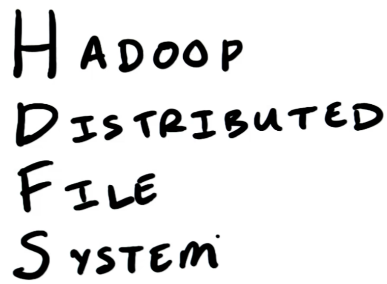
Lesson 2 Notes

Introduction

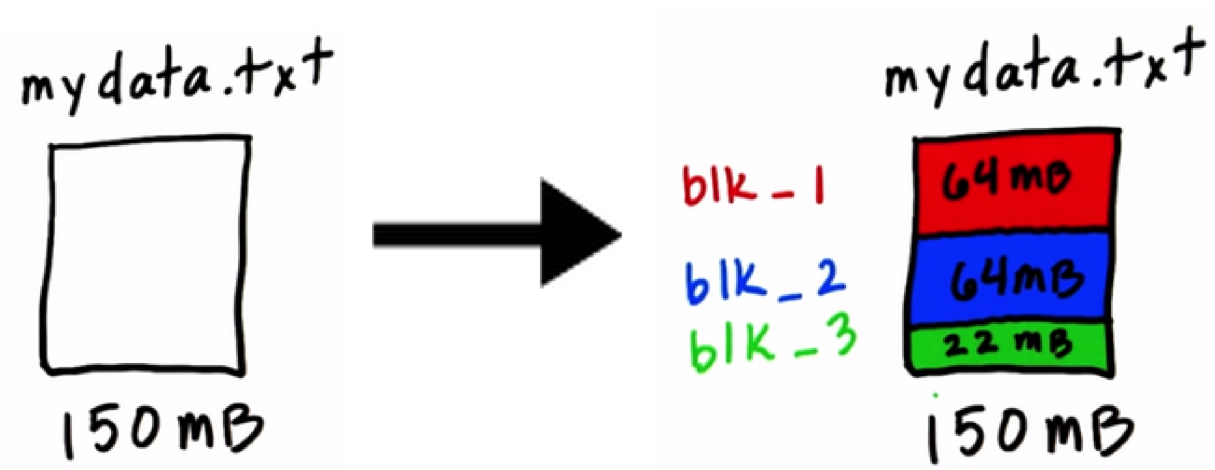
In this lesson we'll take a deeper look at the two key parts of Hadoop - how it stores the data, and how it processes it. Let's start by seeing how data is stored.

HDFS



Files are stored in something called the Hadoop Distributed File System, which everyone just refers to as HDFS. As a developer, this looks very much like a regular filesystem -- the kind you're used to working with on a standard machine. But it's helpful to understand what's going on behind the scenes, so that's what we're going to talk about here.

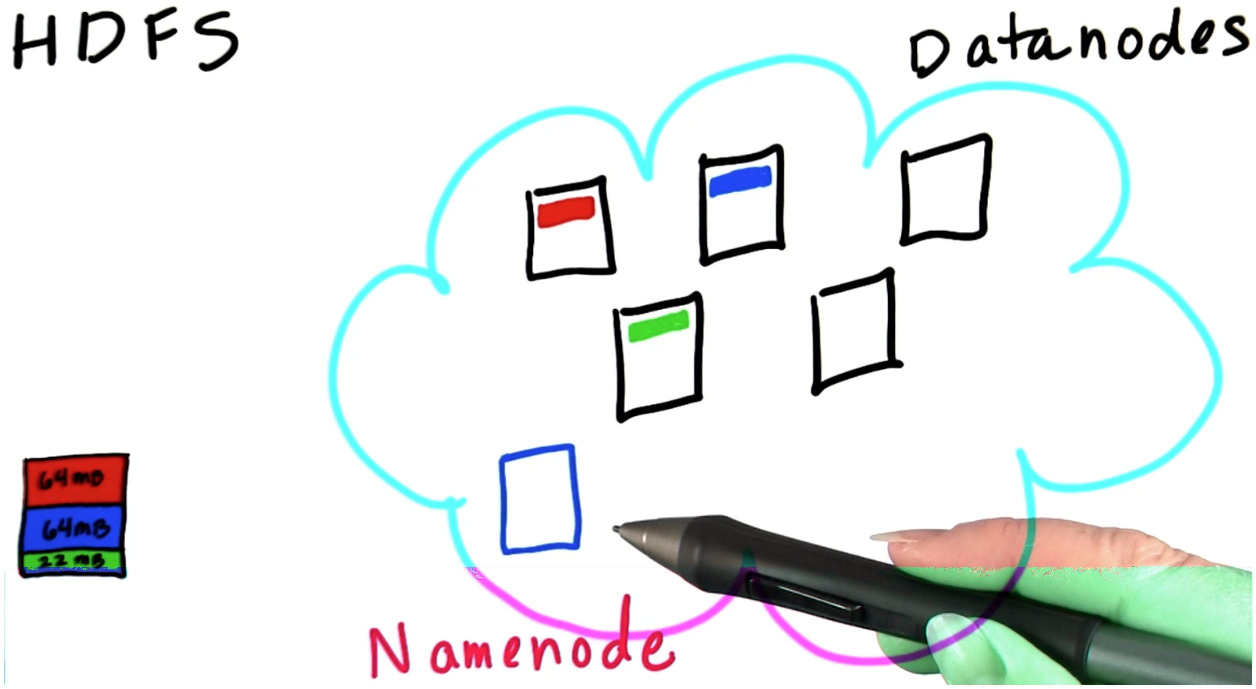
When a file is loaded into HDFS, it's split into chunks, which we call 'blocks'. Each block is pretty big; the default is 64 megabytes. So, imagine we're going to store a file called 'mydata.txt', which is 150 megabytes in size. As it's uploaded to the cluster, it's split into 64 megabyte blocks, and each block will be stored on one node in the cluster. Each block is given a unique name by the system: it's actually just 'blk', then an underscore, then a large number. In this case, the file will be split into three blocks: the first will be 64 megabytes, the second will be 64 megabytes, and the third will be the remaining 22 megabytes.



