



Big Data Science In Spark

Course 73005-EL Slides and Course Notes

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



Big Data Science With Spark

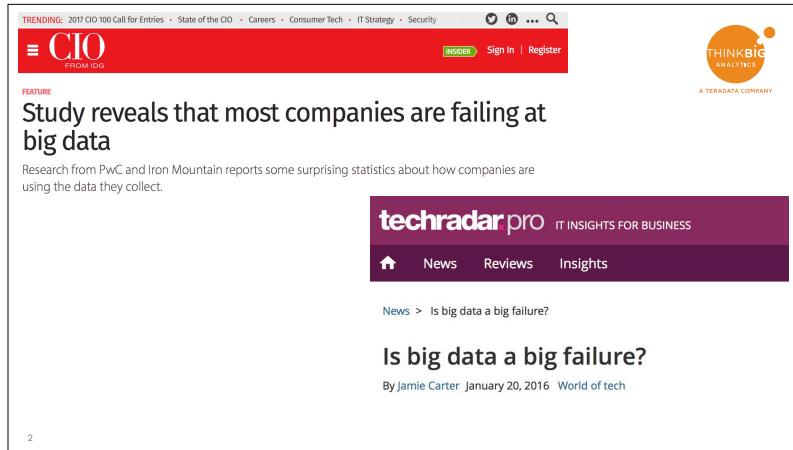
Module 1: Introduction To Big Data

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



A Quick Introduction to Big Data

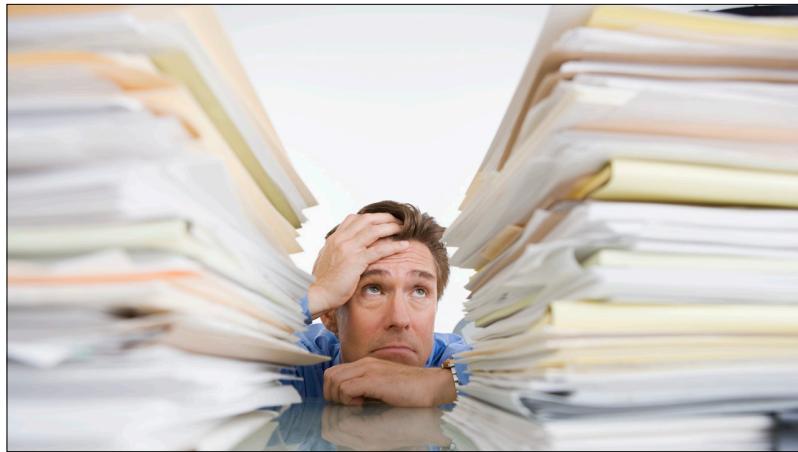
1



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?



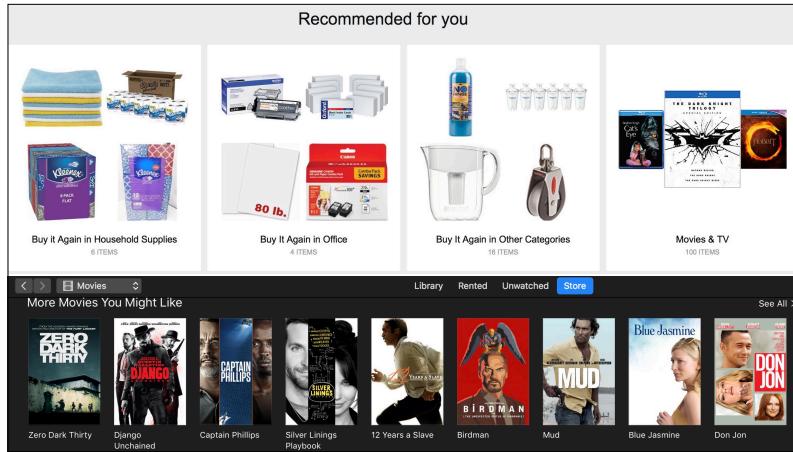
A TERADATA COMPANY

Don't believe the negativity

**Big data is generating sustainable
competitive advantage every day**

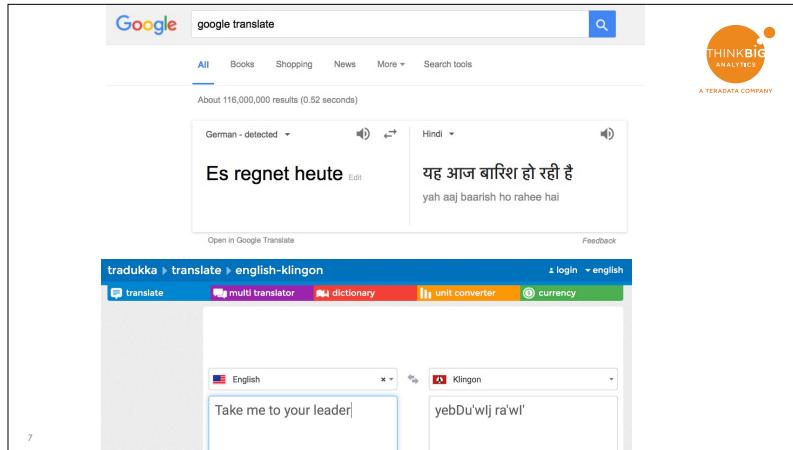
5

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.

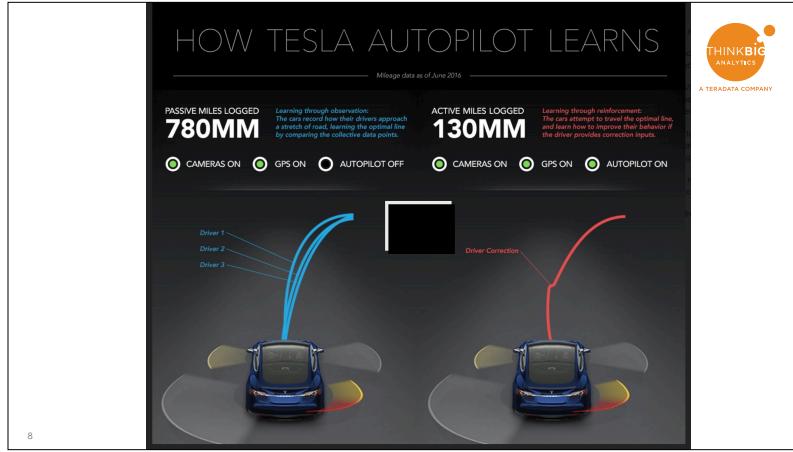
If you saw movies on Netflix or iTunes that were recommended for you, those movie recommendations came from big data.



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 patients 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 Constitutes Big Data?

What do you think constitutes big data?

The Three Vs



A TERADATA COMPANY

- Volume: Terabytes, Petabytes, Exabytes
- Variety: Text, Photos, Binaries
- Velocity: New data is always arriving

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

What Big Data Really Is



A TERADATA COMPANY

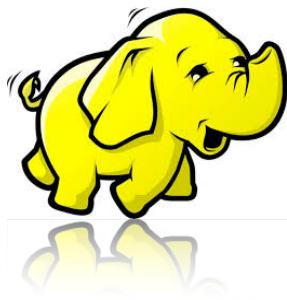
Big Data is anything that
doesn't fit on your laptop
or company servers

12

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?

Where did Hadoop come from?



13

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.

How would you index the web?



Google

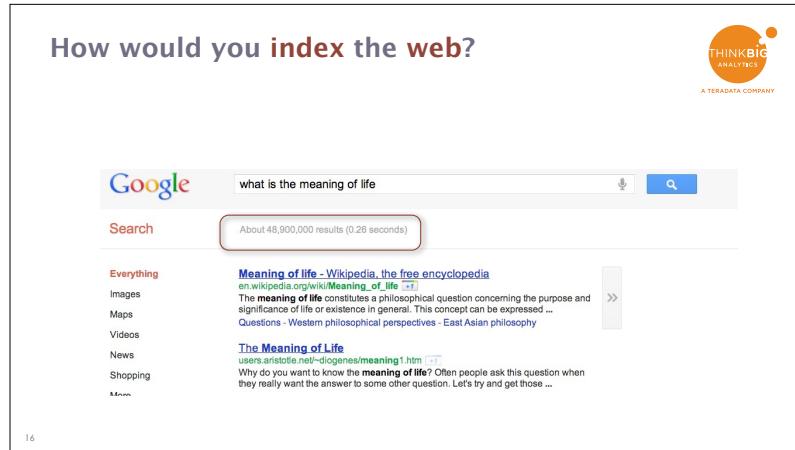
What is the meanin| ♂

what is the meaning of life
what is the meaning of life 42
what is the meaning of pumped up kicks
what is the meaning of halloween
what is the meaning of love
what is the meaning of labor day
what is the meaning of slope
what is the meaning of homecoming
what is the meaning of my last name
what is the meaning of a promise ring

Google Search I'm Feeling Lucky

15

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.



Actually, it computes
an index that maps
terms to pages
in advance.

Google's famous *Page Rank* algorithm.

18

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 just write software; they needed to be published! So in 2003, the Google team published their first paper about the technology they had built:

Google File System is the storage.



A TERADATA COMPANY

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google

2003

ABSTRACT
 We have designed and implemented the Google File System, a scalable distributed system for large factored data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients. While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environments.

1. INTRODUCTION
 We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier

19

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.

MapReduce is the processing framework.



A TERADATA COMPANY

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
 jeff@google.com, sanjay@google.com
Google, Inc.

2004

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity we designed a new

20

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....

Bigtable is the *Big Data scale database*.



A TERADATA COMPANY

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
 Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
 {fayjeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

2006

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time dataerving).

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of struc-

21

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 didn't start with Google...
...but Big Data did**



A TERADATA COMPANY

- Hadoop is an **open source** implementation of Google's data processing infrastructure
- Provides high performance and fault tolerance on **commodity and non-proprietary hardware**
- Mostly **batch-oriented** execution

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.

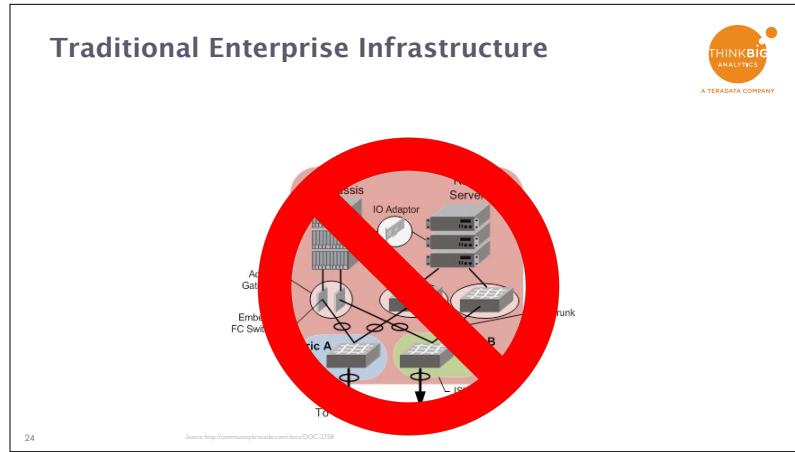
Yahoo!, one of Google's competitors at the time, also needed a search engine, so they re-implemented the Google algorithms themselves. However, they went Google one better; they created an open-source implementation of Google's data processing infrastructure.

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.



Five Ideas Make Hadoop Extraordinary

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



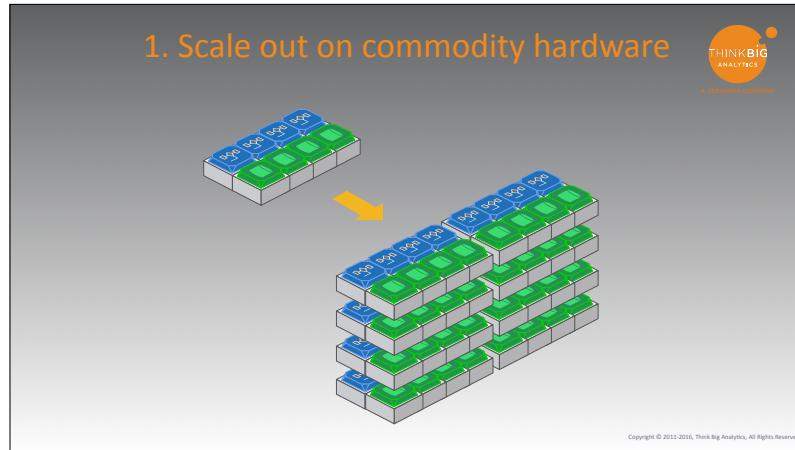
Fundamentally, IT systems architectures have depended on very specialized servers, networks, and storage systems.

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.

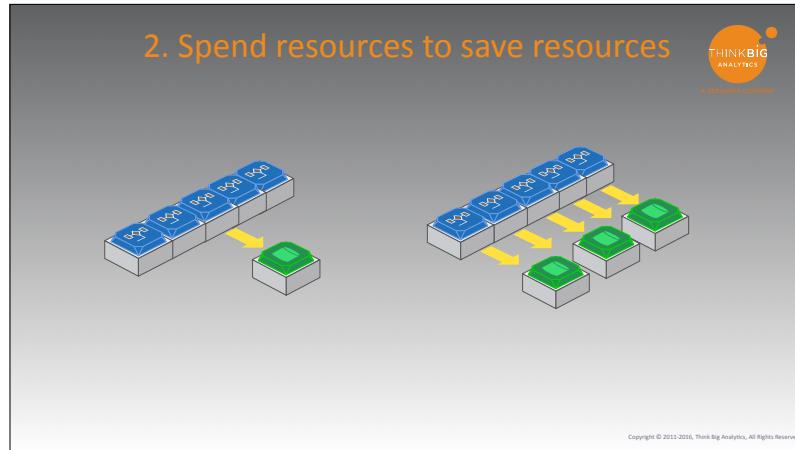


Principle #1: Scale out on commodity hardware.

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.....



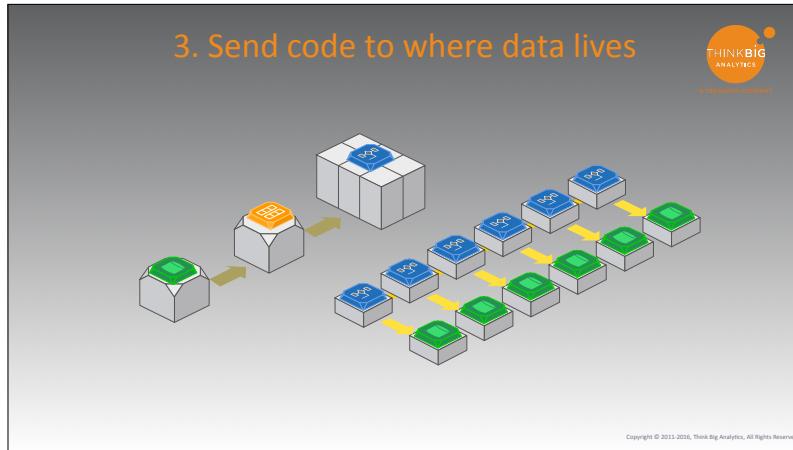
Principle #2 is to spend resources to save resources (usually time)

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....



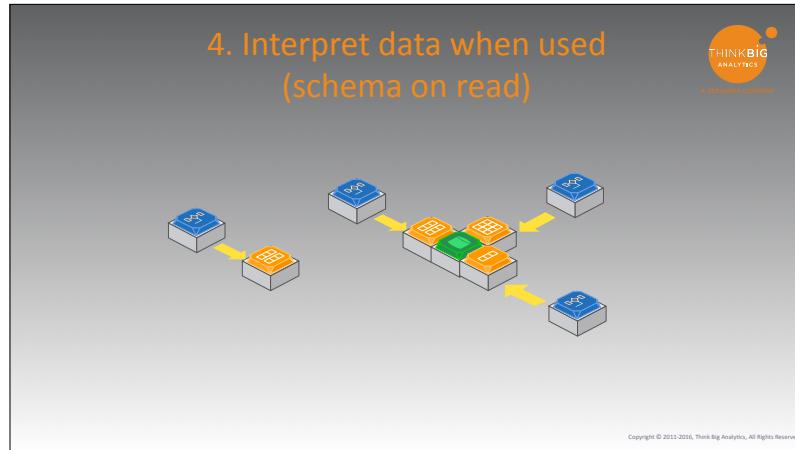
Principle #3: Send code to where the data lives.

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....



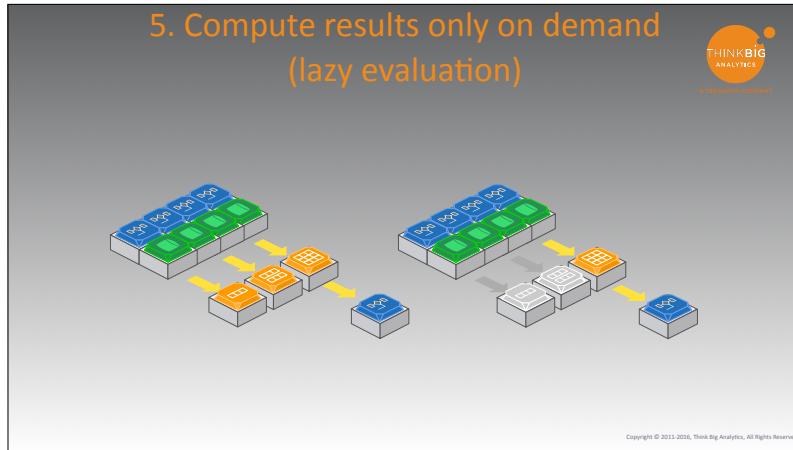
Principal #4 is Interpret Data When Used. This is also called Schema on Read.

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.

Five ideas make Hadoop revolutionary



A TERADATA COMPANY

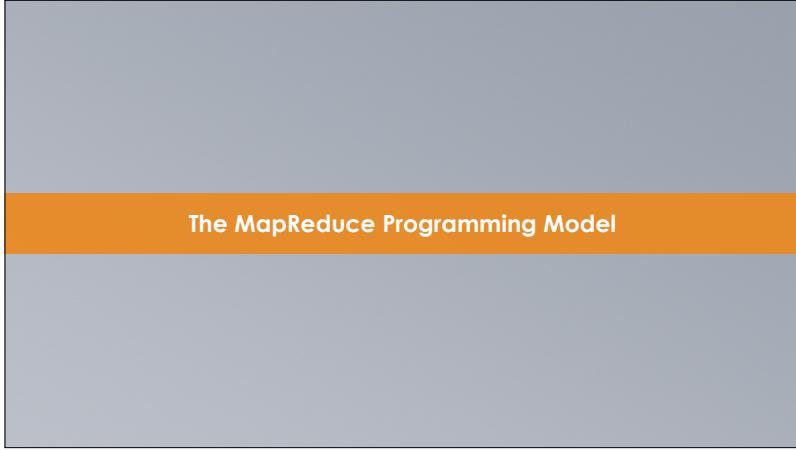
1. Scale out with commodity hardware
2. Spend resources to save resources
3. Send code to where data lives
4. Interpret data only upon use (schema on read)
5. Compute results only as needed (lazy evaluation)

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....



The MapReduce Programming Model

MapReduce.

Now MapReduce has two parts to it....

MapReduce



- Generalization of two operations:
 - *Map* and
 - *Reduce*...

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

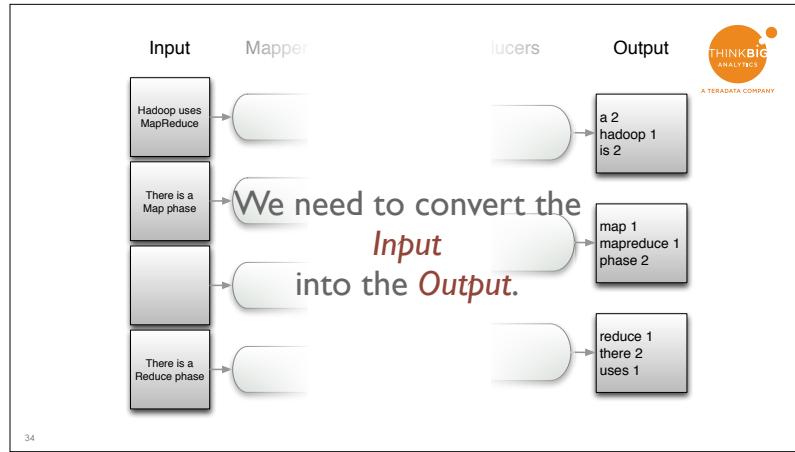
MapReduce in Hadoop



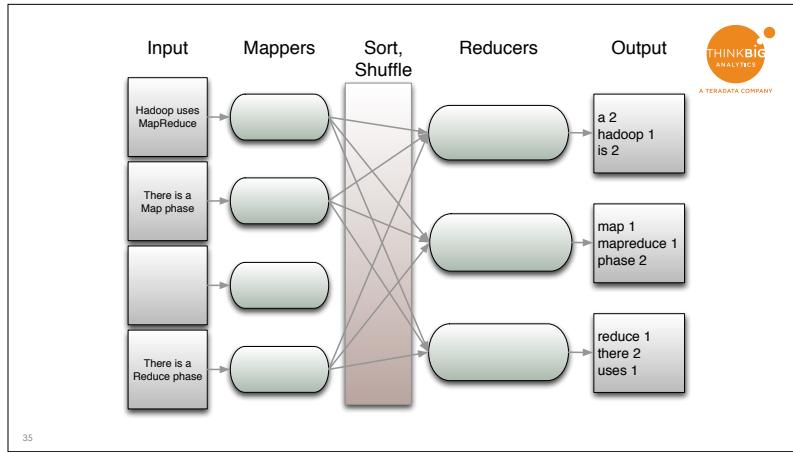
Let's look at how *MapReduce* actually works in *Hadoop*, using *WordCount*.

(The *Hello World* in big data...)

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.

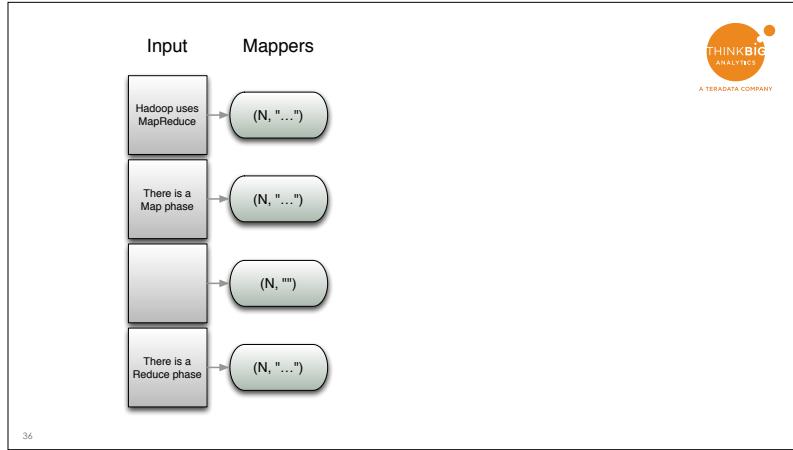


35

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).

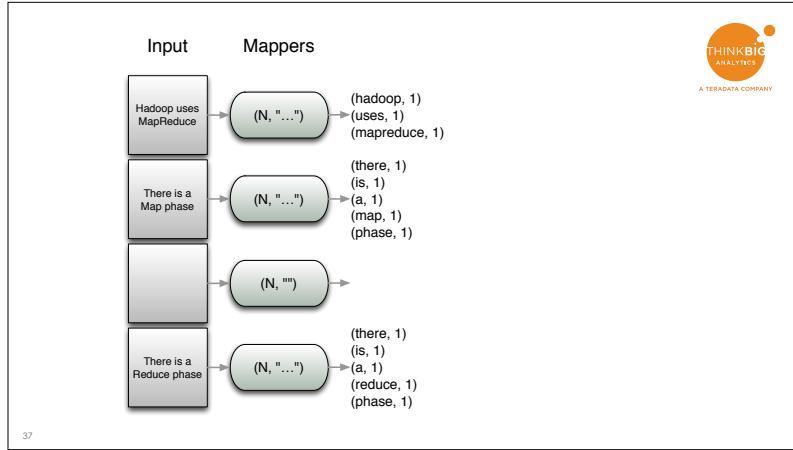
The Mappers emit key-value pairs that will be sorted, then partitioned and “shuffled” to the reducers, where each Reducer will get all instances of a given key (for 1 or more values).

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.

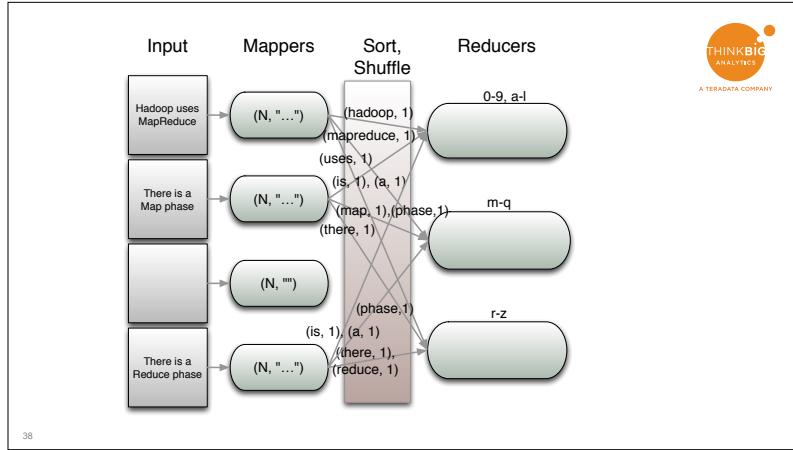
Each mapper will be called repeatedly with key-value pairs, where each key is the position offset into the file for a given line and the value is the line of text. We will ignore the key, tokenize the line of text, convert all words to lower case and count them...



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.

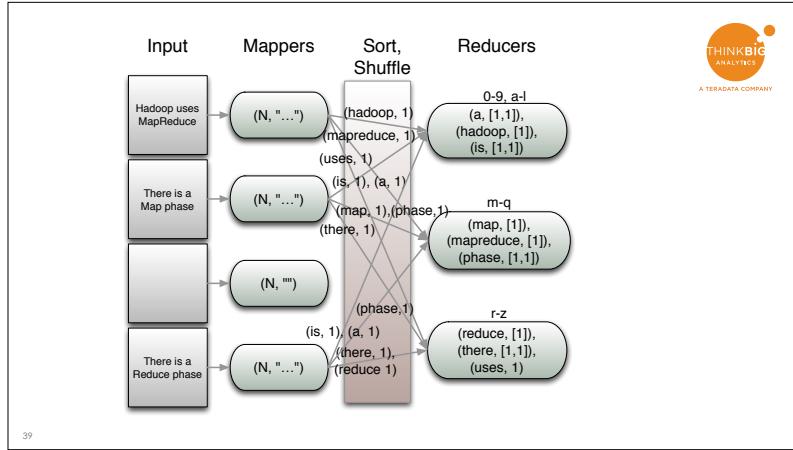
Note how we partitioned the reducers (by first letter of the keys). Also, note that the mapper for the empty doc. emits no pairs, as you would expect.



38

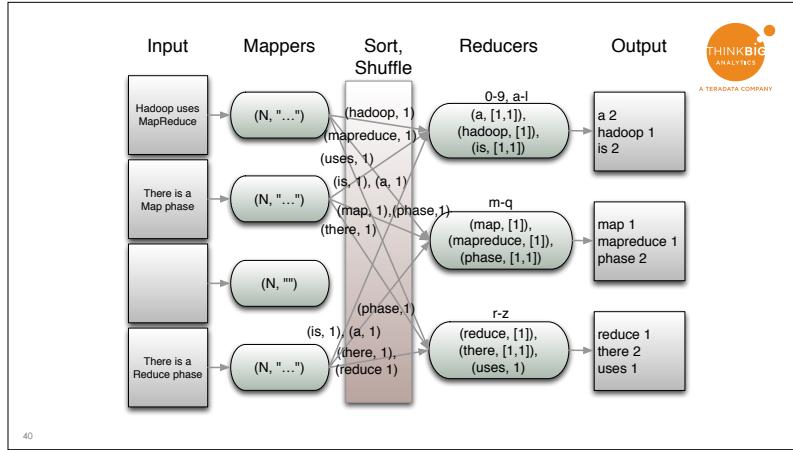
The mappers emit key–value pairs, where each key is one of the words, and the value is the count. In the most naive (but also most memory efficient) implementation, each mapper simply emits (word, 1) each time “word” is seen.

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.



39

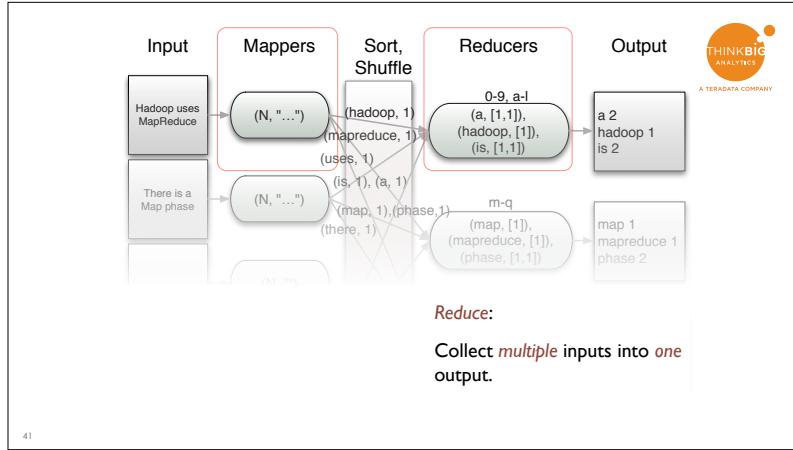
Note how we partitioned the reducers (by first letter of the keys). Also, note that the mapper for the empty doc. emits no pairs, as you would expect.



