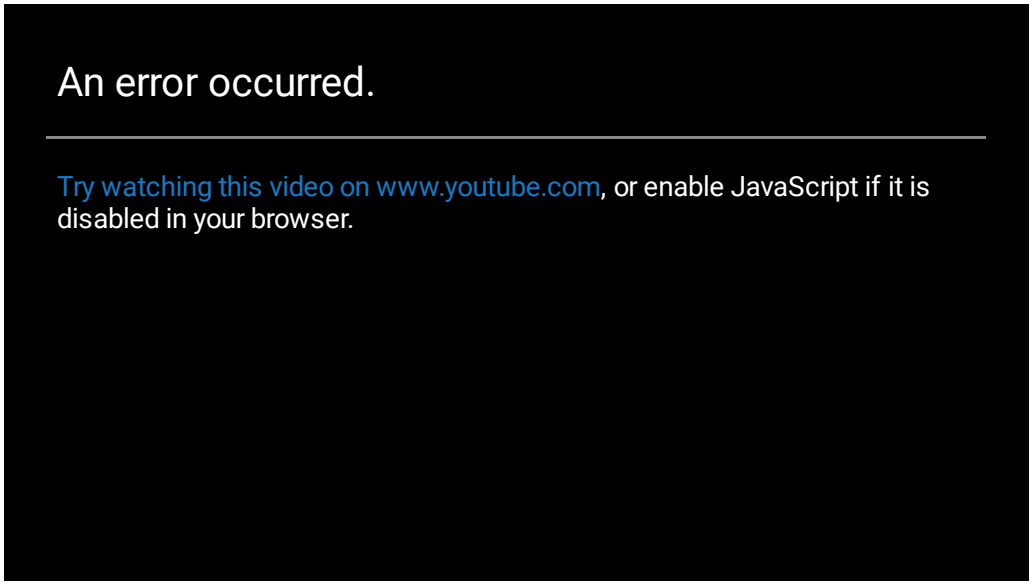
When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. As the mapper task runs, it converts its inputs into lines and feed the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper. By default, the *prefix of a line up to the first tab character* is the key and the rest of the line (excluding the tab character) will be the value. If there is no tab character in the line, then entire line is considered as key and the value is null.

When an executable is specified for reducers, each reducer task will launch the executable as a separate process then the reducer is initialized. As the reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the stdin of the process. In the meantime, the reducer collects the line oriented outputs from the stdout of the process, converts each line into a key/value pair, which is collected as the output of the reducer. By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) is the value.

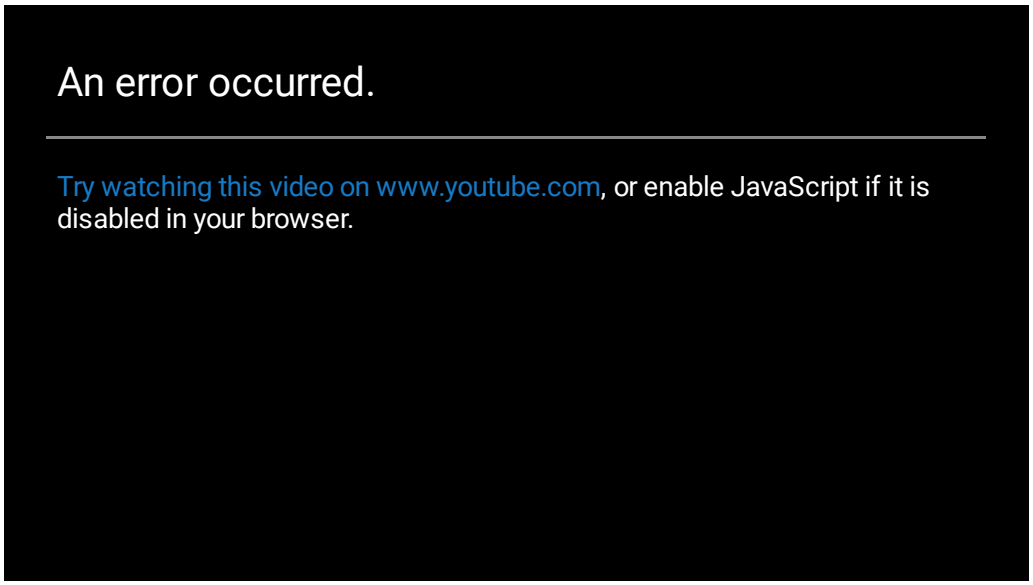
This is the basis for the communication protocol between the Map/Reduce framework and the streaming mapper/reducer. We will learn how to use Python to write MapReduce programs based on Hadoop streaming later.

