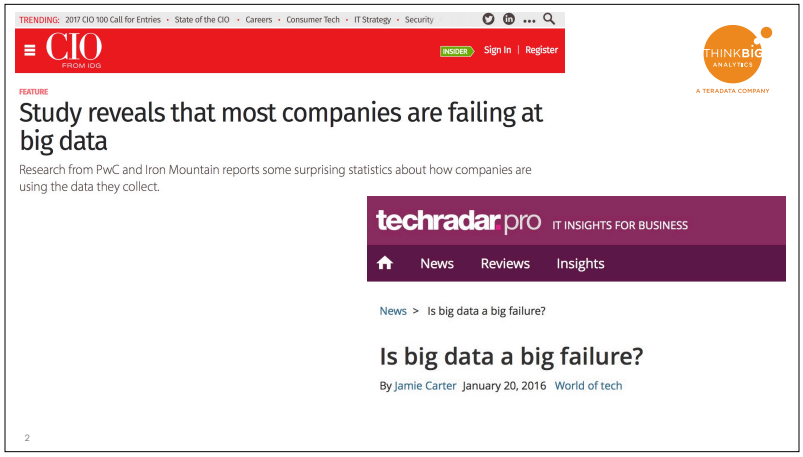
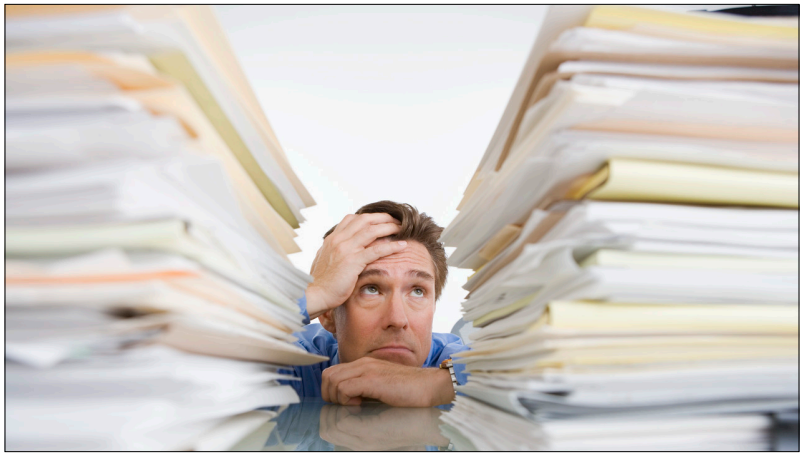
A Quick Introduction to Big Data

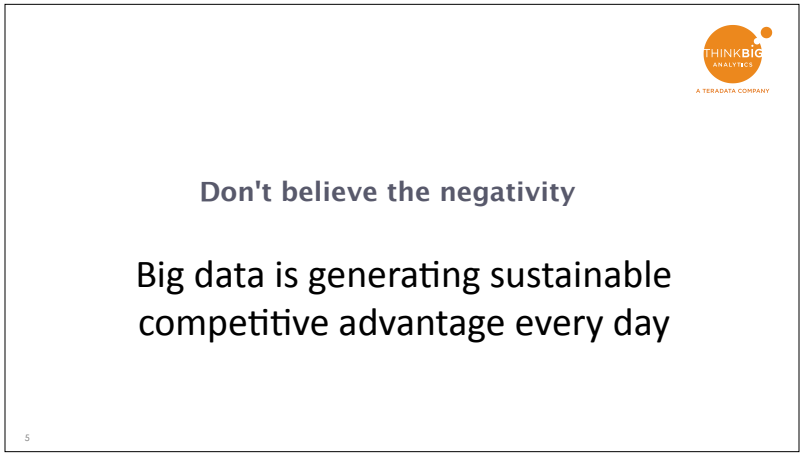


It seems like everywhere we look in technology publications today, people are questioning whether big data has peaked. At the very least, many assert, it's overhyped.

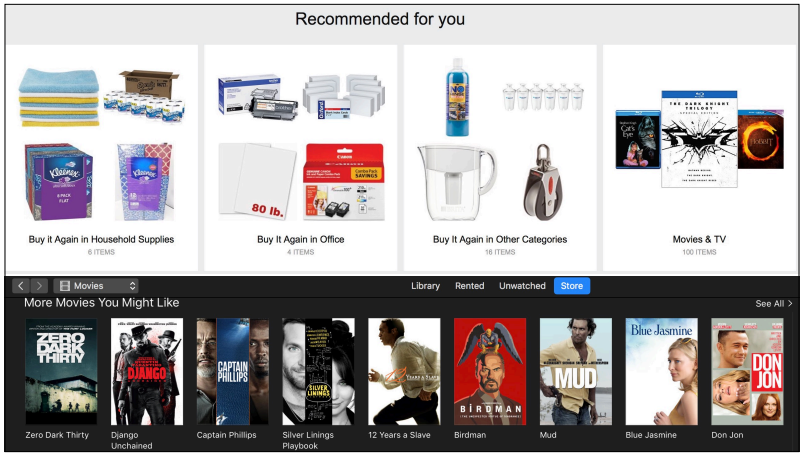
Many analysts assert that Big Data is entering its pets.com phase. For those of you who don't remember, Pets.com was the company that raised $82 million claiming that you wanted to buy all your pet supplies on line. It spent $1.2 million on a Super Bowl ad and promptly went out of business shortly thereafter.



Could it be that Big Data is the next Pets.com?

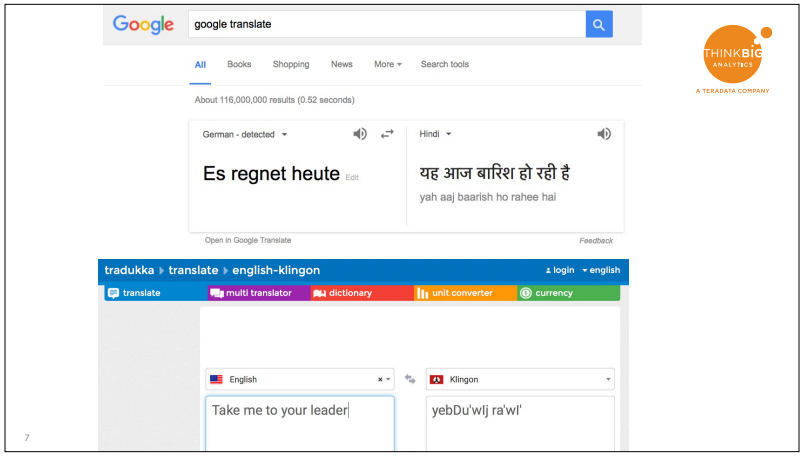


Don't believe the negativity. Big data is generating sustainable competitive advantage every day. How?



If you're an Amazon.com shopper and seen products recommended for you, those product recommendations came from Big Data.

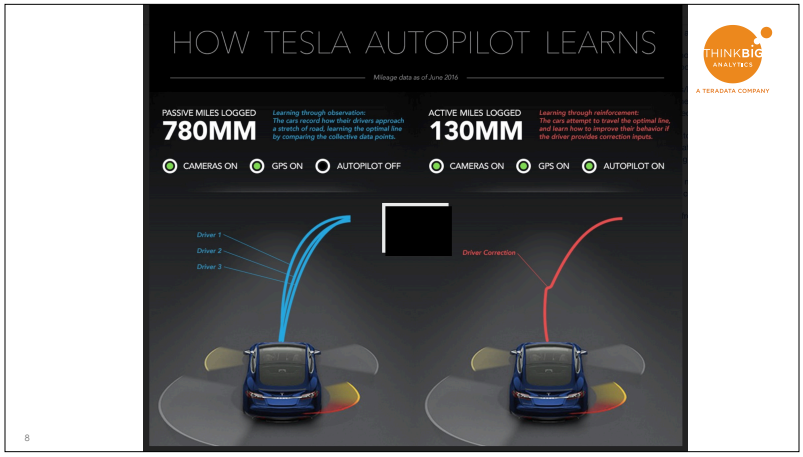
If you saw movies on Netflix or iTunes that were recommended for you, those movie recommendations came from big data.



If you've ever used Google Translate to translate a Web page or a phrase to or from another language, you've used Big Data.

Interestingly, Google Translate doesn't actually learn all the languages it translates; those translations come from massive text correlations among the languages involved. That means that translation services such as Tradukka can even translate from real languages into fictional ones like Klingon, all using big data.

And Big Data doesn't just deliver value on the Internet; it's used to create value in the real world as well. For example take cars that have the ability to steer themselves, like Tesla's Model S and X Autopilot.



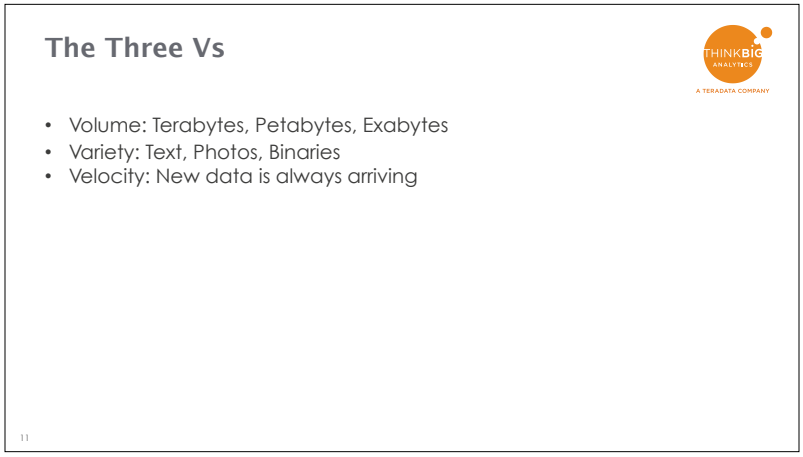
Tesla collects GPS data from its cars at all times, observing the paths they take along various roads. As a result, Tesla provides its cars with Autopilot the optimal path for them to drive. In those cases where the driver decides to override the "optimal" path—perhaps because of a pothole or poor road surface—it records that data too for future drivers. Big data allows our cars to learn from other drivers.

But the ultimate Big Data application is one that touches everyone—health.