The final view of the WordCount process flow for our example.

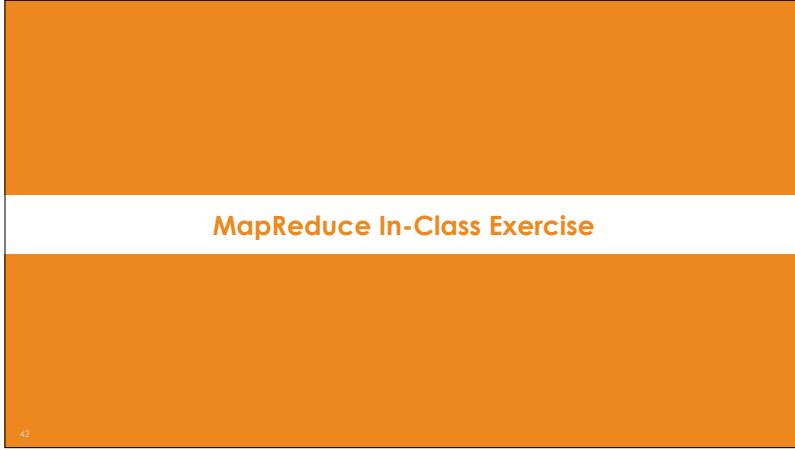
We'll see in more detail shortly how the key–value pairs are passed to the reducers, which add up the counts for each word (key) and then writes the results to the output files.

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?



MapReduce In-Class Exercise

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

Setup



A TERADATA COMPANY

- 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

Exercise Instructions

- Teams perform these three Map/Shuffle-Sort/Reduce steps in order
- Each task must be complete before the next one begins
- When the final reduce task is done, shout out "Done Done!"



1. Map

1. When instructor says "begin", each team leader hands each worker node a single text card to begin. There may be more text cards than workers.
2. For each word in the text, worker nodes should write out a card with the word on it and a one (e.g., **Victory 1**)
3. When the worker node has completed cards for all the text on their input card, they return all their word cards in alphabetical order. This constitutes a **Map**.
4. Any time a worker node returns their cards, hand them the next text card in the stack
5. When all text cards have been handed out and worker nodes are done, shout "**Map Done!**" and move on to Shuffle task.

44

2. Shuffle

1. The leader hands all the Map card decks to a single worker node
2. Worker node merges all the card decks together to create a single new card deck sorted in alphabetical order
3. Once the Shuffle output deck has been sorted, hand the instruction sheet and the output Shuffle deck back to the leader and shouts "**Shuffle Done!**"

3. Reduce

1. The leader splits Shuffle deck into one deck for each worker node and hands each worker node its deck.
2. Worker node draws a card from the input deck and notes the word on it.
3. Worker node draws and counts all cards that have that word on it and notes the total.
4. Worker node writes a new Reduce card containing the word and count (e.g., **Word 3**)
5. If more cards remain in the Shuffle deck, worker node goes to Reduce Step 2. Otherwise it hands the reduce deck back to Job Tracker.
6. When the leader receives all Reduce decks back, shout out "**Done Done!**".

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?

MapReduce in Java



Let's look at *WordCount*
written in the
*MapReduce Java API.**

* This is the MapReduce 1 API usually associated with Hadoop 1.0. MapReduce2 is slightly different.

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

Map Code



A TERADATA COMPANY

```
public class SimpleWordCountMapper
extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
    static final Text word = new Text();
    static final IntWritable one = new IntWritable(1);

    @Override
    public void map(LongWritable key, Text documentContents,
                    OutputCollector<Text, IntWritable> collector, Reporter reporter)
        throws IOException {
        String[] tokens = documentContents.toString().split("\\s+");
        for (String wordString : tokens) {
            if (wordString.length() > 0) {
                word.set(wordString.toLowerCase());
                collector.collect(word, one);
            }
        }
    }
}
```

Let's drill into this code...

47

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).

Map Code

```
public class SimpleWordCountMapper
extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
    static final Text word = new Text();
    static final IntWritable one = new IntWritable(1);

    @Override
    public void map(LongWritable key, Text documentContents,
                    OutputCollector<Text, IntWritable> collector, Reporter reporter)
        throws IOException {
        String[] tokens = documentContents.toString().split("\\s+");
        for (String wordString : tokens) {
            if (wordString.length() > 0) {
                word.set(wordString.toLowerCase());
                collector.collect(word, one);
            }
        }
    }
}
```



Mapper class with 4 type parameters for the input key-value types and output types.

48

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).

Map Code



A TERADATA COMPANY

```
public class SimpleWordCountMapper
extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
    static final Text word = new Text();
    static final IntWritable one = new IntWritable(1);

    @Override
    public void map(LongWritable key, Text documentContents,
                    OutputCollector<Text, IntWritable> collector, Reporter reporter)
        throws IOException {
        String[] tokens = documentContents.toString().split("\\s+");
        for (String wordString : tokens) {
            if (wordString.length() > 0) {
                word.set(wordString.toLowerCase());
                collector.collect(word, one);
            }
        }
    }
}
```

Output key-value objects
we'll reuse.

49

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).

Map Code



```
public class SimpleWordCountMapper
extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
    static final Text word = new Text();
    static final IntWritable one = new IntWritable(1);
    @Override
    public void map(LongWritable key, Text documentContents,
                    OutputCollector<Text, IntWritable> collector, Reporter reporter)
        throws IOException {
        String[] tokens = documentContents.toString().split("\\s+");
        for (String wordString : tokens) {
            if (wordString.length() > 0) {
                word.set(wordString.toLowerCase());
                collector.collect(word, one);
            }
        }
    }
}
```

Map method with input, output "collector", and reporting object.

50

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).

Map Code



A TERADATA COMPANY

```
public class SimpleWordCountMapper
extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
    static final Text word = new Text();
    static final IntWritable one = new IntWritable(1);

    @Override
    public void map(LongWritable key, Text documentContents,
                    OutputCollector<Text, IntWritable> collector, Reporter reporter)
        throws IOException {
        String[] tokens = documentContents.toString().split("\\s+");
        for (String wordString : tokens) {
            if (wordString.length() > 0) {
                word.set(wordString.toLowerCase());
                collector.collect(word, one);
            }
        }
    }
}
```

Tokenize the line, "collect"
each
(word, 1)

51

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).

Reduce Code



A TERADATA COMPANY

```
public class SimpleWordCountReducer
extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {

@Override
public void reduce(Text key, Iterator<IntWritable> counts,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
int count = 0;
while (counts.hasNext()) {
count += counts.next().get();
}
output.collect(key, new IntWritable(count));
}
}
```

Let's drill into this 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.

Reduce Code



A TERADATA COMPANY

```
public class SimpleWordCountReducer  
extends MapReduceBase implements  
Reducer<Text, IntWritable, Text, IntWritable> {  
  
    @Override  
    public void reduce(Text key, Iterator<IntWritable> counts,  
                      OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
        int count = 0;  
        while (counts.hasNext()) {  
            count += counts.next().get();  
        }  
        output.collect(key, new IntWritable(count));  
    }  
}
```

Reducer class with 4 type parameters for the input key-value types and output types.

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



```
public class SimpleWordCountReducer  
extends MapReduceBase implements  
Reducer<Text, IntWritable, Text, IntWritable> {  
  
    @Override  
    public void reduce(Text key, Iterator<IntWritable> counts,  
                      OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
        int count = 0;  
        while (counts.hasNext()) {  
            count += counts.next().get();  
        }  
        output.collect(key, new IntWritable(count));  
    }  
}
```

Reduce method with
input, output, "collector",
and reporting object.

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



A TERADATA COMPANY

```
public class SimpleWordCountReducer
extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {

@Override
public void reduce(Text key, Iterator<IntWritable> counts,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
int count = 0;
while (counts.hasNext()) {
count += counts.next().get();
}
output.collect(key, new IntWritable(count));
}
}
```

Count the counts per word and emit (word, N)

55

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.

Hive



Let's look at WordCount
written in Hive,
the SQL for Hadoop.

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.

Hive Code



A TERADATA COMPANY

```
CREATE EXTERNAL TABLE docs (line STRING)
LOCATION 's3n://thinkbig.academy.aws/data/
shakespeare/input/';

CREATE TABLE word_counts AS SELECT lower(word),
count(1) AS count
FROM
(SELECT explode(split(line, '\\\\W+')) AS word
FROM docs) w
GROUP BY word
ORDER BY count DESC;
```

Let's drill into this code...

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.

Hive Code



A TERADATA COMPANY

```
CREATE EXTERNAL TABLE docs (line STRING)
LOCATION 's3n://thinkbig.academy.aws/data/
shakespeare/input/';
```

```
CREATE TABLE word_counts AS SELECT lower(word),
count(1) AS count
FROM
(SELECT explode(split(line, '\\\\W+')) AS word
FROM docs) w
GROUP BY word
ORDER BY count DESC;
```

58

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.

Hive Code



A TERADATA COMPANY

```
CREATE EXTERNAL TABLE docs (line STRING)
LOCATION 's3n://thinkbig.academy.aws/data/
shakespeare/input/';

CREATE TABLE word_counts AS SELECT lower(word),
count(1) AS count
FROM
(SELECT explode(split(line, '\\\\W+')) AS word
FROM docs) w
GROUP BY word
ORDER BY count DESC;
```

Create the final table and fill it with the results from a nested query of the docs table that performs WordCount on the fly.

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.



A TERADATA COMPANY

Because so many Hadoop *users*
come from *SQL* backgrounds,
Hive is one of the most
essential tools in the ecosystem!!

Hadoop would not have the penetration it has without Hive.



A TERADATA COMPANY

Let's look at *WordCount*
written in *R* for a more concise version

Here's WordCount in R using the rmr2 MapReduce package

This is MapReduce in R version using the
rnr2 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)) )  
}
```

62

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



```
val textFile = sc.textFile("hdfs:///data/shakespeare/input")
val counts = textFile.flatMap(line => line.split(" "))
    .map(word => (word, 1)).
    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://tmp/shakespeare-wc")
```

63

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.

Summary



- Hadoop clusters let us analyze Big Data
- Big Data MapReduce is the original processing model underlying Hadoop
- Applications such as Hive, Pig, and others are allowing more ways to use Hadoop without writing MapReduce code directly

65

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....



**Now that you know what Big Data can do,
what will you do with it?**

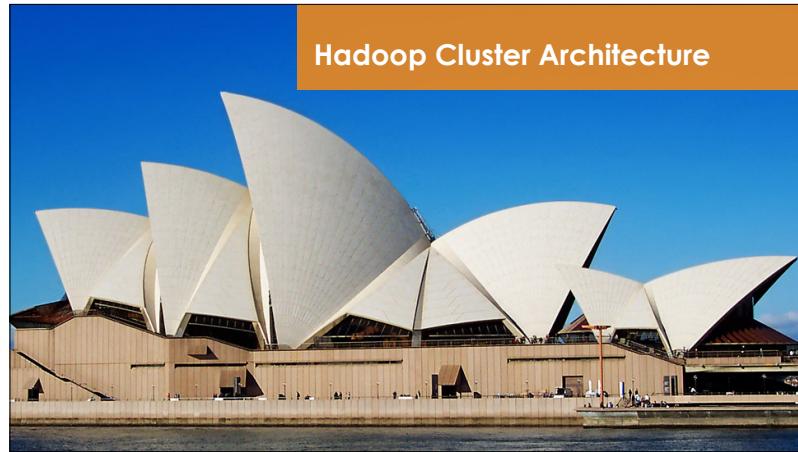
What will you do next?



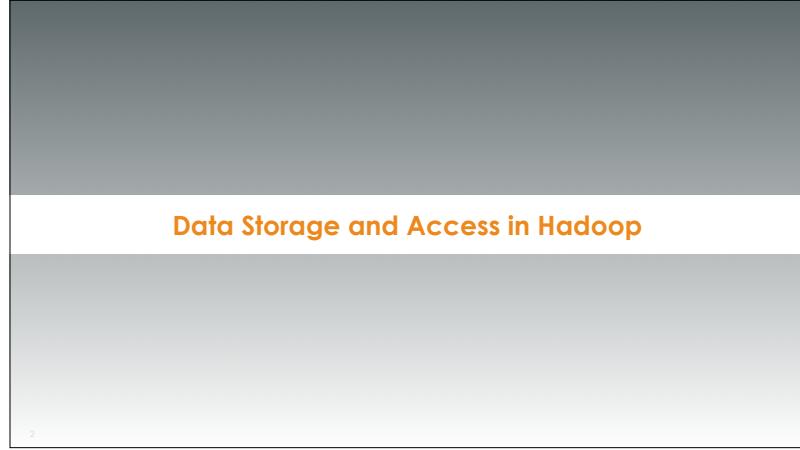
Big Data Science With Spark

Module 2: Hadoop Cluster Architecture

**Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017**



Let's dive into some basic Hadoop architecture



Data Storage and Access in Hadoop

2

We're going to start with Data Storage and Access in Hadoop. We'll begin by asking a question:
What is a cluster?

Hadoop Definitions



A TERADATA COMPANY

A cluster is a
collection of independent servers
connected by a network
managed by software as one system

3

We define a cluster as a collection of independent servers <<click>
connected by a network <<click>>
managed by software as one system <<click>

Now it turns out we can drill down on this definition to define the components of a cluster.

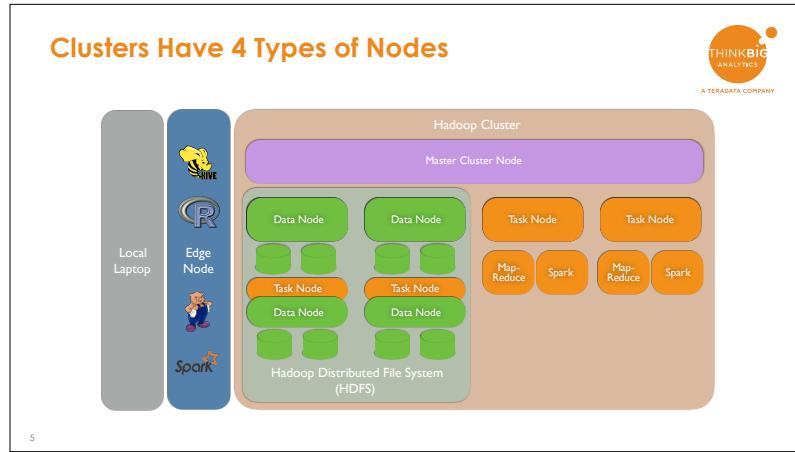
Hadoop Definitions: Nodes	
Term	Definition
node	A generic server with processors, memory, and optionally disks. A collection of these make up a cluster.
master node	The node that directs the rest of the cluster nodes. Typically this node runs programs such as YARN Application Manager and the Name Server.
edge node	A node that connects to both the cluster and data center networks. Typically, this node will run applications such as Hive and Pig. The master node may play the role of an edge node.
worker or core node	All non-master nodes are usually worker nodes that execute Hadoop tasks.
data node	A core node that has disks and participates in HDFS
task node	A core node that runs parallelized application tasks

4

Clusters are made up of nodes, as we just said. So what's a node? It turns out there are many types. Let's walk through each one.

....

A picture will help illustrate this on the next slide.



We start over on the left with whatever local computer you are using. Here's we've called it your local laptop.

That usually connects to an Edge node, which is a node connected to both the cluster and the outer network.

The cluster itself has one or more master nodes, where most of the cluster services run.

The actual work is done by worker nodes of various types, depending on their hardware and the service modules configured on them.

Data nodes have disks attached, while task nodes run executors such as MapReduce and Spark.

Nodes can be both Data and Task nodes at once.

All the data nodes participate in the cluster file system, referred to as the Hadoop Distributed File System.

That's a logical diagram of a cluster. What does a real one look like?



Well here's a good sized cluster. This one resides at Facebook. Facebook has some very large clusters, although Google, Yahoo, LinkedIn and others continually compete for who is running the largest cluster currently.

So let's get you set up with your own clusters.

Getting Started With Hands-On Hadoop



- cd to your exercises/Hadoop-Architecture directory
- Set up your virtual machine and examine the nodes making up your cluster using the directions in
 - 00-Architecture-Setup.md or 00-Architecture-Setup.pdf
 - 01-Architecture-Yarn.md or 01-Architecture-Yarn.pdf
 - 02-Architecture-HDFS.md or 02-Architecture-HDFS.pdf



7

We are now going to perform three labs which will get you used to connecting to your clusters. You should have a directory with all the course materials called sparkclass. Underneath that directory are several other directories including

data
exercises
handouts
images
slides

You'll find these labs in your exercises/Hadoop-Architecture directory.

You can read these labs in either pdf or Markdown form. The PDF form is prettier, but the text in the Markdown version will be easier to copy and paste when you get to complicated commands.

Go ahead and run those three labs. We'll allot about 15 minutes for you to do this.

HDFS Works By Replication



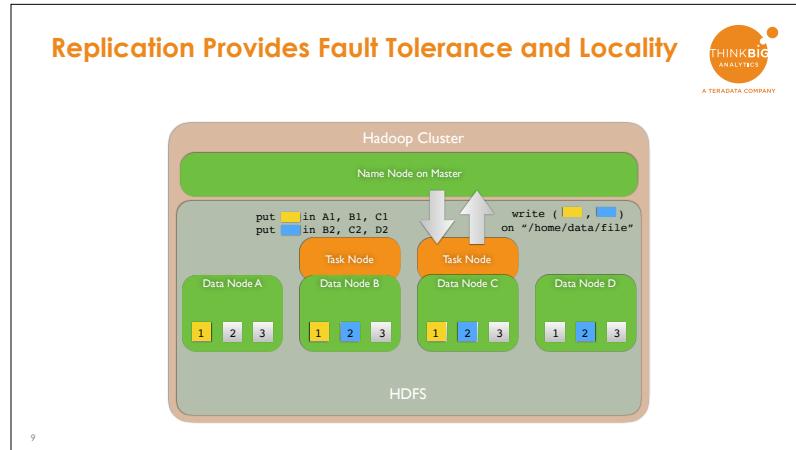
- Hierarchical Unix-like file system for data storage
 - Doesn't have all Unix features
- Splits large files into 64MB or greater blocks
- Two fundamental services
 - Name node for handling names
 - Data nodes for serving blocks
- Blocks are replicated across servers and racks

8

At this point, you'll now have some experience with Yarn and HDFS. Let's tackle what HDFS is about.

On this last bullet, this is an illustration of spending resources to save resources. We replicate blocks not only for reliability, but also for speed of access. It's going to be faster for a node to access a data block the node requesting it is close to the node with the data. As a result, you can gain performance if you replicate your data more. Said another way, you can spend resources to save resources and time.

Let's look at how this works.



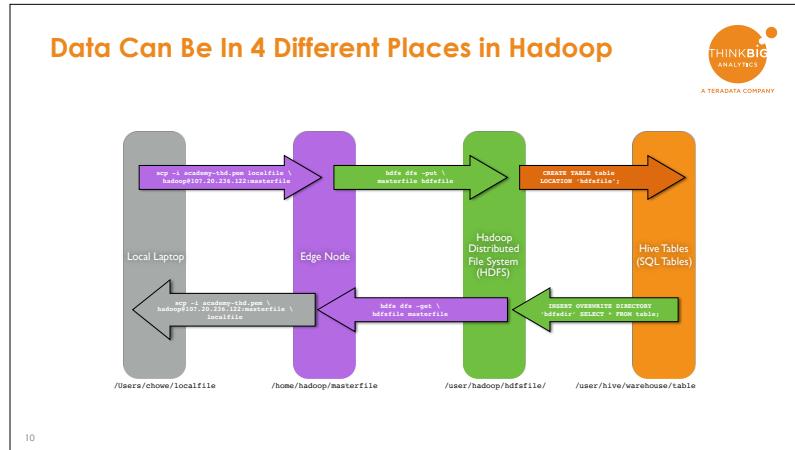
Here we have a five node cluster with one master node <click> and four data nodes.

Over on the right, an application on one of the task nodes asks to write two HDFS blocks, a yellow one and a blue one. That request gets sent to the Name Node. <click>

Unlike in most operating systems, the name node doesn't write that block. Instead, it provides block addresses that specify where the original application should write the blocks. In this case, it says that the yellow block should be written on Data Node A, block 1, Data Node B block 1 and Data Node C block 1. <click>

Meanwhile, the blue block is to be written on Data Node B block 2, Data Node C block 2, and Data Node D block 2.

Once those blocks have been written, we can lose any of the data nodes and still have access to both yellow and blue blocks. In fact, we can lose two nodes and still retain all of the data we wrote.



10

One of the challenges of working with Hadoop is that data can live in 4 different places, all of which look rather similar because they are all Unix-like file systems and have Unix-like names

Data can live on

- your laptop, such as in `/Users/chowe/localfile`
- an Edge node, in a location like `/home/hadoop/masterfile`
- on HDFS in a location like `/user/hadoop/hdfsfile/`
- or in the Hive warehouse in a location like `/user/hive/warehouse/table`

What's difficult is we use different commands to move files among those locations.

1. To move a file from my local laptop to the edge node, I'd type something like `scp -i academy-thd.pem localfile hadoop@107.20.236.122:masterfile` <click>
2. To move a file from the edge node to HDFS, you'd use a command like `hdfs dfs -put masterfile hdfsfile` <click>
3. and finally to move a file from HDFS into the Hive warehouse, we'd use a SQL command like `CREATE TABLE table LOCATION 'hdfsfile';` <click>

We use similar but different commands to bring the data back.

4. To move data from the Hive warehouse to HDFS, we'd do something like `INSERT OVERWRITE DIRECTORY 'hdfsdir' SELECT * FROM table;` <click>

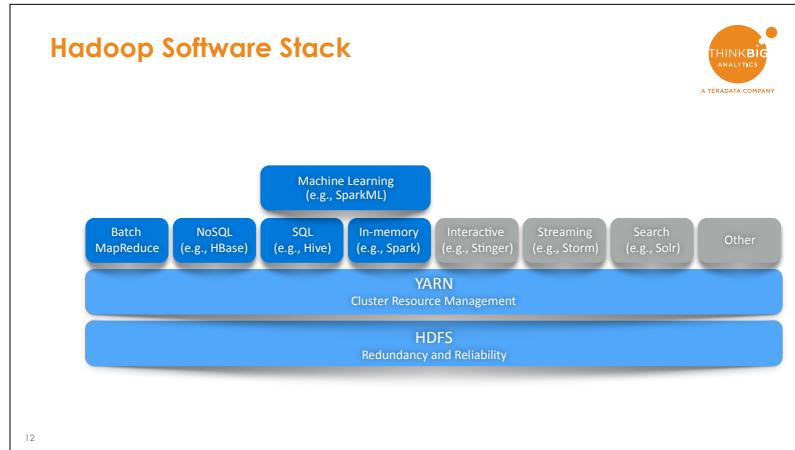
Hadoop Terminal Commands



Shell Command	Meaning
hdfs dfs -ls dest	List files in directory dest
hdfs dfs -put localfile hdfsfile	Copy file from master node local file system to HDFS
hdfs dfs -get hdfsfile localfile	Copy file from HDFS to local file system
hadoop distcp srcURL destURL	Recursively copy the directory specified by srcURL to the directory indicated by destURL using MapReduce
yarn jar jarfile args	Run Java program jarfile with arguments args

11

The first three commands are fairly self-explanatory for listing and moving files. Hadoop distcp is a distributed copy command -- it actually runs a MapReduce job to do the copy, thereby speeding up the movement of large amounts of data by employing the entire cluster to do the moving. Finally, we can have the cluster run Java code by specifying a JAR file to the YARN JAR command.



12

These two components we've discussed in this module—HDFS and YARN—make up the heart of every Hadoop cluster.

HDFS gives us a redundant and reliable data store for big data through replication. <click>
 Yarn provide us with overall cluster resource management, just as any operating system would.
<click>

All the other projects in Hadoop—things like NoSQL databases such as HBase, SQL query engines like Hive, In-memory execution systems like Spark, and even Machine Learning packages like SparkML—run as applications on top of this framework. <click>

Hadoop has more than 50 different application frameworks (and more pop up all the time), so we can't discuss all of them. However, we will discuss the ones in dark blue as part of this course.

Loading Data



A TERADATA COMPANY

- cd to your exercises/Hadoop-Architecture directory
- Explore HDFS and load data into HDFS using the following instructions
 - 03-Architecture-Load-HDFS-Data.md or 03-Architecture-Load-HDFS-Data.pdf



13

Before we start getting into Hadoop applications though, we will need to import our datasets into HDFS. So let's return to our the exercises/Hadoop-Architecture directory and run Lab 03 to load our data into HDFS.

Summary



A TERADATA COMPANY

- A cluster is a collection of independent servers or nodes connected by a network managed by software as one system
- Node types are determined by their function in the cluster
- YARN is the software that runs the cluster
- HDFS provides reliable replicated storage for the cluster

In summary,

A cluster is a
collection of independent servers or nodes
connected by a network
managed by software as one system

Node types are determined by their function in the cluster

YARN is the software that runs the cluster

HDFS provides reliable replicated storage for the cluster



Big Data Science With Spark

Module 3: Spark Hive Fundamentals

Prepared for Elevate

Delivered by Carl Howe, Principal

September 5 through 13, 2017



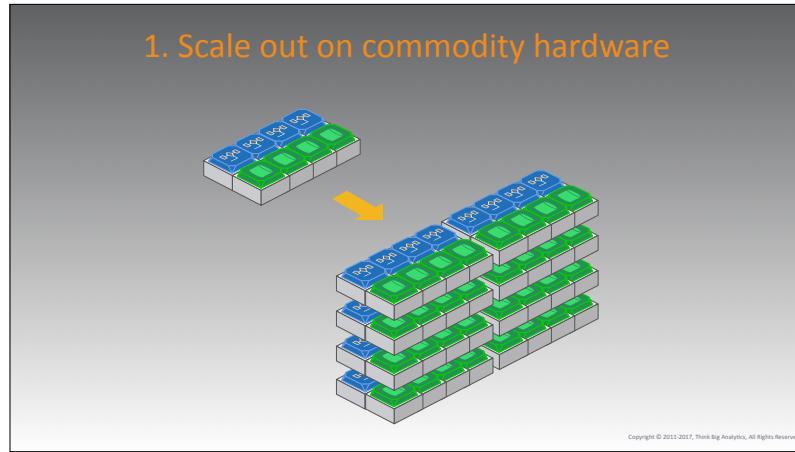
Hive for Spark Developers

Course Agenda



- Hands on Introduction to Hive
- What Hive Is
- Storing Data in Hive
- Probing Data with Hive Queries
- Joining Data and Ordering Output

Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.

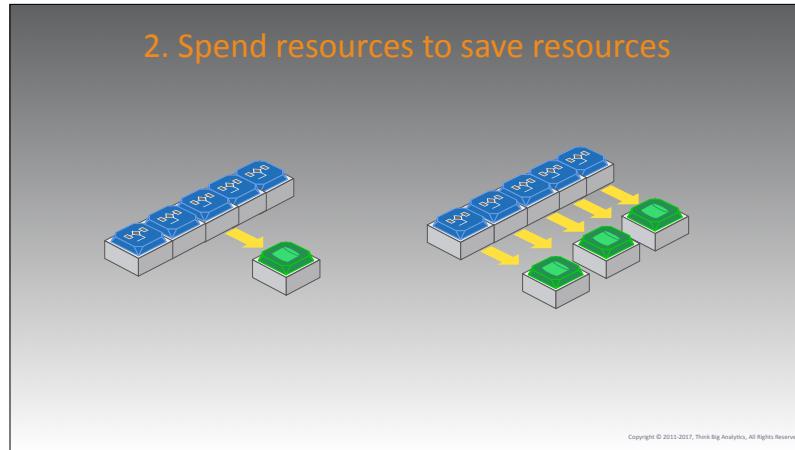


Recall our 5 principles of Hadoop clusters. The first is scale out on commodity hardware.

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.....



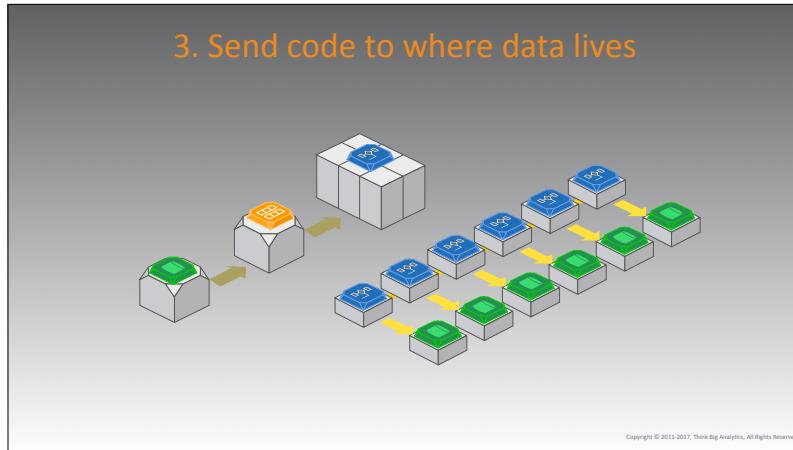
Principle #2 is to spend resources to save resources (usually time)

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....



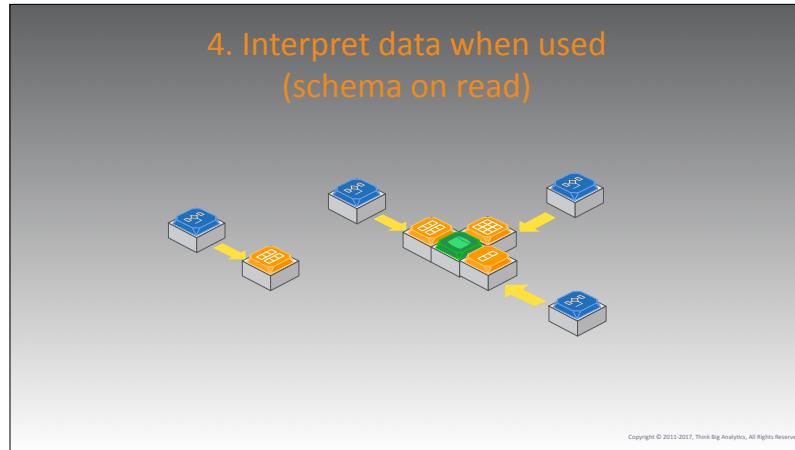
Principle #3: Send code to where the data lives.

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....



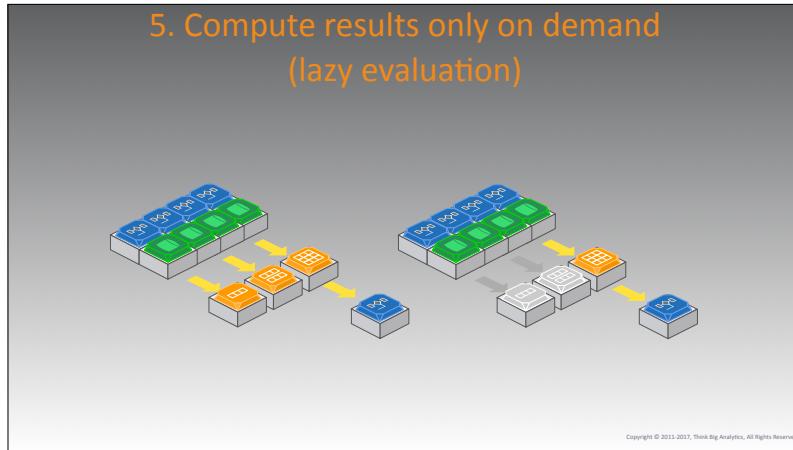
Principal #4 is Interpret Data When Used. This is also called Schema on Read.

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.



What Is *Hive*?

Hive



- A *SQL*-based tool for *data warehousing* using Hadoop clusters.
- Lowers the *barrier* for Hadoop *adoption* for existing SQL apps and users.
- Invented at Facebook.
- Open sourced to Apache in 2008.
- <http://hive.apache.org>

Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.

Hive Elevator Pitch



A TERADATA COMPANY

Hive is a software package that compiles SQL programs into MapReduce jobs. More programmers know SQL than Hadoop; as a result Hive allows more developers to transform big data. Unlike Java and other low-level approaches, Hive automatically generates MapReduce pipelines and speeds up Hadoop development.

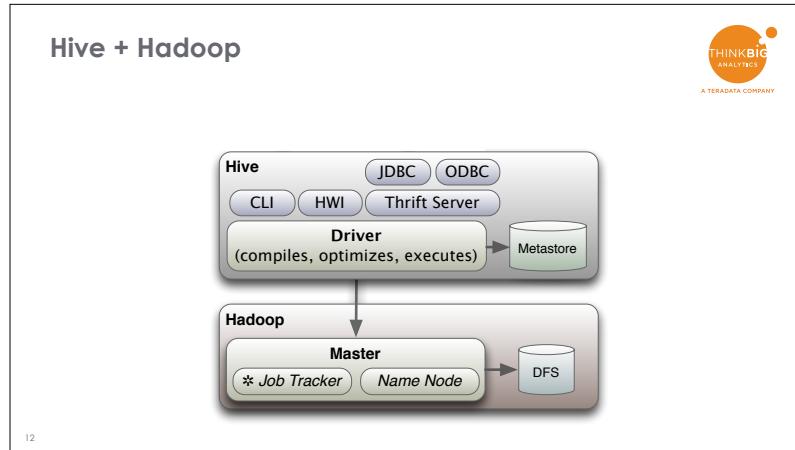
Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.

Repeat after me....



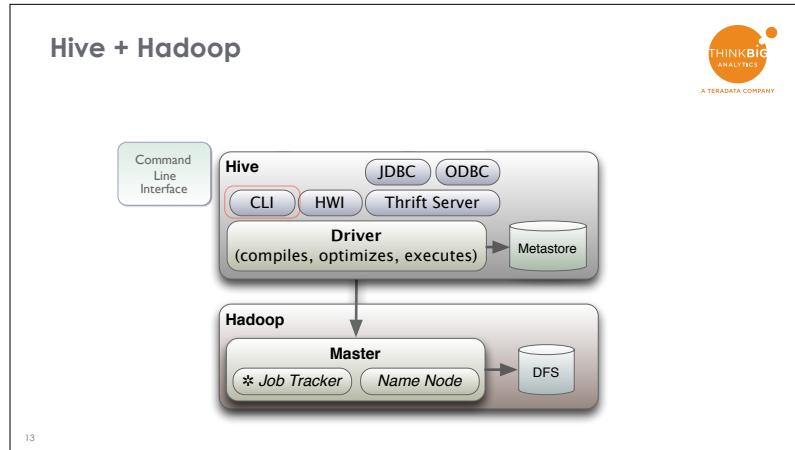
Hive is NOT a database.

Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.



12

Most Hive queries generate MapReduce jobs. (Some operations don't invoke MapReduce, e.g., those that just write updates to the metastore and "select * from table;" queries.) We've omitted some arrows within the Hive bubble for clarity. They go "down", except for the horizontal connection between the driver and the metastore.

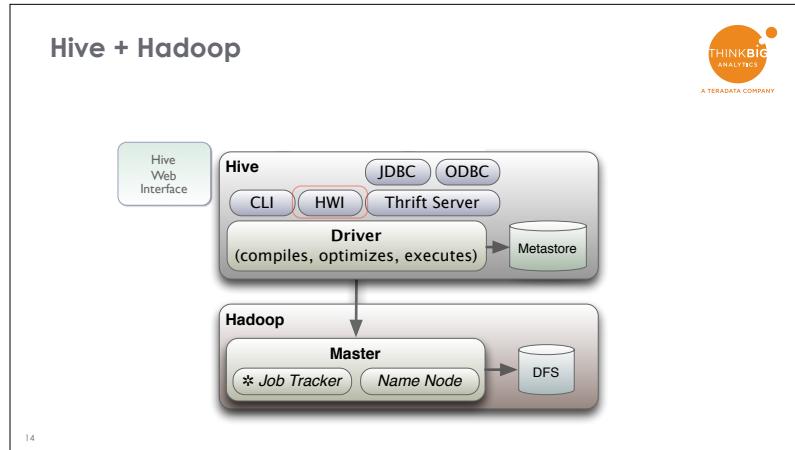


13

CLI = Command Line Interface.

HWI = Hive Web Interface.

We'll discuss the operation of the driver shortly.

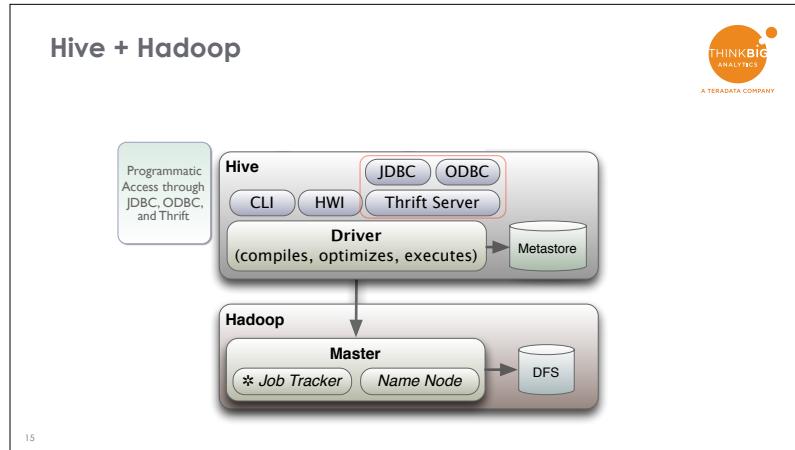


14

CLI = Command Line Interface.

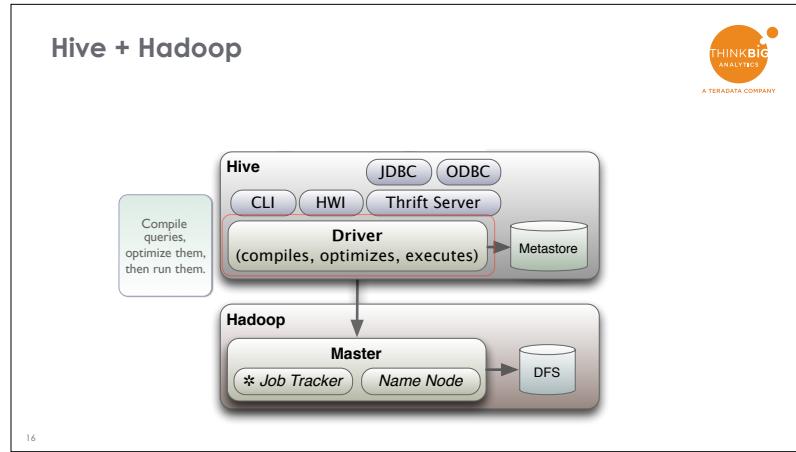
HWI = Hive Web Interface.

We'll discuss the operation of the driver shortly.

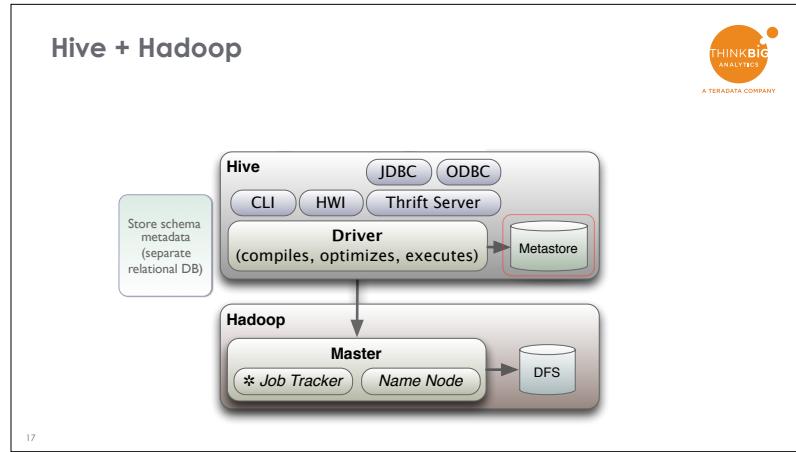


15

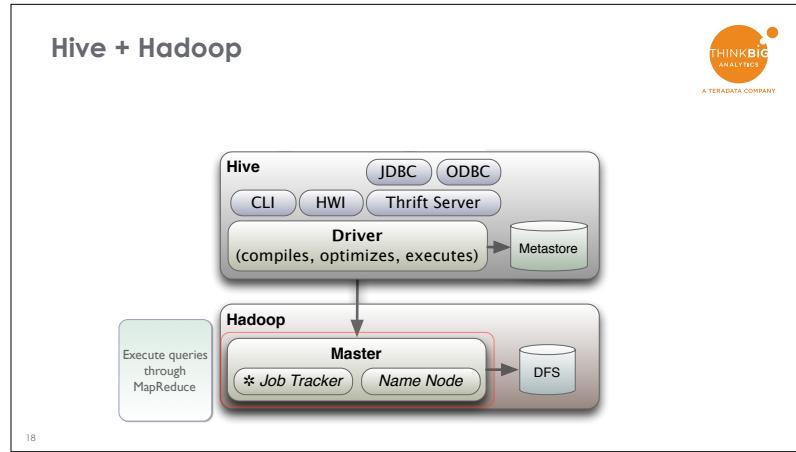
You can also drive Hive from Java programs using JDBC and other languages using ODBC. These interfaces sit on top of a Thrift server, where Thrift is an RPC system invented by Facebook.



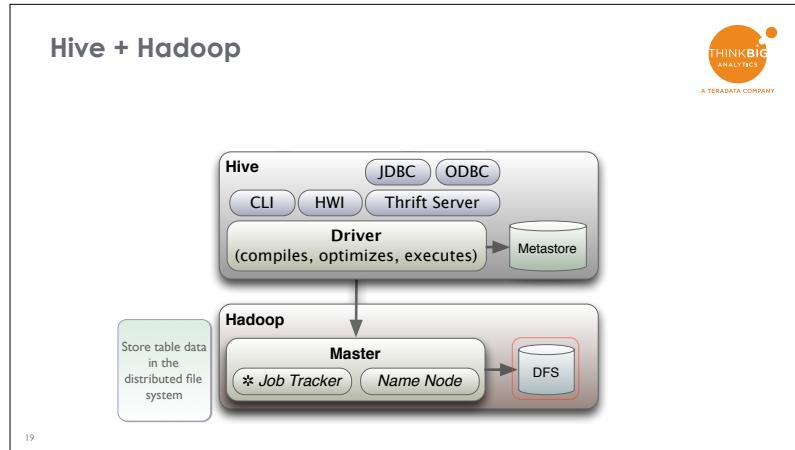
The Driver compiles the queries, optimizes them, and executes them, by *usually* invoking MapReduce jobs, but not always, as we'll see.



A separate relational DB is required to store metadata, such as table schema. We'll discuss it more shortly.



Most Hive queries generate MapReduce jobs. (Some operations don't invoke MapReduce, e.g., those that just write updates to the metastore and "select * from table;" queries.)



Most Hive queries generate MapReduce jobs. (Some operations don't invoke MapReduce, e.g., those that just write updates to the metastore and "select * from table;" queries.) We've omitted some arrows within the Hive bubble for clarity. They go "down", except for the horizontal connection between the driver and the metastore.

Useful Links

Programming Hive



See also the Hive wiki:

<https://cwiki.apache.org/confluence/display/Hive/Home>

This book, written by two former Think Big Analytics consultants, became available in early October 2012. You'll also find lots of documentation the Hive wiki.



Hive Concepts Overview

Databases



- *Databases:*
 - *Namespaces* for *tables*.
 - That's about it...
- *Tables:*
 - Mostly like the SQL tables you know.
 - You define table *schemas* (like *relational* DBs).

There isn't much to the database concept in Hive. You use them keep your "foo" table separate from other people's "foo" tables. Hive tries to give as much of the familiar SQL ideas as it can, especially for tables. We'll see what's the same and what's different.

Table Storage



A TERADATA COMPANY

- Hive can *own and manage* the storage files...
- Or *you can* own and manage them!
 - They are *external* to Hive.
 - Better for sharing with other tools.
- *File, record, and field* formats are configurable.

Batch vs. Interactive?



A TERADATA COMPANY

- Traditional databases are good for OLTP (online transaction processing) use.
- Because Hive uses MapReduce, queries have high latency, often minutes, better for:
- Batch mode workflows.
- Data warehouse applications.
- General OLAP (online analytical processing).

Metastore



- Hive keeps *metadata* about tables, databases, etc. in the *metastore*, a separate *relational* database.
- EMR defaults to using a *MySQL* instance in the cluster, so it's not *persistent*.

By default, when you start an cluster for a job flow, a MySQL instance is created, but it gets deleted when the cluster is terminated.

Metastore



A TERADATA COMPANY

- To retain your metadata, provision a permanent store using [RDS](#) (Relational Data Service), an AWS cloud-based [MySQL](#).
- See [Creating a Metastore Outside the Hadoop Cluster](#) in the [Developer Guide](#).

Using RDS lets you set up a truly persistent store! The “Creating a Metastore Outside the Hadoop Cluster” section (as linked in the API version 2009-11-30 document) in the Developer Guide provides the details for setting this up.



Lab: Hive-Walkthrough

exercises/Hive-Walkthrough



A TERADATA COMPANY

- Where Hive is installed and configured.
- How to run the Hive services, esp. the CLI (command-line interface).
- Basics of how to create, alter and delete tables.
- How to do simple queries.

Walk through the code.



A TERADATA COMPANY

Most of the details today will be found in the *Hive Cheat Sheet* handout and the exercises' *.hql files.

These notes will provide high-level summaries, but we'll spend most of our time working out these other documents, which more easily contain complex HiveQL statements and which are easier to copy from for pasting to the hive CLI.

Course Agenda



- Hands on Introduction to Hive
- **Representing Data in Hive**
- Probing Data with Hive Queries
- Joining Data and Ordering Output
- User Defined Functions
- File and Record Formats
- Calling Outside Programs
- Using Explain
- Conclusion



Hive Databases

We'll start with databases (schemas), then spend most of our time on tables. A later module will briefly discuss indexes and views, two very new features. We'll cover those later.

Creating Databases



A TERADATA COMPANY

- Syntax:

```
CREATE DATABASE IF NOT EXISTS orders
COMMENT 'Tracks user orders';
```

- SCHEMA is an alias for DATABASE.
- COMMENT is optional.
- A DATABASE is effectively just a namespace.
- There is a default database if none used.

I'm omitting some of the more obscure options, including a few that are documented but don't appear to actually work!

Namespace in the sense that we can create tables like “create table db1.table1 ...;”

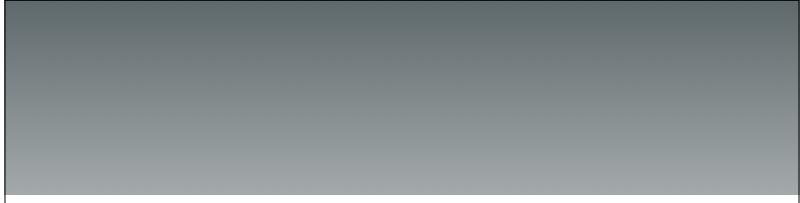
Dropping Databases



- Syntax:

```
DROP DATABASE IF EXISTS orders;
```

There are RESTRICT | CASCADE options shown in the language manual that cause parse errors!



Creating Hive Tables, Part I

[https://cwiki.apache.org/confluence/display/Hive/
LanguageManual+DDL#LanguageManualDDL-CreateTable](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-CreateTable)

Creating **Managed** Tables



A TERADATA COMPANY

- Example:

```
CREATE TABLE IF NOT EXISTS demo1 (
    id INT, name STRING);
```

- IF NOT EXISTS optional.
- Suppresses the warning if the table already exists.

Creating **Managed** Tables



- By default, the data is *stored* under the *root* directory specified by the property:
 - `hive.metastore.warehouse.dir`
- For *Apache Hadoop*, the property defaults to:
`/user/hive/warehouse/`

“`hive.metastore.warehouse.dir`” is a property you can set. This directory is in the distributed file system. You can run `hadoop fs -ls` on this directory.

Creating *Internal* Tables



- Example:

```
CREATE TABLE IF NOT EXISTS demo1 (
    id INT, name STRING);
```

- So, `demo1` under database `mydb` is created in:

`/user/hive/warehouse/mydb.db/demo1`

Note that the directory for the table is under the directory for the database!

Dropping Tables



- Syntax:

```
DROP TABLE IF EXISTS demo1;
```

- The data *is deleted*...
- (unless it's an *external* table, where data is not managed by Hive, as we'll see...).

As we'll see, external tables are not managed by Hive, but managed by you, "external" to Hive.



Lab: Hive-Tables1

exercises/Hive-Tables1



A TERADATA COMPANY

- Create a table.
- Load data into it.
- Run queries and experiment with them.
- See how to drop the table.

Walk through the code.

Hive Schemas (and File Encodings)

<https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-TypeSystem>

41

Specifying Table Schemas



- Example:

```
CREATE TABLE IF NOT EXISTS demo1  
  (id INT, name STRING);
```

- Two columns:

- One of type INT.
- One of type STRING.

Simple Data Types



A TERADATA COMPANY

TINYINT, SMALLINT, INT, BIGINT	1, 2, 4, and 8 byte integers
FLOAT, DOUBLE	4 byte (single precision), and 8 byte (double precision) floating point numbers
BOOLEAN	Boolean
STRING	Arbitrary-length String
TIMESTAMP	(v0.8.0) Date string: "yyyy-mm-dd hh:mm:ss.fffffffff" (The "f"s are nanosecs.)
BINARY	(v0.8.0) a limited VARBINARY type

43

All the types reflect underlying Java types. TIMESTAMP and BINARY are new to v0.8.0. Use an a string for pre-0.8.0 timestamps (or BIGINT for Unix epoch seconds, etc.). BINARY has limited support for representing VARBINARY objects. Note that this isn't a BLOB type, because those are stored separately, while BINARY data is stored within the record.

Complex Data Types



A TERADATA COMPANY

ARRAY	Indexable list of items of the same type. Indices start at 0: <code>orders[0]</code>
MAP	Keys and corresponding values. <code>address['city']</code>
STRUCT	Like C-struct or Java object. <code>name.first, name.last</code>

44

All the types reflect underlying Java types. TIMESTAMP and BINARY are new to v0.8.0. Use an integer type or strings for pre-0.8.0. Use BINARY as the last “column” in a schema to as a way of saying “ignore the rest of this record”.

Complex Schema



A TERADATA COMPANY

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
    city:STRING, state:STRING, zip:INT>);
```

- Uses Java-style “generics” syntax.
 - ARRAY<STRING>
 - MAP<STRING, FLOAT>
 - STRUCT<street:STRING, ...>

45

The <...> is a Java convention. We have to say what type of things the complex values hold. Note that we also name the elements of the STRUCT.

Complex Schema



A TERADATA COMPANY

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT, name and salary
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
        city:STRING, state:STRING, zip:INT>
);
```

46

Let's walk through this...

Complex Schema



A TERADATA COMPANY

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
        city:STRING, state:STRING, zip:INT>
);
```

Arrays of the
same type

47

Let's walk through this...

Complex Schema



A TERADATA COMPANY

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
    city:STRING, state:STRING, zip:INT>
);
```

Paycheck
deductions:
(name, %)

48

Let's walk through this...

Complex Schema



A TERADATA COMPANY

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
    city:STRING, state:STRING, zip:INT>
);
```

Home address

49

Let's walk through this...

Normal Form??



```
subordinates ARRAY<STRING>,
deductions MAP<STRING, FLOAT>,
address STRUCT<street:STRING,
           city:STRING, state:STRING, zip:INT>
```

- We're trading away the benefits of normal form for faster access to all data at once.
- *Very valuable in Big Data systems.*
- <http://queue.acm.org/detail.cfm?id=1563874>

The link is to an ACM Queue article that explains the performance issues with RDBMSs for big data sets and how techniques like denormalizing data help address the performance issues. RDBMSs don't use complex structures. Instead preferring separate tables and using joins with foreign keys. This is dramatically slower than a straight disk scan, but normal form does optimize space utilization.

Storage Formats



A TERADATA COMPANY

- So far, we've used a *plain-text file format*.
- Let's explore its properties.
- We'll see other formats later.

Terminators (Delimiters)



'\n'	Between rows (records)
^A ('\001')	Between fields (columns)
^B ('\002')	Between ARRAY and STRUCT elements and MAP key-value pairs
^C ('\003')	Between each MAP key and value

52

Hive uses the term “terminators” in table definitions, but they are really delimiters or separators between “things”.

“^A” means “control-A”. The corresponding ‘\001’ is the “octal code” for how you write the control character in CREATE TABLE statements.

Specifying Encodings



A TERADATA COMPANY

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
    city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

All the
defaults shown
explicitly!

Keywords you
can't use as
column names!

53

Our original employees table, now written to show all the default values used for the terminators and the fact that it's stored as a text file.

The Actual File Format



A TERADATA COMPANY

```
John Doe^A100000.0^AMary Smith^BTodd  
Jones^AFederal Taxes^C.2^BState Taxes^C.  
05^BInsurance^C.1^A1 Michigan  
Ave.^BChicago^BIL^B60600  
...
```

One line of text.

```
CREATE TABLE employees (  
    name STRING,  
    salary FLOAT,  
    subordinates ARRAY<STRING>,  
    deductions MAP<STRING, FLOAT>,  
    address STRUCT<street:STRING,  
    city:STRING, state:STRING, zip:INT>)
```

The schema, for comparison

This is what's actually stored in the file. The first line, a single record, of the file we'll use is shown here, with all the default delimiters.

How Are Schemas Used?



- In *Relational DBs*, the schema is enforced when data is *loaded* into the table.
 - Called *schema on write*
- Hive can only enforce the schema at *read* (query) time, not *load* time.
 - (Exception is LOAD/INSERT... for *local* tables)
 - Called *schema on read*

55

Traditional relational DBs enforce the schema as data is loaded into tables. This makes queries faster and loads slower. Since Hive has less control over the table contents (e.g., “external” tables), it waits to very the schema when you actually query the data. This slows reads a bit, but it also gives you a lot more flexibility to interpret file contents as you see fit, even using different “interpretations” (schema) at different times for the same data.

We'll explain loading/inserting data into internal tables shortly.



Lab: Hive-Schemas

External Tables



A TERADATA COMPANY

- When you *manage* the data *yourself*:
 - The data is used by other tools.
 - You have a custom ETL process.
 - You can also customize the file format.

```
CREATE EXTERNAL TABLE stocks (
    ymd STRING, ...)
...
LOCATION '/data/stocks';
```

57

Note: ymd = year-month-day. If you use ‘date’ as the name here, you get a compilation error. However, you can use ‘date’ in other places; see the Hive tutorial on the apache site. The LOCATION will be in the cluster, unless you’re running in Hadoop’s local/standalone mode.

Creating *External* Tables



- Example for plain text files:

```
CREATE EXTERNAL TABLE shakespeare_wc (
    word STRING,
    count INT)
ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t'
LOCATION '/data/shakespeare_wc/input';
```

No scheme prefix, e.g., `hdfs://server...`
So, defaults to directory in the cluster.
We own and manage that directory.

Recall that previously we defined an identical `shakespeare_wc` table that was internal and we had to subsequently `LOAD` the data into the table. If we already have the data in our cluster, why duplicate it?!

Note that `LOCATION` is a directory. Hive will just read all the files underneath.

Creating *External* Tables



A TERADATA COMPANY

- The locations can be *local*, in *HDFS*, or in *S3*.
- *Joins* can join table data from *any* such source!

```
...  
LOCATION 'file:///path/to/data';...  
...  
LOCATION 'hdfs://server:port/path/to/data';  
...  
LOCATION 's3n://mybucket/path/to/data';
```

The URI's *scheme*.

So, you might have a table pointing to “hot” data in HDFS, a table pointing to a local temporary file created by an ETL staging process, and some longer-lived data in S3 and do joins on all of them!

Dropping External Tables



A TERADATA COMPANY

- Because you *manage* the data *yourself*:
- The table *contents are not deleted* when you drop the table.
- The table *metadata is deleted* from the *metastore*.

Partitioning



A TERADATA COMPANY

- Way to improve query *performance*.
- A tool for data *organization*.

Partitioning in Hive is similar to partitioning in many DB systems.

Partitions



- Separate *directories* for each partition *column*.

```
CREATE TABLE message_log (
    status STRING, msg STRING, hms STRING)
PARTITIONED BY (
    year INT, month INT, day INT);
On disk:
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...
```

62

This is an INTERNAL table and the directory structure shown will be in Hive's warehouse cluster directory. The actual directories, e.g., .../year=2012/month=01/day=01 (yes, that's the naming scheme), will be created when we load the data, discussed in the next module.

(Note that "hms" is the remaining hours-minutes-seconds...)

Partitions



A TERADATA COMPANY

- Speed *queries* by limiting scans to the correct partitions specified in the WHERE clause.

```
SELECT * FROM message_log  
WHERE year = 2012 AND  
      month = 1 AND  
      day   = 31;
```

In SELECT and WHERE clauses, you use the partitions just like ordinary columns, but they significant performance implications.

Without “Partition Filtering”



A TERADATA COMPANY

```
SELECT * FROM message_log;
```

ALL these directories
are read.

```
...
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...

```

64

Without a WHERE clause that limits the result set, Hive has to read the files in EVERY DIRECTORY ever created for “message_log”. Sometimes, that’s what you want, but the point is that often, you’re likely to do queries between time ranges, so scanning all the data is wasteful.

