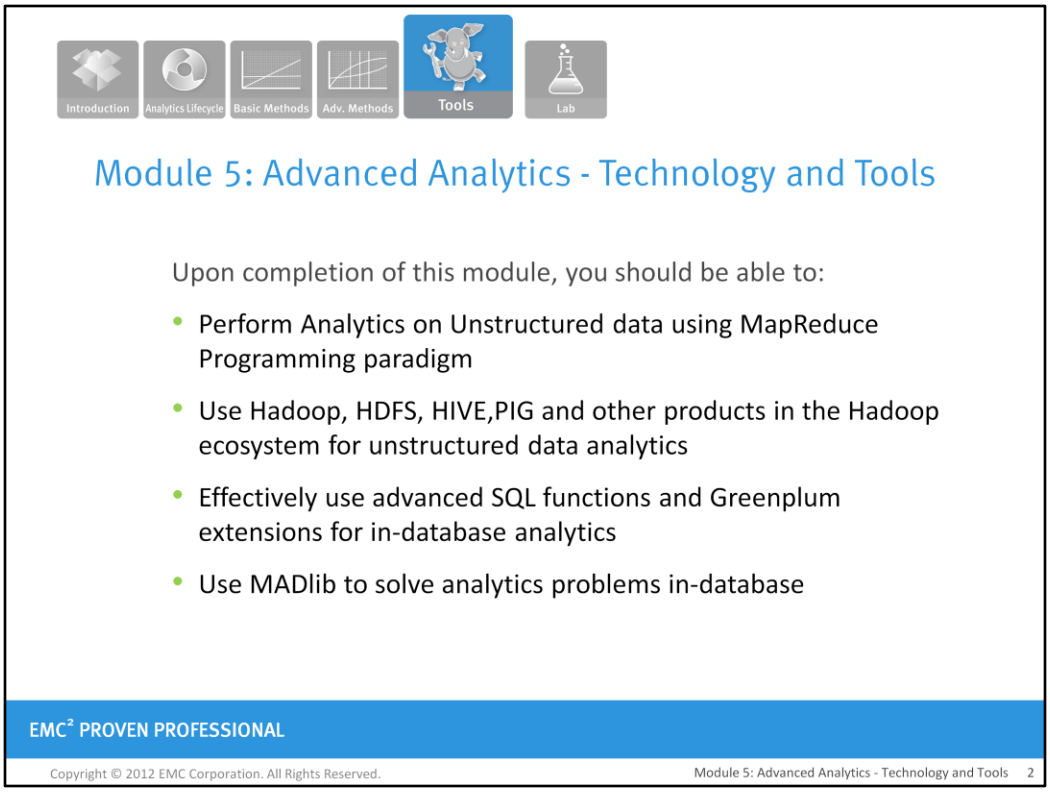
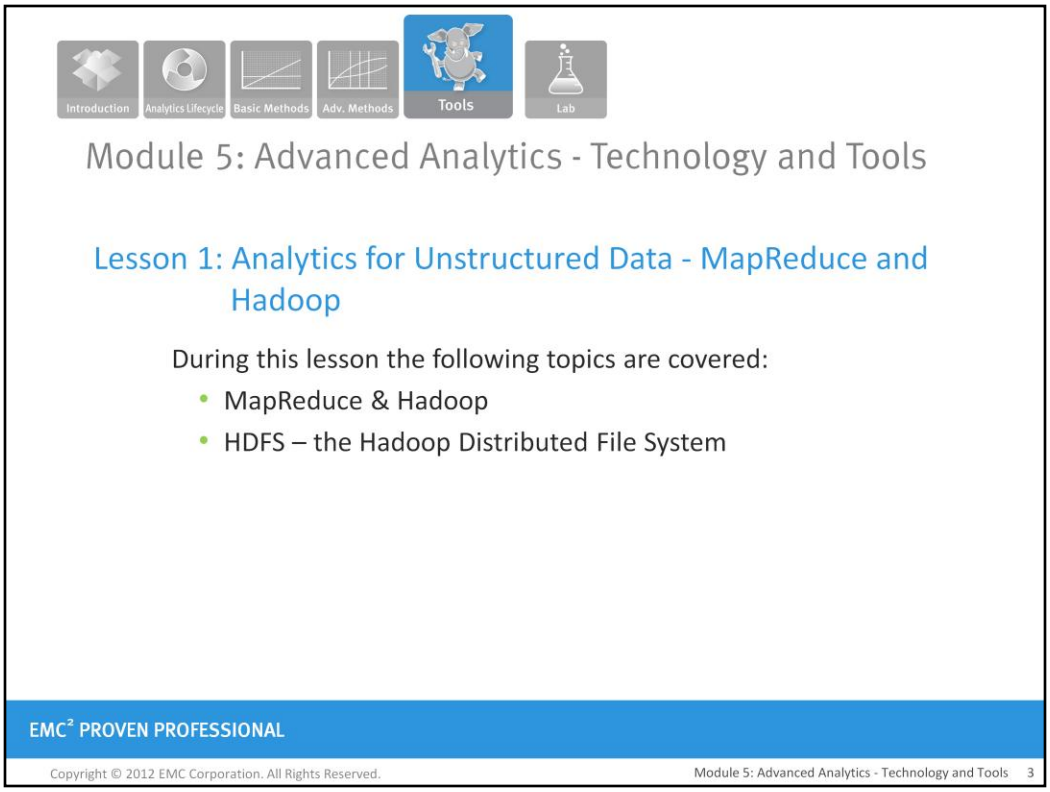
Module 5 – Advanced Analytics - Technology and Tools