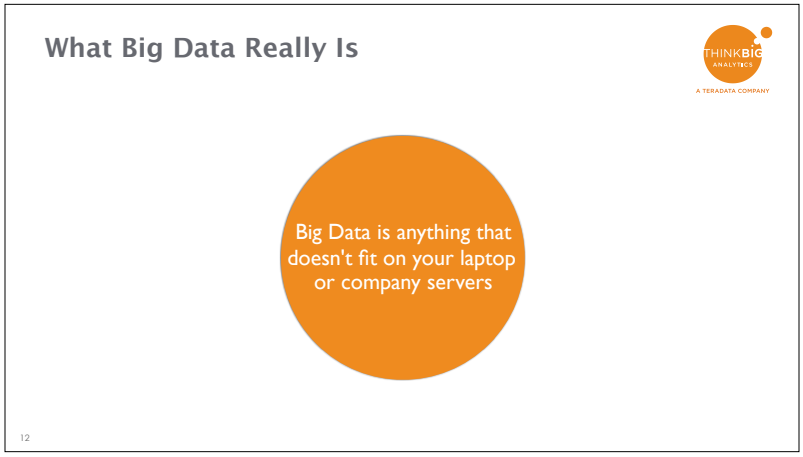
Many of the largest teaching hospitals in the US are pioneering new cancer treatments based on human genomes. This is only possible as the result of collecting the outcomes of tens of thousands of cancer patiences along with their genomic sequences, and then correlating the two data sets. The result: doctors can tailor cancer treatments that optimize outcomes, based on a patient's DNA.

But this begs a question: what constitutes Big Data?

What do you think constitutes big data?

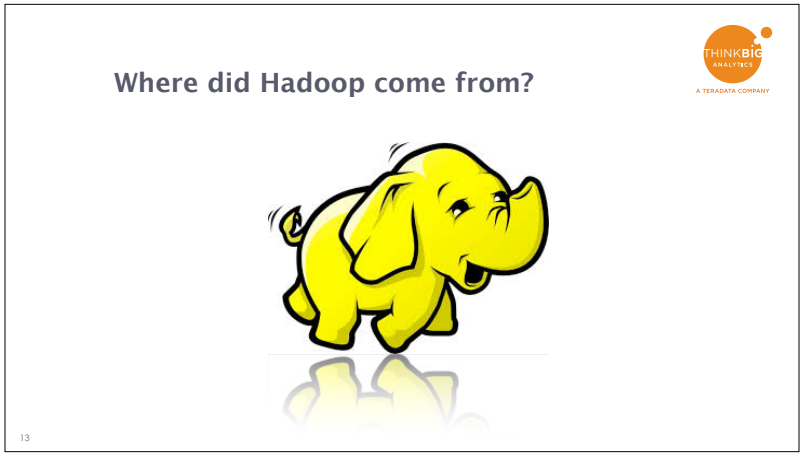


One definition that's often used is the three Vs: Volume, Variety, and Velocity. However, I think a simpler one is just as accurate:



Big Data is really anything that you can't process on your laptop or company servers. Big Data technology means that we are no longer limited by how much hardware we can cram into a single server or PC box.

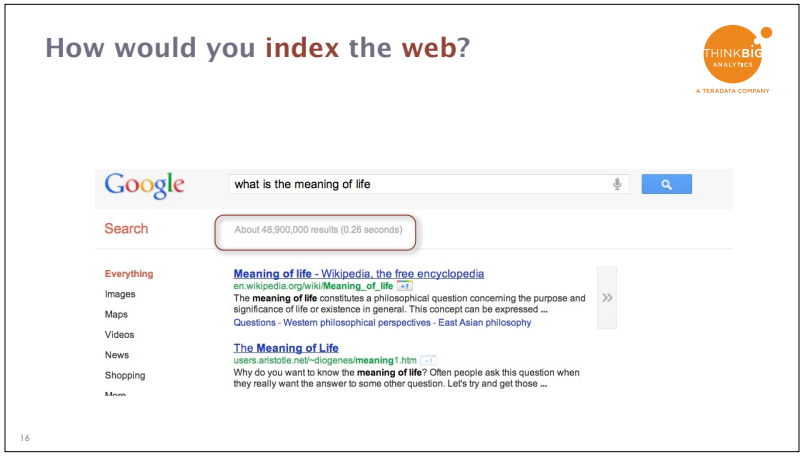
Hadoop is the open source software that makes this all possible. But where did Hadoop come from?



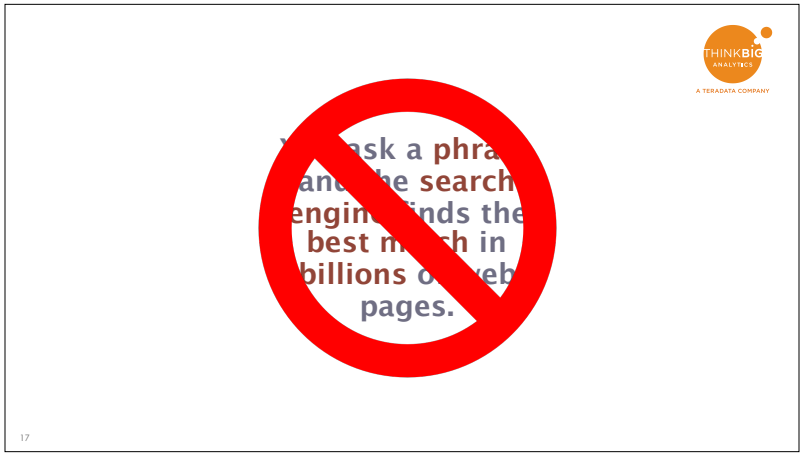
Yes, the name Hadoop came from a child's elephant toy. But Hadoop technology came from something bigger:

The World Wide Web. Now I want you to think back to the olden days of 1996, when Sergei Brin and Larry Page were grad students at Stanford. They had this cool idea that they'd like to provide a search engine for the World Wide Web, and they were going to call it Google. They scrounged up as many old and obsolete PCs as they could find around the lab, stuck them in their office under desks, and started writing software. They named that application Google because it was designed to Index the World Wide Web.

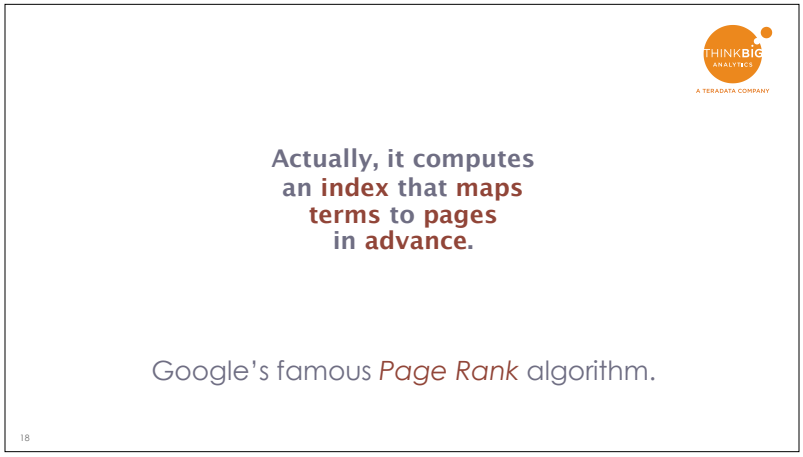
They wanted to be able to find the best answers to questions such as "What is the meaning of life?" on the World Wide Web.



And they wanted to do it fast.

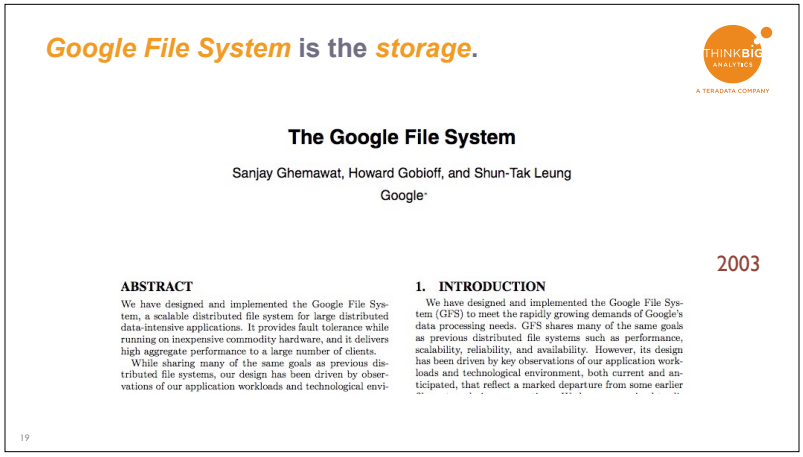


The obvious solution to this would be for you to type in a phrase and Google would find the best match in millions of web pages. However, that would take way too long. It wouldn't be fast.

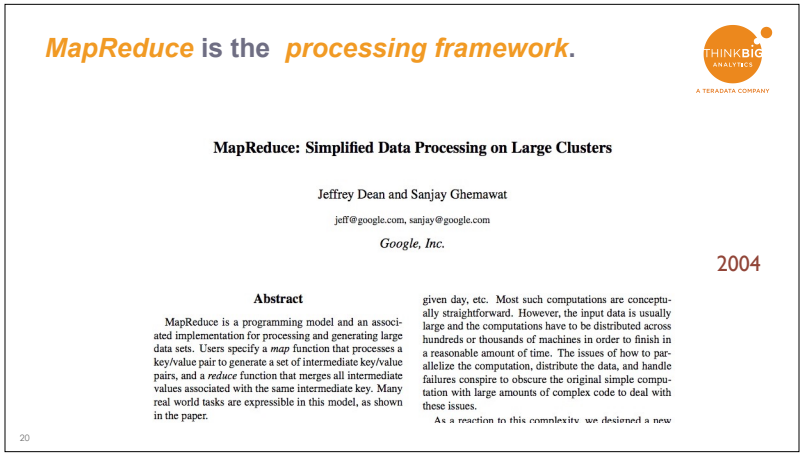


What they did instead was precompute an index that maps terms to pages through the use of a Web spider. The reason they did that was that they could compute this all in advance, and then just search that index when you typed in a query. That, they could make fast.