Filtering by Year



A TERADATA COMPANY

```
SELECT * FROM message_log  
WHERE year = 2012;
```

Just 366 directories
are read.

```
...  
message_log/year=2011/month=12/day=31/  
message_log/year=2012/month=1/day=1/  
...  
message_log/year=2012/month=1/day=31/  
message_log/year=2012/month=2/day=1/  
...
```

65

If you filter by year, you have to read only 365 or 366 directories, that is the days under the months which are under the year.

Filtering by Month



```
SELECT * FROM message_log  
WHERE year = 2012 AND  
      month = 1;  
...  
message_log/year=2011/month=12/day=31/  
message_log/year=2012/month=1/day=1/  
...  
message_log/year=2012/month=1/day=31/  
message_log/year=2012/month=2/day=1/  
...
```

Just 3 directories
are read.

66

If you filter by month, you need to read only those directories for that month, such as 31 directories for January.

Filtering by Day



```
SELECT * FROM message_log
WHERE year = 2012 AND
      month = 1 AND
      day   = 31;
...
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...
```

Just one directory
is read.

67

Finally, if you filter by all three, year, month, and day, Hive only has to read one directory!

The point is that partitions drastically reduce the amount of data Hive has to scan through, but it's only useful if you pick a partitioning scheme that represents common WHERE clause filtering, like date ranges in this example.

External Tables & Partitions



A TERADATA COMPANY

- Still use separate *directories* for each partition.
- ... but the directory path doesn't have the same format constraints.

```
CREATE [EXTERNAL] TABLE stocks (
    ymd STRING, closing_price FLOAT, ...)
PARTITIONED BY (
    exchg STRING, symbol STRING);
```

68

This is an INTERNAL table and the directory structure shown will be in Hive's warehouse cluster directory. The actual directories, e.g., .../year=2012/month=01/day=01, will be created when we load the data, discussed in the next module.

Adding Partitions



- For *external* tables, use ALTER TABLE to specify a *location*.

```
ALTER TABLE stocks
ADD IF NOT EXISTS PARTITION (
    exchg = 'NASDAQ',
    symbol = 'AAPL')
LOCATION '/data/stocks/nasdaq/aapl';
```

- Hive queries will *ignore* a partition if the directory doesn't *exist* yet.

69

We'll see more about ALTER TABLE later.

Technically, we don't have to nest "AAPL" under "NASDAQ", but it's generally a good idea to reflect the data hierarchy this way. (It might improve performance for table scans over multiple "symbol" values, including all of them.)

Ignoring missing partitions is convenient. If you have no data, there's no need for empty directories and/or files. Also, if it's easier, you can create a lot of "future" partitions before you actually have data for them (like a month's worth of `message_log` partitions).



Lab: Hive-Tables2

exercises/Hive-Tables2



A TERADATA COMPANY

- Create an *external* and *partitioned* stocks table.
- List the *partitions*.
- We'll populate the data in the next exercise.

Loading Data Into Tables

[https://cwiki.apache.org/confluence/display/Hive/
LanguageManual+DML](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML)

Loading Data



A TERADATA COMPANY

- Use LOAD to load data from a *file* or *directory*:

```
LOAD DATA LOCAL INPATH '/logs-20120131'  
OVERWRITE INTO TABLE logs  
PARTITION (year=2012, month=1, day=31);
```

- The distributed file system is assumed for the path *unless LOCAL* used.
- Data is *appended unless OVERWRITE* used.
- PARTITION is required if the table uses them.

Loading Data



A TERADATA COMPANY

- Use LOAD to load data from a *file* or *directory*:

```
LOAD DATA LOCAL INPATH '/logs-20120131'  
OVERWRITE INTO TABLE logs  
PARTITION (year=2012, month=1, day=31);
```

- When you use LOCAL, the data is *copied*.
- When you *don't* use LOCAL, the data is *moved*.

Loading Data



- You can use full URLs:

```
LOAD DATA INPATH  
  'hdfs://server/logs-20120131' ...;
```

```
LOAD DATA LOCAL INPATH  
  'file:///logs-20120131' ...;
```

- But `s3n://` URLs aren't supported.

Inserting Data



A TERADATA COMPANY

- Use INSERT to load data from a query:

```
INSERT OVERWRITE TABLE logs
PARTITION (year=2012, month=1, day=31)
SELECT * FROM staged_logs
WHERE year=2012 AND month=1 AND day=31;
```

- Overwrites data when OVERWRITE is used.

Imagine we've staged log data into one table and now we're using this scheme to put into the final, partitioned table.

Inserting Data



- Use `INSERT` to load data from a *query*:

```
INSERT INTO TABLE logs
PARTITION (year=2012, month=1, day=31)
SELECT * FROM staged_logs
WHERE year=2012 AND month=1 AND day=31;
```

- **Appends** data if `INTO` is used (instead of `OVERWRITE`)

Inserting Data



A TERADATA COMPANY

- Use INSERT to load data from a *query*:

```
INSERT INTO TABLE logs
PARTITION (year=2012, month=1, day=31)
SELECT * FROM staged_logs
WHERE year=2012 AND month=1 AND day=31;
```

- Requires PARTITION if the table is partitioned.

Inserting Data



A TERADATA COMPANY

- What if the table is EXTERNAL?
 - The partitions will be written to

```
 ${hive.metastore.warehouse.dir}/
 mydb.db/table/partition=value
```
 - Just like a managed table!
 - *Unless* you create the partitions *first* with ALTER TABLE ...

So, a partitioning query can be used to populate an external table, but if you don't want the default warehouse location used for the data, you have to create the partitions in advance! We'll come back to this with an example shortly.

Inserting Data



- Another syntax supporting *multiple inserts*:

```
FROM staged_logs
INSERT OVERWRITE TABLE logs
PARTITION (year=2012, month=1, day=31)
SELECT *
WHERE year=2012 AND month=1 AND day=31
INSERT OVERWRITE TABLE logs
PARTITION (year=2012, month=2, day=1)
SELECT *
WHERE year=2012 AND month=2 AND day=1;
```

- That could mean a *lot* of INSERT clauses...

Very useful technique for starting with one table and extracting data through SELECTS that get written to many tables or partitions. It also scans “staged_orders” ONCE, which is a performance improvement if this table is huge!

Inserting Data



- *Dynamic partition inserts:*

```
INSERT OVERWRITE TABLE logs
PARTITION (year, month, day)
SELECT ..., year, month, day
FROM staged_logs;
```

- *Last* fields in SELECT must be partition keys.
- ... and must be in the same order.

The data is partitioned “dynamically” based on the values for the select fields. The partitions are created by Hive automatically, based on the partition key values.

Inserting Data



A TERADATA COMPANY

- *Dynamic partition inserts:*

```
INSERT OVERWRITE TABLE logs
PARTITION (year, month, day)
SELECT ..., year, month, day
FROM staged_logs;
```

Requires these properties to be set:

```
set hive.exec.dynamic.partition=true;
set hive.exec.dynamic.partition.mode=nonstrict;
```

82

You have to enable these properties first.

Note: There are additional properties to fine tune the allowed number of partitions, resource utilization in data nodes, etc. See the Tuning chapter of Programming Hive for more details.

Inserting Data



A TERADATA COMPANY

- Mixed *Static* and *Dynamic partition inserts*:

```
INSERT OVERWRITE TABLE logs
PARTITION (year=2012, month, day)
SELECT ..., year, month, day
FROM staged_logs
WHERE year = 2012;
```

- The *dynamic* partition must come *last*.

You can mix static partitions, where you specify a value, and dynamic. The static ones must come before the dynamic partitions.

Inserting Data



- *Create* the partitions *first* for an *external* table.

```
CREATE EXTERNAL TABLE logs (...)  
ALTER TABLE logs  
ADD PARTITION (year=2012, month=1, day=31);  
...  
INSERT INTO TABLE logs  
PARTITION (year, month, day)  
SELECT ..., year, month, day  
FROM staged_logs;
```

Inserting Data while Creating a New Table



- Use AS SELECT in CREATE TABLE:

```
CREATE TABLE aapl
AS SELECT ymd, price_open, price_close
FROM stocks
WHERE symbol = 'AAPL';
```

- The new column names taken from the SELECT statement.
- Can't be used for external tables.

Even though you can't assign new column names, you have a few options, 1) you can use ALTER TABLE to change the names afterwards, 2) you can give the name "aliases" within the SELECT clause.

“Loading” Data into *External* Tables



- Because you own the files, simply adding or replacing files in the table directory “loads” new data.

```
ALTER TABLE logs2
ADD PARTITION (
    year=2012, month=1, day=31)
LOCATION '/logs2/2012/01/31';
```

We'll discuss altering tables in more detail next.
Similarly, if you delete files, the data is deleted.

Writing to Directories



A TERADATA COMPANY

- Use `INSERT ... DIRECTORY` from a *query*:

```
INSERT OVERWRITE LOCAL  
DIRECTORY '/tmp/results'  
SELECT ... FROM ...;
```

- Query results written to one or more *files*.
- *Appends* to the dir. unless `OVERWRITE` is used.
- Writes to the HDFS unless `LOCAL` is used.

If you want the **WHOLE** table, you can just copy the file in the cluster, but if you want a query result, this is the technique. It can write to a new cluster directory or to a local filesystem directory.

The Hive documentation for other forms of this statement.



Lab: Hive-LoadingData

exercises/Hive-LoadingData



A TERADATA COMPANY

- Load data into the `stocks` table created in the previous exercise.
- Insert data while creating a new table.
- Save query results to the local file system.

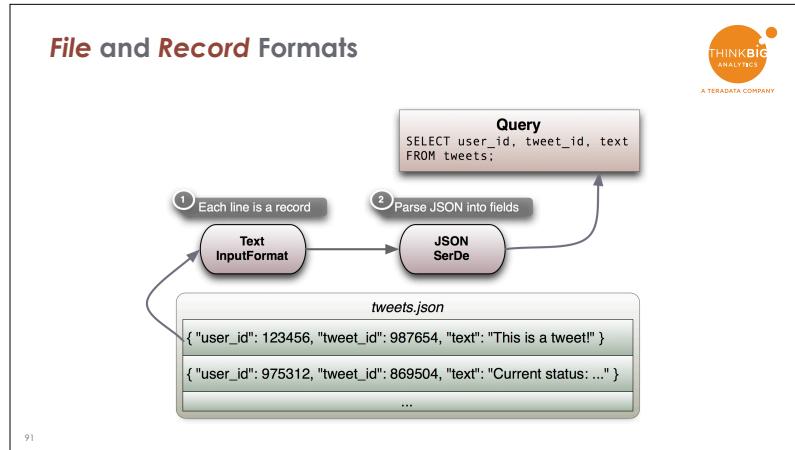
Walk through the code.

File and Record Formats

<https://cwiki.apache.org/confluence/display/Hive/SerDe>

90

How to support new file and record formats. We have used the defaults almost exclusively today, but the file and record formats, and whether or not (and how) you compress files have important benefits for disk space and network IO overhead, MR performance, how easy it is to work with files using other tools (either within or outside the Hadoop “ecosystem”), etc. These choices are usually made by the IT team when architecting the whole system and particular data usage scenarios. We discuss these choices in depth in our Developer Course aimed at Java Developers. Also, the Bonus Material section contains an expanded version of this section.



All the InputFormat does is split the file into records. It knows nothing about the format of those records. The SerDe (serializer/deserializer) parses each record into fields/columns.

File and Record Formats



A TERADATA COMPANY

- INPUTFORMAT and OUTPUTFORMAT:
 - How *records* are stored in *files* and query results are written.
- SERDE: (serializer-deserializer)
 - How *records* are stored in *columns*.

This is an important distinction; how records are encoded in files and how columns/fields are encoded in records.

INPUTFORMATs are responsible for splitting an input stream into records.

OUTPUTFORMATs are responsible for writing records to an output stream (i.e., query results). Two separate classes are used.

SERDEs are responsible for tokenizing a record into columns/fields and also encoding columns/fields into records. Unlike the *PUTFORMATs, there is one class for both tasks.

Built-in File Formats



A TERADATA COMPANY

- The default is TEXTFILE.

```
CREATE TABLE tbl_name (col1 TYPE, ...)  
...  
STORED AS TEXTFILE;
```

We have been using TEXTFILE, the default, all day.

We'll discuss several of the most common options, but there are many more to choice from. In your projects, the whole development team will want to pick the most appropriate formats that balance the various concerns of disk space and network utilization, sharing with other tools, etc.

Built-in File Formats



A TERADATA COMPANY

- SEQUENCEFILE is a binary, space-efficient format supported by Hadoop.

```
CREATE TABLE tbl_name (col1 TYPE, ...)  
...  
STORED AS SEQUENCEFILE;
```

- Easiest to use with pre-existing SEQUENCEFILEs or INSERT ... SELECT.

SEQUENCEFILE is a Hadoop MapReduce format that uses binary encoding of fields, rather than plain text, so it's more space efficient, but less convenient for sharing with non-Hadoop tools.

Built-in File Formats



- Enable SEQUENCEFILE block compression.

```
SET io.seqfile.compression.type=BLOCK;
CREATE TABLE tbl_name (col1 TYPE, ...)
...
STORED AS SEQUENCEFILE;
```

We won't discuss file compression in more detail here. (Our Hadoop training for Java Developers covers this topic in depth.) Compression gives further space and network IO savings. Compressing files by "block" (chunks of rows or bytes), rather than all at once has important practical consequences for MapReduce's ability to split a file into "splits", where each split is sent to a separate Map process. If a file can't be split, then no matter how big it is, it has to be sent to one task, reducing the benefit of a cluster! Block compressed files can be split by MR on block boundaries. Not all compression schemes support block compression. BZip2 does, but GZip does not. SEQUENCEFILEs lend themselves well to block compression.

Built-in File Formats



A TERADATA COMPANY

- RCFILE stores data by *row groups*, then by *columns* within each group:

```
CREATE TABLE tbl_name (col1 TYPE, ...)  
...  
STORED AS RCFILE;
```

- Keeps a “split’s worth” of rows in the same split, but stores by column in the split.
- See <http://en.wikipedia.org/wiki/RCFile>

96

Column-oriented storage is great when you have very long rows and queries typically only need a few columns. Rather than scanning the entire N rows, you just read a smaller amount of data off disk for each column. However, in a distributed system, you might end with rows split across the cluster, so RCFile keeps rows together in split-sized chunks first, but it actually stores the data for those rows in column order, giving you a good compromise of performance tradeoffs.

Custom File Formats



A TERADATA COMPANY

- You might have your data in a *custom format*.

```
CREATE TABLE tbl_name (col1 TYPE, ...)
```

...

```
STORED AS INPUTFORMAT '...' OUTPUTFORMAT '...';
```

You can also use other formats not built into Hive by specifying Java classes that implement them.

Custom File Formats



A TERADATA COMPANY

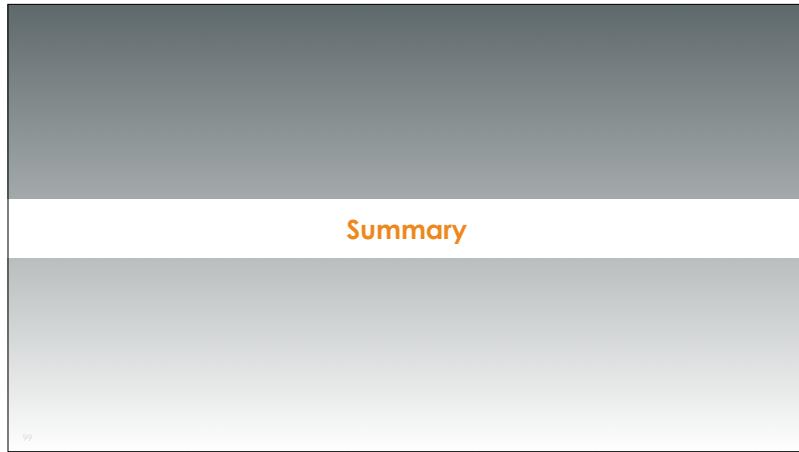
- Must specify both INPUTFORMAT and OUTPUTFORMAT.

```
CREATE TABLE tbl_name (col1 TYPE, ...)  
...  
STORED AS  
INPUTFORMAT  
'org.apache.hadoop.mapreduce.lib.input.TextInputFormat'  
OUTPUTFORMAT  
'org.apache.hadoop.hive.serde2.keyTextOutputFormat';  
Used for query results
```

98

Note that the default INPUTFORMAT is a general Hadoop MapReduce type, while the OUTPUTFORMAT is Hive-specific type.

If you specify INPUTFORMAT, you must also specify OUTPUTFORMAT.



Hive Advantages



- Provides the Hive Warehouse and Metadata for creating cluster-wide tables in Hadoop clusters.
- *Indispensable* for users with *SQL* experience.
- The basis of many 3rd-party *analyst* tools

100

I can't emphasize the first point enough. Almost every Hadoop cluster has Hive, even if programmers never use it. Why? Because it's THE repository for traditional SQL tables in a Hadoop cluster. Even if you do all your work in Spark or Pig, you still want Hive for its Warehouse and Metastore.

Hive is truly indispensable for SQL programmers because it provides a familiar interface, and it's the basis of many 3rd party tools.

Hive Advantages



A TERADATA COMPANY

- *Simplifies* many **common** data analysis **tasks**.
 - *Far simpler* than Java MapReduce programming.
 - *Optimizer* for common scenarios.
 - *Extensible* with Java plugins.
- *Integration* with other Hadoop tools.

Hive Disadvantages



- *Not a complete SQL implementation:*
 - Transactions, row updates, etc.
 - ... But some will be added over time.
- Hadoop batch mode has *high latency*.
 - But latency amortized over big queries.
- *Documentation* isn't very user friendly

Summary



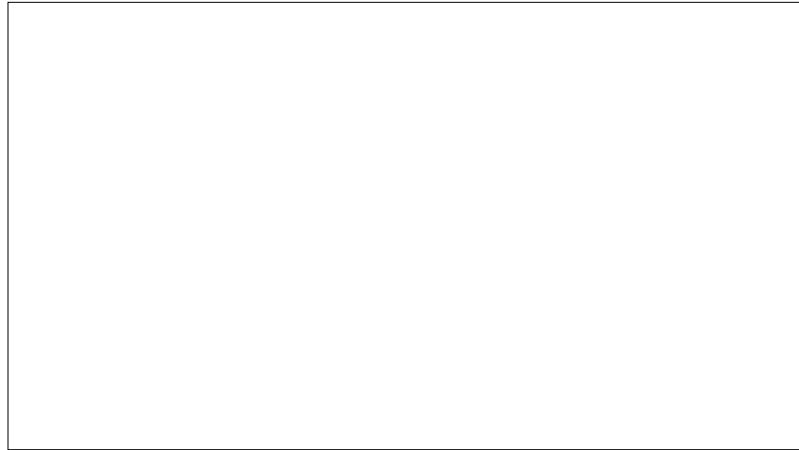
- If you use Hadoop clusters, you need to know Hive
- Hive allows you to apply familiar SQL tools with raw files of many formats using schema on read
- Hive's ability to create partitions can dramatically speed up queries
- We will frequently use Hive's metadata and metastore from other tools such as Spark



Big Data Science With Spark

Module 4: Big Data Bottlenecks

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



This is intentionally a blank slide. It provides us an opportunity to ask a question to kick off the presentation.

What was the importance of January 1, 1983? Do you know?

The New York Times

N.Y. / Region

WORLD U.S. N.Y. / REGION BUSINESS TECHNOLOGY SCIENCE HEALTH SPORTS OPINION

NEWS SUMMARY

NEWS SUMMARY, SATURDAY, JANUARY 1, 1983

Published: January 1, 1983

International Soviet troops will stay in Afghanistan until the Soviet Union's conditions for their withdrawal are met, the Government said in a statement released through Tass, the official press agency.

The missing news: This was the dawn of the Internet networking era (TCP/IP)

preparing to accept a face-saving settlement to extract more than 100,000 Soviet troops from a seemingly intractable war. (Page 1, Column 6.) Poles turned out to beat the deadline for expiring ration coupons and lined up for bread, for vodka for New Year's Eve

Facebook Twitter

PRINT REPRINTS

2

It was the dawn of the Internet networking era. It was when the ARPAnet was split from the MILnet, and when all the host computers were required to use a new protocol: TCP/IP to communicate. TCP/IP was a reinvention of networking that refuted Bill Joy's famous 4 myths (next slide)

Bill Joy's 4 Myths of Distributed Computing

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure



TCP/IP was built because none of these things were actually true

The real world has limitations and bottlenecks

3

Bill Joy's 4 myths were actually the guiding rules for networks before the 1980s. Prior to this, architects assumed that was that a sufficiently smart network could take care of all the errors might occur (i.e., the network was reliable), that they could ignore the latency of the network, they could assume they'd have all the bandwidth they needed, and they didn't need to worry about security because the network was "sealed off" from the bad things in the world.

The reality of the world is that none of these things are true of internets. TCP/IP was built to deal with real networks, not ideal ones. The real world has limitations and bottlenecks.



Understanding Big Data Bottlenecks

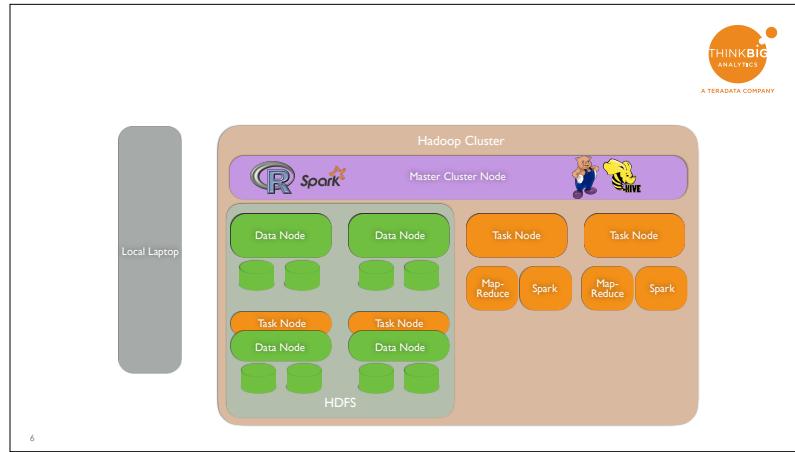
4

The same is true of Big Data. Hadoop exists in a real world of computing. The real world has limitations and bottlenecks. We need to understand these to be successful architects and developers.

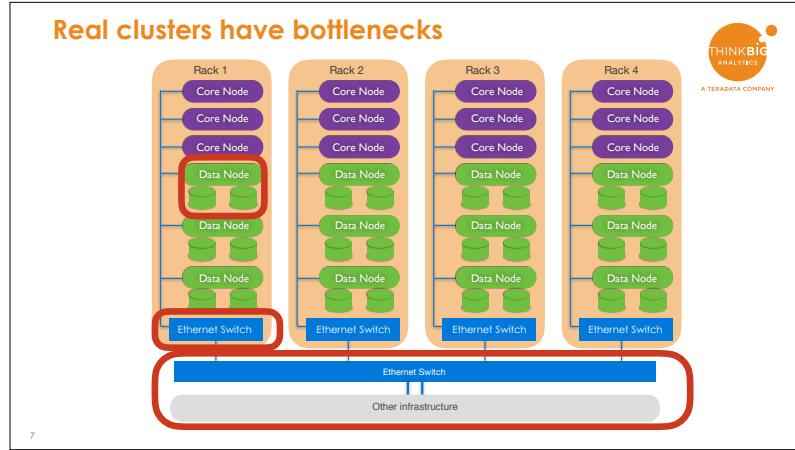
Bottlenecks define the performance of clusters



- Every component of a cluster has performance limitations
- Your slowest components define what you can do
- Understanding speed bottlenecks can spell the difference between successful Hadoop projects and failed ones



Remember this diagram? This is a logical architecture of a cluster. However, there is also a physical architecture that is more complex.



This is how a real cluster works.

We have core nodes <click>
 and we have data nodes <click>
 Those are tied together by an Ethernet <click>
 And the nodes are all put in a rack <click>
 The racks are then replicated <click>
 And the racks are networked together by another larger switch which connects it to the rest of the infrastructure <click>

Now where are the bottlenecks?

Well, the obvious one is the disk. <click>
 However, less obvious is the ethernet switch in the rack <click>
 And most neglected of all is the ethernet switch that connects to the rest of the infrastructure. <click> Often it can be very difficult to find out what the bandwidth of that link is because it sometimes is managed by a different organization (networking versus IT).

So how can we sort out the effects of all these possible bottlenecks?

Clusters must minimize disk use to be fast



A TERADATA COMPANY

	Nanoseconds	Time to move 1TB (seconds)	Time to move 1PB (days)
CPU instruction	0.2		
Uncached memory read	0.6	5	0.1
1MB SSD read (3300 MB/sec)	318,000	333 (More than 5 minutes)	3.9
1MB LAN read (10Gb ethernet)	800,000	800 (More than 10 minutes)	9.3 (More than 1 week)
1MB disk read (no seek, 6Gb SATA)	1,333,000	1,333 (More than 20 minutes)	15.4 (More than 2 weeks)
Disk head seek	3,000,000		

8

Here are some constants that every programmer should know. We're going to begin by assuming a 5 GHz CPU, where each instruction takes 0.2 nanoseconds or 200 picoseconds. <click>

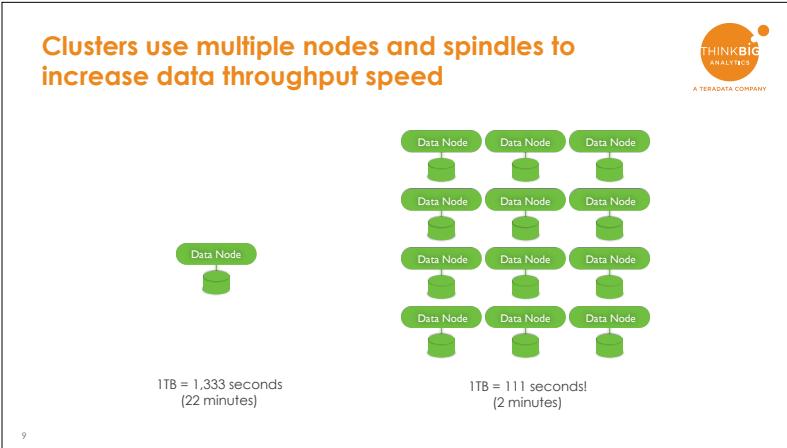
Unfortunately, our memory doesn't run as fast as our processor. It takes 0.6 nanoseconds to deliver a row of data. That means that it takes 5 seconds to read a Terabyte out of memory and more than 2 hours—about 0.1 days—to read a petabyte. And that's using RAM. <click>

Let's look at Solid State Drives or SSDs. Those are fast, right? Well, sort of. Your typical enterprise SSD reads data at about 3,300 MB/second. That means it takes 333 minutes—that's more than 5 minutes—to read a Terabyte from an SSD. It also means it takes nearly 4 days to read a Petabyte. Hmm.

We spoke about Ethernet being a possible bottleneck earlier. A 1MB read on 10Gb Ethernet takes about 800 microseconds. That means it takes 800 seconds to read a TByte and more than a week to read a Petabyte over 10Gb Ethernet.

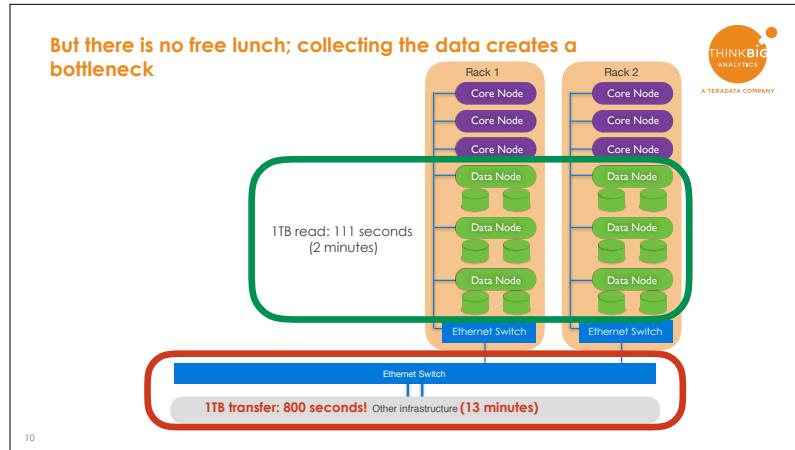
Disks are even slower, with our one Terabyte read taking more than 20 minutes and a PByte read taking more than 2 weeks.

And all of this assumes that we have no disk seeks. Disks impose even more delay to the process. It takes about 3 milliseconds to move a disk head. Suffice it to say, that seems like a



We should be able to speed up our terabyte read by sharing the data across multiple disks and then reading from multiple spindles at once. So if a single node takes 22 minutes to read a terabyte, it should only require 2 minutes for 12 data nodes to read that same terabyte (assuming it was appropriately distributed across those disks).

However....



Collecting that data, however, will require sending the information over the ethernet switch. Unfortunately, when we calculate how long that will take, that increases our time to read our terabyte back up to 13 minutes. Our bottleneck has moved from the disk to the ethernet.