EMC² PROVEN PROFESSIONAL



The objectives for this module are shown here.



Lesson 1 introduces the idea behind MapReduce processing, and then describes how Hadoop implements this algorithm.

Hadoop's value really shines in combination with HDFS: the Hadoop Distributed File System.

This lesson covers Data Management: the processing and development of frameworks to work on unstructured data in the terabyte range, and presents extensions to Hadoop that leverage its capabilities.

Lesson 2 covers:

- Hive and Pig – Hadoop query languages
- HBase – a BigTable workalike using Hadoop
- Mahout – machine learning algorithms and Hadoop MapReduce

Putting the Data Analytics Lifecycle into Practice

- From Module 4 – Lifecycle
  - *Phase 1: Discovery*
  - *Phase 2: Data Preparation*
  - Phase 3: Model Planning
  - Phase 4: Model Building
  - Phase 5: Results & Key Findings
  - Phase 6: Operationalize

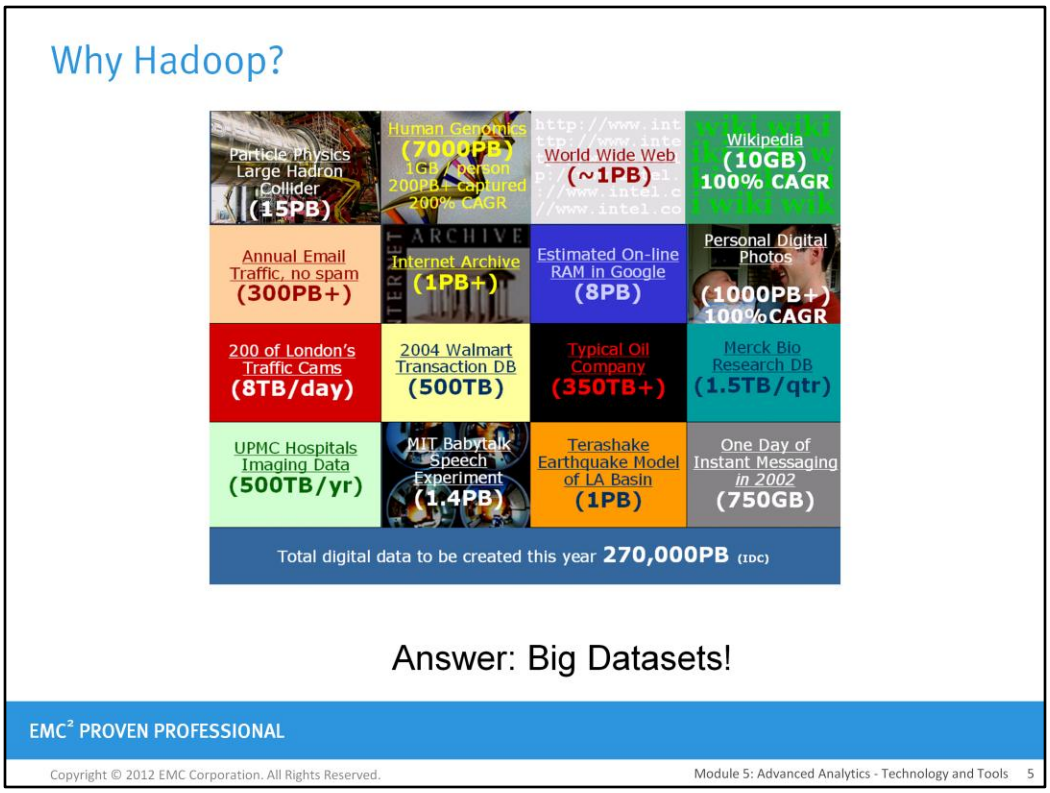
- You have "big data," how can you make it suitable for analysis?
  - That is, obtain results & key findings in a timely manner?
- Combine Phases 1 and 2