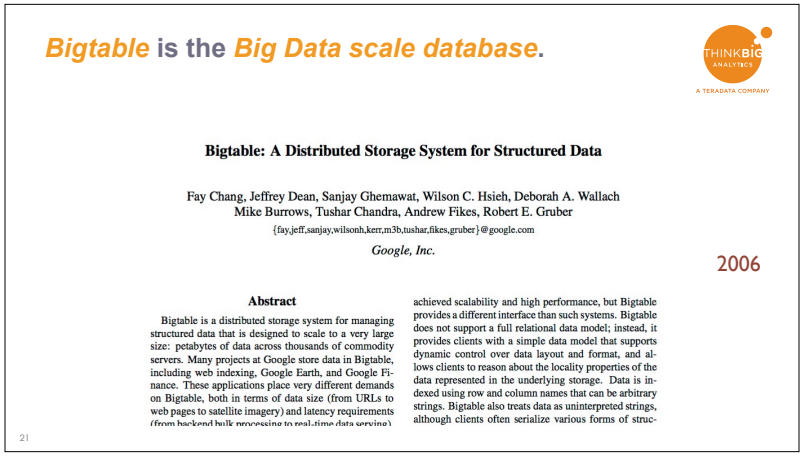
Being good graduate students, however, they didn't didn't just write software; they needed to be published! So in 2003, the Google team published their first paper about the technology they had built:



It was called the Google File System. This was a distributed file system that provided horizontal scalability and resiliency when file blocks are duplicated around the cluster.

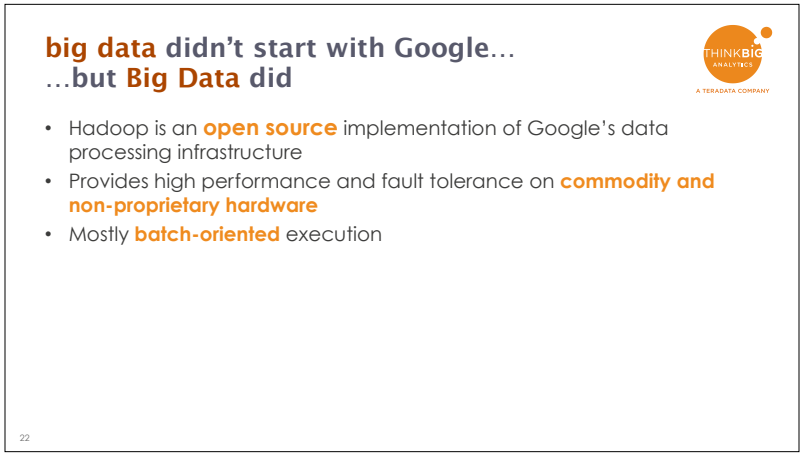


The second paper, published in 2004, was the algorithm they used to distribute the work of computing the index throughout a large cluster of servers. That was called MapReduce and we'll be talking more about that shortly. And finally....



They published a 2006 paper on a new distributed storage system that didn't use a relational model. Google called it BigTable, but today, we'd simply refer to it as a NoSQL database.

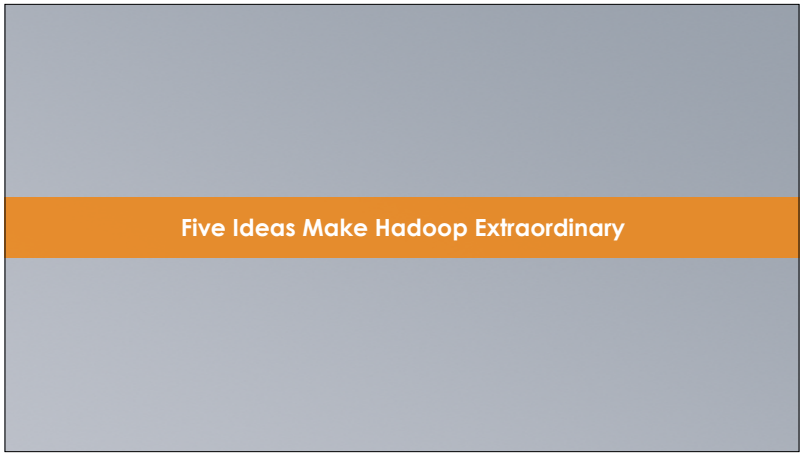
Now to be fair, big data (small b small d) didn't start with Google. However,



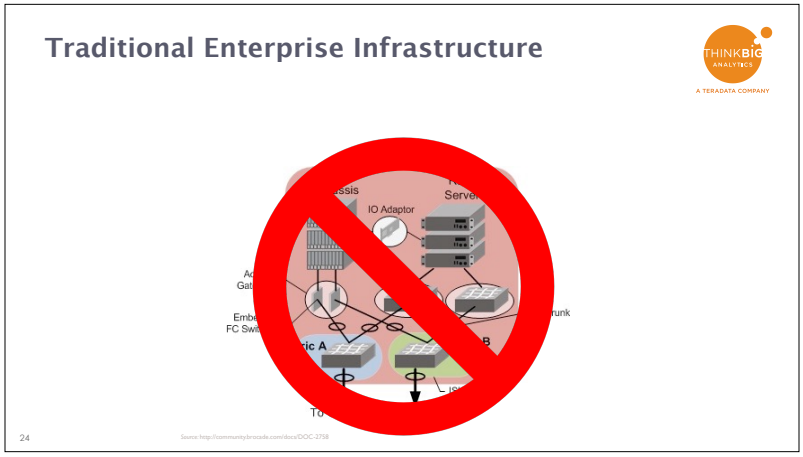
Big Data (capital B capital D) did start with Google. They created the technological foundation for what is now called Hadoop. However, while they published papers on the algorithms, Google kept its code to itself.

Yahoo!, one of Google's competitors at the time, also needed a search engine, so they re-implemented the Google algorithms themselves. However, they went Google one better; they created an open-source implementation of Google's data processing infrastructure.

The great thing about Hadoop is that it provides high performance and fault tolerance on commodity and non-proprietary hardware. You don't need special gear to run Hadoop; all you need is a cluster of commodity servers and networks.



So what makes Hadoop special? What has made it different from traditional IT architectures?



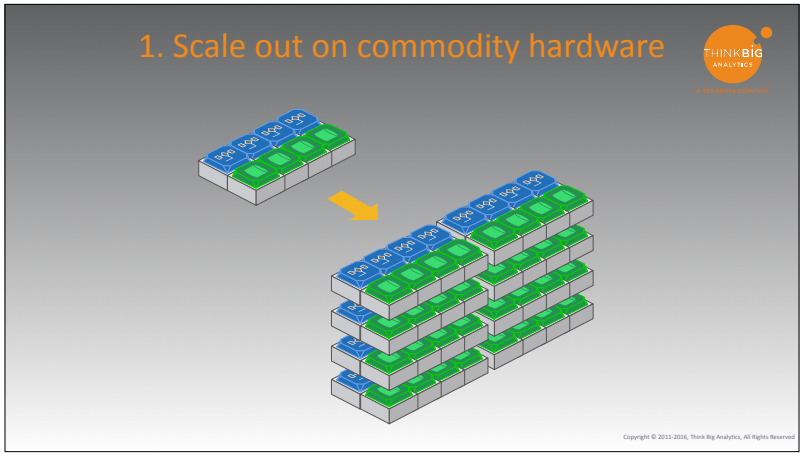
Fundamentally, IT systems architectures have depended on very specialized servers, networks, and storage systems.

Look inside any data center and you'll likely find a wide variety of server types. There's a collection of beefy rack servers. There's another rack that's filled with server blades. Over there is a big high availability server for the database. Oh, and we also have specialized servers that are just for storage and databases, made by companies like Teradata, EMC, and Hitachi.

The same is true of networks. Most data centers have one network for communication (likely Ethernet), but also have a parallel but different network for storage (usually FiberChannel or its derivatives).

What's wrong with specialization you say? Nothing, so long as you never upgrade it and don't care about how much it costs. But when your big million dollar server runs out of capacity, what do you do? You move it off the data center floor with a forklift and bring in an even more expensive server. Oh, and you probably have to reconfigure or rewrite some of your software too.

There has to be a better way, and there is. It's called Hadoop. And it differs from traditional IT infrastructure in 5 specific ways.

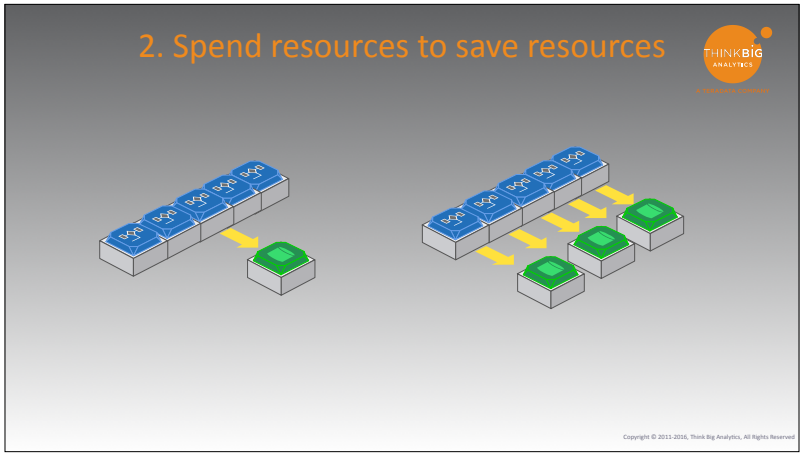


Principle #1: Scale out on commodity hardware.

Hadoop doesn't use specialized servers. It uses garden-variety Intel servers you can buy from any of 100 different server vendors.

Further, when you need to upgrade, you don't need forklift upgrades. Instead, Hadoop uses arrays of commodity hardware configured in what's called scale-out. That means if you need to handle 50% more traffic, you don't replace anything or change any software -- you simply buy 50% more servers. <<click for build>>

This is a similar idea to our principle #2.....



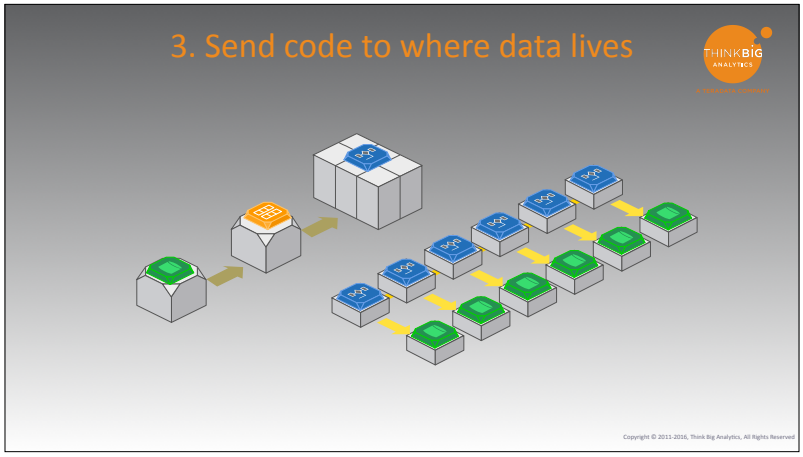
Principle #2 is to spend resources to save resources (usually time)

Typical IT infrastructures have been carefully optimized to make best use of hardware resources. Because Hadoop uses cheap commodity hardware instead of expensive specialized gear, hardware isn't our most precious resource -- it's a commodity! So we are willing to buy extra copies of cheap hardware to save time and to make our computations easier to build. <<click for build>>

An example is thinking about HDFS, the Hadoop Distributed File System. An expensive dedicated storage box would likely use a disk array configured with expensive RAID controllers for reliability. Hadoop takes a different approach, as you'll hear when we talk about HDFS. It simply stores multiple copies of the files on cheap commodity disk drives. If one fails, the software simply routes around it, uses a good copy and creates another copy for further redundancy.