Rules of good cluster performance



- Ingest into HDFS first using as parallel an operation as possible
- Do sanity checks on how long imports should take
- Use in-memory tools if possible (more on that in a bit)
- Turn big data into small data as soon as possible
- Focus exports on small data, not big

11

Bottlenecks will define how fast our cluster programs will run. But there are some ways we can avoid those bottlenecks.

Ingest into HDFS first using as parallel an operation as possible. Use tools like hadoop df distcp, sqoop, and tdch to ingest data, not single-node tools like hdfs dfs get.

Do sanity checks on how long imports should take. If someone says they are ingesting a petabyte in a few hours, don't be afraid to ask questions about how they are doing that. Petabytes are more about days than minutes.

Use in-memory tools if possible (more on that in a bit). Memory is many orders of magnitude faster than disk. It would be awesome if we had a tool that focused on keeping data in memory instead of disk. And Spark is just that tool.

Turn big data into small data as soon as possible. The best way to make a program run fast is to filter or sample your data first before processing it. Very few algorithms work significantly better with a petabyte than with a gigabyte. Don't be shy about filtering and sampling.

Focus exports on small data, not big. Hadoop isn't about doing ETL, but really about finding the analytical needle in the Big Data haystack. The art of data science is turning Big Data into insight, which if it's going to be used for decision-making, has to be small. Do that in the cluster before you export your data.



Big Data Science With Spark

Module 5: Spark Architecture And Concepts

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



Spark Architecture and Concepts

1

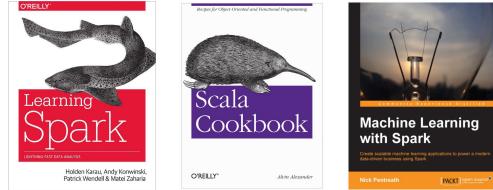


What we'll cover



- Architecture
- Comparing Spark with MapReduce
- How to start the spark-shell
- Running some hands-on Spark code

References



Indispensable. Note that some of the details have changed since the 2nd Edition.

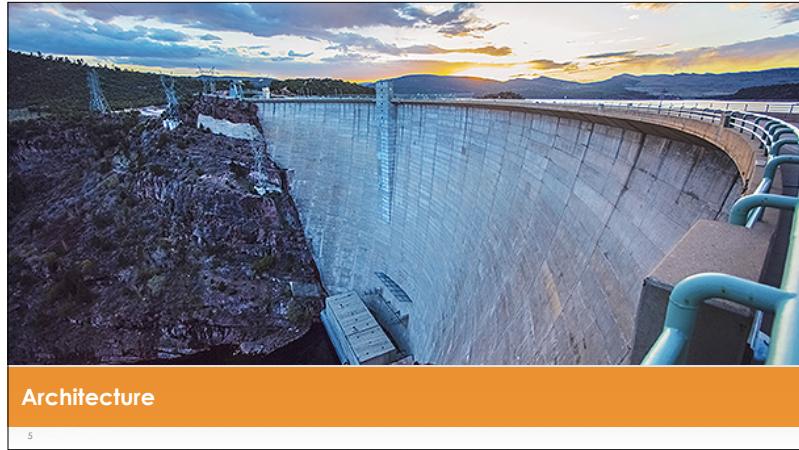


A TERADATA COMPANY

Other references

- Official docs
 - <https://spark.apache.org/docs/latest/>
- Quick Start
 - <https://spark.apache.org/docs/latest/quick-start.html>
- Programming Guides
 - <https://spark.apache.org/docs/latest/programming-guide.html>
 - MLlib: <https://spark.apache.org/docs/latest/mllib-guide.html>
 - Spark SQL: <https://spark.apache.org/docs/latest/sql-programming-guide.html>
 - GraphX: <https://spark.apache.org/docs/latest/graphx-programming-guide.html>
 - Streaming: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

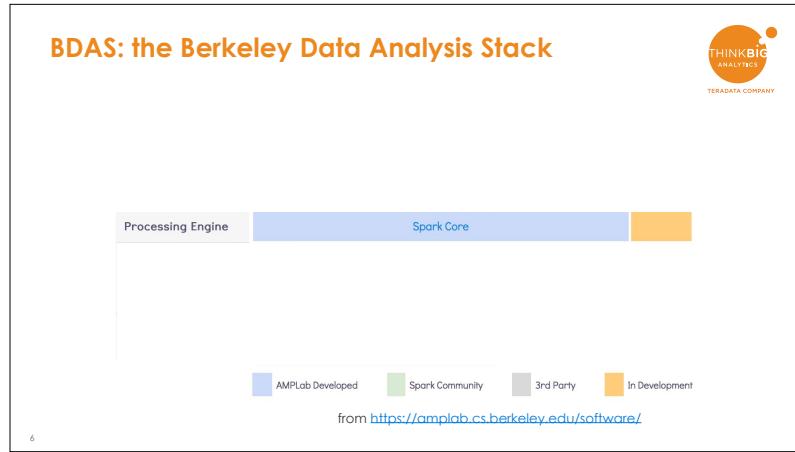
Because Spark changes so quickly—a new release comes along about every 3 months—the best references are the official Spark documentation. Just go to <http://spark.apache.org/docs/latest> to stay up to date.



Architecture

5

Spark's power comes from its Architecture. The Hadoop ecosystem evolved from the three main Google papers describing their equivalents of HDFS, MapReduce, and NoSQL. Other tools such as Hive, Pig, Kafka, and Storm simply evolved on top of that.



In contrast, Spark was designed by the Berkeley AMPLab, not as an add-on, but a complete software stack.

At the heart of Spark is the Spark Core. It contains the basic processing engine of Spark. However, that core was designed to run on top of other components. <<click to next build>>.

Now Spark can happily run on top of the traditional Hadoop components of YARN and HDFS. However, Spark was originally designed to use its own cluster OS, Mesos. And while Spark was designed to use HDFS as its primary storage, the AMPLab also added an in-memory accelerator to that called Tachyon.

And Berkeley didn't just work on the infrastructure of clusters. They also thought about the stack above the Spark Core and developed tools for applications. <<click for next build>>

On top of the core, Berkeley added a host of applications, which we'll discuss on the next slide.

In short, Spark is unlike Hadoop because all its components were designed to work together instead of simply evolving.

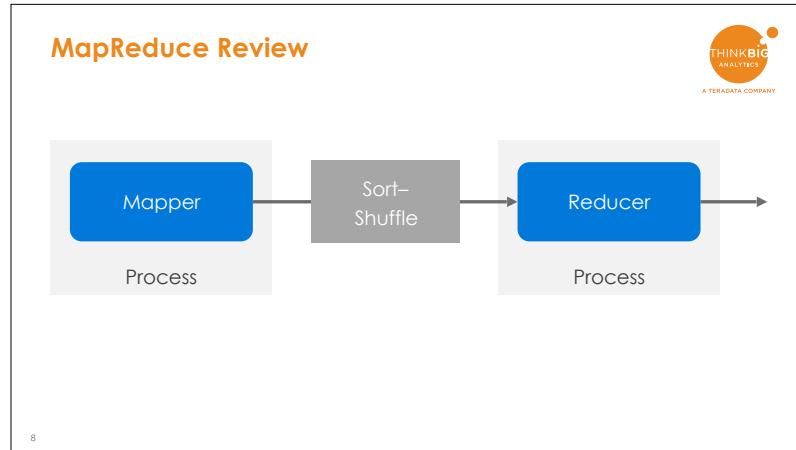
Spark: A full stack ecosystem designed for clusters



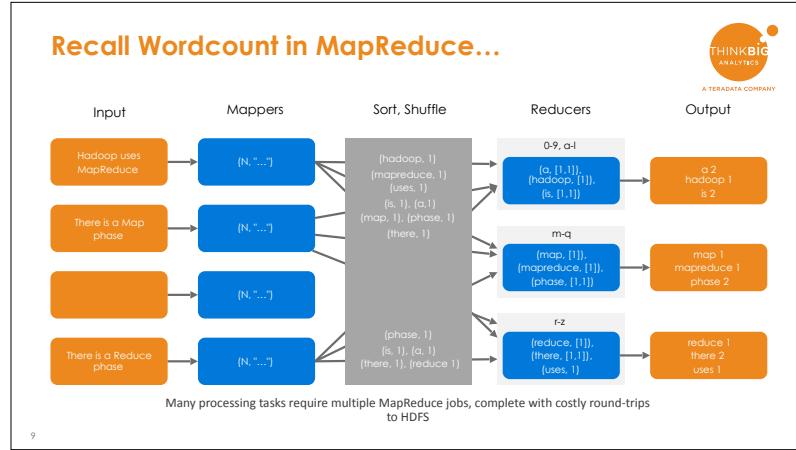
- Written in Scala (a functional Java Virtual Machine (JVM) language)
- Four officially supported APIs:
 - Scala
 - Java
 - Python
 - R (as of v1.4)
- Major components
 - General-purpose Spark for Big Data
 - Spark Streaming
 - SparkSQL and Dataframes Support
 - GraphX for graph analysis
 - SparkML for machine learning
- All Spark components are already **designed to run in parallel on Hadoop**

7

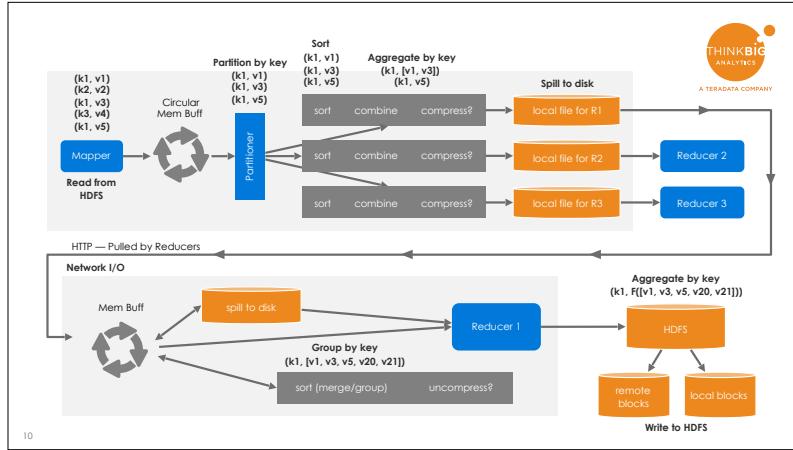
Spark is written in Scala, which is a Java Virtual Machine language. It natively supports four languages instead of Hadoop's one (Java). It hosts a wealth of applications and most importantly, every Spark application is **designed to run in parallel on a cluster**.



If you recall our discussion of WordCount when we talked about Big Data, It consisted of three major steps: Map, Sort-Shuffle, and Reduce. However, as you'll recall, more than that was going on.



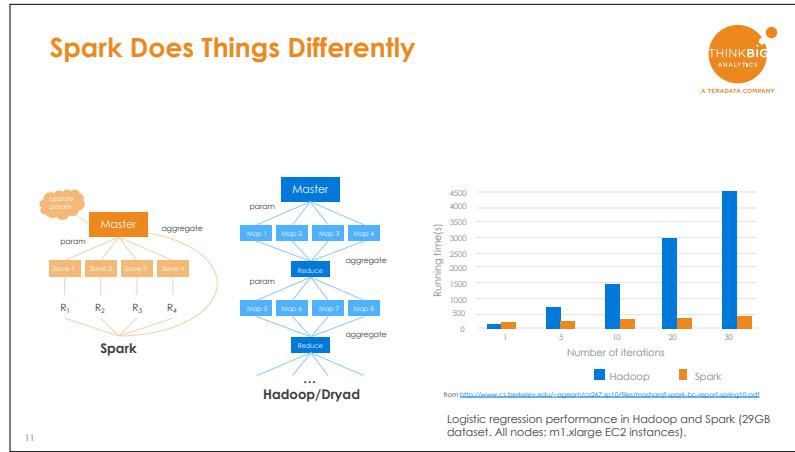
You remember that MapReduce was designed all around key-value pairs that contained bits and pieces of our Big Data. What we glossed over, though, is how we use these key-value pairs when they may contain terabytes or petabytes of information, and when cluster memory may only be measured in gigabytes.



In fact, what actually goes on is that MapReduce is designed so that it doesn't have to store those key value pairs in RAM. Instead, it has mechanisms to spill data to disk at every step of its processing.

The mapper writes to a circular memory buffer that spills to disk when it gets close to filling or at the end of the map process. The partitioner splits the K-V pairs into the same number of streams as there are reducers and each stream is sorted, the Combiner is applied (if there is one) and compressed (if turned on). The mapper's task tracker informs the job tracker when results are available, which informs the reducer's task tracker. The reducer then starts pulling data over using HTTP (even before the mapper is finished). The pulled data is stored in memory, spilling to disk as needed. The streams from the different mappers are merged (it's called a sort, but it's really a merge) after uncompressing, if necessary. The final merge step, reading from memory and disk, is fed into the reducer, which outputs to HDFS (or other...), with one replica of each block stored locally and the other replicas sent to other cluster nodes. Note that the reducer receives key-value pairs that are key-[val1, val2, ...]. Deciding what vals belong with each key is called "grouping".

So while it might appear that MapReduce only uses HDFS for input and output, in fact it may be using disks all through the computation. Making this situation even worse is the fact that many Big Data jobs require multiple MapReduce steps.



Spark on the other hand does things differently. Instead of locking the user into a fixed Map-Shuffle-Reduce computational model, it allows developers to create arbitrary computations, represented as Directed Acyclical Graphs or DAGs. That combined with its in-memory storage allows Spark to outperform MapReduce in many Big Data workloads.

Why is Spark fast?



- Caches data and intermediate results in RAM
- Multithreaded Executors live for entire duration of application
- Much more flexible execution plans — more complicated DAGs can avoid expensive trips to/from HDFS

12

There are three reasons that Spark is fast.

The one that everyone knows is that Spark caches data in RAM. That's true and it does speed things up.

What a lot of people aren't aware of though is that some of Spark's speed comes from not always spinning Java Virtual Machines up and down. In MapReduce, a new JVM is started for each Map, Shuffle, and Reduce phase. In Spark, though, the JVMs are started once at the beginning of a job and only destroyed when the job is over. This makes various Spark job phases run much faster.

Finally, Spark has much more flexible execution plans. Instead of being locked into a Map-Shuffle-Reduce sequence, it can run arbitrary DAGs and optimize across the entire DAG. That means that Spark can often eliminate entire branches of computations if they aren't needed by the results requested.

An analogy for MapReduce versus Spark



13

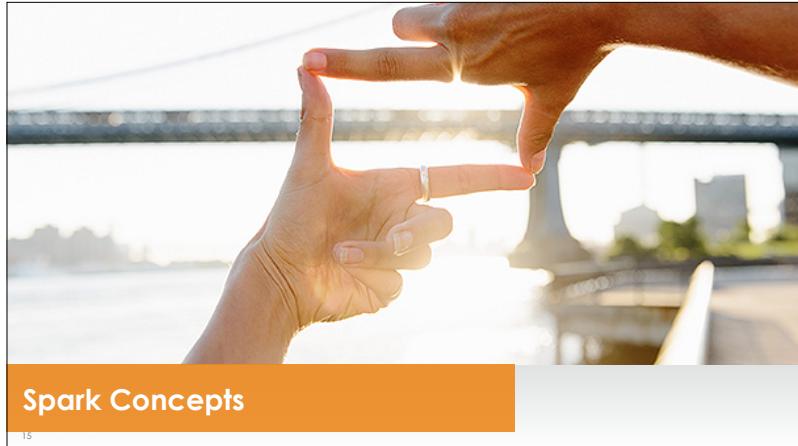
As an analogy, think of a MapReduce job as akin to a car traveling across a busy city with stoplights. The stoplights assure that the vehicle proceeds safely, but they do require it to stop periodically. In contrast, Spark is more like traveling on a superhighway. It has fixed entrances and exits, but in between, the vehicle can just speed along.

Industry has committed to Spark

The screenshot shows a news article from CIO magazine. The title is "IBM commits to Apache Spark compute engine". The article discusses IBM's commitment to Apache Spark, mentioning its integration into IBM Bluemix and its donation of IBM SystemML machine learning technology to the Spark open source ecosystem. The author is Thor Straszheim, and the date is 01.08.2014. The page includes a sidebar with related links and a footer with navigation icons.

14

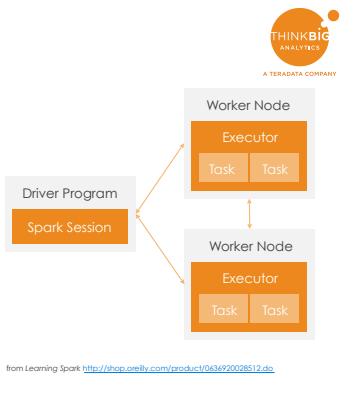
While there may be competing in-memory technologies—Apache Flink, for example, has many of the same characteristics as the Spark execution engine—the industry has put its money on Spark to date. IBM alone has roughly 3,500 Spark developers on staff already. And it is just one of many IT vendors supporting Spark.



How does Spark actually work then?

Spark Components

- Driver program
 - Contains the “main” function for your application
 - Communicates with the cluster through a Spark Session
- Jobs
 - Encompasses all the work to compute the requested RDDs
 - Composed of one or more stages
 - Stages are broken into Tasks for execution
- Executors
 - Persistent, multithreaded processes which run the tasks



from Learning Spark <http://shop.oreilly.com/product/0636920028512.do>

16

First, we'll define some terms.

The Spark Driver program contains the “main” function for your application and communicates with the cluster through a Spark Session.

Jobs encompass all the work to compute the requested distributed datasets, which are called RDDs. A job consists of one or more stages. Stages, in turn, are broken into Tasks for execution.

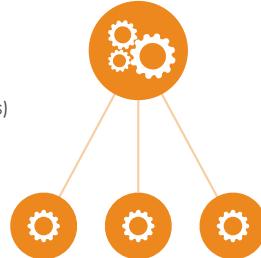
Persistent, multithreaded processes called Executors actually run the tasks. Each worker node making up a Spark cluster will have one or more Spark executors.

Three Spark Deployment Options



A TERADATA COMPANY

- local: runs on the local machine
 - master local (for single core)
 - master local[N] (for N cores)
- standalone: Spark's built-in cluster mode (you are responsible for launching master and worker processes)
 - master spark://host:port
- YARN: uses YARN to launch master and worker processes in containers
 - master yarn
(related: Mesos mode)
 - master mesos



17

Spark can run in three ways:

1. local mode which only runs on one processor, albeit using multiple cores.
2. standalone mode, which is where Spark owns the cluster. No other jobs other than Spark ones run in standalone mode. YARN isn't run either.
3. YARN mode. This is the most common mode, and it's one where the Spark driver and executors run in containers managed by YARN.

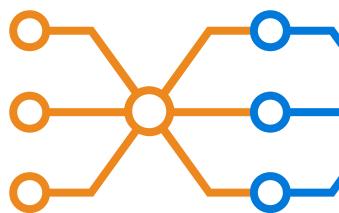
Mesos mode is also available in case you are running a Mesos cluster, but that isn't common.

Connecting Spark and Hadoop



A TERADATA COMPANY

- While Spark doesn't require Hadoop, it commonly leverages HDFS since it lacks its own storage layer
- Properly configured Hadoop client tools on each node running Spark will ensure access to HDFS, etc.
- Copy Hive's `hive-site.xml` file into Spark's `conf/` directory to allow SparkSQL to access Hive's metastore



18

Spark is independent of Hadoop, but needs to connect to it because it needs a storage layer.

While Spark doesn't require Hadoop, it commonly leverages HDFS since it lacks its own storage layer

Properly configured Hadoop client tools on each node running Spark will ensure access to HDFS, etc.

Copy Hive's `hive-site.xml` file into Spark's `conf/` directory to allow SparkSQL to access Hive's metastore

Spark Web UI

- Launched by Spark shell, pyspark, etc.
- Runs on port 4040 by default
- Displays information on running and completed jobs and other cluster details

The screenshot shows the Spark Web UI interface. At the top, there's a navigation bar with tabs: Jobs (which is selected), Stages, Storage, Environment, and Executors. To the right of the navigation is a logo for 'THINKBIG ANALYTICS A TERADATA COMPANY'. Below the navigation, the main content area is titled 'Spark Jobs' with a link '(?)'. It shows a summary: Total Uptime: 3.3 min, Scheduling Mode: FIFO, Completed Jobs: 1. There's a link to 'Event Timeline'. Below this is a table titled 'Completed Jobs (1)'. The table has columns: Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. One row is present: Job Id 0, Description count at <console>:24, Submitted 2015/06/15 23:18:11, Duration 0.1 s, Stages: Succeeded/Total 1/1, Tasks (for all stages): Succeeded/Total 4/4.

19

A Web user interface for Spark is started up whenever a Spark job runs. Typically this will be available on port 4040. However, if another job is still running, the Spark UI may be started on a sequential port afterward, such as 4041 or 4042. Spark will usually tell you what port its GUI is running on.

This user interface has a huge amount of information about what Spark is doing and how the applications are running. You should spend time exploring it sometime when you have run a few Spark jobs and see all the information it provides.

Getting started with Spark in Scala



A TERADATA COMPANY

To run bin/spark-shell on exactly four cores, use:

```
$ ./bin/spark-shell --master local[4]
```

Or, to also add code.jar to its classpath, use:

```
$ ./bin/spark-shell --master local[4] --jars code.jar
```

For a complete list of options, run spark-shell --help. Behind the scenes, spark-shell invokes the more general [spark-submit script](#).

Getting started with Spark in Python



A TERADATA COMPANY

For example, to run bin/pyspark on exactly four cores, use:

```
$ ./bin/pyspark --master local[4]
```

Or, to also add code.py to the search path (in order to later be able to import code), use:

```
$ ./bin/pyspark --master local[4] --py-files code.py
```

For a complete list of options, run pyspark --help. Behind the scenes, pyspark invokes the more general [spark-submit script](#).



Spark and MapReduce Comparison Exercise

22

We are now going to perform three labs which will get you used to connecting to your clusters. You should have a directory with all the course materials called sparkclass. Underneath that directory are several other directories including

- data
- exercises
- handouts
- images
- slides

You'll find these labs in your exercises/Hadoop-Architecture directory.

You can read these labs in either pdf or Markdown form. The PDF form is prettier, but the text in the Markdown version will be easier to copy and paste when you get to complicated commands.

Go ahead and run those three labs. We'll allot about 15 minutes for you to do this.

Comparing Spark and MapReduce Speed

Vagrant version



- vagrant ssh edge to your home directory (cd ~)
 - Time and run MapReduce Pi Estimator for 100000 samples

```
time hadoop jar /usr/lib/hadoop-0.20-mapreduce/hadoop-examples.jar pi 4 100000
```
 - Your run time is probably around 25 seconds
-
- Now run the same example under Spark. This example will actually run 400000 samples

```
sudo su
sed 's/INFO/WARN/g' /vagrant/latest-spark/conf/log4j.properties.template \
>/vagrant/latest-spark/conf/log4j.properties
exit
time $SPARK_HOME/bin/run-example --master local[*] SparkPi 4
```
 - Your run time is probably around 8 seconds

Comparing Spark and MapReduce Speed

EMR version



- ssh to your home directory on the cluster (cd ~)
- Time and run MapReduce Pi Estimator for 100000 samples

```
time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.7.3-amzn-3.jar pi 4 100000
```

- Your run time is probably around 28 seconds

- Now run the same example under Spark.
- Turn down Spark logging by typing the following

```
sudo su
sed 's/INFO/WARN/g' /etc/spark/conf/log4j.properties.template >/etc/spark/conf/log4j.properties
exit
time /usr/lib/spark/bin/run-example --master local[*] SparkPi 4
```

- Your run time is probably around 7 seconds
- Note that user time may be greater than real time, showing parallel execution

Summary



A TERADATA COMPANY

- Spark offers a faster, in-memory alternative to MapReduce
- Spark's advantages include
 - Fully integrated software stack including SQL and machine learning
 - 4 supported languages
 - New concepts for how to code for cluster
- Despite its integrated view of software, Spark coexists with traditional Hadoop infrastructure



Big Data Science With Spark

Module 6: Spark In Scala

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



Spark in Scala

1

With preliminaries out of the way, let's dive into Spark in Scala. We choose Scala because it's the language Spark was written in. As a result, it poses the cleanest interface to the Spark environment.

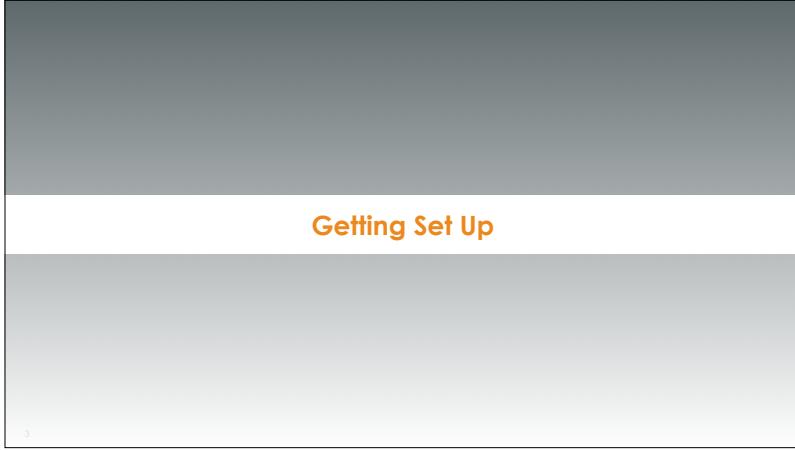


What We'll Cover



- The Spark execution model and Spark Sessions
- The Spark shell
- Scala immutables and mutables
- Resilient Distributed Datasets (RDDs)
- External datasets
- RDD operations: Transformations and Actions
- Function literals (aka closures or anonymous functions)
- Persistence
- Some Spark examples

We'll begin by covering these fundamental topics.



Getting Set Up

Before we begin, though, we should get you set up.

How to upgrade Spark software using our flash drive

Vagrant Version



A TERADATA COMPANY

We included some upgrade scripts as part of our file distribution that should now be in /vagrant. You should:

1. Log into your edge node
2. cd /vagrant
3. source ./spark-setup-edge.sh
4. Log into your control node
5. cd /vagrant
6. source ./spark-setup-control.sh

At this point, you should be up and ready to go with Spark 2.2.0 on your cluster. Your SPARK_HOME variable has been set to /vagrant/latest-spark.

If you are using a local two-node VirtualBox cluster created using Vagrant, you'll want to execute a couple of scripts to get your environment set up with the latest version of Spark (Spark 2.2.0)

No Upgrade Required

EMR Version



Your Amazon EMR instance should already be configured with version 2.2.0 of Spark.

Please note that Spark has been installed in /usr/lib/spark.

The shell variable \$SPARK_HOME has not been set by EMR. Should you need to set the variable SPARK_HOME in later exercises, you will want to use the value /usr/lib/spark.

At this point, you should be up and ready to go with Spark 2.2.0 on your cluster.

If you are using Amazon Web Services Elastic Map Reduce instances, your instance should be all set up to use Spark 2.2.0.

If you ever need to do a manual upgrade, do this



The following assumes you have downloaded a version of Spark into your Vagrant directory. In this case, we will assume it to be `latest-spark`

1. Set `SPARK_HOME`: `export SPARK_HOME=/your-spark-installation-location`
(e.g., `/vagrant/latest-spark` or `/mnt/latest-spark`)
2. Add Spark to your `$PATH`: `export PATH=$SPARK_HOME/bin:$PATH`
3. Copy over your `hive-site.xml` to the Spark conf directory:
`cp /etc/hive/conf/hive-site.xml $SPARK_HOME/conf/`
4. Set your `HADOOP_CONF_DIR` variable so that you can use yarn mode:
`export HADOOP_CONF_DIR=/etc/hadoop/conf`
5. (optional) For Spark versions before 2.0, reduce your Spark logging from INFO to WARN.
`sudo sed -i.bak 's/INFO/WARN/' /usr/lib/spark/conf/log4j.properties`
6. (optional) If you are running 2.0 or later, the logging default is WARN and you can adjust this at the command line:
`sc.setLogLevel(newLevel)`

6

One of the best features of Spark is that it is easy to switch to using a different version than the one installed on your cluster. The process really consists of only 4 essential steps:

1. Setting your `SPARK_HOME` shell variable to point at the directory where your Spark installation lives.
2. Adding `$SPARK_HOME/bin` to your execution path.
3. Copying over your `hive-site.xml` file to your Spark configuration directory
4. Setting the shell variable `$HADOOP_CONF_DIR` to point at your Hadoop configuration directory.

If you are using a Spark version prior to version 2, the default is for Spark to log all INFO messages to the console, which can definitely interfere with seeing what your program is doing. You can avoid this issue by setting your logging level to WARN by editing your `log4j.properties` file, as shown in step 5.

If you are running Spark 2.0 or later, the default logging level is WARN. You can change it to other levels, say ERROR, using a simple spark-shell command, `sc.setLogLevel("ERROR")`.