There's a daemon, or piece of software, running on each of these cluster nodes called the

DataNode, which takes care of storing the blocks.

Now, clearly we need to know which blocks make up the file. That's handled by a separate machine, running a daemon called the NameNode. The information stored on the NameNode is known as the 'metadata'.



QUIZ - Are there problems?

That's fine as far as it goes, but there are some problems with this. Take a look at the diagram, and see if you can spot where we might run into trouble.

[ ] Network failure between nodes
[ ] Disk failure on DataNode
[ ] Not all DataNodes are used
[ ] Block sizes are different
[ ] Disk failure on NameNode

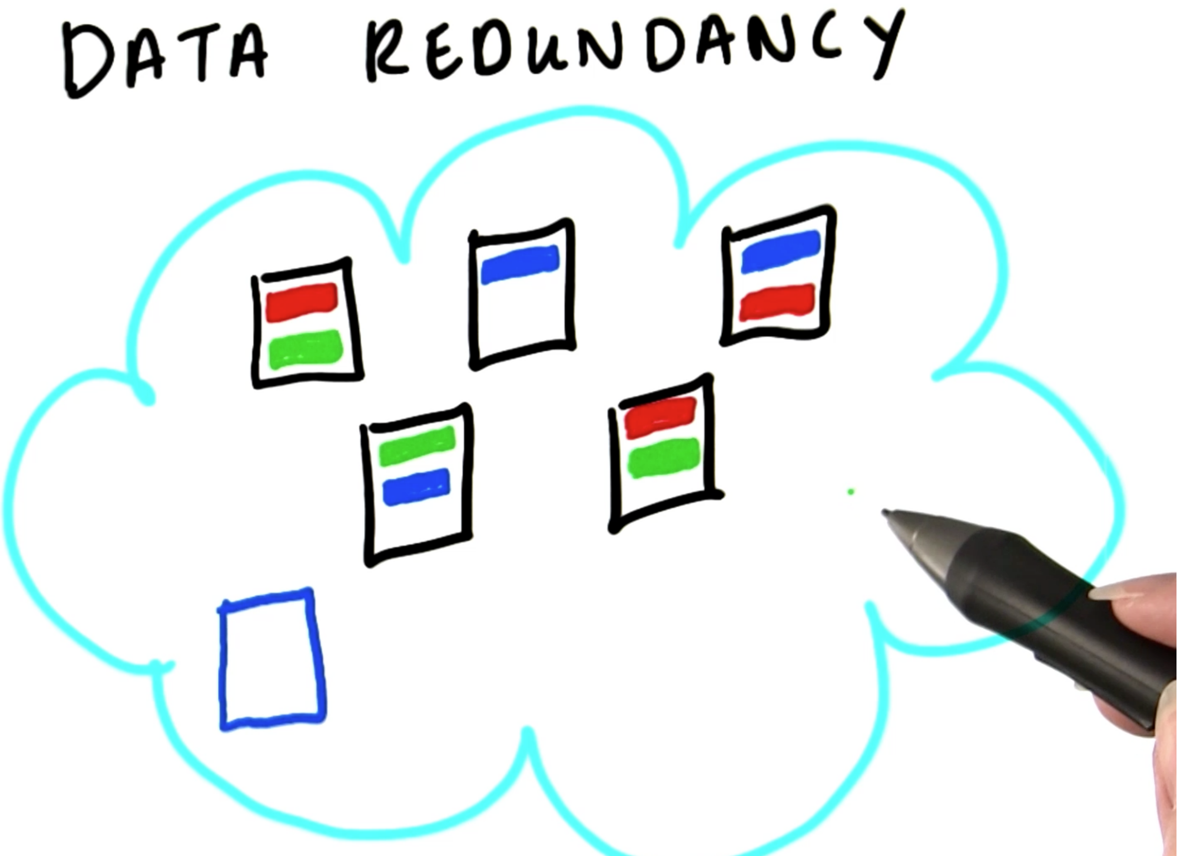
Answer:

'Some nodes are not used' is not a problem, since they can be used for different files, neither is different block size. Network and disk failures are certainly a problem, let's look into this in more detail

Data Redundancy

The problem with things right now is that if one of our nodes fails, we're left with missing data for the file. If this node goes away, for example, we've got a 64 megabyte hole in the middle of mydata.txt. And, of course, similar problems with any other files which have blocks stored on that node.

To solve this problem, Hadoop replicates each block three times as it's stored in HDFS. So blk_1 doesn't just live here, it's also stored perhaps here and here. blk_2 isn't just here, but also maybe here and here. And similarly for blk_3. Hadoop just picks three random nodes, and puts one copy of the block on each of the three. Well, actually, it's not a totally random choice, but it's close enough for us right now.



Now, if a single node fails, it's not a problem because we have two other copies of the block on other nodes. And the NameNode is smart enough that if it sees that any of the blocks are under-replicated, it will arrange to have those blocks re-replicated on the cluster so we're back to having three copies of them again.

QUIZ - Any problems now?

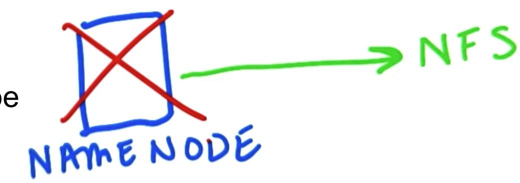
OK, so, we've taken care of what happens if one of our DataNodes fails. But there's another obvious single point of failure here. What happens if the NameNode has a hardware problem?

data on HDFS may be inaccessible [ ]
data on HDFS may be lost forever [ ]
there is no problem [ ]