With a 3TB disk costing only about $90, it makes sense to just keep more copies. We're spending resources to save resources.

Our third principle is the first that really comes about because of Big Data....



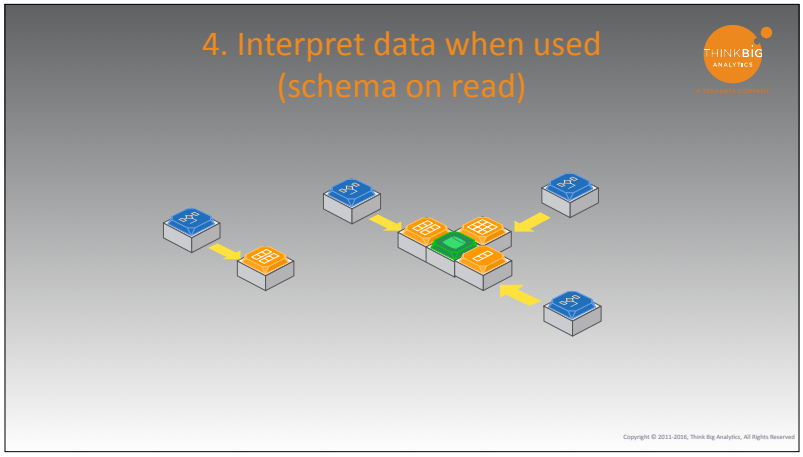
Principle #3: Send code to where the data lives.

In the olden days of mainframe computing, when a company would update its billing records for all its customers, the mainframe would send instructions to the disk controllers to fetch the data. The disk controllers would in turn ask the disks, and all that data would roll into the CPU. The CPU would do its updates, and then send the data back down the channel to the disk controller who would promptly write it to disk.

That worked fine when databases were megabytes or perhaps gigabytes. After all, that wasn't that much bigger than the programs we were running.

But today's clusters deal with Big Data -- data measured in the Terabyte and Petabyte range. That data takes a long time to move, no matter how fast your processors. Meanwhile, our programs haven't really grown that much. So instead of moving the data to the CPU, Hadoop moves the code to where the data lives. <<click for build>>. After all, the code is only megabytes, but the data is terabytes -- moving the cost is thousands of times faster.

Principle #4 is the one that tends to flummox some Database Admins....



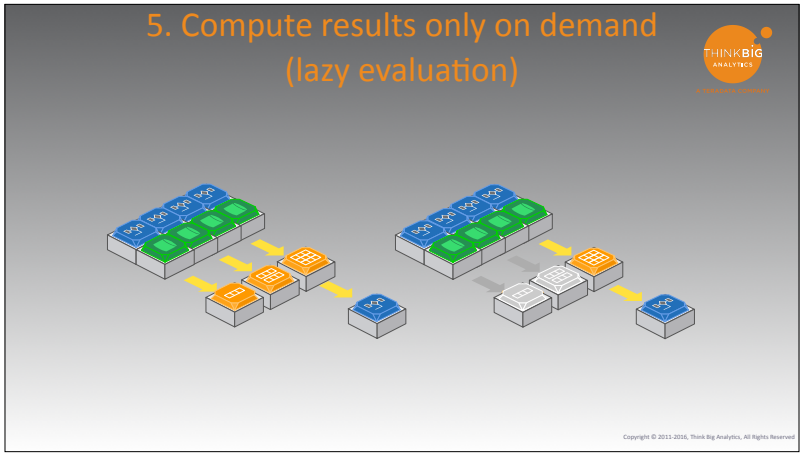
Principal #4 is Interpret Data When Used. This is also called Schema on Read.

Traditional databases relied on a different model: Schema on Write. The programmer would define a schema when she or he created a table, and the database would reserve space on the disk according to that schema to ensure that it would have room for those fields. That's Schema on Write.

With Schema on Read, we don't do anything when a schema is created. Instead, we use schemas to interpret existing files. That means that for a single file, one program could interpret that file as 2 tables <<click for build>>, another program could interpret that same file as only one table <<click for build>, a third program might interpret that as 3 tables. <<click for build>>.

This approach gives us incredible flexibility in how we process Big Data. It also means we can work with datasets even when they have no table schemas at all.

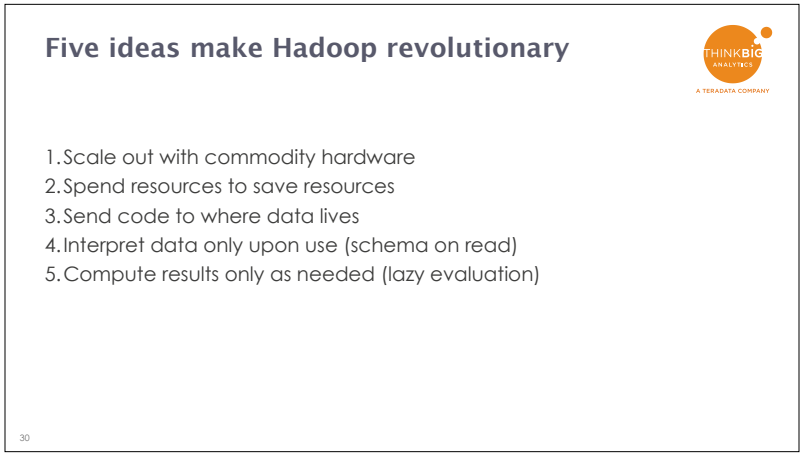
Finally, we have principle #5....



Principle #5 is to compute results only on demand (also known as lazy evaluation).

Rather than running every computation on a schedule, whether its results will be used or not, Hadoop clusters only run computations when their results are requested <<click for build>>. It's kind of like running the computation backwards -- we wait for a request and do the bare minimum to satisfy that request.

Think of this approach being like a college student who never attends class and only studies the materials the night before an exam. If the student knows that the professor will only ask about a certain topic, he or she can simply study that topic and leave the rest for another time (or exam). That's lazy evaluation.

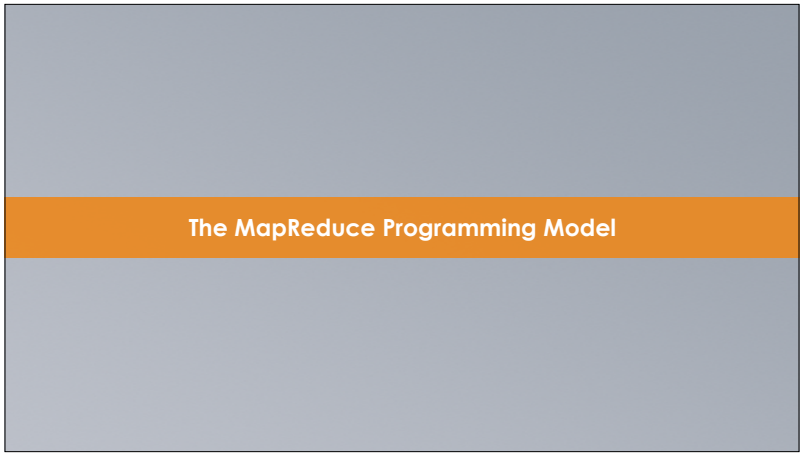


So those are our 5 principles:

1. Scale out with commodity hardware
2. Spend resources to save resources
3. Send code to where data lives
4. Interpret data only upon use (schema on read)
5. Compute results only as needed (lazy evaluation)

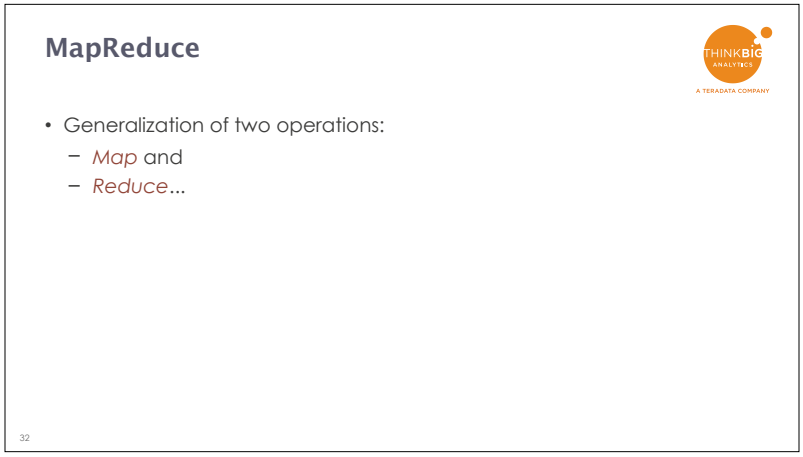
We will see those principles in action throughout this course.

So now, let's look at the first and fundamental programming model behind Hadoop....

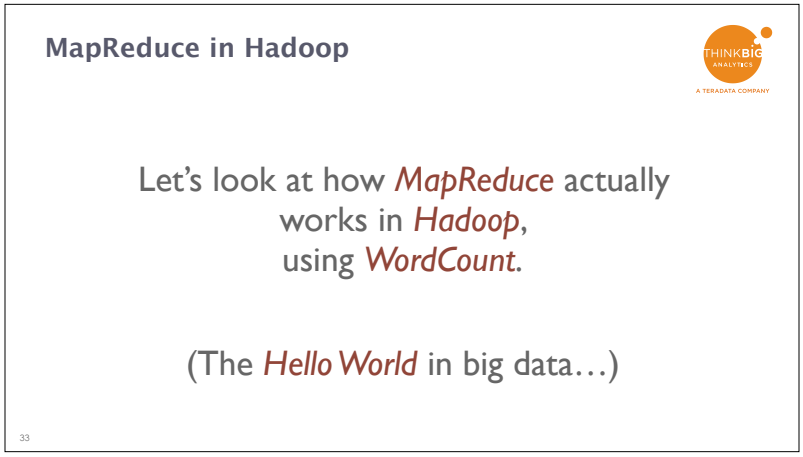


MapReduce.