A TERADATA COMPANY

Spark smoke test

Vagrant Version

- Go to your edge node by typing vagrant ssh edge
- On the edge node, type: spark-shell
- You should see the following:

```
$ spark-shell
17/08/28 19:06:38 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling
back to uploading libraries under SPARK HOME.
Spark context Web UI available at http://ip-172-31-57-176.ec2.internal:4040
Spark context available as 'sc' (master = yarn, app id = application_1503599820296_0009).
Spark session available as 'spark'.
Welcome to
    / \ \
   /   \ \
  /     \ \
 /       \ \
/         \ \
version 2.2.0

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_141)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

7

If you're using vagrant, Go to your edge node by typing vagrant ssh edge. On the edge node, type: spark-shell. You should see the following.

Note that the spark-shell provides you with the Web address of the Spark user interface. It also has provided you with both a spark context (sc) and a spark session (spark). You'll need those to issue commands to the spark cluster.



 A TERADATA COMPANY

Spark smoke test

EMR Version

- Connect to your cluster by typing
`ssh -i ~/sparkcourse.pem hadoop@$MY_IP_ADDRESS`
- Type spark-shell and you should see the following:

```
$ spark-shell
17/08/28 19:06:38 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling
back to uploading libraries under SPARK_HOME.
Spark context (Web UI available at http://ip-172-31-57-176.ec2.internal:4040)
Spark context available as 'sc' (master = yarn, app id = application_1503599820296_0009).
Spark session available as 'spark'.
Welcome to
```

```
    / \   / \   / \   / \
   /   \ /   \ /   \ /   \
  /     \ /     \ /     \ /     \
 /       \ /       \ /       \ /       \
/         \ /         \ /         \ /         \
/           \ /           \ /           \ /           \
/             \ /             \ /             \ /             \
/               \ /               \ /               \ /               \
/                 \ /                 \ /                 \ /                 \
/                   \ /                   \ /                   \ /                   \
/                     \ /                     \ /                     \ /                     \
/                       \ /                       \ /                       \ /                       \
/                         \ /                         \ /                         \ /                         \
/                           \ /                           \ /                           \ /                           \
/                             \ /                             \ /                             \ /                             \
/                               \ /                               \ /                               \ /                               \
/                                 \ /                                 \ /                                 \ /                                 \
/                                   \ /                                   \ /                                   \ /                                   \
/                                     \ /                                     \ /                                     \ /                                     \
/                                       \ /                                       \ /                                       \ /                                       \
/                                         \ /                                         \ /                                         \ /                                         \
/                                           \ /                                           \ /                                           \ /                                           \
/                                             \ /                                             \ /                                             \ /                                             \
/                                               \ /                                               \ /                                               \ /                                               \
/                                                 \ /                                                 \ /                                                 \ /                                                 \
/                                                   \ /                                                   \ /                                                   \ /                                                   \
/                                                     \ /                                                     \ /                                                     \ /                                                     \
/                                                       \ /                                                       \ /                                                       \ /                                                       \ /                                                       \
/                                                         \ /                                                         \ /                                                         \ /                                                         \ /
```

version 2.2.0

```
Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_141)
Type in expressions to have them evaluated.
Type :help for more information.
```

8 scala>

If you're using EMR, Go to your edge node by typing

```
ssh -i ~/sparkcourse.pem hadoop@$MY_IP_ADDRESS
```

On your cluster node, type `spark-shell`. You should see the following.

Note that the spark-shell provides you with the Web address of the Spark user interface. It also has provided you with both a spark context (sc) and a spark session (spark). You'll need those to issue commands to the spark cluster.



RDDs, Transforms, and Actions

With setup done, let's jump into the fundamental components of basic Spark: RDDs, transforms, and actions.

Execution Model and Spark Contexts



- Spark programs execute in two different places
 - In a single Spark **driver program** on the master or edge node
 - On **executor nodes** in the cluster
- Driver code runs **serially**
- Task code runs in **parallel**
- Every Spark program creates this environment on startup and references it through an object called the **Spark Session** (or in older versions, a **Spark Context**).
- Java Virtual Machines start on cluster worker nodes as Tasks when a Spark Session is created and only die when the driver program ends
- Only one Spark Session or Spark Context can exist in a program

```
val spark = SparkSession  
  .builder()  
  .appName("Spark Example")  
  .config("spark.driver.memory", "3g")
```

10

Spark programs execute in two different places

1. In a single Spark driver program on the master or edge node
2. On executor nodes in the cluster

Driver code runs serially, while Task code runs in parallel. Every Spark program creates this environment on startup and references it through an object called the Spark Session (or in older versions, a Spark Context).

Java Virtual Machines start on cluster worker nodes as Tasks when a Spark Session is created and only die when the driver program ends

Only one Spark Session or Spark Context can exist in a program.

We can create a spark session using the following Scala chain:

```
val spark = SparkSession  
  .builder()  
  .appName("Spark Example")  
  .config("spark.driver.memory", "3g")
```

The builder method creates the spark session, appName then names that session, and the config method then sets various spark variables. In this case, we're setting the driver memory size to 3 gigabytes. We could similarly set the executor memory sizes using another config statement.

The Spark shell



- **spark-shell** creates a Scala interpreter for Spark, along with a context in the variable `sc`
- At the prompt can write Scala code and have it interpreted immediately.

```
spark-shell --master yarn
Setting default log level to "WARN".
To adjust log level, use sc.setLogLevel(newLevel).
Spark context available as 'sc' (master = yarn, app id = application_1477402950617_0001).
Spark session available as 'spark'.
Welcome to

    / \ \
   /   \ \
  /     \ \
 /       \ \
/         \ \
version 2.0.1

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_45)
Type in expressions to have them evaluated.
Type :help for more information.

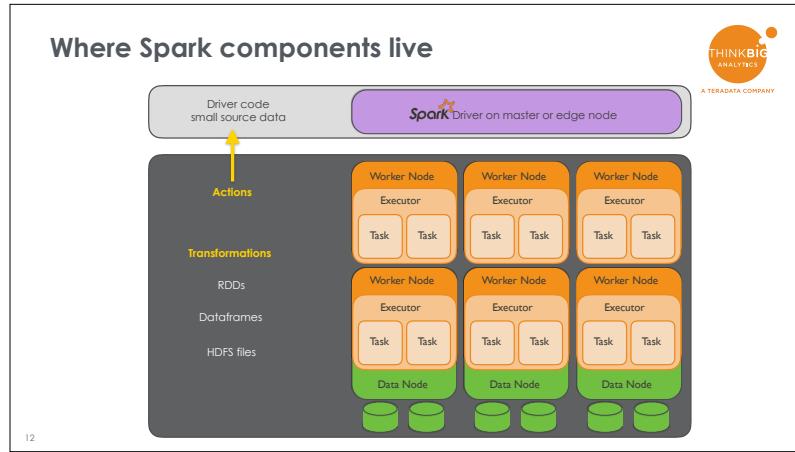
scala> val shakes = sc.textFile("/data/shakespeare/input")
shakes: org.apache.spark.rdd.RDD[String] = hdfs://data/shakespeare/input MapPartitionsRDD[1] at textFile at <console>:24

scala> shakes.take(1)
res0: Array[String] = Array(" 1 KING HENRY IV")
```

11

The spark shell is a REPL, or Read Evaluate Print Loop program. It reads what you type and then interprets it as Scala code.

The spark-shell is one of the biggest differences between the Spark Scala programming model and a traditional Java one. Scala has a REPL to ease development; at the time of this writing Java has none (although one is promised in Java 9).



If we look at this visually, the light gray box represents the edge node, depending on your cluster configuration. That's where the Spark driver lives. It runs serially there on one node.

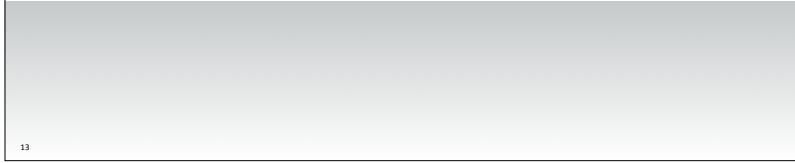
In the dark gray box, we have the rest of the cluster data and worker nodes. That's where HDFS files are distributed over the disks connected to the data nodes. It's also where Spark DataFrames and RDDs exist.

Transformations are Spark methods that execute in parallel on the cluster. They run in parallel.

Actions are Spark methods that take data from the cluster and bring that data back to the driver. Because they must bring the data back to the driver, they force all the nodes involved to synchronize their efforts. They create a momentary serialization.



Walkthrough: 01-spark-basics-lab.{scala,md}



13

Let's go hands-on with Spark with our first Scala lab. Two versions exist: one in Markdown and one in pure Scala code. We recommend you bring up one of those versions in a regular text editor (i.e., Notepad on Windows PCs,TextEdit or your favorite development text editor on a Mac) to view the text while having your terminal window along side it for typing commands.

Hands-On Lab: First Scala Program: Lab 02-spark-first-program.md



We saw in our walkthrough that we count the number of stock quotes for AAPL using code that looks like this:

```
val rdd = sc.textFile("hdfs:///data/stocks-flat/input")
val aapl = rdd.filter( line => line.contains("AAPL") )
aapl.count
```

Try using that pattern to count the number of lines in the file /data/shakespeare/input that contain the word "king". You'll want to use the `toLowerCase` function before the `contains` function to ensure all the text is lower case.

A solution is on the next slide or in the lab script

Hands-On Lab: First Scala Program



A TERADATA COMPANY

One solution looks like this

```
val rdd = sc.textFile("hdfs:///data/shakespeare/input")
val kings = rdd.filter( line => line.toLowerCase().contains("king"))
kings.count
```

You should have gotten 4773 lines that reference a king in all of Shakespeare. That's almost 3% of the 175,376 lines in all of Shakespeare's plays and poems.



15

Scala immutables and mutables



- Variables can be of two types in Scala
 - **Immutable:** Can't be changed once created
 - **Mutable:** Can be changed and modified after creation
- Immutable variables are created by declaring them **val**
- Mutable variables are created by declaring them as **var**

```
val immutable = "Immutables can never be changed once created"  
var mutable = "Mutables can be modified after creation"
```

16

Variables can be of two types in Scala

Immutable: Can't be changed once created

Mutable: Can be changed and modified after creation

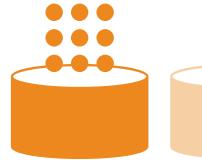
This is probably the most profound addition to cluster programming in the 21st century. By explicitly allowing the programmer to tell Scala which variables are immutable, Scala knows it can copy those variables throughout the cluster safely without affecting the validity of the computation. That means frequently used variables are likely to be local to a node and avoids excessive network traffic fetching variables. This is one of the fundamental insights within Spark: Immutables allow cluster programs to run faster because of greater data locality.

Immutable variables are created by declaring them **val**

Mutable variables are created by declaring them as **var**

Resilient Distributed Datasets (RDDs) are the primary parallel data structure

- Data is represented as Resilient Distributed Datasets (“RDDs”)
 - Can be created on the fly in the driver (e.g., `parallelize()`)
 - Could be a file on disk, HDFS path, result set, etc.
 - RDDs are immutable and cannot be modified once created
- New RDDs can be created from existing ones via **Transformations** which are executed **lazily**
- **Actions** compute a result based on an RDD. The result can be returned to the driver environment or saved to disk or elsewhere



17

Data is represented as Resilient Distributed Datasets (“RDDs”)

Can be created on the fly in the driver (e.g., `parallelize()`)

Could be a file on disk, HDFS path, result set, etc.

RDDs are immutable and cannot be modified once created

New RDDs can be created from existing ones via Transformations which are executed lazily

Actions compute a result based on an RDD. The result can be returned to the driver environment or saved to disk or elsewhere

Parallelized collections create RDDs



- The transformation `parallelize` turns a local collection into an RDD
- Once in an RDD, that parallelized collection can be operated on in parallel
- You can control the number of partitions that `parallelize` uses through a second argument, e.g., `sc.parallelize(data, 10)`
- Default number of partitions used is automatically set by your cluster

```
val data = Array(1, 2, 3, 4, 5)           // driver Array
val distData = sc.parallelize(data)        // Transformation into RDD
distData.take(3)                         // Action: take first 3 elements
res4: Array[Int] = Array(1, 2, 3)         // result
```

18

The transformation `parallelize` turns a local collection into an RDD

Once in an RDD, that parallelized collection can be operated on in parallel

You can control the number of partitions that `parallelize` uses through a second argument, e.g., `sc.parallelize(data, 10)`

Default number of partitions used is automatically set by your cluster



More commonly, file inputs create RDDs

- Spark can create RDDs from any storage source supported by Hadoop
 - Local file system, HDFS, Cassandra, HBase, Amazon S3
 - If using local file systems, the file must be replicated or shared on all worker nodes
 - Spark happily reads directories, compressed files, and paths specified by regular expressions
 - Like parallelize, textFile and its friends accept a second argument with the number of partitions to parallelize the input across

```
scala> val shakes = sc.textFile("hdfs:///data/shakespeare/input")
shakes: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>:21
scala> shakes.take(1)
res1: Array[String] = Array(" 1 KING HENRY IV")
```

19

Spark can create RDDs from any storage source supported by Hadoop

Local file system, HDFS, Cassandra, HBase, Amazon S3

If using local file systems, the file must be replicated or shared on all worker nodes

Spark happily reads directories, compressed files, and paths specified by regular expressions

Like parallelize, textFile and its friends accept a second argument with the number of partitions to parallelize the input across

RDD Operations: Transformations and Actions



- **Transformations** convert an RDD into another RDD
- **Actions** often turn an RDD into something else
- Only **actions** cause evaluation

```
val lines = sc.textFile("hdfs://data/shakespeare/input")      // Transformation
val lineLengths = lines.map(s => s.length)                  // Transformation
val totalLength = lineLengths.reduce((a, b) => a + b)        // Action
```

Transformations convert an RDD into another RDD

Actions often turn an RDD into something else

Only actions cause evaluation

Common Transformations



A TERADATA COMPANY

Assume:

```
val rdd = sc.parallelize(Array("This is a line of text", "And so is this"))  
map — apply a function while preserving structure  
    rdd.map( line => line.split("\\s+") )  
    → Array(Array(This, is, a, line, of, text), Array(And, so, is, this))
```



flatMap — apply a function while flattening structure

```
rdd.flatMap( line => line.split("\\s+") )  
→ Array(This, is, a, line, of, text, And, so, is, this)
```



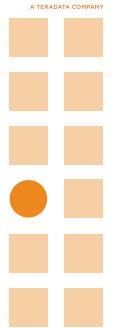
filter — discard elements from an RDD which don't match a condition

```
rdd.filter( line => line.contains("so") )  
→ Array(And so is this)
```



Common Transformations

```
val pair_rdd = rdd.flatMap( line => line.split("\\s+")) .map( word  
=> (word.toLowerCase, 1) )  
+ Array((this,1), (is,1), (a,1), (line,1), (of,1), (text,1), (and,  
1), (so,1), (is,1), (this,1))  
  
groupByKey — group values by key  
pair_rdd.groupByKey  
→ Array((this,CompactBuffer(1, 1)),  
(is,CompactBuffer(1, 1)), (line,CompactBuffer(1)),  
(so,CompactBuffer(1)), (a,CompactBuffer(1)),  
(text,CompactBuffer(1)), (of,CompactBuffer(1)),  
(and,CompactBuffer(1)))  
  
reduceByKey — apply a function to each value for each key  
pair_rdd.reduceByKey( v1, v2 ) => v1 + v2 )  
→ Array((this,2), (is,2), (line,1), (so,1), (text,1),  
(a,1), (of,1), (and,1))
```



More Transformations



A TERADATA COMPANY

For RDDs

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- sample
- union
- intersection
- distinct
- cartesian
- pipe
- coalesce

For PairRDDs:

- groupByKey
- reduceByKey
- aggregateByKey
- sortByKey
- join
- cogroup
- keys
- values
- repartition
- repartitionAndSortWithinPartitions

see <http://spark.apache.org/docs/2.2.0/programming-guide.html#transformations>

Common Spark Actions

- **collect** — gather results from nodes and return
- **first** — return the first element of the RDD
- **take(N)** — return the first N elements of the RDD
- **saveAsTextFile** — write the RDD as a text file
- **saveAsSequenceFile** — write the RDD as a SequenceFile
- **count** — count elements in the RDD
- **countByKey** — count elements in the RDD by key
- **foreach** — process each element of an RDD
 - (e.g., `rdd.collect.foreach{println}`)

See <http://spark.apache.org/docs/2.2.0/programming-guide.html#actions>



A TERADATA COMPANY



Spark Actions

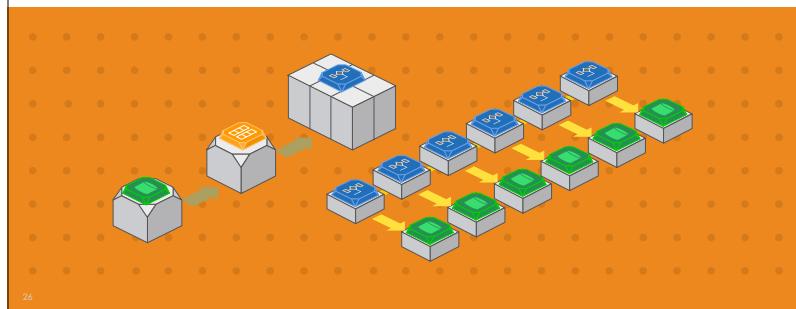
reduce
collect
count
first
take
takeSample
takeOrdered

saveAsTextFile
saveAsSequenceFile
saveAsObjectFile
countByKey
foreach

See <http://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#actions>



An important idea: Function Literals
(also known as Anonymous Functions
and Lambda Functions)
allow us to move code to worker nodes



Function definitions are pretty common in most languages



A TERADATA COMPANY

- A function definition lets us name a stored operation that takes typed arguments and returns a typed result
- Scala can infer result types and even argument types

```
def add(x:Int, y:Int):Int = {           // Takes two Int arguments and returns an Int
    return x + y
}
println(add(42,13))

// Implicit typing and return
def add(x:Int, y:Int) = {                //result type is inferred
    x + y                                // "return" keyword is optional
}

//Curly braces are optional on single line blocks
def add(x:Int, y:Int) = x + y
```

At least two ways exist to define functions and methods



- => is a definition or *transformation* operator; it takes its left hand arguments and transforms them using the right hand expression
- Types can be explicit or implied

```
/ Many ways exist to define functions and methods. Here are several
val even = (i: Int) => { i % 2 == 0 } // explicit long form
val even: (Int) => Boolean = i => { i % 2 == 0 }
val even: Int => Boolean = i => ( i % 2 == 0 )
val even: Int => Boolean = i => i % 2 == 0
val even: Int => Boolean = _ % 2 == 0 // _ means first argument

// implicit result approach
val add = (x: Int, y: Int) => { x + y }
val add = (x: Int, y: Int) => x + y

// explicit result approach
val add: (Int, Int) => Int = (x,y) => ( x + y )
val add: (Int, Int) => Int = (x,y) => x + y
```

Anonymous functions allow us to do without the name



A TERADATA COMPANY

- **Function literals** (a term from Jason Schwartz who wrote the O'Reilly Learning Scala book) are functions defined and passed in-line without giving them a name
- Function literals also go by the names of **anonymous functions** or **lambda functions**

```
// a function with a name greeting
val greeting = (x: String) => "Hello " + x
val names = List("Joe", "Mary", "Barbara")
names.map(greeting)

// should get
//   res2: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)

// Now let's get rid of the name greeting
names.map((x: String) => "Hello " + x)
// should get the same answer:
//   res2: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

Scala provides shorthand representations



- => allows us to define a function quickly
- _ provides a quick argument reference by appearance
 - First _ means first argument, second _ means second argument

```
// Define maximize to compute the maximum of two integers using a function literal
// instead of def. Function literals are also referred to as
// anonymous or lambda functions
val maximize = (a: Int, b: Int) => if (a > b) a else b // Two arguments function literal
maximize(5, 3)

// Define doubler as an immutable variable whose value is a function
// (note we aren't using def)
val doubler = (x: Int) => x * 2      // This assigns a function literal to
                                         // immutable doubler
doubler(4)

// OK, now use the _ placeholder for the doubler function argument
// Read this as 'doubler is defined as a function literal that takes an integer and
// returns an integer; the function definition value multiplies its argument times 2.

val doubler: (Int) => Int = _ * 2
doubler(4)
```

Underscores imply anonymous function arguments



- Assuming the compiler can infer the argument types, you can use an underscore to both define an anonymous function and stand for its first argument
- If there is more than one argument, the arguments are assigned sequentially
- Use with caution; _ means many things in Scala and is very sensitive to context

```
/// Because we can infer the types, we can use the underscore shortcut
// for this function definition
names.map("Hello " + _)

// We can also use this for our even number tester
val nums = Array(1, 2, 3, 4, 5)
nums.map(_ % 2 == 0)

// Should produce
// Array[Boolean] = Array(false, true, false, true, false)
// This may be harder to understand; it's a reducing function
// similar to that used in MapReduce

nums.reduceLeft(_ + _) // first plus second argument
Int = 15
```

Hands-On Lab: Functional Programming Lab 03-spark-transformations-program.{md, scala}



Type the following into your spark-shell

```
val fib = Array(1, 2, 3, 5, 8, 13, 21, 34)
```

Using the REPL, use both named functions and anonymous functions to do the following:

- Compute all the squares
- Return those squares that are divisible by 3

You'll want to use the .map and .filter transformations on fib to invoke your functions

Answers are on the next slide

A large, solid orange circle containing a white question mark, centered on the slide.

Hands-On Lab: Functional Programming



A TERADATA COMPANY

```
// Compute all the squares and sum all the  
values provided  
  
val fib = Array(1, 2, 3, 5, 8, 13, 21, 34)  
  
def square(x: Int) = x * x  
def divisible(x: Int) = (x % 3 == 0)  
  
// First using named functions  
  
fib.map(square).filter(divisible)  
  
// Now use functional literals  
  
fib.map(x => x*x).filter(_ % 3 == 0)  
  
// Result is Array[Int] = Array(9, 441)
```

Persistence lets you reuse RDDs without recomputing them



- Executor memory is limited, so Spark carefully manages it using a least-recently-used (LRU) algorithm.
- Spark caches not only RDDs themselves, but the transformations that created them
- Because it remembers how to recreate every RDD, Spark can destroy them at any time to reclaim memory
- This can require a lot of re-computation if you need to use an RDD more than once
- Spark allows you to mark RDDs that you expect to reuse as **persistent**
- This is very important when doing iterative algorithms

```
rdd.persist()  
rdd.cache()
```

34

Caching is one of the secrets of creating high-performing Spark applications. The cache or persist function (both work) allows you to specify which of your datasets should be retained in memory.

This technique is most useful in iterative programs that reference the same data structure many many times.

However, you should always make sure you time your program with and without explicit caching. Often Spark will do a better job of managing its memory than you will because it can optimize all aspects of the DAG, not just the piece you are looking at.

Also, you should be aware that because of lazy evaluation, you will gain no performance benefit the first reference after your persist command; the first persist is the one that puts the RDD into memory and saves it. However, the second and subsequent references should run faster.

Persistence Comes In Several Forms



Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as serialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than serialized objects, especially when using a fast serializer , but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon . Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory. If you plan to use Tachyon as the off heap store, Spark is compatible with Tachyon out-of-the-box.

35

You have many options in how to persist your RDDs. `MEMORY_ONLY` is the default and is the most common one. In those cases where your dataset is quite large, however, you may wish to specify `MEMORY_ONLY_SER`. Most of the other options are for special cases or where you have long-running computations that you can't afford to re-run in case of a node failure.



Spark Lab: 04-spark-functions-lab.{scala,md}

Wordcount Operations, MapReduce Style



- Split input line on whitespace
- Construct (word, 1) key-value pairs
- Group by key
- Sort by key
- Merge and sort by key ("merge-sort")
- Group values by key
- Sum values

Finally, A common mistake some programmers make is simply copying the functions they used in a MapReduce program over into their Spark code. For example, if we were simply emulating MapReduce wordcount, we'd be tempted to write our code as shown on the next slide.

Wordcount Operations, MapReduce Style



A TERADATA COMPANY

If we translated that directly to Spark we'd get....

```
flatMap( line => line.split("\\s+") )  
map( word => (word, 1) )  
groupByKey(...)  
sortByKey(...)  
repartitionAndSortWithinPartitions...  
aggregateByKey...
```

Don't do this.

Break Out of MapReduce Jail



A TERADATA COMPANY

```
map( line => line.split("\\s+") )  
map( word => (word, 1) )  
groupByKey(...)  
sortByKey(...)  
repartition(1).sortWithinPartitions(...)  
aggregateByKey(...)
```

Don't do this.

Spark's Execution Model Is General Purpose; Take Advantage Of It



```
flatMap( line => line.split("\\s+") )  
map( word => (word, 1) )  
groupByKey(...)  
sortByKey(...)  
repartition( sortWithinPartitions...  
aggregateByKey...  
  
text.flatMap( line => line.split("\\W+") ).  
map( word => (word.toLowerCase(), 1) ).  
reduceByKey( (v1, v2) => v1 + v2 )  
  
// or  
  
text.flatMap( _.split("\\W+") ).  
map(word => (word.toLowerCase(), 1) ).  
reduceByKey(_ + _)
```

40

Don't be afraid to take advantage of Spark's flexibility and rewrite your algorithm with a more flexible DAG.



Spark WordCount Lab: 04-spark-wordcount.{md,scala}

41

This lab is optional -- we touched on Wordcount already in lab 3, but if you want to step through it in detail, this is where to do it.

Summary



A TERADATA COMPANY

- Spark provides a new full-stack cluster development environment
- Resilient distributed datasets (RDDs) provide in-memory storage of data across the cluster instead of spilling to disk
- Transformation operations don't execute until an action is called
- Functional programming allows us to push code to the cluster nodes dynamically

This has been a quick introduction to Spark at the RDD level. It's a new way of thinking about cluster computing and it has huge uptake in the Hadoop community. However, the best is yet to come, as we'll see when we begin working with Spark SQL.



SparkSQL and Dataframes

Spark SQL

1



What We'll Cover



- Overview
- SQLContext and HiveContexts
- Creating data frames
- Dataframe operations
- Running SQL programmatically
- Data sources including interoperation with Hive

History: SparkSQL Replaces Shark



- Hive had/has an execution engine for Spark called Shark
 - set `hive.execution.engine=spark`
 - (please note: this does not work on the EMR Hive because it wasn't compiled for this option)
- However, much of its internals still planned for a MapReduce dataflow
- Instead of just making Shark bigger, the Berkeley people started over again with SparkSQL

Remember All Those Actions and Transformations?



A TERADATA COMPANY

Transformations

```
map  
filter  
flatMap  
mapPartitions  
mapPartitionsWithIndex  
sample  
union  
intersection  
distinct  
cartesian  
pipe  
coalesce  
groupByKey  
reduceByKey  
aggregateByKey  
sortByKey  
join  
cogroup  
keys  
values  
repartition  
repartitionAndSortWithinPartitions
```

Actions

```
reduce  
collect  
count  
first  
take  
takeSample  
takeOrdered  
saveAsTextFile  
saveAsSequenceFile  
saveAsObjectFile  
countByKey  
foreach
```

Aren't they terribly low level?

Wouldn't it be great if we had standard data manipulation operations?

SparkSQL Gives Programmers Higher Level Abstractions for RDDs



DataFrame:
a distributed collection
of data organized into
rows and columns (i.e.,
like relational tables)

DataFrames are easy to work with



A TERADATA COMPANY

- Created from
 - an existing RDD
 - a Hive table
 - data sources
 - code
- Rows are distributed throughout the cluster
- Think of them as an in-memory Hive table

```
// Spark versions 2.0 and later:  
val df = spark.read.json("/data/spark-resources-data/people.json")  
  
// Spark versions before 2.0  
// Existing Spark context is assumed to be in sc  
val sqlContext = new org.apache.spark.sql.SQLContext(sc)  
  
val df = sqlContext.read.json("/data/spark-resources-data/people.json")
```

For Spark Versions 2.0 and Above, Access SQL Operations through your Spark Session



- Accessing Hive tables requires that you copy your Hive config file into the Spark configuration directory, e.g.
`cp /usr/lib/hive/conf/hive-site.xml /usr/lib/spark/conf`
- Spark-shell automatically creates a Spark session object called `spark`.
- Create your own Spark session using the `SparkSession` method.