More videos that may help you understand MapReduce

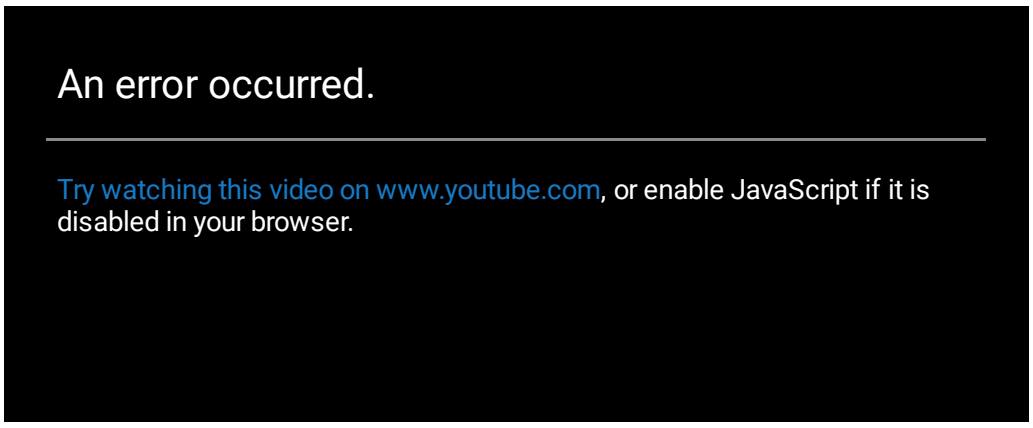
1. What is MapReduce?



2. Learn MapReduce with Playing Cards



3. Hadoop Streaming in Python, hadoop streaming tutorial



Word frequency

Let's now get some hands-on experience of MapReduce using Hadoop streaming.

Suppose that you want to know **the words in a text file and how frequently each word occurs**.

You could write a Python program to calculate the frequencies for you.

When you click the panel on the right you will get a terminal connection to a server. There is Python program called "frequency.py" on this server - it reads a file from standard input and prints the frequency of each word in the file to standard output. Read through the program to make sure you understand it.

There is also a sample text file called "text.txt". You can open the file to inspect its contents.

To run frequency.py with text.txt as input, enter the following into the terminal:

```
$ python frequency.py < text.txt
```

This command tells Python to run the program "frequency.py", and use "text.txt" as the input to the program (that's what the `<` is doing).

You should see the words in the file, and the frequency of each word.

So far this just an ordinary Python program working with an ordinary file in the server's local file system. But what if the file is stored in a Hadoop cluster? Then you'll need to use a different technique. In the next two slides we'll see how to do it using MapReduce.

Word frequency, split

Now let's turn this program into one that can run on a file stored in Hadoop. We need to break it into a **mapper** program and a **reducer** program.

When you click the panel on the right you'll get a terminal connection to a server. On the server (in the local file system), are two Python programs, called "mapper.py" and "reducer.py". You can open and view them.

mapper.py takes a file from standard input and prints each word with frequency 1 to standard output. Why frequency 1? We'll see why shortly.

reducer.py takes a file of word-frequency pairs from standard input and prints the total frequency of each word to standard output.

We can simulate how Hadoop will handle this. The file "text.txt" from the previous slide has been split into two halves, and saved as the files "textA.txt" and "textB.txt". This is to simulate the distribution of the file across a Hadoop cluster. (You can open and view them.)

First, we apply mapper.py to textA.txt, and send the results to a file called results.txt (this file will be automatically created for you, by the local file system):

```
$ python mapper.py < textA.txt > results.txt
```

This command tells Python to run the program mapper.py, using the file textA.txt as the input, and then send the results to the file results.txt (that's what `>` is doing).