EMC² PROVEN PROFESSIONAL

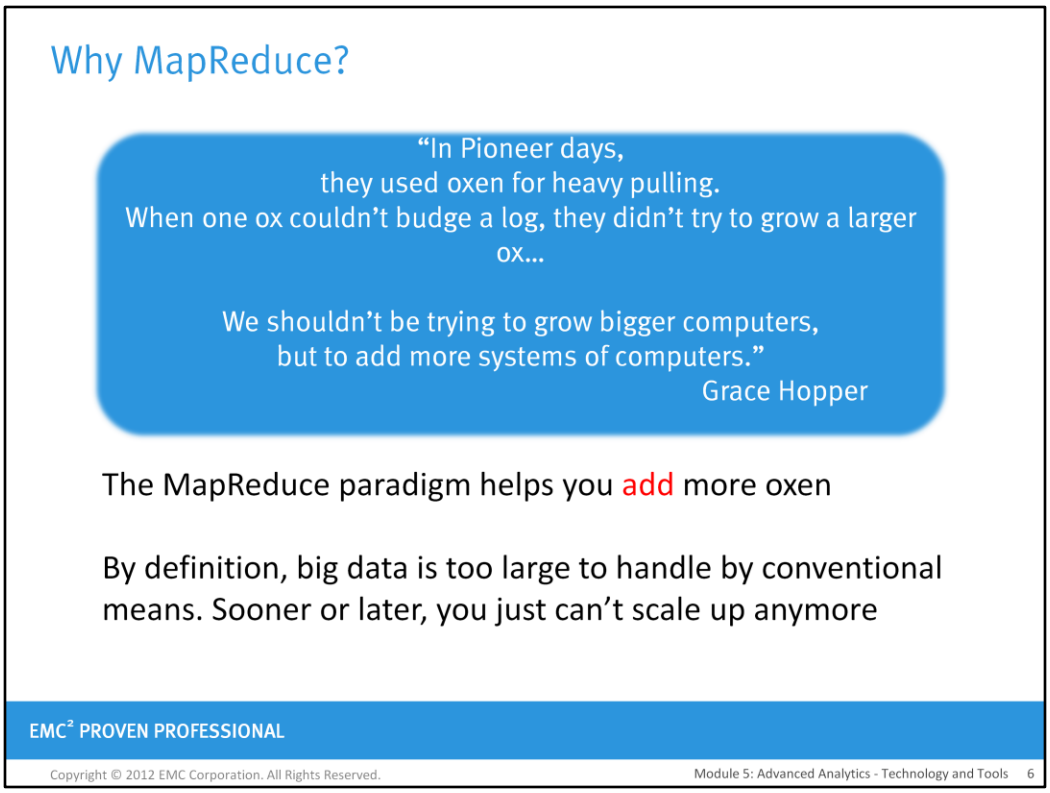
In the previous module, we discussed various ways to mine data for information. In this module, you're going to take a step back and look at another acquire-parse-filter method.