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```



For Spark version prior to 2.0: SQLContexts and HiveContexts

- SparkSQL and the DataFrame interface have their own contexts
- You can create two types of contexts
 - SQLContext: a context for a private SQL Metastore in Spark
 - HiveContext: a context to access the Hive Metastore
- Accessing a Hive context requires that you copy your Hive config file into the Spark configuration directory, e.g.

```
cp /usr/lib/hive/conf/hive-site.xml /usr/lib/spark/conf
```

```
val sc: SparkContext // An existing SparkContext.  
val sqlContext = new org.apache.spark.sql.SQLContext(sc)  
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

Note: as of Spark 1.6.0, SQL operations produce the same results regardless of whether you use SQLContext or HiveContext

Creating a dataframe from a JSON file



- Just as with Hive, we can create DataFrames from files
- All the normal Hadoop file types are supported
- Simplified ingestion wrappers exist for text, JSON, parquet, and others

```
// Existing Spark session is assumed to be in spark
val df = spark.read.json("/data/spark-resources-data/people.json")
// Displays the content of the DataFrame to stdout
df.show()
```

Creating a dataframe from a Parquet file



A TERADATA COMPANY

- Just as with JSON, we can create DataFrames from Parquet files too

```
val df = spark.read.parquet("/data/spark-resources-data/users.parquet")
df.show()

val s3flights = spark.read.parquet("s3n://thinkbig.academy.aws/ontime.parquet")
```

For Spark 2.0 and Later: Use the Spark Session To apply SQL



- Just as with Hive, we can create DataFrames from files
- All the normal Hadoop file types are supported
- Simplified ingestion wrappers exist for text, JSON, parquet, and others

```
// Existing Spark context is assumed to be in sc
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

spark.sql("USE carl")           // Select my database
val stocks = spark.sql("SELECT * FROM STOCKS")
stocks.show()
```



For Spark versions before 1.6.0: A HiveContext allows us to read Hive files

- Just as with Hive, we can create DataFrames from files
- All the normal Hadoop file types are supported
- Simplified ingestion wrappers exist for text, JSON, parquet, and others

```
// Existing Spark context is assumed to be in sc
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

hiveContext.sql("USE carl")           // Select my database
val stocks = hiveContext.sql("SELECT * FROM STOCKS")
stocks.show()
```

But DataFrames allow us to bypass SQL too



- Most of the typical operations you'd do in SQL are available as dataframe operations
- Doing these operations in Scala allows us to use its DAG optimizer and lazy evaluator and to persist DataFrames we want to keep around
- The following works on a parquet dataset of 160M rows (160GB uncompressed)

```
// Existing Spark session is assumed to be in spark
val s3flights = spark.read.parquet("s3://thinkbig.academy.aws/ontime/parquet")
// Now let's trim that down to 2013 and only some columns

val flights = s3flights.select("year", "month", "dayofmonth", "carrier", "tailnum",
"actuelapsedetime", "origin", "dest", "deptime", "arrdelayminutes").
filter(s3flights("year") === 2013)

val numflights = flights.groupBy("carrier").count // number of flights by carrier
// now average delay by carrier
val avgdelays = flights.groupBy("carrier").mean("arrdelayminutes")
```

13

Lab: 01-sparkSQL-walkthrough.{md,scala}

Are you ready for some SQL?



A TERADATA COMPANY

- SQL statements can be executed directly on Hive tables from Scala using the sql method
- If the Hive table is partitioned, it honors the partitions

```
// Existing Spark session is assumed to be in spark
spark.sql("SELECT name FROM employees WHERE address.zip = 60500").show()

spark.sql("SELECT ymd, price_open, price_close FROM stocks WHERE symbol = 'AAPL' AND
exchg = 'NASDAQ' LIMIT 20").show()

spark.sql("SELECT year(s.ymd), avg(s.price_close) FROM stocks s WHERE s.symbol =
'AAPL' AND s.exchg = 'NASDAQ' GROUP BY year(s.ymd) HAVING avg(s.price_close) NOT
BETWEEN 50.0 AND 100.0").show()
```



The spark-sql shell

- Hive contexts understand HiveQL, but use a different execution engine
- Compare the speed of SparkSQL with Hive running the same queries
- A good way to do this is to run the command spark-sql
- It uses a HiveContext by default if you have Hive configured

```
// from the Unix shell type the following:  
spark-sql  
USE carl;  
SELECT year(s.ymd), avg(s.price_close) FROM stocks s WHERE s.symbol = 'AAPL' AND s.exchg = 'NASDAQ' GROUP BY year(s.ymd) HAVING avg(s.price_close) NOT BETWEEN 50.0 AND 100.0;
```

We can run SQL on DataFrames from RDDs too!



- You can build your own DataFrames from RDDs and query them with SQL statements
- In fact, DataFrames are RDDs

```
case class Person(name: String, age: Long)

/// Create an RDD of Person objects and register it as a table.
val people = spark.read.textFile("hdfs://data/spark-resources-data/people.txt").
  map(_.split(",")).                                         // split by commas
  map(p => Person(p(0), p(1).trim.toInt)). // first field is the person string, second is an integer
  toDF()
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are DataFrames and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

17

Lab: 02-sparkSQL-schemas.{md,scala}

Quick exercise



Using the Scala wordcount as an example, count up the number of words in all of Shakespeare

If you know how to use regular expressions, can you suggest a better way to split up the words in Shakespeare so we don't get all the punctuation?

Can you modify your function to count all words longer than 10 characters? The function length gives you the length of a string.



Copyright © 2011-2017, ThinkBig Analytics. All Rights Reserved

Summary



- SparkSQL gives you the language of HiveQL/SQL with the speed of Spark
- Dataframes with SparkSQL are the underpinnings of most of the high-level application libraries in Spark
- We'll see these abstractions put to use as we explore those applications

– <https://spark.apache.org/docs/latest/>



Big Data Science With Spark

Module 7: Spark Machine Learning Through Examples

Prepared for Elevate

Delivered by Carl Howe, Principal

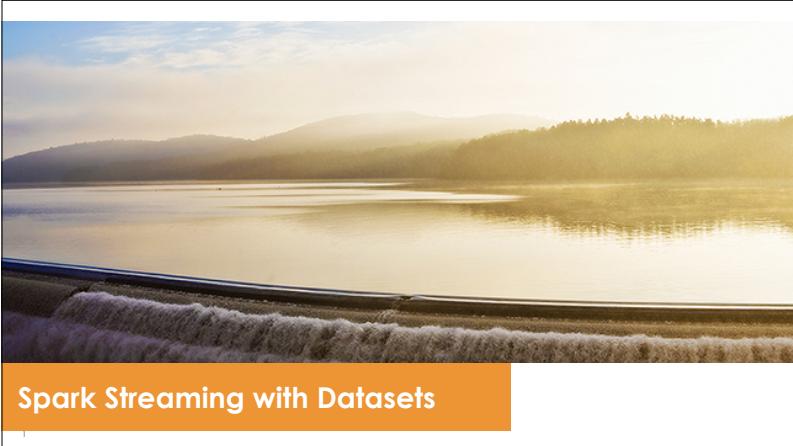
September 5 through 13, 2017



Big Data Science With Spark

Module 8: Spark Streaming

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017

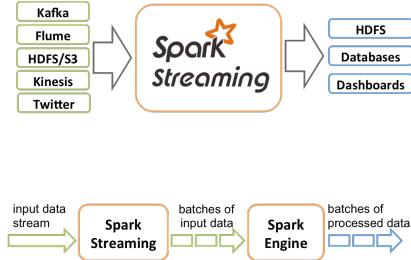


Spark Streaming with Datasets

What is Spark Streaming?



- Extension of Spark core
- Takes data real-time data from a variety of sources
- Writes data to HDFS, databases, and dashboards
- Processes that data in *microbatches*
- Can use Spark Machine Learning and Graph Processing on batches



2

Extension of Spark core

Takes data real-time data from a variety of sources

Writes data to HDFS, databases, and dashboards

Processes that data in microbatches

Can use Spark Machine Learning and Graph Processing on batches

Examples of Streaming Datasets



A TERADATA COMPANY

- Internet of Things (IoT) data
- Credit card transactions
- Files being ingested

Fundamental concepts behind Spark Streaming



- Two ways of handling Streaming
 - RDD-based using DStreams
 - SQL-based using DataSets (Alpha as of Spark 2.1.1)
- Which you choose will depend on data typing and programming model

4

Two ways of handling Streaming

RDD-based using DStreams

SQL-based using DataSets (Alpha as of Spark 2.1.1)

Which you choose will depend on data typing and programming model



DStreams Example (Unstructured Streaming)

Spark Supports Multiple Stream Types



- HDFS directories
- Network connections

Additional advanced sources can be processed in stand-alone programs, but not from the spark-shell

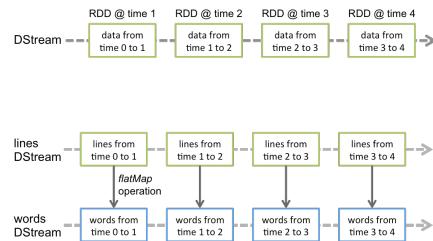
- Kafka
- Flume
- Kinesis

What is a DStream?



A TERADATA COMPANY

- A Discretized Stream (DStream) is a series of RDDs
- Each RDD represents a microbatch of data for a certain time interval
- All operations on DStreams are applied to each microbatch



Spark Streaming Programs Have Two Components



- Receivers pull in data from an external source
- DStreams are created by receivers to create processing pipelines

**The programmer must allocate an executor core
for every Receiver and DStream**

Example: `spark-shell --master local[2]`
allows one Receiver and one DStream

Structure of a DStreams Streaming program



A TERADATA COMPANY

1. Define the input sources by creating input DStreams.
2. Write transformation and output operations for those DStreams.
3. Receive data and process it using `streamingContext.start()`.
4. Wait for the processing to end using
`streamingContext.awaitTermination()`.
5. You can manually stop streaming by calling
`streamingContext.stop()`.

File Streaming Example

A full discussion is in 01-Spark-DStreaming-Files.scala



```
// DStreams Wordcount example from files
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

// Create the context
val ssc = new StreamingContext(sc, Seconds(10))

// Create the FileInputDStream on the directory and use the
// stream to count words in new files created
val lines = ssc.textFileStream("/tmp/streaming-input")
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

Network Streaming



A TERADATA COMPANY

- We can also create DStreams for network ports
- Instead of sc.textFileStream, we use sc.socketTextStream with a socket number
- Everything else remains the same

Example: Ode to Wordcount on a DStream

A full discussion is in 02-Spark-DStreaming-Network.scala



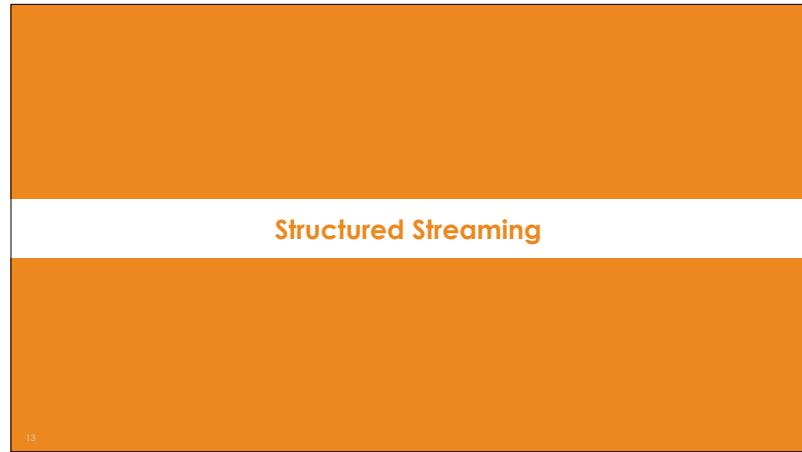
```
// DStreams Wordcount example
import org.apache.spark_
import org.apache.spark.streaming_
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent from a starvation scenario.
val ssc = new StreamingContext(sc, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))

// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
```



Structured Streaming

What is Structured Streaming with Datasets?



- As DStreams are to RDDs, Structured Streaming is to SparkSQL

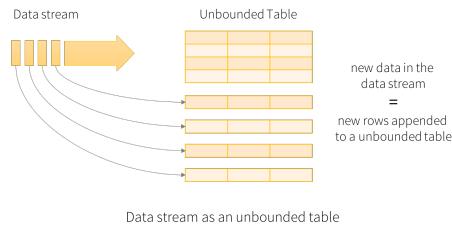
	Spark Core	SparkSQL
Concept	Discretized Streams (DStreams)	Datasets
Non-streaming Analog	RDDs	Dataframes
Example operations	map, flatMap, reduce, filter, union	select, groupBy, join, arbitrary SQL queries
Outputs	RDDs	Files and Tables

Structured Streaming Concept



A TERADATA COMPANY

- A Dataset is simply a DataFrame that can grows with time
- New data from the data stream input appends rows to the table

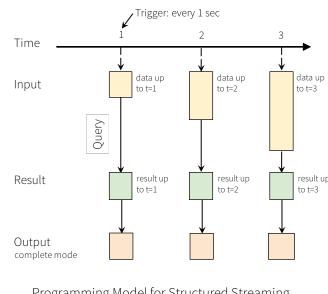


Structured Streaming Concepts



A TERADATA COMPANY

- A query on the input will generate a "Result Table"
- New rows get appended to the input table every interval
- Every interval, the query gets run to update the results table
- Whenever results get updated, we write changed result rows to an external sync



Programming Model for Structured Streaming

Outputs Are What Get Written to External Storage



The output can be defined in a different mode:

- **Complete Mode:** the entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.
- **Append Mode:** Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.
- **Update Mode:** Only the rows that were updated in the Result Table since the last trigger will be written to the external storage (available since Spark 2.1.1). Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger. If the query doesn't contain aggregations, it will be equivalent to Append mode.

Example: Wordcount using Structured Streaming

A full discussion is in 03-Spark-SSstreaming-Network.scala



A TERADATA COMPANY

```
import org.apache.spark.sql.functions._  
import org.apache.spark.sql.SparkSession  
  
// for a standalone Spark program, you'd need  
// val spark = SparkSession  
//   .builder()  
//   .appName("StructuredNetworkWordCount")  
//   .getOrCreate()  
  
import spark.implicits._  
  
// Create DataFrame representing the stream of input lines from connection to localhost:9999  
val lines = spark.readStream.format("socket").option("host", "localhost").option("port", 9999).load()  
  
// Split the lines into words  
val words = lines.as[String].flatMap(_.split(" "))  
  
// Generate running word count  
val wordCounts = words.groupBy("value").count()  
  
// Start running the query that prints the running counts to the console  
val query = wordCounts.writeStream.outputMode("update").format("console").start()  
query.awaitTermination()  
ssc.start() // Start the computation  
ssc.awaitTermination() // Wait for the computation to terminate
```

Structured Streaming Pluses and Minuses



Pluses

Let's developers use SQL to analyze streaming data

Minuses

Datasets

Creates datasets that are easily manipulated

Dataframes

Works well with HDFS and network sources

select, groupBy, join, arbitrary SQL queries

RDDs

Files and Tables

Summary



A TERADATA COMPANY

- Spark does streaming through micro-batching of data
- Spark has two models for streaming: DStreams and Structured Streaming
- At present, DStreams are more mature, but Structured Streaming holds promise for the future.



Big Data Science With Spark

Module 9: Spark In Python

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



Spark in Python

Now that we know how to program Spark in Scala, let's look at the same material in Python. Most people find this easier than Scala because they already know Python.

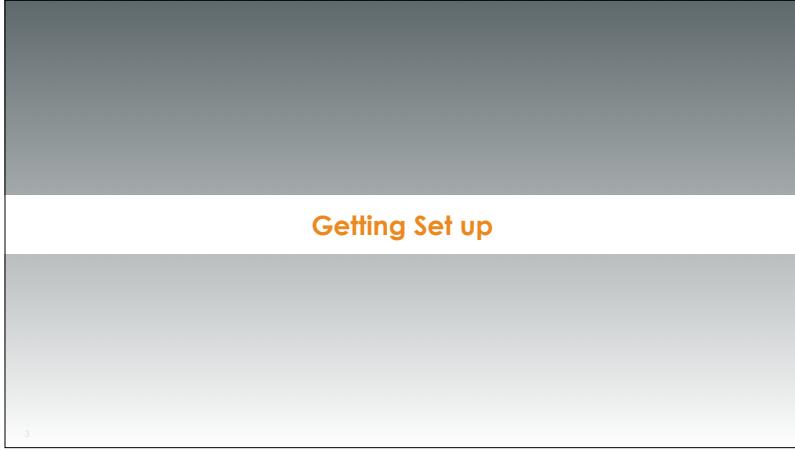


What We'll Cover



- The Spark execution model and Spark Contexts
- The PySpark shell
- Resilient Distributed Datasets (RDDs)
- External datasets
- RDD operations: Transformations and Actions
- Lambda functions
- Persistence
- Some Spark examples

Just as with Scala, we'll cover these fundamental topics.



Getting Set up

As with Scala, we'll get set up first.

How to upgrade Spark software using our flash drive

Vagrant Version



A TERADATA COMPANY

We included some upgrade scripts as part of our file distribution that should now be in /vagrant. You should:

1. Log into your edge node
2. cd /vagrant
3. source ./spark-setup-edge.sh
4. Log into your control node
5. cd /vagrant
6. source ./spark-setup-control.sh

At this point, you should be up and ready to go with Spark 2.2.0 on your cluster. Your SPARK_HOME variable has been set to /vagrant/latest-spark.

If you are using a local two-node VirtualBox cluster created using Vagrant, you'll want to execute a couple of scripts to get your environment set up with the latest version of Spark (Spark 2.2.0)

No Upgrade Required

EMR Version



Your Amazon EMR instance should already be configured with version 2.2.0 of Spark.

Please note that Spark has been installed in /usr/lib/spark.

The shell variable \$SPARK_HOME has not been set by EMR. Should you need to set the variable SPARK_HOME in later exercises, you will want to use the value /usr/lib/spark.

At this point, you should be up and ready to go with Spark 2.2.0 on your cluster.

If you are using Amazon Web Services Elastic Map Reduce instances, your instance should be all set up to use Spark 2.2.0.

If you ever need to do a manual upgrade, do this



The following assumes you have downloaded a version of Spark into your Vagrant directory. In this case, we will assume it to be `latest-spark`

1. Set `SPARK_HOME`: `export SPARK_HOME=/your-spark-installation-location`
(e.g., `/vagrant/latest-spark` or `/mnt/latest-spark`)
2. Add Spark to your `$PATH`: `export PATH=$SPARK_HOME/bin:$PATH`
3. Copy over your `hive-site.xml` to the Spark conf directory:
`cp /etc/hive/conf/hive-site.xml $SPARK_HOME/conf/`
4. Set your `HADOOP_CONF_DIR` variable so that you can use yarn mode:
`export HADOOP_CONF_DIR=/etc/hadoop/conf`
5. (optional) For Spark versions before 2.0, reduce your Spark logging from INFO to WARN.
`sudo sed -i.bak 's/INFO/WARN/' /usr/lib/spark/conf/log4j.properties`
6. (optional) If you are running 2.0 or later, the logging default is WARN and you can adjust this at the command line:
`sc.setLogLevel(newLevel)`

6

One of the best features of Spark is that it is easy to switch to using a different version than the one installed on your cluster. The process really consists of only 4 essential steps:

1. Setting your `SPARK_HOME` shell variable to point at the directory where your Spark installation lives.
2. Adding `$SPARK_HOME/bin` to your execution path.
3. Copying over your `hive-site.xml` file to your Spark configuration directory
4. Setting the shell variable `$HADOOP_CONF_DIR` to point at your Hadoop configuration directory.

If you are using a Spark version prior to version 2, the default is for Spark to log all INFO messages to the console, which can definitely interfere with seeing what your program is doing. You can avoid this issue by setting your logging level to WARN by editing your `log4j.properties` file, as shown in step 5.

If you are running Spark 2.0 or later, the default logging level is WARN. You can change it to other levels, say ERROR, using a simple spark-shell command, `sc.setLogLevel("ERROR")`.

PySpark smoke test



Vagrant version

- On the edge node, type: `pyspark --master yarn`
 - You should see the following:

version 2.2.0

Using Python version 2.6.6 (r266:84292, Jan 22 2014 09:42:36)
SparkSession available as 'spark'.

- Some warnings may appear as well; don't worry about those now

If you're using vagrant, Go to your edge node by typing `vagrant ssh edge`. On the edge node, type: `pyspark`. You should see the following.

PySpark smoke test

EMR version

- On the edge node, type: `pyspark --master yarn`
 - You should see the following:

```
[hadoop@ip-172-31-57-176 hadoop]$ pyspark
Python 2.7.12 (default, Sep 1 2016, 22:14:00)
[GCC 4.8.3 20140911 (Red Hat 4.8.3-9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
17/08/30 23:42:13 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling back to uploading libraries under SPARK_HOME.
Welcome to
    ____          _ _   _ 
   / \ \ \_ \ \ \ \ \ \ \ \ \ \ \ 
  /   \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
 /     \ \ \ \ \ \ \ \ \ \ \ \ \ 
/       \ \ \ \ \ \ \ \ \ \ \ \ 
/         \ \ \ \ \ \ \ \ \ \ \ 
/           \ \ \ \ \ \ \ \ \ 
/             \ \ \ \ \ \ \ 
/               \ \ \ \ \ \ 
/                 \ \ \ \ \ 
/                   \ \ \ \ 
/                     \ \ 
/                       \ 
version 2.2.0

Using Python version 2.7.12 (default, Sep 1 2016 22:14:00)
SparkSession available as 'spark'.
>>>
```

- Some warnings may appear as well; don't worry about those now

8



RDDs, Transforms, and Actions

With setup done, let's jump into the fundamental components of basic Spark: RDDs, transforms, and actions.

Execution Model and Spark Contexts



- Spark programs execute in two different places
 - In a single Spark **driver program** on the master or edge node
 - On **executor nodes** in the cluster
- Driver code runs **serially**
- Task code runs in **parallel**
- Every Spark program creates this environment on startup and references it through an object called the **Spark Context**
- Java Virtual Machines start on cluster worker nodes as Tasks when a Spark Context is created and only die when the driver program ends
- Only one Spark Context can exist in a program

```
spark = SparkSession\  
    .builder()  
    .appName("PythonPi")  
    .getOrCreate()
```

10

Spark programs execute in two different places

1. In a single Spark driver program on the master or edge node
2. On executor nodes in the cluster

Driver code runs serially, while Task code runs in parallel. Every Spark program creates this environment on startup and references it through an object called the Spark Session (or in older versions, a Spark Context).

Java Virtual Machines start on cluster worker nodes as Tasks when a Spark Session is created and only die when the driver program ends

Only one Spark Session or Spark Context can exist in a program.

We can create a spark session using the following Scala chain:

```
val spark = SparkSession  
    .builder()  
    .appName("Spark Example")  
    .config("spark.driver.memory", "3g")
```

The builder method creates the spark session, appName then names that session, and the config method then sets various spark variables. In this case, we're setting the driver memory size to 3 gigabytes. We could similarly set the executor memory sizes using another config statement.

The PySpark shell



A TERADATA COMPANY

- **pyspark** creates a Python interpreter for Spark, along with a context in the variable `sc`
- At the prompt can write Python code and have it interpreted immediately.

```
pyspark
Python 2.7.12 (default, Sep 1 2016, 22:14:00)
[GCC 4.8.3 20140911 (Red Hat 4.8.3-9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
17/08/30 23:42:13 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling back to
uploading libraries under SPARK_HOME.
Welcome to
   ____          _ _   _ 
  / \ \    / \ / \ / \ / \ 
 /   \  /   \  /   \  /   \
/     \ /     \ /     \ /     \
\     / \     / \     / \     /
 \ / \ / \ / \ / \ / \ / \ / \ 
   version 2.2.0

Using Python version 2.7.12 (default, Sep 1 2016 22:14:00)
SparkSession available as 'spark'.
>>> shakes = sc.textFile("hdfs:///data/shakespeare/input/")
>>> shakes.take(1)
[u't1 KING HENRY IV']
```

11

Options for running Python with Spark



A TERADATA COMPANY

To run bin/pyspark locally on exactly four cores, use:

```
$ ./bin/pyspark --master local[4]
```

Or, to also add code.py to the search path (in order to later be able to import code), use:

```
$ ./bin/pyspark --master local[4] --py-files code.py
```

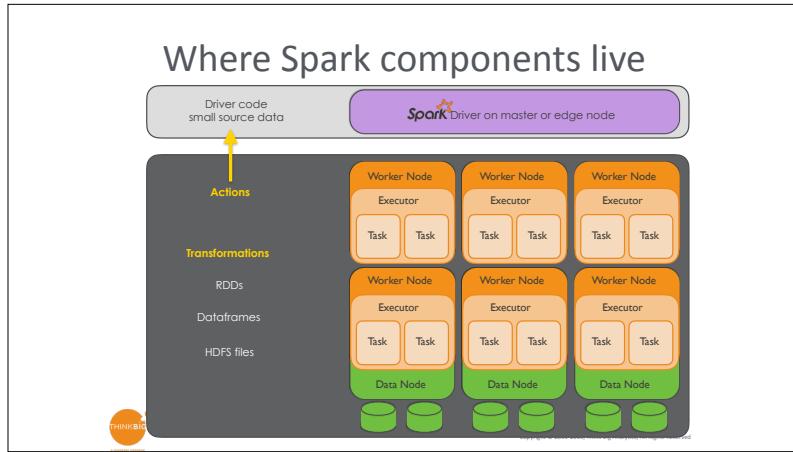
For a complete list of options, run pyspark --help. Behind the scenes, pyspark invokes the more general spark-submit script.

It is also possible to launch the PySpark shell in IPython, the enhanced Python interpreter. PySpark works with IPython 1.0.0 and later. To use IPython, set the PYSPARK_DRIVER_PYTHON variable to ipython when running bin/pyspark:

```
$ PYSPARK_DRIVER_PYTHON=ipython ./bin/pyspark
```

You can customize the ipython command by setting PYSPARK_DRIVER_PYTHON_OPTS. For example, to launch the IPython Notebook with PyLab plot support:

```
$ PYSPARK_DRIVER_PYTHON=ipython PYSPARK_DRIVER_PYTHON_OPTS="notebook" ./bin/pyspark
```



If we look at this visually, the light gray box represents the edge node, depending on your cluster configuration. That's where the Spark driver lives. It runs serially there on one node.

In the dark gray box, we have the rest of the cluster data and worker nodes. That's where HDFS files are distributed over the disks connected to the data nodes. It's also where Spark DataFrames and RDDs exist.

Transformations are Spark methods that execute in parallel on the cluster. They run in parallel.

Actions are Spark methods that take data from the cluster and bring that data back to the driver. Because they must bring the data back to the driver, they force all the nodes involved to synchronize their efforts. They create a momentary serialization.



Walkthrough: 01-pyspark-basics-lab

14

Let's go hands-on with Spark with our first Python lab. Two versions exist: one in Markdown and one in pure Python code. We recommend you bring up one of those versions in a regular text editor (i.e., Notepad on Windows PCs,TextEdit or your favorite development text editor on a Mac) to view the text while having your terminal window along side it for typing commands.

Hands-On Lab: First Scala Program: Lab 02-python-first-program.md



We saw in our walkthrough that we count the number of stock quotes for AAPL using code that looks like this:

```
rdd = sc.textFile("hdfs:///data/stocks-flat/input")
aapl = rdd.filter( line => line.contains("AAPL") )
aapl.count
```

Try using that pattern to count the number of lines in the file /data/shakespeare/input that contain the word "king". You'll want to use the `toLowerCase` function before the `contains` function to ensure all the text is lower case.

A solution is on the next slide or in the lab script

Hands-On Lab: First Scala Program



A TERADATA COMPANY

One solution looks like this

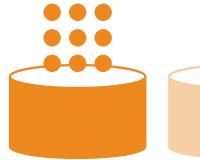
```
rdd = sc.textFile("hdfs:///data/shakespeare/input")
kings = rdd.filter(lambda line: "king" in line.lower())
kings.count()
```

You should have gotten 4773 lines that reference a king in all of Shakespeare. That's almost 3% of the 175,376 lines in all of Shakespeare's plays and poems.



Resilient Distributed Datasets (RDDs) are the primary parallel data structure

- Data is represented as Resilient Distributed Datasets (“RDDs”)
 - Can be created on the fly in the driver (e.g., `parallelize()`)
 - Could be a file on disk, HDFS path, result set, etc.
 - RDDs are immutable and cannot be modified once created
- New RDDs can be created from existing ones via **Transformations** which are executed **lazily**
- **Actions** compute a result based on an RDD. The result can be returned to the driver environment or saved to disk or elsewhere



17

Data is represented as Resilient Distributed Datasets (“RDDs”)

Can be created on the fly in the driver (e.g., `parallelize()`)

Could be a file on disk, HDFS path, result set, etc.

RDDs are immutable and cannot be modified once created

New RDDs can be created from existing ones via Transformations which are executed lazily

Actions compute a result based on an RDD. The result can be returned to the driver environment or saved to disk or elsewhere

Parallelized collections create RDDs



- The transformation `parallelize` turns a local collection into an RDD
- Once in an RDD, that parallelized collection can be operated on in parallel
- You can control the number of partitions that `parallelize` uses through a second argument, e.g., `sc.parallelize(data, 10)`
- Default number of partitions used is automatically set by your cluster

```
data = [1, 2, 3, 4, 5]           # driver Array
distData = sc.parallelize(data)    # Transformation into RDD
distData.take(3)                 # Action: take first 3 elements
[1, 2, 3]
```

The transformation `parallelize` turns a local collection into an RDD

Once in an RDD, that parallelized collection can be operated on in parallel

You can control the number of partitions that `parallelize` uses through a second argument, e.g.,
`sc.parallelize(data, 10)`

Default number of partitions used is automatically set by your cluster

More commonly, file inputs create RDDs



- Spark can create RDDs from any storage source supported by Hadoop
 - Local file system, HDFS, Cassandra, HBase, Amazon S3
 - If using local file systems, the file must be replicated or shared on all worker nodes
 - Spark happily reads directories, compressed files, and paths specified by regular expressions
 - Like parallelize, textFile and its friends accept a second argument with the number of partitions to parallelize the input across