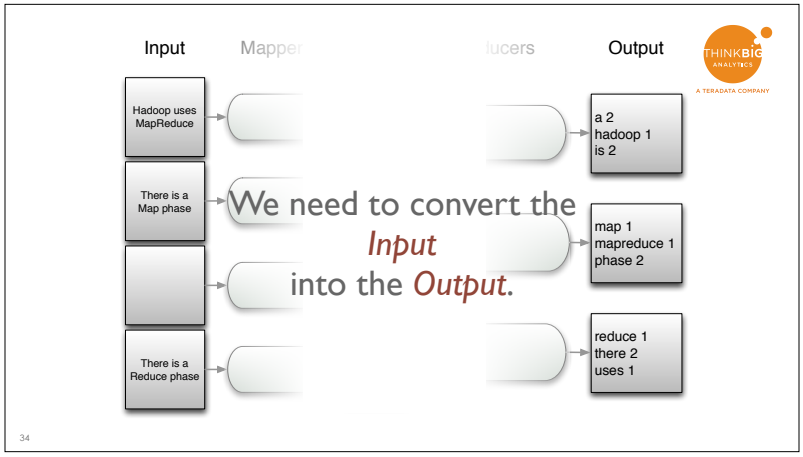
Now MapReduce has two parts to it....



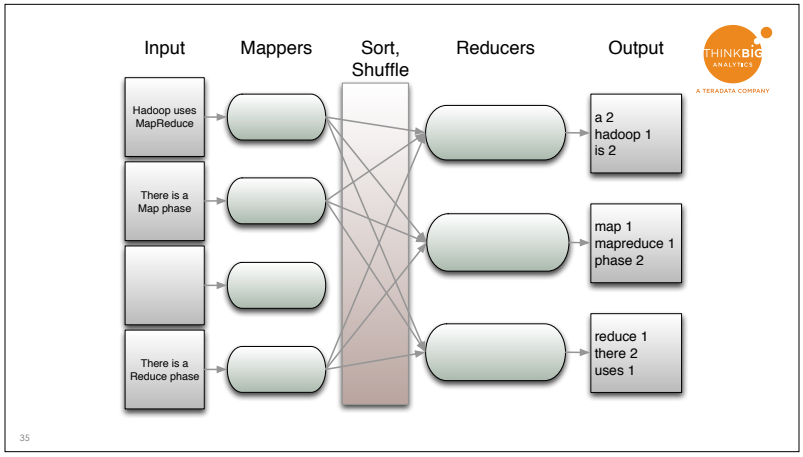
Map and Reduce. Actually, there's another piece called the Shuffle/Sort, but we'll get to that later.



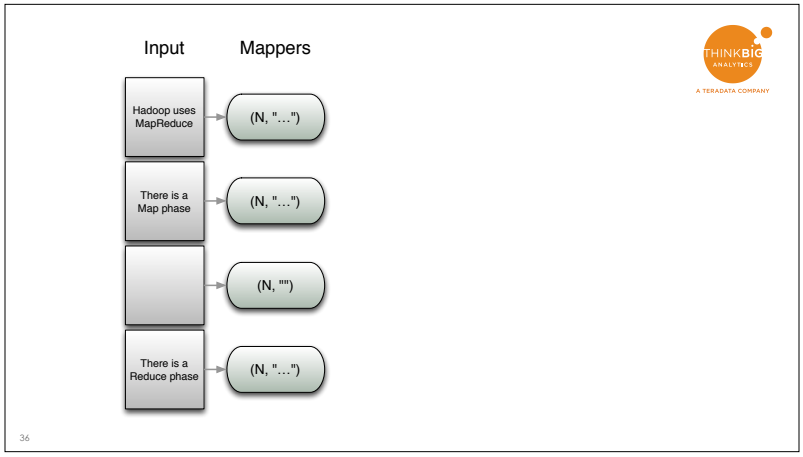
To better understand this, let's look at how MapReduce actually works in Hadoop.



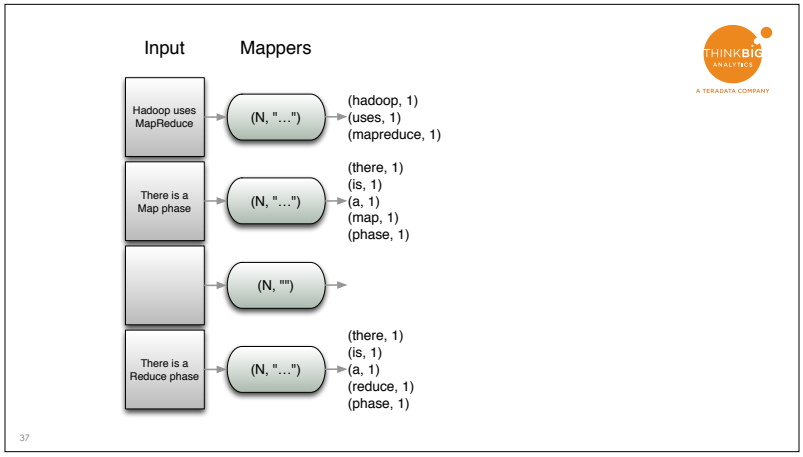
Four input documents, one left empty, the others with small phrases (for clarity…). The word count output is on the right (we'll see why there are three output "documents"). We need to get from the input on the left-hand side to the output on the right-hand side.



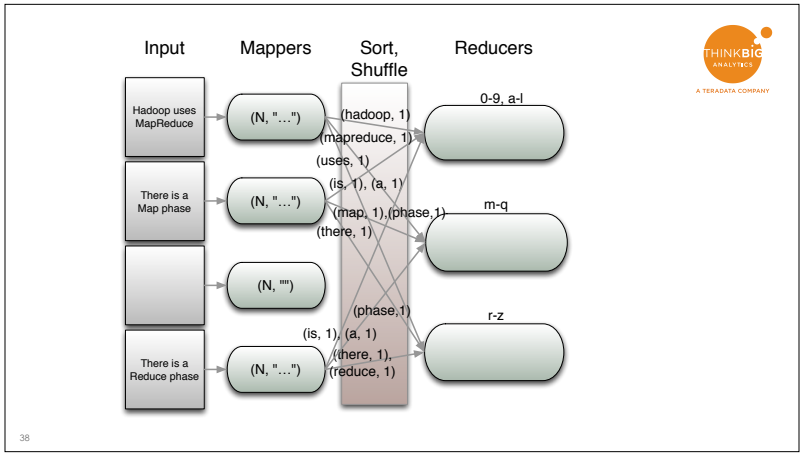
Here is a schematic view of the steps in Hadoop MapReduce. Each Input file is read by a single Mapper process (default: can be many-to-many, as we'll see later).
The Mappers emit key-value pairs that will be sorted, then partitioned and "shuffled" to the reducers, where each Reducer will get all instances of a given key (for 1 or more values).
Each Reducer generates the final key-value pairs and writes them to one or more files (based on the size of the output).



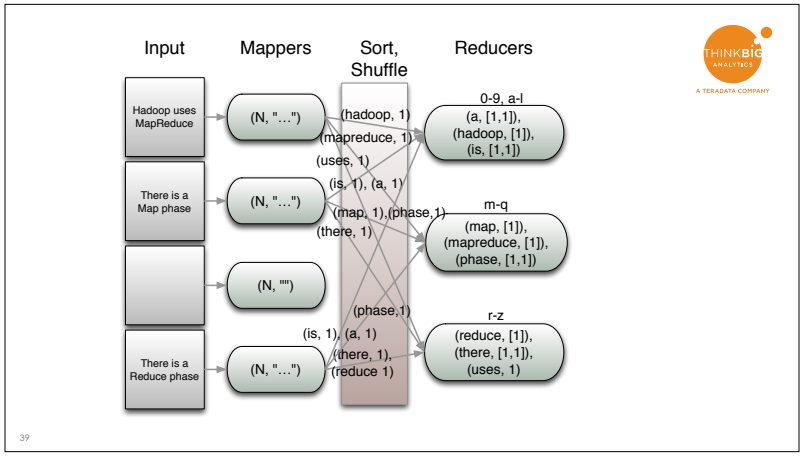
Each document gets a mapper. I'm showing the document contents in the boxes for this example. Actually, large documents might get split to several mappers (as we'll see). It is also possible to concatenate many small documents into a single, larger document for input to a mapper.
Each mapper will be called repeatedly with key-value pairs, where each key is the position offset into the file for a given line and the value is the line of text. We will ignore the key, tokenize the line of text, convert all words to lower case and count them...



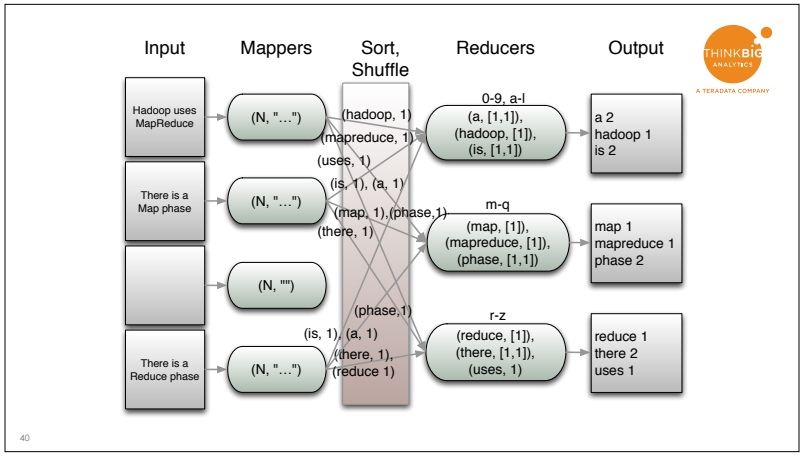
The mappers emit key-value pairs, where each key is one of the words, and the value is the count. In the most naive (but also most memory efficient) implementation, each mapper simply emits (word, 1) each time “word” is seen.
The mappers themselves don’t decide to which reducer each pair should be sent. Rather, the job setup configures what to do and the Hadoop runtime enforces it during the Sort/Shuffle phase, where the key-value pairs in each mapper are sorted by key (that is locally, not globally or “totally”) and then the pairs are routed to the correct reducer, on the current machine or other machines.
Note how we partitioned the reducers (by first letter of the keys). Also, note that the mapper for the empty doc. emits no pairs, as you would expect.