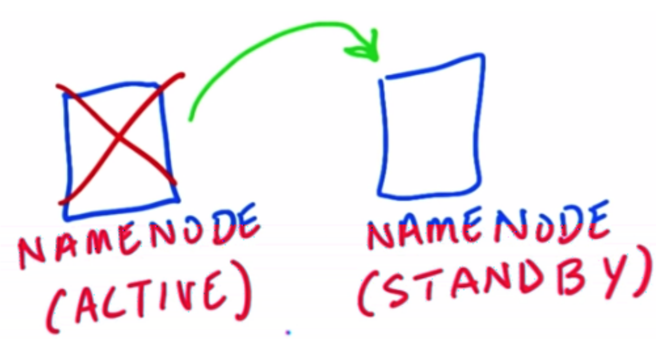
Answer:
If there is a network failure, the data will not be accessible temporarily. If the disk of the single NameNode would fail, data on HDFS would be lost permanently

NameNode High Availability

For a long time, the NameNode was a single point of failure in Hadoop. If it died , the entire cluster was inaccessible. And if the metadata on the NameNode was lost, the entire cluster's data was lost. Sure, you've still got all the blocks on the DataNodes, but you've no way of knowing which block belongs to which file without the metadata. So to avoid that problem, people would configure the NameNode to store the metadata not just on its own hard drive but also somewhere else on the network using NFS, which is a method of mounting a remote disk on the NameNode. That way, even if the NameNode burst into flames, there would be a copy of the metadata somewhere else on the network.



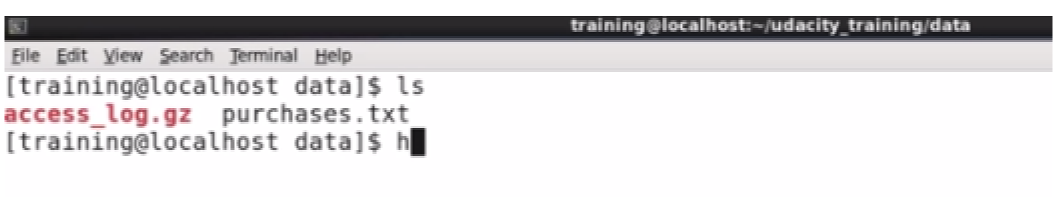
Active and Standby NameNode



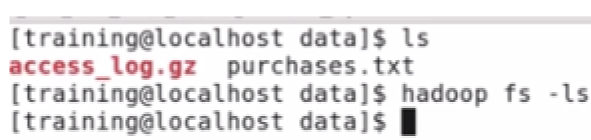
These days, the problem of the NameNode being a single point of failure has been solved. Most production Hadoop clusters now have two NameNodes: one Active, and one Standby. The Active NameNode works as before, but the Standby can be configured to take over automatically if the Active one fails. That way, the cluster will keep running if any of the nodes -- even one of the NameNodes -- fails.

Ian's now going to show you a demonstration of how to use HDFS.

DEMO of HDFS



So, here I have a directory on my local machine, which contains a couple of files, and I want to put one of them into hdfs. All of the commands which interact with the Hadoop file system start with Hadoop FS. So first of all, let's see what we



have in hdfs to start with. I do that by saying hadoop fs minus ls. That gives me a listing of what's in my home directory on the Hadoop cluster. Because I'm logged in to the local machine as a user called training, my home directory in hdfs is /user/training. And as you can see, there's nothing there. So now, let's upload our purchases.txt file. We do that with hadoop fs minus put purchases.txt. Hadoop fs minus put takes a local file and places it into hdfs.

```
[training@localhost data]$ hadoop fs -put purchases.txt
```

Since I'm not specifying a destination filename, it'll be uploaded with the same filename. So, it takes a few seconds to upload. And now I can do another hadoop fs minus ls, and we can see that that file is now in hdfs.

```
[training@localhost data]$ hadoop fs -ls
Found 1 items
-rw-r--r--   1 training supergroup  211312924 2013-09-12 21:16 purchases.txt
```

I can take a look at the last few lines of the file by saying, hadoop fs minus tail, and then the filename, and that just displays the last few lines on the screen for me.

```
[training@localhost data]$ hadoop fs -tail purchases.txt
31      17:59   Norfolk Toys    164.34  MasterCard
2012-12-31      17:59   Chula Vista     Music   380.67  Visa
2012-12-31      17:59   Hialeah Toys    115.21  MasterCard
2012-12-31      17:59   Indianapolis    Men's Clothing  158.28  MasterCard
2012-12-31      17:59   Norfolk Garden  414.09  MasterCard
2012-12-31      17:59   Baltimore       DVDs    467.3   Visa
2012-12-31      17:59   Santa Ana       Video Games     144.73  Visa
2012-12-31      17:59   Gilbert Consumer Electronics    354.66  Discover
2012-12-31      17:59   Memphis Sporting Goods  124.79  Amex
2012-12-31      17:59   Chicago Men's Clothing  386.54  MasterCard
2012-12-31      17:59   Birmingham      CDs     118.04  Cash
2012-12-31      17:59   Las Vegas       Health and Beauty       420.46  Amex
2012-12-31      17:59   Wichita Toys    383.9   Cash
2012-12-31      17:59   Tucson  Pet Supplies    268.39  MasterCard
2012-12-31      17:59   Glendale        Women's Clothing        68.05   Amex
2012-12-31      17:59   Albuquerque     Toys    345.7   MasterCard
2012-12-31      17:59   Rochester       DVDs    399.57  Amex
2012-12-31      17:59   Greensboro      Baby    277.27  Discover
2012-12-31      17:59   Arlington       Women's Clothing        134.95  MasterCard
2012-12-31      17:59   Corpus Christi  DVDs    441.61  Discover
```

There's also a hadoop fs minus cat, which will display the entire contents of the file and we'll use that later. There are plenty of other hadoop fs commands and as you'll probably have started to realize, they closely mirror standard UNIX file system commands. So, if I want to rename the file, for example, I can say hadoop fs minus mv, which moves purchases.txt, in this case, to newname.txt.

```
[training@localhost data]$ hadoop fs -mv purchases.txt newname.txt
[training@localhost data]$ hadoop fs -ls
Found 1 items
-rw-r--r--   1 training supergroup  211312924 2013-09-12 21:16 newname.txt
[training@localhost data]$ hadoop fs -rm newname.txt
```

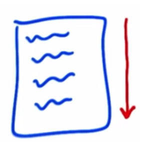
If I want to delete a file, hadoop fs minus rm will remove that file for me. So, let's get rid of newname.txt from hdfs.

```
[training@localhost data]$ hadoop fs -rm newname.txt
Deleted newname.txt
```

I create a directory in hdfs by saying hadoop fs minus mkdir and then the directory name, and now let's upload purchases.txt and place it in the myinput directory so that it's ready for processing by hdfs. Once I've done that, hadoop fs minus ls myinput will show me the contents of that directory. And just as I expected, there's the file.