This method is called **MapReduce**, and it **reflects a pattern for processing Big Data and extracting only the data you need**.

The MapReduce paradigm fits well with various Data Analytics processes. In our Data Analysis lifecycle, Map and Reduce represent activities from the 1st two phases. Following Ben Fry's model in *Seven Stages of Data Visualization*, the actions would be part of the Acquire/Parse/Filter stages (Map [Acquire/Parse] and Reduce [Filter]).



Consider the examples of different classes of "big data." If we are to achieve anything with data of this size, we need a way both to store and access this information such that we can retrieve information in finite time.

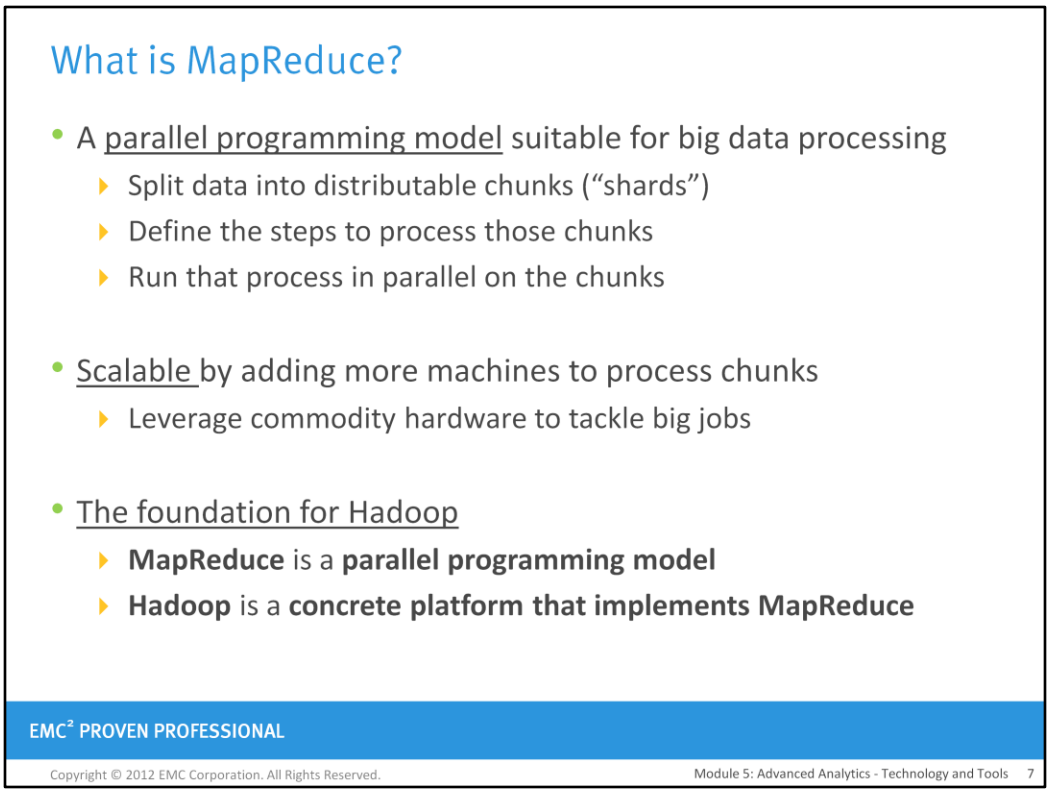


You see from Grace Hopper's quote that the fundamental paradigm of MapReduce is the **reduction in time to complete a given task by breaking it down into stages and then executing those stages in parallel.**

Such an activity is sometimes called the "master/slave" or "master/worker" pattern, and has been known for a while.