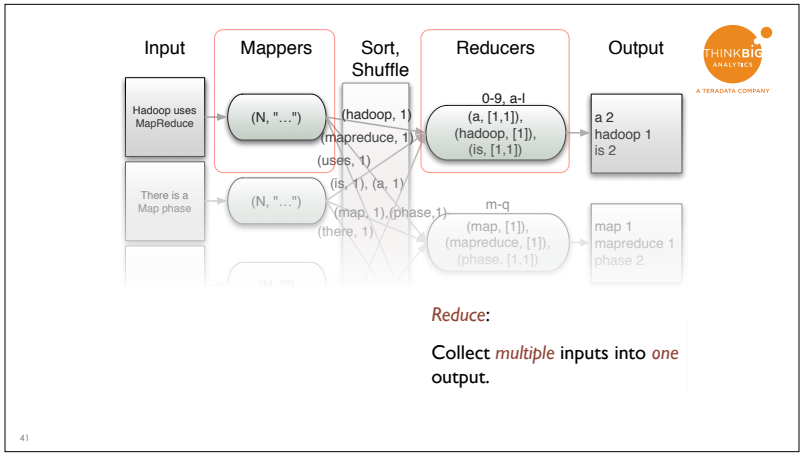
The mappers emit key-value pairs, where each key is one of the words, and the value is the count. In the most naive (but also most memory efficient) implementation, each mapper simply emits (word, 1) each time “word” is seen.
The mappers themselves don’t decide to which reducer each pair should be sent. Rather, the job setup configures what to do and the Hadoop runtime enforces it during the Sort/Shuffle phase, where the key-value pairs in each mapper are sorted by key (that is locally, not globally or “totally”) and then the pairs are routed to the correct reducer, on the current machine or other machines.



Note how we partitioned the reducers (by first letter of the keys). Also, note that the mapper for the empty doc. emits no pairs, as you would expect.

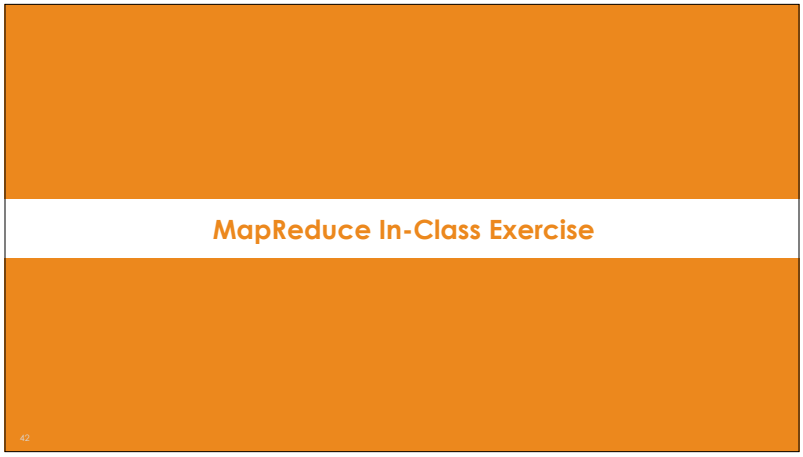


The final view of the WordCount process flow for our example.
We'll see in more detail shortly how the key-value pairs are passed to the reducers, which add up the counts for each word (key) and then writes the results to the output files.
The the output files contain one line for each key (the word) and value (the count), assuming we're using text output. The choice of delimiter between key and value is up to you. (We'll discuss options as we go.)



To recap, a “map” transforms one input to one output, but this is generalized in MapReduce to be one to 0-N. The output key-value pairs are distributed to reducers. The “reduce” collects together multiple inputs with the same key into common buckets and then applies a reduction operator to them.

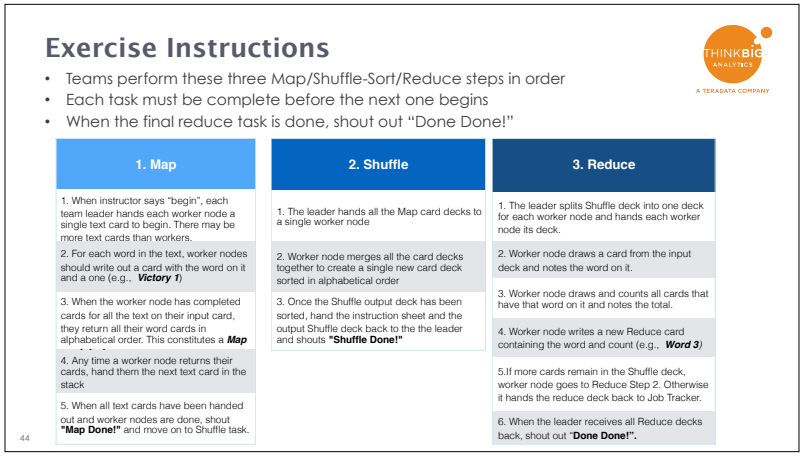
Does everyone understand the steps involved in a MapReduce computation? Yes? You're sure?



Let's find out by doing an exercise. This won't require computers.

Setup

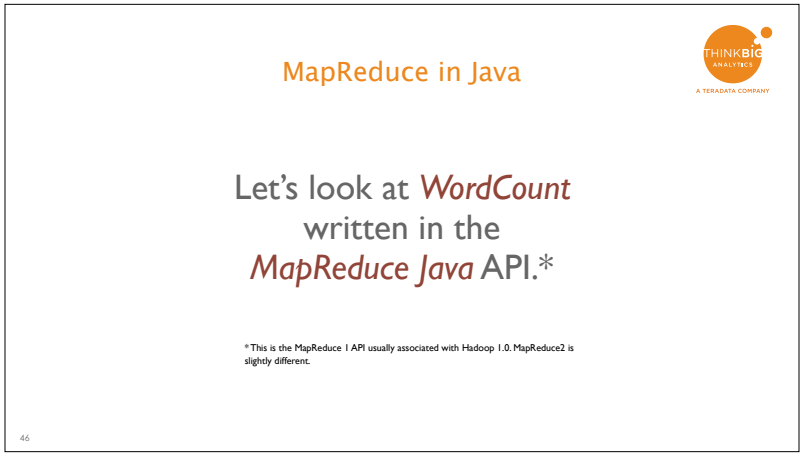
- Break up into teams of at least 4 people per team
- Designate a person to be the job tracker; everyone on the team should be able to reach the job tracker
- Make sure everyone has blank cards and a marker



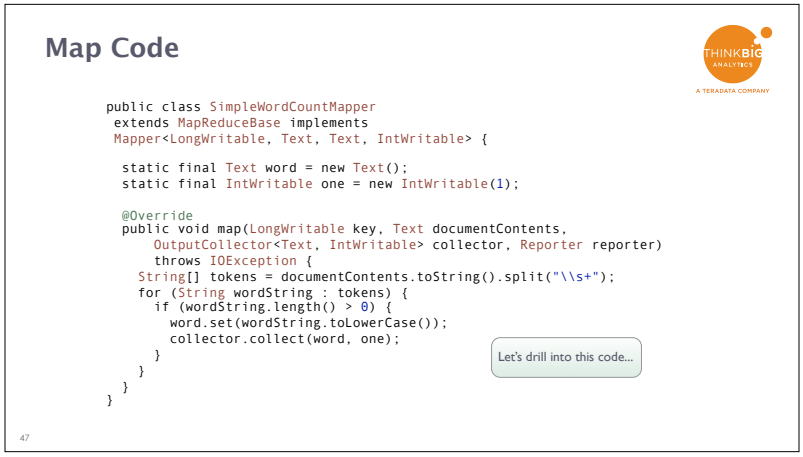
The instructor should time the teams and may wish to award prizes for those who finish first with the correct answers. Instructors are provided answers separately.

Discussion

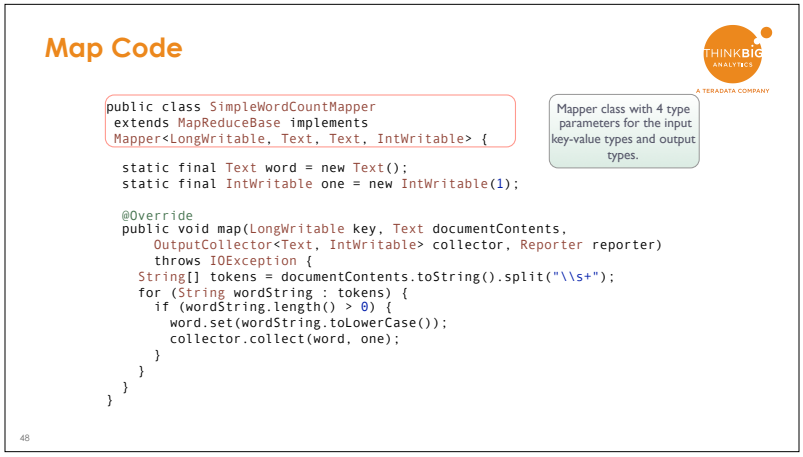
- Did your MapReduce team get the right answer?
- Who was busiest during this process?
- Who was most idle?
- If we were processing a few megabytes of text with 1,000 people, would this process have made more sense?
- Where are the bottlenecks to making a big job go faster?



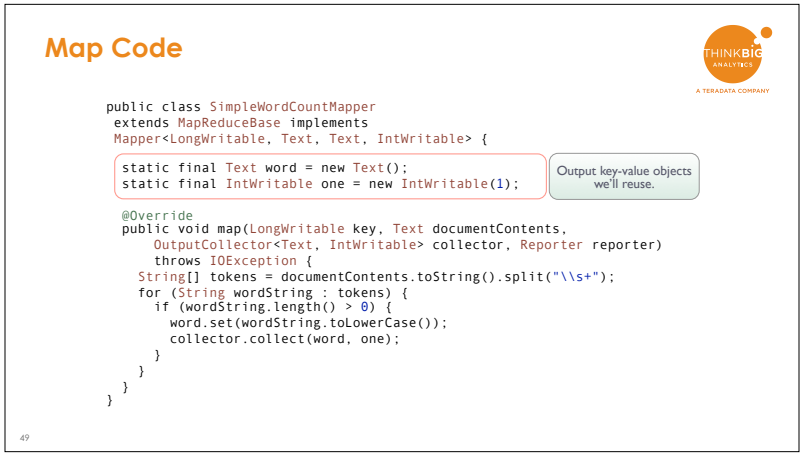
Here's WordCount in the Java API. We'll omit setup and related code.