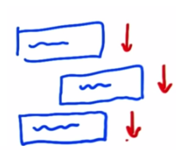
```
[training@localhost data]$ hadoop fs -mkdir myinput
[training@localhost data]$ hadoop fs -put purchases.txt myinput
[training@localhost data]$ hadoop fs -ls
Found 1 items
drwxr-xr-x   - training supergroup          0 2013-09-12 21:16 myinput
[training@localhost data]$ hadoop fs -ls myinput
Found 1 items
-rw-r--r--   1 training supergroup  211312924 2013-09-12 21:16 myinput/purchases.txt
[training@localhost data]$ █
```

MapReduce

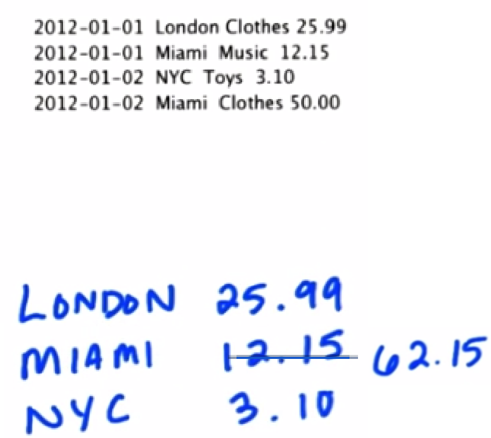


Thanks, Ian. OK, now we've seen how files are stored in HDFS, let's discuss how that data is processed with MapReduce. Say I had a large file. Processing that serially from the top to the bottom could take a long time.

Instead, mapreduce is designed to be a very parallelized way of managing data, meaning that your input data is split into many pieces, and each piece is processed simultaneously. To explain, let's take a real-world scenario.



Let's imagine we run a retailer with thousands of stores around the world. And we have a ledger which contains all the sales from all the stores, organized by date. We've been asked to calculate the total sales generated by each store over the last year.



Now, one way to do that would be just to start at the beginning of the ledger and, for each entry, write the store name and the amount next to it. For the next entry, I need to see if I've already got that store written down; if I have, I can add the amount to that store. If not, I write down a new store name and that first purchase. And so on, and so on.

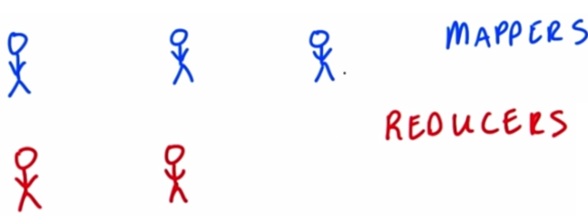
Hashtables

Typically, this is how we'd solve things in a traditional computing environment: we'd create some kind of associative array or hashtable for the stores then process the input file one entry at a time.

What problems do you see with such approach, if you run this on 1 TB of data?
[ ] it will not work
[ ] you might run out of memory
[ ] it will take a long time
[ ] the end result might be incorrect

Answer:
First of all, you got millions and millions of sales to process. So it's going to take an awfully long time for your computer to first read the file from a disk and then to process. Also, the more stores you have, the longer it takes you to check my total sheet, find the right store, and add the new value to the running total for that store. But again, it would take a long time and you may even run out of memory to hold your array if you really do have a huge number of stores. So instead, let's see how you would do this as a MapReduce job.

Mappers and Reducers



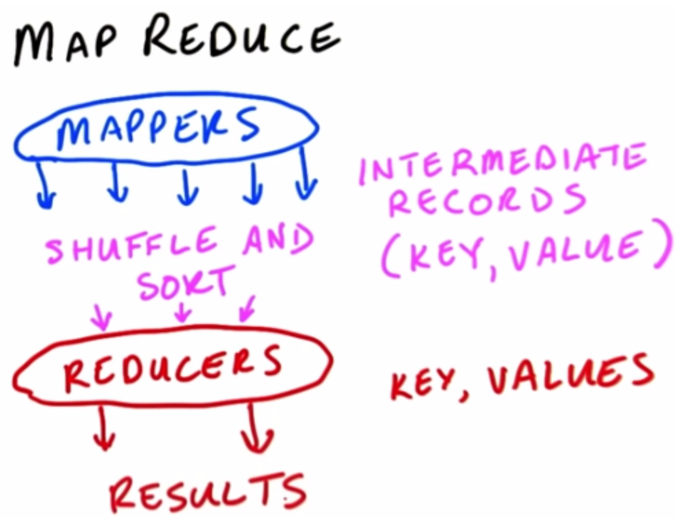
We'll take the staff in the accounts department and split them into two groups, We'll call them the Mappers and the Reducers. Then we'll break the ledger down into chunks, and give each chunk to one of the Mappers. All of the Mappers can work at the same time, and each one is working on just a small fraction of the overall data.

Here's what a Mapper will do. They will take the first record in their chunk of the ledger, and on an index card they'll write the store name as the heading. Underneath, they'll write the sale amount for that record. Then they'll take the next record, and do the same thing. As they're writing the index cards, they'll pile them up so that all the cards for one particular store go on the same pile. By the end, each Mapper will have a pile of cards per store.