The definition of Big Data asserts that the data is simply too large to handle by conventional means. The usual example is when an RDBMS is initially used to store that data. As performance needs increase, organizations purchase more powerful hardware and then more systems to share the data retrieval and processing. And yet the data keeps on growing.

What to do?



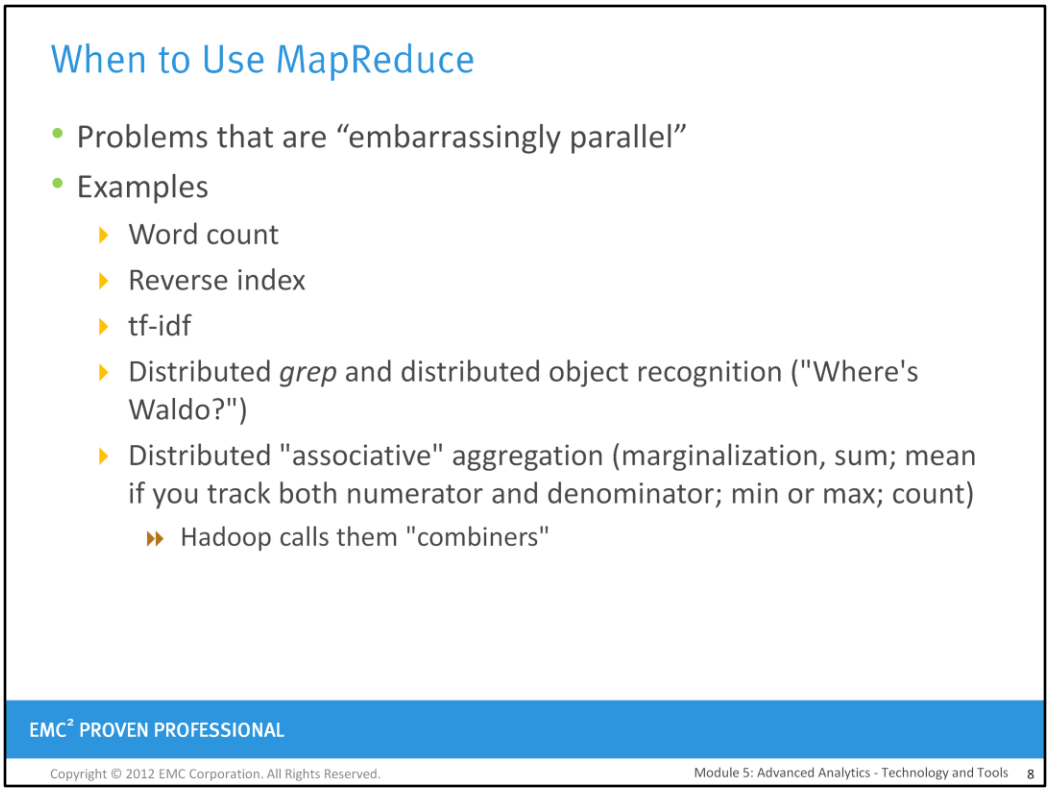
As we've said, the idea of MapReduce isn't new (more like old wine in new bottles). What is **new in Google's MapReduce is the parallel processing of data as well as computation**.

Here, **a set of worker tasks each work on a subset of data where each subset is (usually) physically and logically distinct one from the other**.

In the world of databases, this separation is often called "sharding." Sharding is a technique whereby data is partitioned such that i/o operations can proceed without concern for other users, nor will these queries conflict with other users such that the data must be locked.

A classic example of this is separating a database into country of origin: queries against the US, Canada and Mexico run independently. The difference is that it's the data that's distributed as well as the computation.

MapReduce also borrows some elements of functional programming. All data elements in MapReduce are *immutable*; changing an input (key, value) pair does not change the input files. All communication occurs by generating new output (key, value) pairs and then forwarding the output to a new phase. **At base, MapReduce programs transform lists of input data elements into lists of output data element, and do so twice: Once for the Map and once for the Reduce.**



For which kinds of problems is MapReduce most suited? Very simply, problems that are “embarrassingly parallel.“ In these situations, the problem can be decomposed into tasks that can be broken into “chunks” that can be distributed, processed, and recombined without communicating with each other.