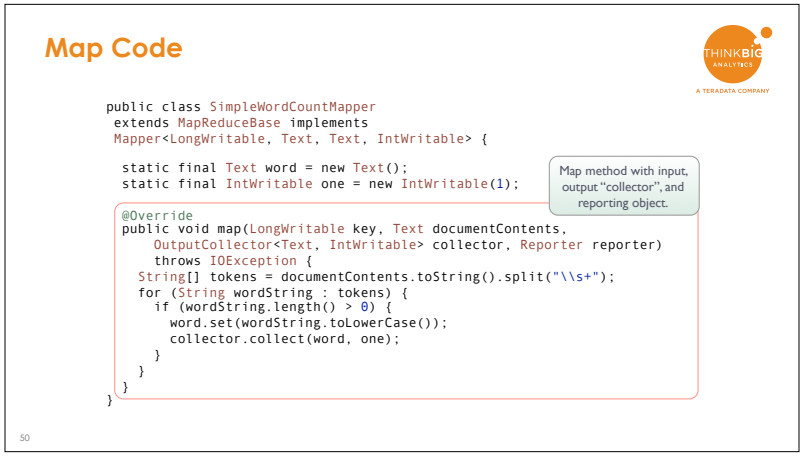
This is the Map step. “map” is passed each line of text, the key is the position offset into the file, which we ignore. We tokenize into words by splitting on whitespace (which doesn’t properly handle punctuation!), convert each word to lower case, and emit a key value pair: (word, 1).



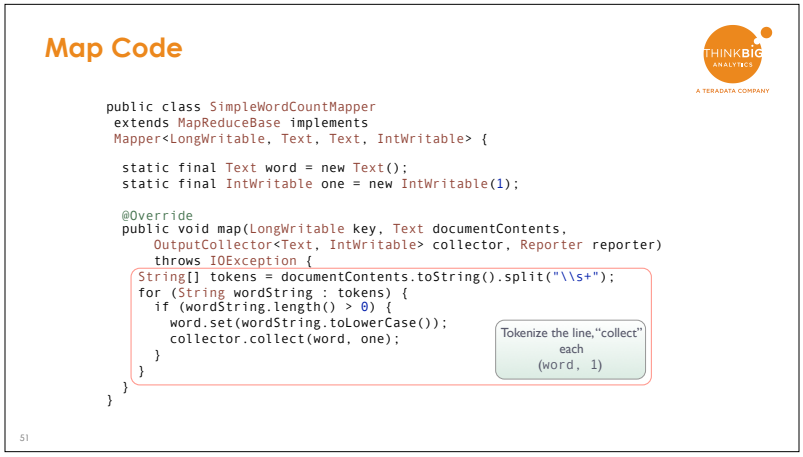
This is the Map step. “map” is passed each line of text, the key is the position offset into the file, which we ignore. We tokenize into words by splitting on whitespace (which doesn’t properly handle punctuation!), convert each word to lower case, and emit a key value pair: (word, 1).



This is the Map step. “map” is passed each line of text, the key is the position offset into the file, which we ignore. We tokenize into words by splitting on whitespace (which doesn’t properly handle punctuation!), convert each word to lower case, and emit a key value pair: (word, 1).



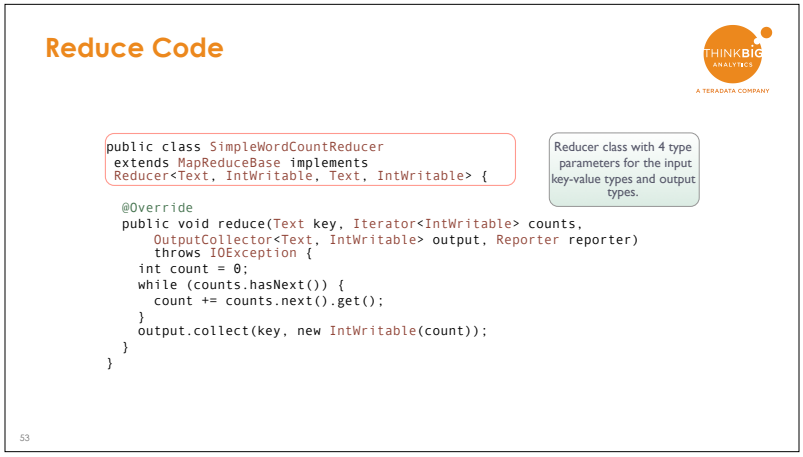
This is the Map step. “map” is passed each line of text, the key is the position offset into the file, which we ignore. We tokenize into words by splitting on whitespace (which doesn’t properly handle punctuation!), convert each word to lower case, and emit a key value pair: (word, 1).



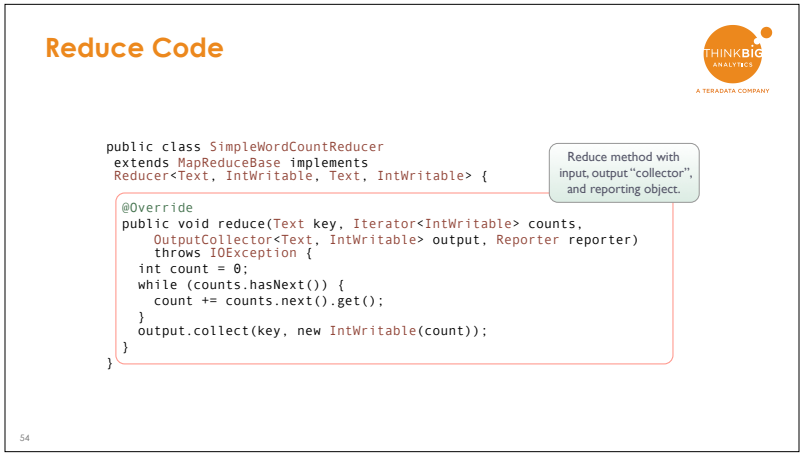
This is the Map step. “map” is passed each line of text, the key is the position offset into the file, which we ignore. We tokenize into words by splitting on whitespace (which doesn’t properly handle punctuation!), convert each word to lower case, and emit a key value pair: (word, 1).

This is the Reduce step. “reduce” is passed each word (the key) and a collection of all the counts for that word (a list of “1s” in our implementation). It simply sums the list and emits the final word and count.

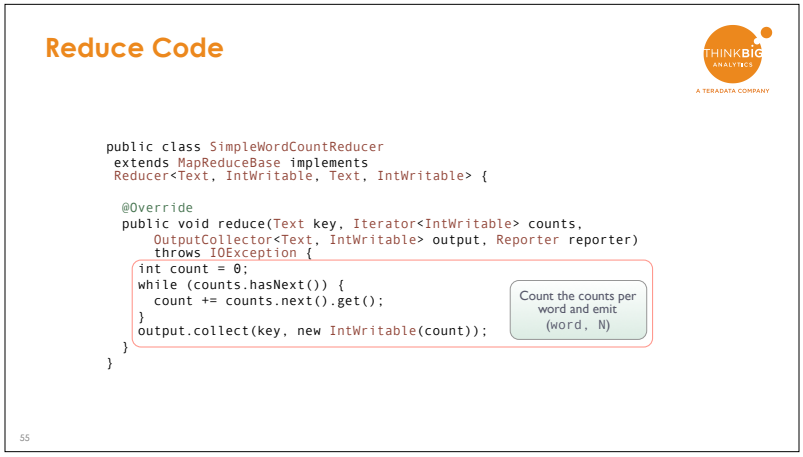
Reduce Code



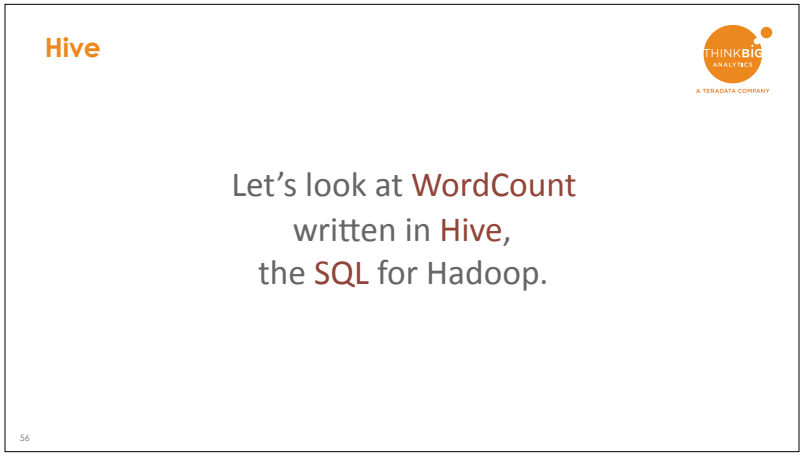
This is the Reduce step. “reduce” is passed each word (the key) and a collection of all the counts for that word (a list of “1s” in our implementation). It simply sums the list and emits the final word and count.



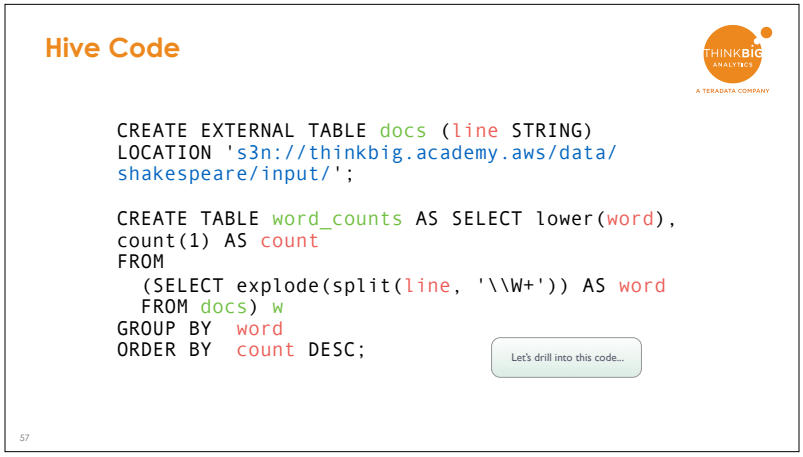
This is the Reduce step. “reduce” is passed each word (the key) and a collection of all the counts for that word (a list of “1s” in our implementation). It simply sums the list and emits the final word and count.