Once the Mappers have finished, the Reducers can collect their sets of cards. We tell each Reducer which stores they're responsible for. The Reducers go to all the Mappers and retrieve the piles of cards for their own stores. It's fast for them to do, because each Mapper has separated the cards into a pile per store already. Once the Reducers have retrieved all their data, they collect all the small piles per store and create a large pile per store. Then they start going through the piles, one at a time. All they have to do at this point is add up all the amounts on all the cards in a pile, and that gives them the total sales for that store, which they can write on their final total sheet. And to keep things organized, each Reducer goes through his or her set of piles of cards in alphabetical order.

MapReduce

And that's MapReduce! The Mappers are programs which each deal with a relatively small amount of data, and they all work in parallel. The Mappers output what we call 'intermediate records', which in this case were our index cards. Hadoop deals with all data in the form of records, and records are key-value pairs. In this example, the key was the store name, and the value was the sale total for that particular piece of input. Once the Mappers have finished, a phase of MapReduce called the 'Shuffle and Sort' takes place. The shuffle is the movement of the intermediate data from the Mappers to the Reducers and the combination of all the small sets of records together, and the sort is the fact that the Reducers will organize the sets of records -- the piles of index cards in our example -- into order. Finally, the Reduce phase works on one set of records -- one pile of cards -- at a time; it gets the key, and then a list of all the values, it processes those values in some way (adding them up in our case) and then it writes out its final data for that key.



QUIZ: Final result

Since the intermediate data is only sorted per Reducer, how could we get the final results in total sorted order?

[ ] can't be done
[ ] have only one Reducer
[ ] merge the result files after the job

Answer:

You could either have a single reducer, or merge the result files after the job

QUIZ: Two reducer problem

Assume you have a job which has two Reducers. The Mappers output the following keys:
Apple, Banana, Carrot, Grape
Which keys will go to the first of the two Reducers?

[ ] Apple and Banana
[ ] Apple and Carrot
[ ] Carrot and Grape
[ ] Apple and Grape
[ ] We don't know, but two will go to each Reducer
[ ] We don't know, and it's possible that one Reducer will not get any of the keys

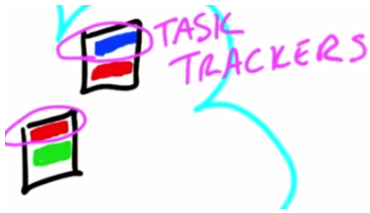
Answer:

Since there is no guarantee that each reducer will get same number of keys, it might be that one of them will get none. For more information on how this works see the links instructor notes.

Daemons of MapReduce

So we've seen conceptually how MapReduce works. In the next lesson, we'll talk about how to actually write code to perform MapReduce jobs on the cluster, but before we do that it's useful to know where the code will actually run.

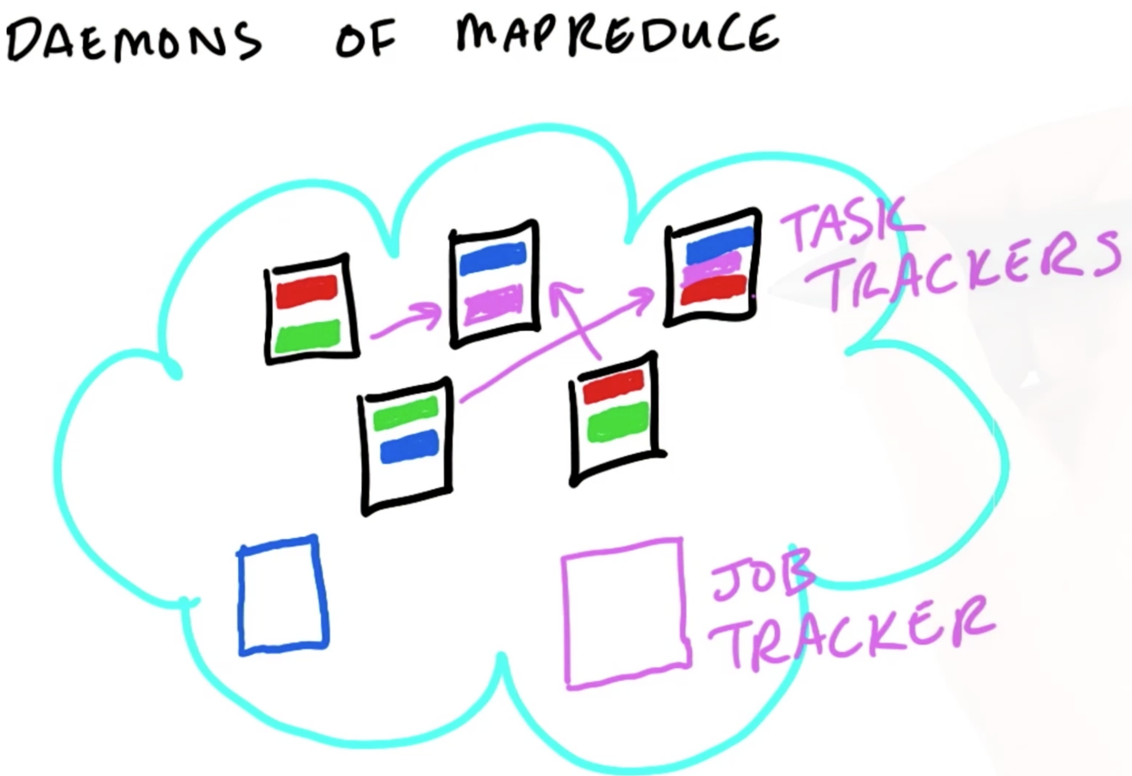
Just as with HDFS, there are a set of daemons -- which are basically just pieces of code which run all the time -- that control MapReduce on the cluster. When you run a MapReducejob, you submit the code to a machine called the JobTracker. That splits the work into Mappers and Reducers, and those Mappers and Reducers will run on the cluster nodes. Running the actual Map tasks and Reduce tasks is handled by a daemon called the



TaskTracker, which runs on each of the slave nodes in the cluster. Notice that since the TaskTrackers run on the same machines as the DataNodes, the Hadoop framework will be able to have Map tasks work on pieces of data that are stored on the same machine, which will save a lot of network traffic.