Examples of such problems are listed on the slide. We’ve met some of these functions before: reverse index, tf-idf, distributed grep (pattern matching).

Marginalization occurs when you use aggregate statistics over certain variables as a way of reducing the number of variables in an analysis by focusing on the aggregation of other variables.

Consider this example.

Assume we have a table where each object has attributes "state", “climate" (wet, dry, temperate), and "% of cats in the US“ (*pct_cats*). This table provides us with the number of cats broken out by climate and state. If we sum the *pct_cats* across each state, then we are calculating the marginal probability of cats in each state. If we sum *pct_cats* across each value of climate, we are calculating the marginal probability of cats as a function of climate.

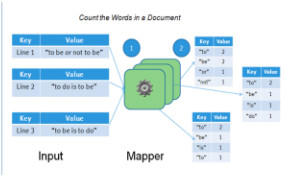
In statistical terms, the table gives you P(cats | state, climate). Marginalization is thus to aggregate this table up to either P(cats | state), or P(cats | climate), depending. For the SQL inclined, the statement

SELECT SUM(pct_cats) from table GROUP BY state;

would perform this action for the values of state.

The Map part of MapReduce

- Transform
  - (Map) input values to output values: <k1,v1> → <k2,v2>
- Input – Key/Value Pairs
  - For instance, Key = line number, Value = text string
- Map Function
  - Steps to transform input pairs to output pairs
    - For example, count the different words in the input
- Output – Key/Value Pairs
  - For example, Key = <word>, Value = <count>
- Map output is the input to Reduce

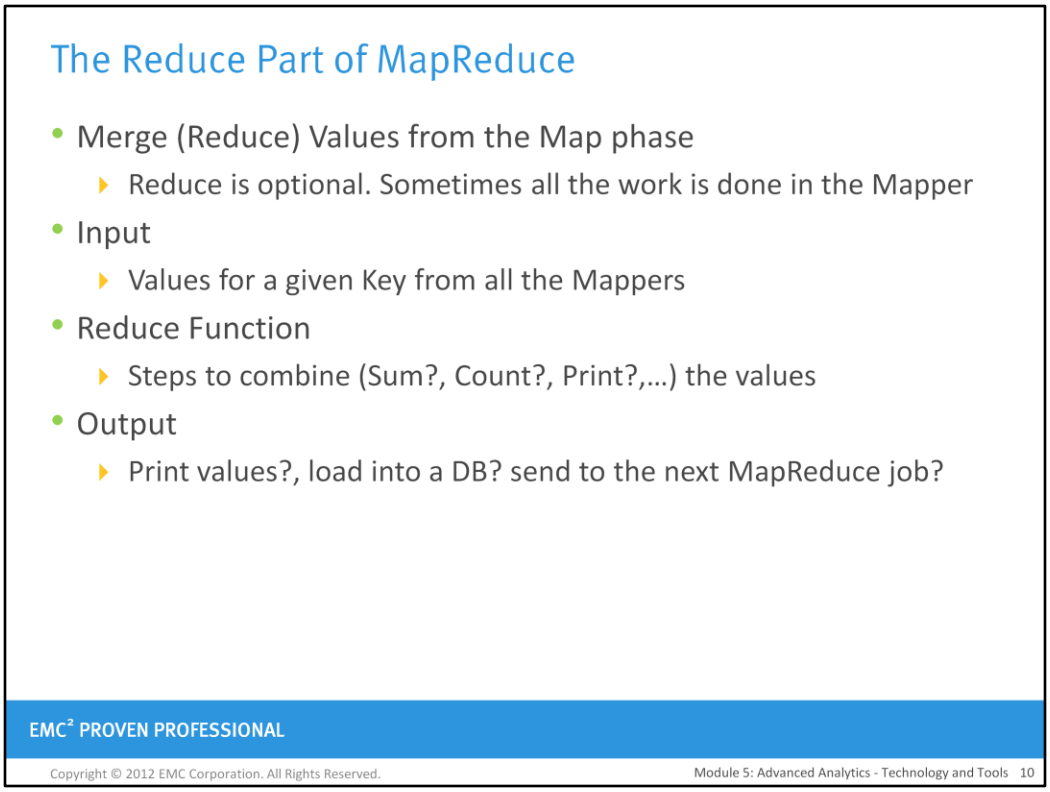


EMC² PROVEN PROFESSIONAL

The first phase of any MapReduce job is to, well, map.