Next, we do the same for textB.txt, except we *append* the results, by using `>>` instead of `>` (we want to keep the results that are already there):

```
$ python mapper.py < textB.txt >> results.txt
```

The file results.txt now contains the output of the mapper applied to textA.txt, and the output of the mapper applied to textB.txt. Finally, we apply reducer.py to results.txt to get the total frequency of each word across the two files:

```
$ python reducer.py < results.txt
```

This command tells Python to run the program reducer.py using the file results.txt as input.

You should see the same results as we got in the previous slide, when we applied a single Python program to the single text file.

What we've done here is simulated, on the local file system on the server, what MapReduce does when working with a distributed file in HDFS. In the next slide we'll use MapReduce.

Word frequency, on Hadoop

Now let's get this to run as a MapReduce job on Hadoop.

When you click the panel on the right you will get a terminal connection to a server that has Hadoop installed and running, and also YARN (which is a part of Hadoop that is needed to run MapReduce jobs).

Preparation

We need to copy the file text.txt into HDFS.

First, create your default working directory in HDFS:

```
$ hdfs dfs -mkdir -p /user/user
```

Next, let's create a directory in which to keep the input files of our MapReduce job. Call it "input":

```
$ hdfs dfs -mkdir /user/user/input
```

Upload the file "text.txt" into HDFS /user/user/input:

```
$ hdfs dfs -put text.txt /user/user/input
```

Running the job

Now let's run the MapReduce job.

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar -file ~/mapper.py -mapper ~
```

The Hadoop streaming utility will create a MapReduce job, submit the job to an appropriate cluster and monitor the progress of the job until it completes.

The results of the job will be put in the HDFS folder /output. You can view the results there. You will see a file with the name "part-00000", which contains the result.

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

Alternatively, you could move the results folder to your local home directory and view them from there:

```
$ hdfs dfs -get output output
$ cat output/part-00000
```

Working with CSV

In the word frequency example, we worked with a text file. Now let's try working with a CSV file.

When you click the panel on the right you will get a terminal connection to a server that has, in your home directory on this server, a CSV file called "employees.csv". You can open the file to inspect its contents. The fields in the file are as follows:

```
employee_id (integer)
first_name (string)
last_name (string)
email (string)
phone_number (string)
hire_date (date)
salary (integer)
```

Maximum salary

Suppose we want to know **the maximum salary of employees**. How might we do it using a mapper and reducer? The mapper is applied to only part of the file, so it cannot calculate the overall maximum salary. But it can return each of the salaries, and then we can get the reducer find the maximum of those salaries.

On the server you will see files called "mapper.py" and "reducer.py" - these are mappers and reducers written in Python which will do what we want.

You can see the results by running the following command:

```
$ python mapper.py < employees.csv | python reducer.py
```

This command tells Python to execute the program mapper.py, using the file employees.csv as input, and then **pipe** the results as input to reducer.py (we use `|` to indicate that we want the results piped to the next process).

Once we're happy that the mapper and reducer are working correctly we could run them as a MapReduce job on Hadoop, as per the previous slide. But we won't do that - for the purpose of learning the concept of mapping reducing the above should suffice.

An alternative method

You may have noticed that mapper.py is a bit lazy - it doesn't calculate the maximum salary among the salaries that it has, it just returns each salary paired with 1, and lets reducer.py do the rest.

We could write an alternative mapper which returns just a single <key, value> pair, containing the

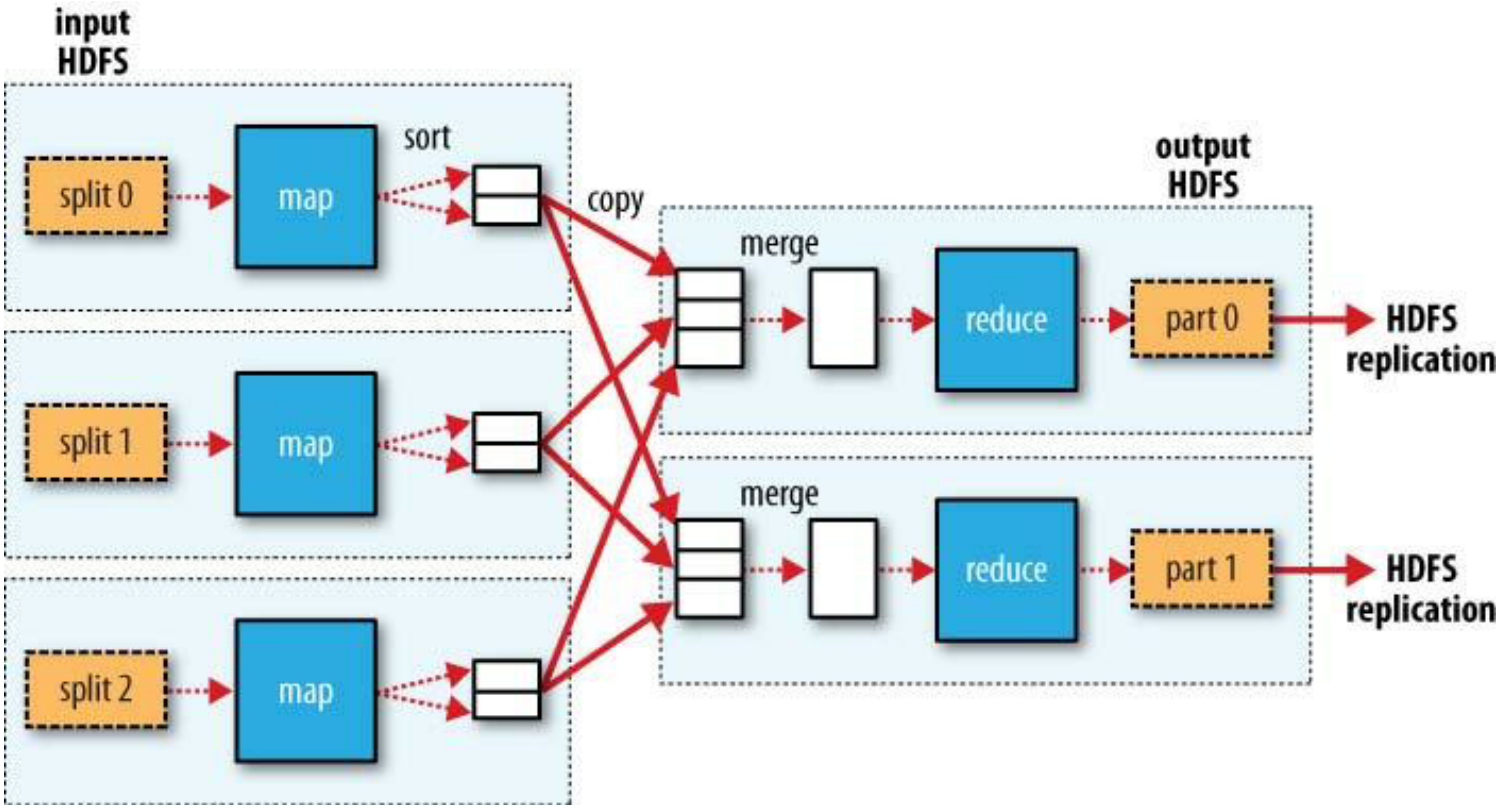
maximum salary among the salaries that it has. Then the reducer just needs to find the maximum values of these local maxima. This means that less data needs to be sent across the Hadoop cluster for the reducer to reduce.

You will also see, on the server, a Python program called mapper2.py, which does this. Try it out - you should get the same result:

```
$ python mapper2.py < employees.csv | python reducer.py
```

More Detailed MapReduce Dataflow

The figure below shows an example of the MapReduce data flow with 3 mappers and 2 reducers.



This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as “the shuffle,” as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time.

In the shuffling phase (introduced here):

- **All key/value pairs are sorted before being presented to the reducer function**
- **All key/value pairs sharing the same key are sent to the same reducer**

These can be done in the background as a part of the MapReduce mechanism when using Hadoop Streaming, and we don't need to implement any method to fulfill these.

Test with multiple reducers

Here, we use the same problem as the previous slide for an example.

It worth noting that:

- To run the python codes with Hadoop Streaming, you need to add `#!/usr/bin/python` in the top line of your `*.py` files. This tells Hadoop where to find the python interpreter.
- We changed both `mapper.py` and `reducer.py` , the previous codes are in the comments.

- In this test the mapper outputs <salary, null> as the <key,value> pair.
    - When using Hadoop Streaming, by default, the *prefix of a line up to the first **tab character*** (\t) is the key and the rest of the line (excluding the tab character) will be the value. If there is no tab character in the line, then **the entire line** is considered as **key** and the **value** is **null**. More details can be found here.

Now, let's start the test!

Preparation

We need to copy the file empolyees.csv into HDFS.

```
$ hdfs dfs -mkdir -p /user/user
```

```
$ hdfs dfs -mkdir /user/user/input
```

```
$ hdfs dfs -put employees.csv /user/user/input
```

Running the job with a single Reducer

Run the MapReduce job:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar -file ~/mapper.py -mapper ~
```

List the content of the output folder:

```
$ hdfs dfs -ls output
```

Then, you will find that the reducer outputs the result in a file named 'part-00000'.

Show the content of the file 'part-00000':

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

You will get the correct answer: The maximum salary is $ 24000

Running the job with 3 Reducers

We should first remove the output folder before doing the next test.

```
$ hdfs dfs -rm -r output
```

To run the job with three reducers, we should add the setting `-numReduceTasks 3` to the command, and we have:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar -file ~/mapper.py -mapper ~
```

This time, you will find that there are three output files in the output folder (part-00000, part-00001 and part-00002), one for each reducer.

```
$ hdfs dfs -ls output
```

```
[user@sahara ~]$ hdfs dfs -ls output
Found 4 items
-rw-r--r--   1 user supergroup          0 2020-07-30 01:53 output/_SUCCESS
-rw-r--r--   1 user supergroup         31 2020-07-30 01:53 output/part-00000
-rw-r--r--   1 user supergroup         31 2020-07-30 01:53 output/part-00001
-rw-r--r--   1 user supergroup         31 2020-07-30 01:53 output/part-00002
```

To show the contents of these three files together:

```
$ hdfs dfs -cat output/part-*
```

You will find that there are three lines, and only one has the correct answer. However, we just want to get one line output as the final result (Imagine that we have 1000 reducers, it is not a good idea to find the maximum value within 1000 possible output files). Therefore, this job fails in this case.

Next, you can modify the current codes back to the previous slide, test it with 3 reducers and compare the results. What can you find? :)

```
[user@sahara ~]$ hdfs dfs -ls output
Found 4 items
-rw-r--r--   1 user supergroup          0 2020-08-01 08:45 output/_SUCCESS
-rw-r--r--   1 user supergroup          0 2020-08-01 08:45 output/part-00000
-rw-r--r--   1 user supergroup          0 2020-08-01 08:45 output/part-00001
-rw-r--r--   1 user supergroup         31 2020-08-01 08:45 output/part-00002
```

Why does 'running the job with a single Reducer' work?

Although the mapper output many <key, value> pairs with different keys, all the <key, value> pairs are allocated to the same reducer (because there is only one reducer). Therefore, the reducer can get **the maximum salary of employees**.

In a distributed computing scenario, it is common to have multiple reducers. Therefore, we have to consider this case when designing our mapper and reducer programs.

Try it: salary holders

Now try an example yourself, using the same CSV file as the previous slide.

Suppose you want **a list of distinct salaries and, for each of those salaries, the last names of employees on that salary**. Try writing a mapper and reducer that will do that for you.

When you click the panel on the right you'll get a terminal connection to a server that has the file "employees.csv" in your home directory in the local file system. Here are the fields in the file again:

```
employee_id (integer)
first_name (string)
last_name (string)
email (string)
phone_number (string)
hire_date (date)
salary (integer)
```

Files called mapper.py and reducer.py have been started for you. Your task is to finish writing those files.

You can test your mapper and reducer by running the following command:

```
$ python mapper.py < employees.csv | python reducer.py
```