As we saw, each Mapper processes a portion of the input data known as an 'InputSplit', and by default Hadoop will use an HDFS block as the InputSplit for each Mapper. It will try to make sure that a Mapper works on data which is on the same machine as the block itself, so in an ideal world, the Mapper which processes a block will run on one of the machines which actually stores that block. If block 2 needs processing, for example, it will ideally be processed on this machine, this machine, or this machine. That won't always be possible, because the TaskTrackers on all three machines may already be busy, in which case the data will be streamed to another node for processing, but it should happen the majority of the time.

The Mappers read the input data, and produce intermediate data which the Hadoop framework then passes to the Reducers -- that's the shuffle and sort. The Reducers process that data, and write their final output back to HDFS.



So let's have Ian run a job on our cluster.

Running a Job

It's often the case that MapReduce code is written in Java. However, to make things a little easier for us, we've actually written our mapper and reducer in Python instead. And we can do that thanks to a feature called Hadoop streaming, which allows you to write your code in pretty much any language you'd like. So first of all, let's double-check that we have our input data in HDFS. So, if I Hadoop fs minus ls, then there's my input directory. And if I look at that directory, then yes, there's purchases.txt in there. And in my local directory, I have mapper.py and reducer.py, that's the code for the mapper and reducer, written in Python. We'll look at the actual code in the next lesson.

```
training@localhost:~/udacity_training/code
File  Edit  View  Search  Terminal  Help
[training@localhost code]$ hadoop fs -ls
Found 1 items
drwxr-xr-x   - training supergroup          0 2013-09-12 21:16 myinput
[training@localhost code]$ hadoop fs -ls myinput
Found 1 items
-rw-r--r--   1 training supergroup  211312924 2013-09-12 21:16 myinput/purchases.txt
[training@localhost code]$ ls
mapper.py  reducer.py
[training@localhost code]$
```

Okay, to submit the job we have to give this rather cumbersome command. We say hadoop jar, a path to a jar, then I specify the mapper, I specify the reducer, I need to say -file, for both the mapper and the reducer code. I specify the input directory in HDFS and I specify the output directory to which the reducers will write their output data. And we're calling that joboutput.

```
[training@localhost code]$ hadoop jar /usr/lib/hadoop-0.20-mapreduce/contrib/streaming/hadoop-streaming-2.0
0-mr1-cdh4.1.1.jar -mapper mapper.py -reducer reducer.py -file mapper.py -file reducer.py -input myinput -o
tput joboutput
```

I hit Enter and off we go. Hadoop's pretty verbose, as you can see. As the job runs, you'll see a bunch of output which shows us how far along the job is. It turns out that for this job Hadoop will be running four mappers. And our virtual machine here can only run two at a time. So the job is going to take longer than it would on a larger cluster. Actually, that's worth mentioning here. With the size of the data we have for this example which is only 200 megs, realistically, we could probably have solved this problem faster by just importing the data into a relational database and querying it from there. And that's often the case when we're developing and testing code. Because the test data sets are pretty small, Hadoop isn't necessarily the optimal tool for the job. But when we're done testing and we need to process our full production data, that's when Hadoop really comes into its own. So, as you can see the job is now nearly complete, and when the job has finished we'll see that the last line tells me that the output directory is called joboutput.

```
training@localhost:~/udacity_training/code
File Edit View Search Terminal Help
13/09/12 21:23:25 INFO snappy.LoadSnappy: Snappy native library loaded
13/09/12 21:23:25 INFO mapred.FileInputFormat: Total input paths to process : 1
13/09/12 21:23:25 INFO streaming.StreamJob: getLocalDirs(): [/var/lib/hadoop-hdfs/cache/training/mapred/lo
l]
13/09/12 21:23:25 INFO streaming.StreamJob: Running job: job_201309111631_0001
13/09/12 21:23:25 INFO streaming.StreamJob: To kill this job, run:
13/09/12 21:23:25 INFO streaming.StreamJob: UNDEF/bin/hadoop job  -Dmapred.job.tracker=0.0.0.0:8021 -kill
b_201309111631_0001
13/09/12 21:23:25 INFO streaming.StreamJob: Tracking URL: http://0.0.0.0:50030/jobdetails.jsp?jobid=job_20
09111631_0001
13/09/12 21:23:26 INFO streaming.StreamJob:  map 0%  reduce 0%
13/09/12 21:23:36 INFO streaming.StreamJob:  map 16%  reduce 0%
13/09/12 21:23:39 INFO streaming.StreamJob:  map 24%  reduce 0%
13/09/12 21:23:42 INFO streaming.StreamJob:  map 33%  reduce 0%
13/09/12 21:23:46 INFO streaming.StreamJob:  map 42%  reduce 0%
13/09/12 21:23:49 INFO streaming.StreamJob:  map 50%  reduce 0%
13/09/12 21:23:59 INFO streaming.StreamJob:  map 75%  reduce 0%
13/09/12 21:24:00 INFO streaming.StreamJob:  map 85%  reduce 25%
13/09/12 21:24:03 INFO streaming.StreamJob:  map 94%  reduce 25%
13/09/12 21:24:06 INFO streaming.StreamJob:  map 100%  reduce 25%
13/09/12 21:24:09 INFO streaming.StreamJob:  map 100%  reduce 33%
13/09/12 21:24:12 INFO streaming.StreamJob:  map 100%  reduce 73%
13/09/12 21:24:15 INFO streaming.StreamJob:  map 100%  reduce 81%
13/09/12 21:24:18 INFO streaming.StreamJob:  map 100%  reduce 89%
13/09/12 21:24:21 INFO streaming.StreamJob:  map 100%  reduce 96%
13/09/12 21:24:24 INFO streaming.StreamJob:  map 100%  reduce 100%
13/09/12 21:24:26 INFO streaming.StreamJob: Job complete: job_201309111631_0001
13/09/12 21:24:26 INFO streaming.StreamJob: Output: joboutput
```