```
>>> shakes = sc.textFile("hdfs:///data/shakespeare/input")
>>> shakes.take(1)
[u'\t1 KING HENRY IV']
>>>
```

19

Spark can create RDDs from any storage source supported by Hadoop

Local file system, HDFS, Cassandra, HBase, Amazon S3

If using local file systems, the file must be replicated or shared on all worker nodes

Spark happily reads directories, compressed files, and paths specified by regular expressions

Like parallelize, textFile and its friends accept a second argument with the number of partitions to parallelize the input across

RDD Operations: Transformations and Actions



- **Transformations** convert an RDD into another RDD
- **Actions** often turn an RDD into something else
- Only **actions** cause evaluation

```
lines = sc.textFile("hdfs:///data/shakespeare/input")      # Transformation
lineLengths = lines.map(lambda s: len(s))                 # Transformation
totalLength = lineLengths.reduce(lambda a, b: a + b)       # Action
totalLength    # print result
# should be 5167385
```

Transformations convert an RDD into another RDD

Actions often turn an RDD into something else

Only actions cause evaluation

Common Transformations

```
Assume: rdd = sc.parallelize(["This is a line of text", "And so is this"])
```

map — apply a function while preserving structure

```
rdd.map( lambda line: line.split() ).collect()  
+ [['This', 'is', 'a', 'line', 'of', 'text'], ['And', 'so', 'is', 'this']]
```

flatMap — apply a function while flattening structure

```
rdd.flatMap( lambda line: line.split()).collect()  
+ ['This', 'is', 'a', 'line', 'of', 'text', 'And', 'so', 'is', 'this']
```

filter — discard elements from an RDD which don't match a condition

```
rdd.filter( lambda line: "so" in line ).collect()  
+ ['And so is this']
```



Common Transformations

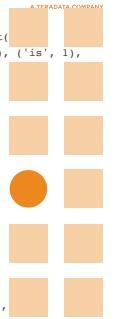
```
pair_rdd = rdd.flatMap(lambda line: line.split()).map(lambda word: (word.lower(), 1)).collect()  
+ [('this', 1), ('is', 1), ('a', 1), ('line', 1), ('of', 1), ('text', 1), ('and', 1), ('so', 1), ('is', 1),  
('this', 1)]
```

groupByKey — group values by key

```
pair_rdd.groupByKey().collect()  
+ [('a', <py4j.java_gateway.ResultIteratorable object at 0x20f7e50>), ('and',  
<py4j.java_gateway.ResultIteratorable object at 0x1fa05d0>), ('this',  
<py4j.java_gateway.ResultIteratorable object at 0x20f7b10>), ('is',  
<py4j.java_gateway.ResultIteratorable object at 0x1fd20d0>), ('so',  
<py4j.java_gateway.ResultIteratorable object at 0x1fd2190>), ('line',  
<py4j.java_gateway.ResultIteratorable object at 0x20d6e90>), ('text',  
<py4j.java_gateway.ResultIteratorable object at 0x20d6e10>), ('of',  
<py4j.java_gateway.ResultIteratorable object at 0x20459d0>)]
```

reduceByKey — apply a function to each value for each key

```
pair_rdd.reduceByKey(lambda v1, v2: v1 + v2).collect()  
+ [('a', 1), ('and', 1), ('this', 2), ('is', 2), ('line', 1), ('text', 1), ('of', 1)]
```



Spark Transformations



A TERADATA COMPANY

• map

• filter

• flatMap

• mapPartitions

• mapPartitionsWithIndex

• sample

• union

• intersection

• distinct

• cartesian

• pipe

• coalesce

For PairRDD:

• groupByKey

• reduceByKey

• aggregateByKey

• sortByKey

• join

• cogroup

• keys

• values

• repartition

• repartitionAndSortWithinPartitions

see <http://spark.apache.org/docs/2.2.0/programming-guide.html#transformations>

Common Spark Actions

- **collect** — gather results from nodes and return
- **first** — return the first element of the RDD
- **take(N)** — return the first N elements of the RDD
- **saveAsTextFile** — write the RDD as a text file
- **saveAsSequenceFile** — write the RDD as a SequenceFile
- **count** — count elements in the RDD
- **countByKey** — count elements in the RDD by key
- **foreach** — process each element of an RDD
 - (e.g., `rdd.collect.foreach{println}`)

See <http://spark.apache.org/docs/2.2.0/programming-guide.html#actions>



Spark Actions

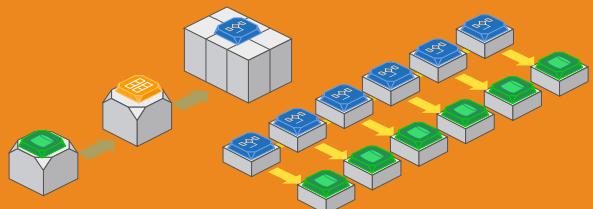
- reduce
 - collect
 - count
 - first
 - take
 - takeSample
 - takeOrdered
-
- saveAsTextFile
 - saveAsSequenceFile
 - saveAsObjectFile
 - countByKey
 - foreach

See <http://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#actions>

25



An important idea: Function Literals
AKA Anonymous Functions
AKA Lambda Functions
allow us to move code to worker nodes



Function definitions are pretty common in most languages



A TERADATA COMPANY

- A function definition lets us name a stored operation that takes arguments and returns a result
- Python function result types are determined by the types of the arguments and the function operations

```
def add(x, y):
    return x + y

add(42,13)

# Shorter version all on one line
def add(x, y): return x + y
```

Anonymous functions allow us to do without the name



- **Function literals** are functions defined and passed in-line without giving them a name
- Function literals also go by the names of **anonymous functions** or **lambda functions**

```
# a named function greeting
def greeting(x): return "Hello " + x
greeting("Joe")

# an anonymous function whose definition is assigned to variable greeting
greeting = lambda x: "Hello " + x
greeting("Joe")
```

Anonymous functions allow us to do without the name



- Either type of function can be used for Spark functions

```
names = ["Joe", "Mary", "Barbara"]
# comprehension for a local list; local lists don't have the map method defined.
[greeting(x) for x in names]

# Now do this in Spark
names = sc.parallelize(["Joe", "Mary", "Barbara"])
names.map(greeting).collect()

# should get
#   ['Hello Joe', 'Hello Mary', 'Hello Barbara']

# Now let's get rid of the name greeting

names.map(lambda x: "Hello " + x).collect()
# should get the same answer:
#   ['Hello Joe', 'Hello Mary', 'Hello Barbara']
```

Hands-On Lab: Functional Programming Lab 03-python-transformations-program.{md, scala}



Type the following into pyspark

```
fib = sc.parallelize([1, 2, 3, 5, 8, 13, 21, 34])  
Using the REPL, use both named functions  
and anonymous functions to do the following:
```

- Compute all the squares
- Return those squares that are divisible by 3

You'll want to use the .map and .filter
transformations on fib to invoke your
functions

Answers are on the next slide

A large, solid orange circle containing a white question mark, centered on the slide.

Hands-On Lab: Functional Programming



A TERADATA COMPANY

```
# Compute all the squares and sum all the values
# provided
fib = sc.parallelize([1, 2, 3, 5, 8, 13, 21, 34])

def square(x):
    return(x * x)
def divisible(x):
    return(x % 3 == 0)

# First using named functions
fib.map(square).filter(divisible).collect()
[9, 441]
# Now use functional literals

fib.map(lambda x: x*x).filter(lambda x: x % 3 ==
0).collect()

# Result is Array[Int] = Array(9, 441)
[9, 441]
```

31





Persistence lets you reuse RDDs without recomputing them

- Spark carefully manages its memory
- Spark caches not only RDDs themselves, but the transformations that created them
- Because it remembers how to recreate every RDD, Spark can destroy them at any time
- This can require a lot of re-computation if you need to use an RDD more than once
- Spark allows you to mark RDDs that you expect to reuse as **persistent**
- This is very important when doing iterative algorithms

```
# persist an RDD by calling persist or cache on it, e.g.,  
rdd.persist()  
or  
rdd.cache()
```

32

Caching is one of the secrets of creating high-performing Spark applications. The cache or persist function (both work) allows you to specify which of your datasets should be retained in memory.

This technique is most useful in iterative programs that reference the same data structure many many times.

However, you should always make sure you time your program with and without explicit caching. Often Spark will do a better job of managing its memory than you will because it can optimize all aspects of the DAG, not just the piece you are looking at.

Also, you should be aware that because of lazy evaluation, you will gain no performance benefit the first reference after your persist command; the first persist is the one that puts the RDD into memory and saves it. However, the second and subsequent references should run faster.

Persistence Comes In Several Forms



Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it effective in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory. If you plan to use Tachyon as the off heap store, Spark is compatible with Tachyon out-of-the-box.

33

You have many options in how to persist your RDDs. `MEMORY_ONLY` is the default and is the most common one. In those cases where your dataset is quite large, however, you may wish to specify `MEMORY_ONLY_SER`. Most of the other options are for special cases or where you have long-running computations that you can't afford to re-run in case of a node failure.



Spark Lab: 04-python-functions-lab.{scala,md}

Wordcount operations



A TERADATA COMPANY

- Split input line on whitespace
- Construct $(word, 1)$ key-value pairs
- Group by key
- Sort by key
- Merge and sort by key ("merge-sort")
- Group values by key
- Sum values

Don't translate 1:1 from MapReduce

```
flatMap( lambda line: line.split() )
map( lambda word: (word, 1) )
groupByKey(..)
sortByKey(..)
repartitionAndSortWithinPartitions
aggregateByKey..
```

```
# Use this instead
text_file = sc.textFile("hdfs:///data/shakespeare/input")
word_counts = text_file \
.flatMap(lambda line: line.split()) \
.map(lambda word: (word, 1)) \
.reduceByKey(lambda a, b: a + b)
```



A TERADATA COMPANY

Don't be afraid to take advantage of Spark's flexibility and rewrite your algorithm with a more flexible DAG.

Summary



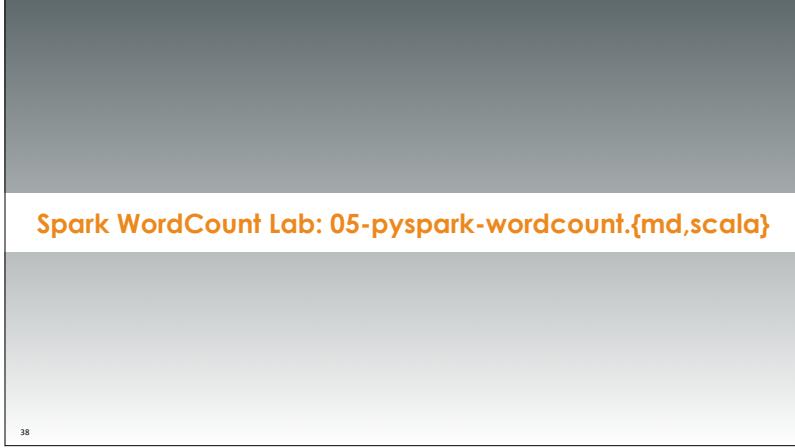
- Spark provides a new full-stack cluster development environment
- Resilient distributed datasets (RDDs) provide in-memory storage of data across the cluster instead of spilling to disk
- Transformation operations don't execute until an action is called
- Functional programming allows us to push code to the cluster nodes dynamically

However.....

SparkSQL and Dataframes are how you will usually want to use Spark



Copyright © 2011-2017, ThinkBig Analytics. All Rights Reserved



Spark WordCount Lab: 05-pyspark-wordcount.{md,scala}

38

This lab is optional -- we touched on Wordcount already in lab 3, but if you want to step through it in detail, this is where to do it.

Summary



A TERADATA COMPANY

- Pyspark and Python on Spark provide a similar experience to Scala, but without the type anxiety
- Most of what you've learned in Scala is directly transferrable, provided you remember to use lambdas instead of Scala's shortcuts
- Be sure to check the Python Spark API documentation for differences with Scala



Big Data Science With Spark

Module 10: Spark In R

Prepared for Elevate
Delivered by Carl Howe, Principal
September 5 through 13, 2017



Spark In 

1

For vagrant clusters



A TERADATA COMPANY

Run these scripts in on your edge node and control node respectively. Each may take as much as 30-40 minutes to run, so start now.

- install-r-studio-edge.sh*: This script installs both R and RStudio Server on the edge node. Once it has been run, you should be able to get to RStudio at the address <http://edge:8787> and to log in with user name **vagrant** and password **vagrant**. Because of the need to download all the packages and compile them, this will take almost an hour to run.
- install-r-control.sh*: We don't need RStudio server on the control node, so this script only installs R and the packages. It won't take quite as long as the edge node, but still requires some time to run.



Agenda



A TERADATA COMPANY

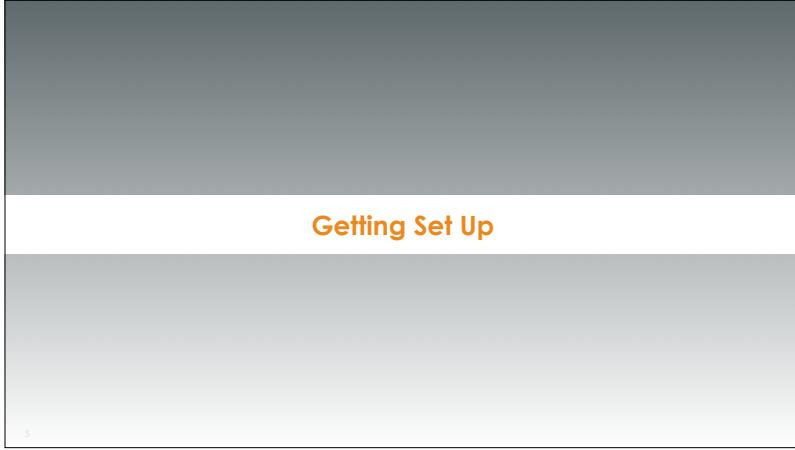
- What's different about Spark when using R
- Getting set up
- Adapting to the SparkR API
- Rethinking the Spark interface
- Comparisons with Scala and Python interfaces

What's Different about R on Spark?



A TERADATA COMPANY

- Think Data Science instead of Data Engineering
 - No RDD API
 - High level operations focused on DataFrames and SQL
 - Easy access to Machine Learning libraries
- Multiple libraries with which to use Spark
 - SparkR
 - sparklyr



Getting Set Up

If you ever need to do a manual upgrade, you do this



The following assumes you have downloaded a version of Spark into your Vagrant directory. In this case, we will assume it to be spark-2.0.1-bin-hadoop2.6

1. **Set SPARK_HOME:** export SPARK_HOME=/vagrant/spark-2.0.1-bin-hadoop2.6
2. **Add Spark to your path:** export PATH=\$SPARK_HOME/bin:\$PATH
3. **Copy over your hive-site.xml to the Spark conf directory:**
cp /etc/hive/conf/hive-site.xml \$SPARK_HOME/conf/
4. **Set your HADOOP_CONF_DIR variable so that you can use yarn mode:**
export HADOOP_CONF_DIR=/etc/hadoop/conf
5. **(optional) For Spark versions before 2.0, reduce your Spark logging from INFO to WARN.**
sudo sed -i.bak 's/INFO/WARN/' /usr/lib/spark/conf/log4j.properties
6. **(optional) If you are running 2.0 or later, the logging default is WARN and you can adjust this at the command line:**
sc.setLogLevel(newLevel)

Spark R smoke test

- On the edge node, type: `sparkR --master yarn`
- You should see the following:



A TERADATA COMPANY

```
[vagrant@edge vagrant]$ sparkR --master yarn
R version 3.3.3 (2017-03-06) -- "Another Orange"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Loading java with spark-submit command /vagrant/spark-2.0.1-bin-hadoop2.6/bin/spark-submit "sparkr-shell" /tmp/RtmpFBYVP6/
backend_port5222fd1cb
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).

Welcome to
   _/\_ 
  / \ \_ 
 /   \ \_ 
 \   / \_ 
  \ / \_ 
   \_ \_ 
version 2.2.0

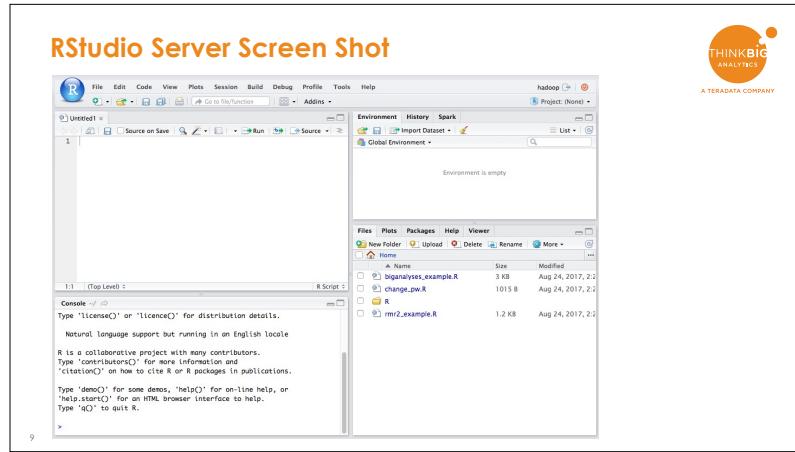
7 > SparkSession available as 'spark'.
```

7

RStudio Server: Point Your Browser To Port 8787



- The RStudio Server Integrated Development Environment is available as a Web page at \$MY_IP_ADDRESS:8787
- Log in using user name hadoop, password hadoop
- You should see what's shown on the next page.



RStudio Server Environment Variables



RStudio Server doesn't necessarily have all the necessary Spark environment variables, so you'll have to start your programs with the following preamble

```
if (nchar(Sys.getenv("SPARK_HOME")) < 1) {  
  Sys.setenv(SPARK_HOME = "/usr/lib/spark")      # or wherever your Spark  
  install lives  
}  
if (nchar(Sys.getenv("HADOOP_CONF_DIR")) < 1) {  
  Sys.setenv(HADOOP_CONF_DIR = "/etc/hadoop/conf")    # or wherever your Hadoop  
  lives  
}  
if (nchar(Sys.getenv("YARN_CONF_DIR")) < 1) {  
  Sys.setenv(YARN_CONF_DIR = "/etc/hadoop/conf")      # or wherever your YARN config  
  lives  
}  
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))  
sparkR.session(master = "yarn", sparkConfig = list(spark.driver.memory = "3g"))
```

10 sparkR.session is what creates your Spark context for you. :

To Run under RStudio Server, Go To Port 8787



A TERADATA COMPANY

To run sparkR under RStudio Server, simply run these commands after pointing your browser to port 8787:

```
if (nchar(Sys.getenv("SPARK_HOME")) < 1) {  
  Sys.setenv(SPARK_HOME = "/usr/lib/spark")      # or wherever your Spark  
  install lives  
}  
if (nchar(Sys.getenv("HADOOP_CONF_DIR")) < 1) {  
  Sys.setenv(HADOOP_CONF_DIR = "/etc/hadoop/conf")    # or wherever your Hadoop  
  lives  
}  
if (nchar(Sys.getenv("YARN_CONF_DIR")) < 1) {  
  Sys.setenv(YARN_CONF_DIR = "/etc/hadoop/conf")      # or wherever your YARN config  
  lives  
}  
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))  
sparkR.session(master = "yarn", sparkConfig = list(spark.driver.memory = "3g"))
```

sparkR.session is what creates your Spark context for you. :

11

Comparing DataFrames in Scala, Python, and R



```
val df = spark.read.json("examples/src/main/resources/people.json")
// Displays the content of the DataFrame to stdout
df.show()
```

This is the Python version

```
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
```

And here's R

```
df <- read.df("examples/src/main/resources/people.json", "json")
# Displays the content of the DataFrame to stdout
showDF(df)
```

Copyright © 2012-2017, Think Big Analytics, All Rights Reserved

Comparing SQL queries in Scala, Python, and R



```
val df = spark.sql("SELECT * FROM STOCKS")
// Displays the content of the DataFrame to stdout
df.show()
```

This is the Python version

```
df = spark.sql("SELECT * FROM STOCKS")
# Displays the content of the DataFrame to stdout
df.show()
```

And here's R

```
df <- sql("SELECT * FROM STOCKS")
# Displays the content of the DataFrame to stdout
showDF(df)
```

What happened to RDDs?



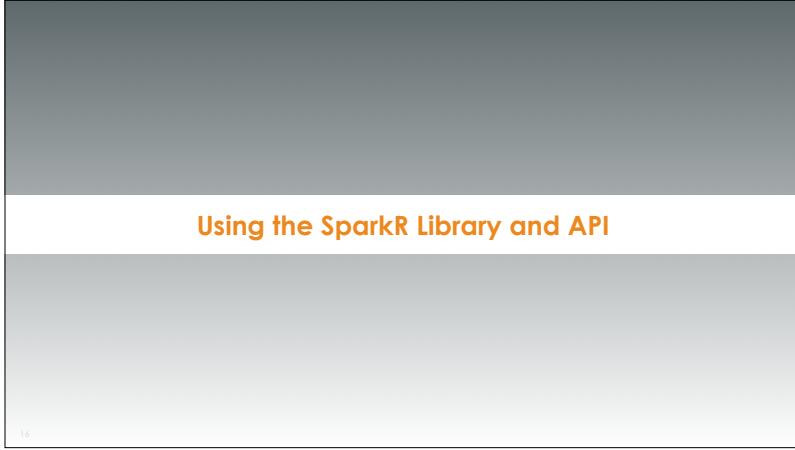
- Spark's R API doesn't support RDDs because they are too low level
- R focuses its API on Datasets, DataFrames, and other higher-level objects

You Have Two Ways To Use Spark From R



- ▶ SparkR: Library that uses R interfaces to the same Spark APIs used by Scala
- ▶ sparklyr: Library that translates dplyr operations to SQL statements operated by the cluster

We'll look at the first, which is more common. We'll then walk through the second with in code.



Using the SparkR Library and API

The SparkR API



- Provides most of the same grouping and aggregation functions most R programmers are familiar with in packages such as `plyr` and `dplyr`.
- Some of the functions are spelled differently, so pay close attention to capitalization, underscores, and dots in the examples.

You Must Distinguish Two Types of Dataframes



A TERADATA COMPANY

- R natively supports a **data.frame** type, which is implemented as a list of columns.
- SparkR supports a different dataframe type called a **DataFrame**, which is implemented within the Spark Cluster as an RDD of Rows.
- You must use different functions for the two different types!

```
head(faithful)
##   eruptions waiting
## 1      3.600     79
## 2      1.800     54
## 3      3.333     74
## 4      2.283     62
## 5      4.533     85
## 6      2.883     55

str(faithful)
## 'data.frame': 272 obs. of  2 variables:
##   $ eruptions: num  3.6 1.8 3.33 2.28 4.53 ...
##   $ waiting  : num  79 54 74 62 85 55 88 85 51 85 ...

df <- as.DataFrame(faithful) # note this is
# as.DataFrame, not as.data.frame
# Displays the first part of the SparkDataFrame
head(df)
## you see the same numbers but it takes longer.
##   eruptions waiting
##1      3.600     79
##2      1.800     54
##3      3.333     74
```

Reading in DataFrames



A TERADATA COMPANY

- `read.df` is the primary function to read a DataFrame
- Accepts a variety of formats, including
 - CSV
 - JSON
 - ORC
 - Parquet
- If you just want rows of text, you can use `read.text`

```
df <- read.df("/data/flightdata/parquet-trimmed/", "parquet")
```

Actual sparkR code



A TERADATA COMPANY

This is all a lot like SparkSQL in Scala, but with different syntax

```
> df <- read.df("/data/spark-resources-data/people.json", "json")
17/07/20 10:26:36 WARN datasource.DataSource: Error while looking for metadata directory.
> showDF(df)
+-----+
| age   name|
+-----+
| null|Michael|
| 30  | Andy|
| 19  | Justin|
+-----+
> printSchema(df)
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
> names(df)
[1] "age"  "name"
> df %>% select("name") %>% showDF()
+---+
| name|
+---+
| Michael|
| Andy|
| Justin|
+---+
```

Library magrittr gives you a pipeline operator
similar to how we would use "." in Python or Scala



```
library(magrittr)
> df %>% select(df$name, df$age + 1) %>% showDF()
+-----+
|   name|(age + 1.0)|
+-----+
| Michael|      null|
|   Andy|      31.0|
| Justin|      20.0|
+-----+-----+
```



```
> df %>% filter(df$age > 21) %>% showDF()
+-----+
| age|name|
+-----+
| 30|Andy|
+-----+
```



```
> df %>% groupBy("age") %>% count() %>% showDF()
+-----+
| age|count|
+-----+
|  0|     1|
| 19|     1|
| null|    1|
| 30|     1|
+-----+
```

**WARNING: Don't be fooled into thinking that
these are dplyr operations**

These are SparkR Functions



A TERADATA COMPANY

A screenshot of an RStudio interface. On the left is a sidebar with a tree view of available functions. In the center is the main workspace, which shows the R documentation for the 'filter' function. The title 'filter [SparkR]' is at the top. Below it are sections for 'DESCRIPTION', 'USAGE', and 'ARGUMENTS'. The 'DESCRIPTION' section says 'Filter the rows of a SparkDataFrame according to a given condition.' The 'USAGE' section shows two examples: 'filter(x, condition)' and 'filter(x, condition)'. The 'ARGUMENTS' section describes 'x' as 'A SparkDataFrame to be sorted.' and 'condition' as 'The condition to filter on. This may either be a Column expression or a string containing a SQL statement.' A red oval highlights the 'filter' function in both the sidebar and the main documentation area.

filter [SparkR]

DESCRIPTION

Filter the rows of a SparkDataFrame according to a given condition.

USAGE

```
## S4 method for signature 'SparkDataFrame,characterOrColumn'
filter(x, condition)
```

```
## S4 method for signature 'SparkDataFrame,characterOrColumn'
where(x, condition)
```

filter(x, condition)

where(x, condition)

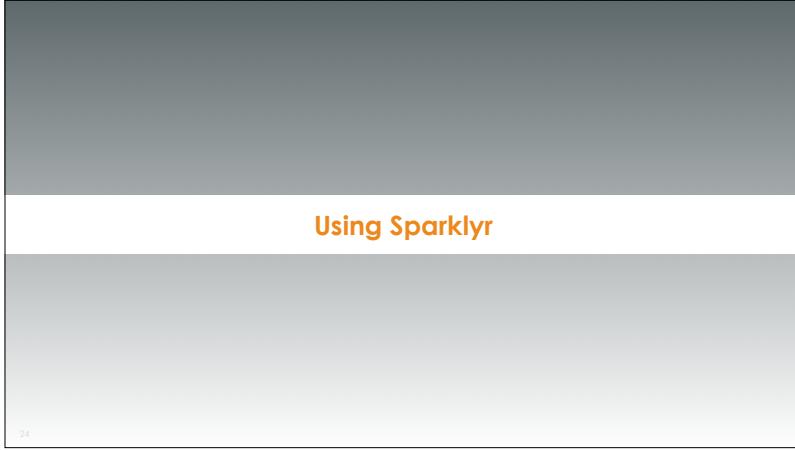
ARGUMENTS

x A SparkDataFrame to be sorted.

condition The condition to filter on. This may either be a Column expression or a string containing a SQL statement

Lab: 02-SparkR-API.{md,R}





Using Sparklyr

Sparklyr Hides Spark Behind dplyr Primitives



- Developed by the RStudio team
- Provides a complete `dplyr` backend that connects to Spark
- Filters and aggregates Spark DataSets and brings them into R for analysis and visualization.
- Allows use Spark's of distributed machine learning library from R.
- Allows extensions that call the full Spark API and provide interfaces to Spark packages.
- Still relatively early in its maturity

Example sparklyr code



- Sparklyr works as a back end to `dplyr`.
- Acts as if it is part of the Hadley Wickham *tidyverse* packages for Data Science
- Once your Spark connection is set up, it mostly just looks like `dplyr` pipelines
- However, due to lazy evaluation, you must *collect* your results before printing or graphing them.

```
library(sparklyr)
sc <- spark_connect(master = "yarn", version = "2.2.0")
install.packages(c("nycflights13"))
library(dplyr)
# Copy nycflights to cluster
flights_tbl <- copy_to(sc, nycflights13::flights,
"flights")
flights_tbl %>% filter(dep_delay == 2) %>% head()
delay <- flights_tbl %>%
  group_by(tailnum) %>%
  summarise(count = n(), dist = mean(distance), delay =
mean(arr_delay)) %>%
  filter(count > 20, dist < 2000, !is.na(delay)) %>%
  collect
```

Sparklyr Supports Machine Learning Too





Summary



- Python and R environments for Spark exist and work
- Many of the same principles you've learned in Scala work similarly in these languages
- The DataFrame API is more likely to be compatible across languages than the RDD one
- All non-Scala languages have some "gotchas" in either functionality or performance