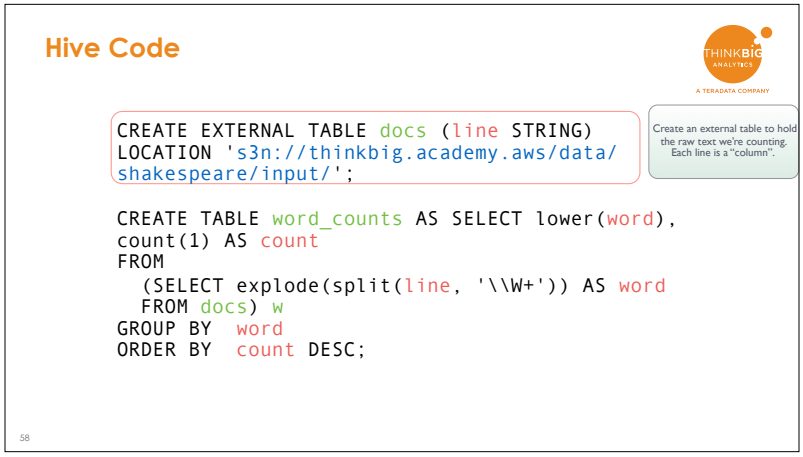
This is the Reduce step. “reduce” is passed each word (the key) and a collection of all the counts for that word (a list of “1s” in our implementation). It simply sums the list and emits the final word and count.



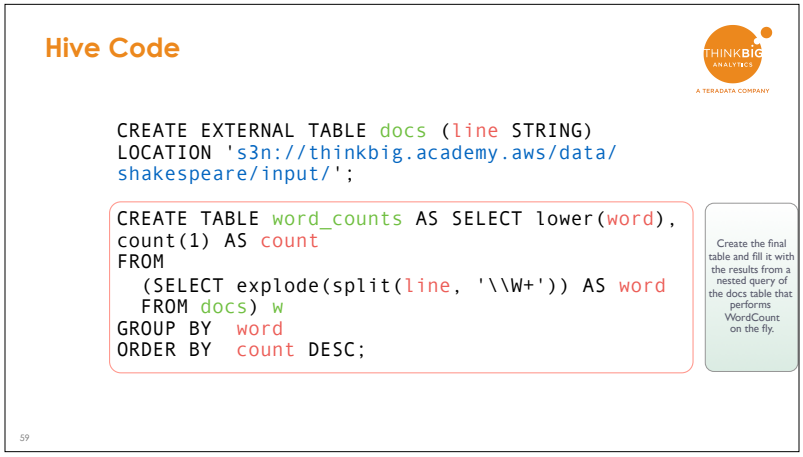
Here's WordCount in the high-level SQL tool for Hadoop, Hive, which was created at Facebook and open sourced to the Apache project in 2008.



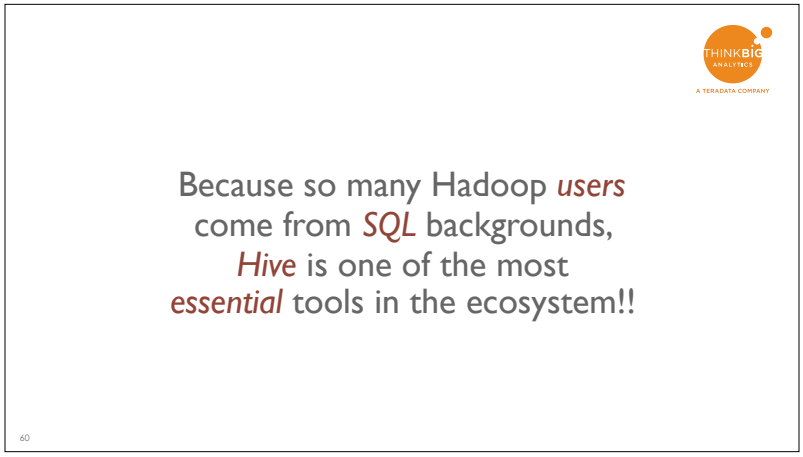
Create a table that holds the documents to word count. These are just arbitrary text documents! We treat each line as a “column” named “line”. Then we load the docs into the table. Finally, we create a word_counts table and load it with a nested query from the docs table that splits into words, groups and orders them, and counts their occurrences.



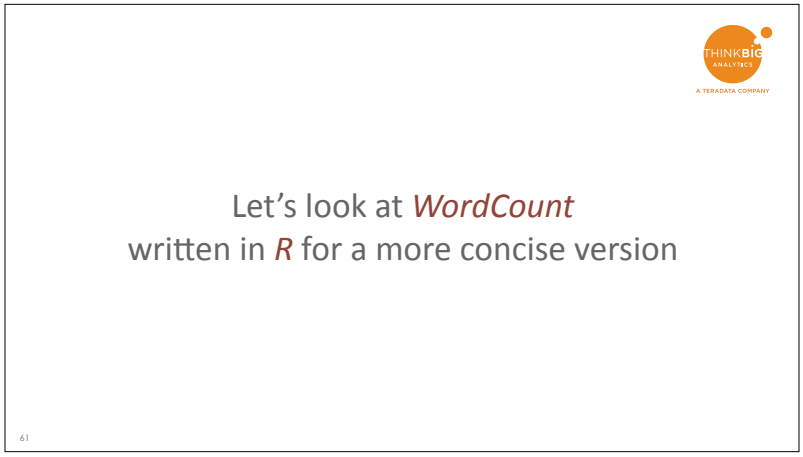
Create a table that holds the documents to word count. These are just arbitrary text documents! We treat each line as a “column” named “line”. Then we load the docs into the table. Finally, we create a word_counts table and load it with a nested query from the docs table that splits into words, groups and orders them, and counts their occurrences.



Create a table that holds the documents to word count. These are just arbitrary text documents! We treat each line as a “column” named “line”. Then we load the docs into the table. Finally, we create a word_counts table and load it with a nested query from the docs table that splits into words, groups and orders them, and counts their occurrences.



Hadoop would not have the penetration it has without Hive.



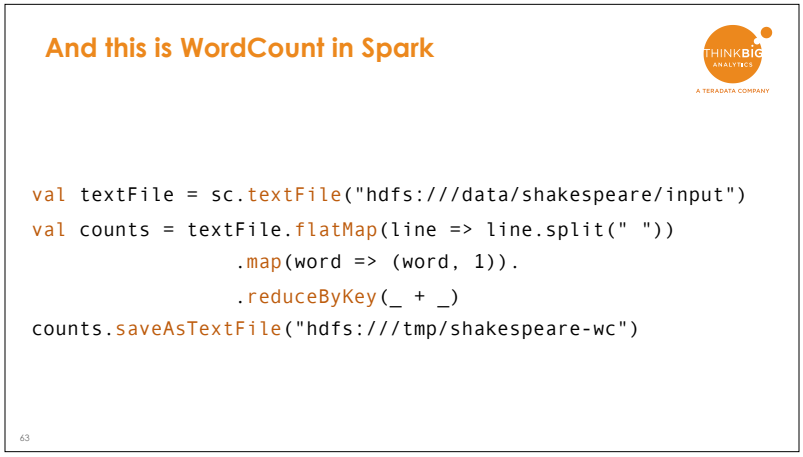
Here's WordCount in R using the rmr2 MapReduce package

This is MapReduce in R version using the rmr2 package


```
map.wc = function(k,lines) {
  words.list = strsplit(lines, '\\W+')
  words = unlist(words.list)
  return( keyval(words, 1) )
}

reduce.wc = function(word,counts) {
  return( keyval(word, sum(counts)) )
}
```

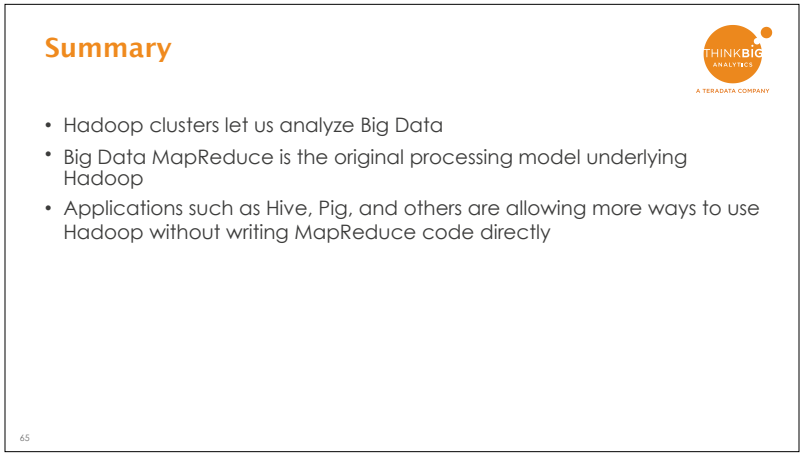

Here's WordCount in R using the rmr2 MapReduce package. It's simple -- just two functions, one for map and the other for reduce. Vector math simplifies the process.



And this is WordCount in Spark.

The fact that we can write Hadoop code in many different languages and frameworks is part of the power of open source software. Anyone can extend it with more tools. So while we refer to Hadoop as a single technology, it's a lot like a Swiss Army knife.

Hadoop is one technology, but it includes many tools for many different applications and use cases. You may not need to know all of them, but you'll want to know the most frequently used ones.



So in summary, <<read the slide>>

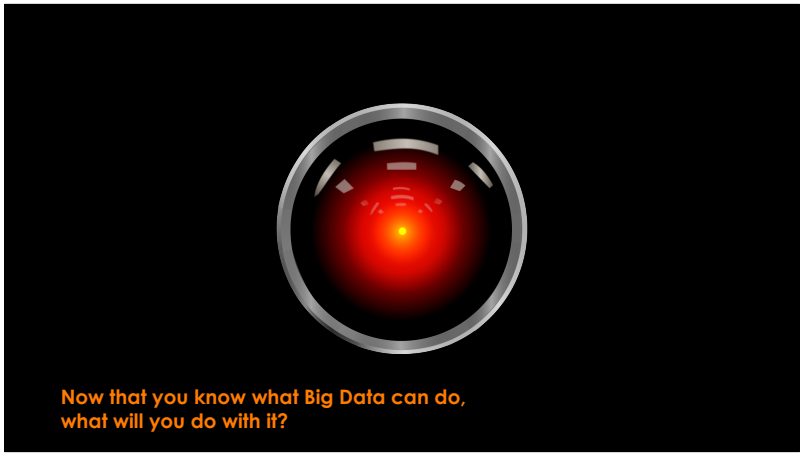
And remember that there are five things that make Hadoop different from traditional IT infrastructure:

Summary: Five ideas make Hadoop revolutionary

1. Scale out with commodity hardware
2. Spend resources to save resources
3. Send code to where data lives
4. Interpret data only upon use (schema on read)
5. Compute results only as needed (lazy evaluation)

<<read the 5 things>>

So now that you know what Big Data can do....



What will you do next?