Let's take a look at what we've got in there. Hadoop fs minus ls, shows me that yes I do have a job output directory. And if we look at the job output directory, you'll see that it contains three things. It contains a file called _SUCCESS, which just tells me that the job has successfully completed. It contains a directory called _logs, which contains some log information about what happened during the job's run. And then, it contains a file called part-00000. That file is the output from the one reducer that we had for this job.

```
[training@localhost code]$ hadoop fs -ls
Found 2 items
drwxr-xr-x   - training supergroup          0 2013-09-12 21:24 joboutput
drwxr-xr-x   - training supergroup          0 2013-09-12 21:16 myinput
[training@localhost code]$ hadoop fs -ls joboutput
Found 3 items
-rw-r--r--   1 training supergroup          0 2013-09-12 21:24 joboutput/_SUCCESS
drwxr-xr-x   - training supergroup          0 2013-09-12 21:23 joboutput/_logs
-rw-r--r--   1 training supergroup       2296 2013-09-12 21:24 joboutput/part-00000
[training@localhost code]$ hadoop fs
```

Let's take a look at that by saying hadoop fs minus cat part 00000, and we'll pipe that to less on our local machine.

```
[training@localhost code]$ hadoop fs -cat joboutput/part-00000 |
```

That's the contents of that file, which is the output from our reducer. It's the sum total sales broken down by store exactly as we want it.

```
Albuquerque	10052311.42
Anaheim	10076416.36
Anchorage	9933500.4
Arlington	10072207.97
Atlanta	9997146.7
Aurora	9992970.92
Austin	10057158.9
Bakersfield	10031208.92
Baltimore	10096521.45
Baton Rouge	10131273.23
Birmingham	10076606.52
Boise	10039166.74
Boston	10039473.28
Buffalo	10001941.19
Chandler	9919559.86
Charlotte	10112531.34
Chesapeake	10038504.92
Chicago	10062522.07
Chula Vista	9974951.34
Cincinnati	10139505.74
Cleveland	10067835.84
Colorado Springs	10061105.87
Columbus	10035241.03
Corpus Christi	9976522.77
Dallas	10066548.45
Denver	10031534.87
Detroit	9979260.76
Durham	10153890.21
```

```
El Paso	10016409.97
Fort Wayne	10132594.02
Fort Worth	10120830.65
Fremont	10053242.36
Fresno	9976260.26
Garland	10071043.92
Gilbert	10062115.19
Glendale	10044493.97
Greensboro	10033781.39
Henderson	10053416.05
Hialeah	10047052.76
Honolulu	10006273.49
Houston	10042106.27
Indianapolis	10090272.77
Irvine	10084867.45
Irving	10133944.08
Jacksonville	10072003.33
Jersey City	9920141.87
Kansas City	9968118.73
Laredo	10144604.98
Las Vegas	10054257.98
Lexington	10084510.95
Lincoln	10069485.4
Long Beach	10006380.25
Los Angeles	10084576.8
Louisville	10008566.47
Lubbock	9958119.15
Madison	10032035.54
```

```
Memphis	10038565.32
Mesa	10053642.6
Miami	9947316.07
Milwaukee	10064482.65
Minneapolis	10011757.78
Nashville	9961450.51
New Orleans	9949257.75
New York	10085293.55
Newark	10144052.8
Norfolk	10088563.17
North Las Vegas	10029652.51
Oakland	9947292.52
Oklahoma City	10118986.25
Omaha	10026642.34
Orlando	10074922.52
Philadelphia	10190080.26
Phoenix	10079076.7
Pittsburgh	10090124.82
Plano	10046103.61
Portland	10007635.77
Raleigh	10061442.54
Reno	10079955.16
Richmond	9992941.59
Riverside	10006695.42
Rochester	10067606.92
Sacramento	10123468.18
Saint Paul	10057233.57
San Antonio	10014441.7
```

```
Portland	10007635.77
Raleigh	10061442.54
Reno	10079955.16
Richmond	9992941.59
Riverside	10006695.42
Rochester	10067606.92
Sacramento	10123468.18
Saint Paul	10057233.57
San Antonio	10014441.7
San Bernardino	9965152.04
San Diego	9966038.39
San Francisco	9995570.54
San Jose	9936721.41
Santa Ana	10050309.93
Scottsdale	10037929.85
Seattle	9936267.37
Spokane	10083362.98
St. Louis	10002105.14
St. Petersburg	9986495.54
Stockton	10006412.64
Tampa	10106428.55
Toledo	10020768.88
Tucson	9998252.47
Tulsa	10064955.9
Virginia Beach	10086553.5
Washington	10139363.39
Wichita	10083643.21
Winston-Salem	10044011.83
(END)
```

Incidentally, if you want to retrieve data from HFDS and put it onto your local disk, you can do that with Hadoop fs minus get. Hadoop fs minus get is the opposite of Hadoop fs minus put. It just pulls data from HDFS and puts it on the local disk. So as you can see, now I have my local file.txt on my local disk And I can manipulate that however I'd like.

```
[training@localhost code]$ hadoop fs -get joboutput/part-00000 mylocalfile.txt
[training@localhost code]$ ls
mapper.py  mylocalfile.txt  reducer.py
[training@localhost code]$ less mylocalfile.txt
```

That Hadoop job command we typed was pretty painful to have to remember. So to save you time, we've created an alias in the demo virtual machine that you'll be downloading. You can just type hs followed by four arguments, the mapper script, the reducer script, the input directory, and the output directory.

```
[training@localhost code]$ hs mapper.py reducer.py myinput joboutput
```

Here's one important thing to note, though. When you're running a Hadoop job, the output directory must not already exist. And as you can see, if we try and run the command with an existing directory. In this case, job output. Hadoop refuses to run the job.

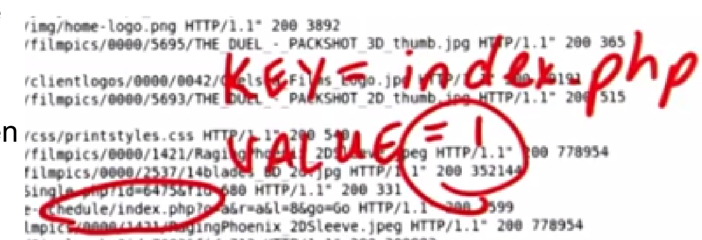
```
packageJobJar: [mapper.py, reducer.py, /tmp/hadoop-training/hadoop-unjar7772862338865304886/] [] /tmp/strea
job275688370291485201.jar tmpDir=null
13/09/12 21:27:26 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications s
ould implement Tool for the same.
13/09/12 21:27:26 INFO mapred.JobClient: Cleaning up the staging area hdfs://0.0.0.0:8020/var/lib/hadoop-hd
s/cache/mapred/mapred/staging/training/.staging/job_201309111631_0002
13/09/12 21:27:26 ERROR security.UserGroupInformation: PriviledgedActionException as:training (auth:SIMPLE)
cause:org.apache.hadoop.mapred.FileAlreadyExistsException: Output directory hdfs://0.0.0.0:8020/user/traini
g/joboutput already exists
13/09/12 21:27:26 ERROR streaming.StreamJob: Error launching job , Output path already exists : Output dire
tory hdfs://0.0.0.0:8020/user/training/joboutput already exists
Streaming Command Failed!
```

This is actually a feature of Hadoop. It's designed to stop you inadvertently deleting or overwriting data that's already in the cluster. But as you can see, if we specify a different directory, which doesn't already exist, then the job will begin just fine.

Processing Logs

The example we just talked about was calculating the average sales per store. And there are lots of other things we can do with MapReduce that are actually quite similar, conceptually, to that. For example, log processing is really quite similar. Imagine you have a set of log files from a Web server which look like this, and you want to know how many times each page has been hit.

Well, it's really similar to the sales per store. Your Mapper will read a line of the log file at a time, and will extract the name of the page -- like index.html, for example.



Its intermediate data will have the name of the page as the key, and a 1 as the value, because you've found one hit to the page at that position in the log. When all the Mappers are done, the Reducers will get the keys, and a list of all the values for each particular key. They can then just add all the '1's up for a key and that will tell them the total number of hits to that page on the Web site. Simple, but far more efficient than writing a stand-alone program to go through all the logs from start to finish if you have hundreds of gigabytes to process.

Practice makes perfect

And that's just the start of what you can do with MapReduce. Things like fraud detection, recommender systems, item classification… there are many, many applications of MapReduce, but they all start with those simple concepts. And they all share some basic characteristics: there's a lot of data to be processed, and the work can be parallelized -- you don't have to just start at the beginning and slog through to the end.

Perhaps the hardest thing to learn when you're new to Hadoop is how to solve problems by thinking in terms of MapReduce. It's a very different way of processing data compared to how you're probably used to working and, honestly, the best way to learn is by practice. In the next lesson we'll write the code to solve our sales-by-store problem, and you'll start to work on other MapReduce problems.

Virtual Machine Setup

We've provided a virtual machine with CDH, Cloudera's distribution of Hadoop, pre-installed. We say that this VM is running a cluster in 'pseudo-distributed mode'. That means it's a complete Hadoop cluster running on a single machine. It's a great way to write and test code, because it really is a complete Hadoop cluster… just one which is running on a single machine. The VM also contains our sample datasets and sample solutions to the problems we're going to ask you to solve. If you haven't already downloaded it, now would be a good time to do so. You can find instructions on how to do that in the Instructor Notes for this lesson.

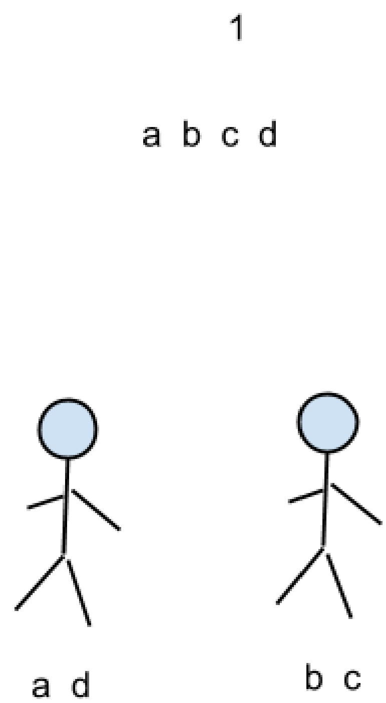
Once you've downloaded and started up the VM, we'd like you to try uploading a data set into HDFS and running a MapReduce job yourself. The exercise instructions document in the Instructor Notes section gives you step-by-step instructions on what to do (Instructions document). Have fun!

Conclusion

So, that's the end of the lesson. You learned about how Hadoop uses HDFS to store data, and the basic principles behind MapReduce. In the next lesson, we'll look at the MapReduce code itself; by the end of the lesson you'll be ready to write your own programs to analyze data.

Number of Reducers

One thing worthy of note is that you, as a developer, specify how many Reducers you want for your job. The default is to have a single Reducer, but for large amounts of data it often makes sense to have many more than one. Otherwise, that one Reducer will end up having to process a huge amount of data from the Mappers. The Hadoop framework decides which keys get sent to each Reducer, and there's no guarantee that each Reducer will get the same number of keys. The keys which go to a particular Reducer are sorted, but each Reducer writes its own file into HDFS. So if, for example, we had four keys: a, b, c, and d, and two Reducers, then one Reducer might get keys a and d, the other might get b and c. So the results would be sorted within each Reducer's output, but just joining the files together wouldn't produce completely sorted output.



QUIZ:

Before we move on, though, which of the following types of problem do you think are good candidates to solve with MapReduce?

[ ] Detecting anomalous behavior from a log file
[ ] Calculating returns from a large number of stock portfolios
[ ] Very large matrix inversion
[ ] (…something else)

Answer: The answer is that all but matrix multiplication are good candidates to solve with MapReduce. The reason matrix inversion is not, is that matrix manipulation tends to require holding the entire contents of both matrices in memory at once, rather than processing individual portions. You can do it with MapReduce, but it turns out to be quite difficult.