In the **mapping phase, raw data is transformed into a set of <key, value> pairs.**

For example, the key might be a line number and the value a text string. The map function will transform this input into a series of output pairs: in this case, a record containing each unique word in the input (the key), and a count of its occurrences (the value). So, for example, the input string “to be or not to be” would result in a series of output records such as [to 2 ; be 2; or 1; not 1] (assume that the “;” characters represents a newline).



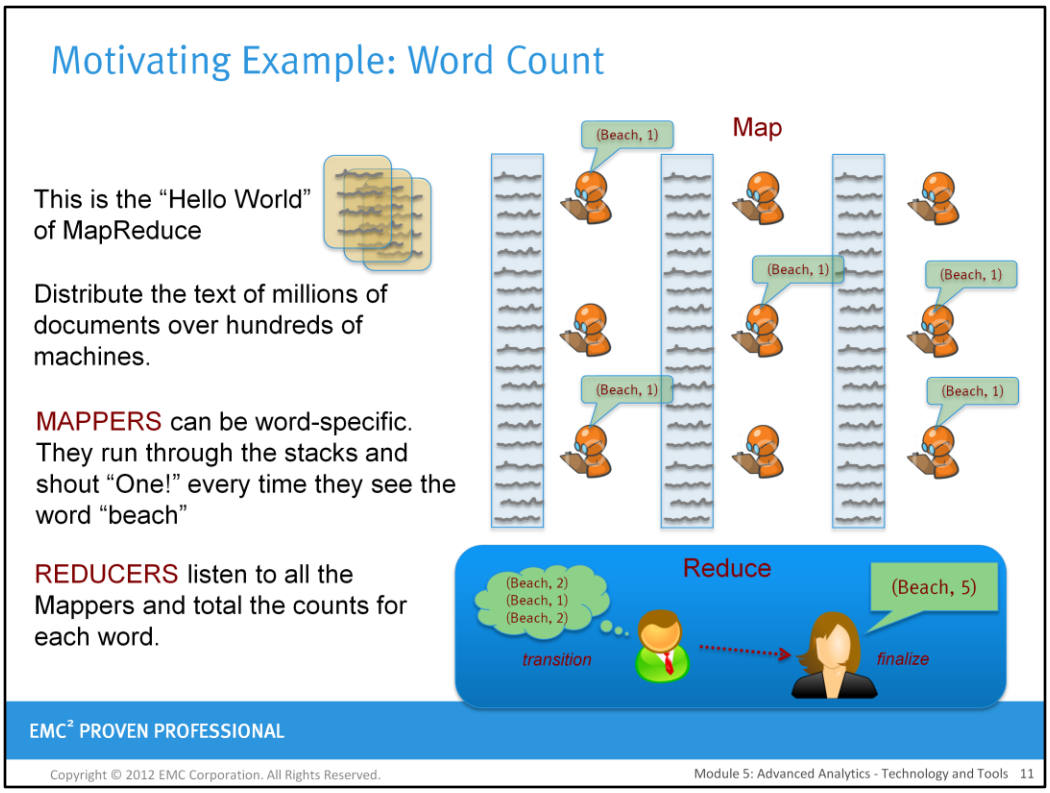
Well, you've finished mapping, and now what?

The next task is to run the Reduce step on the input from the Mapper.

Sometimes all the work is done in the Mapper, and you're done. Otherwise, you use the Reducer to combine the input from the Mapper. Maybe you count it, print it, load it into a database, or save it in a file and load it into R for more analysis.

The MapReduce framework supports a Combiner function as well. In this case, this function would take Mapper output and produce a second output stream for the Reducer component.

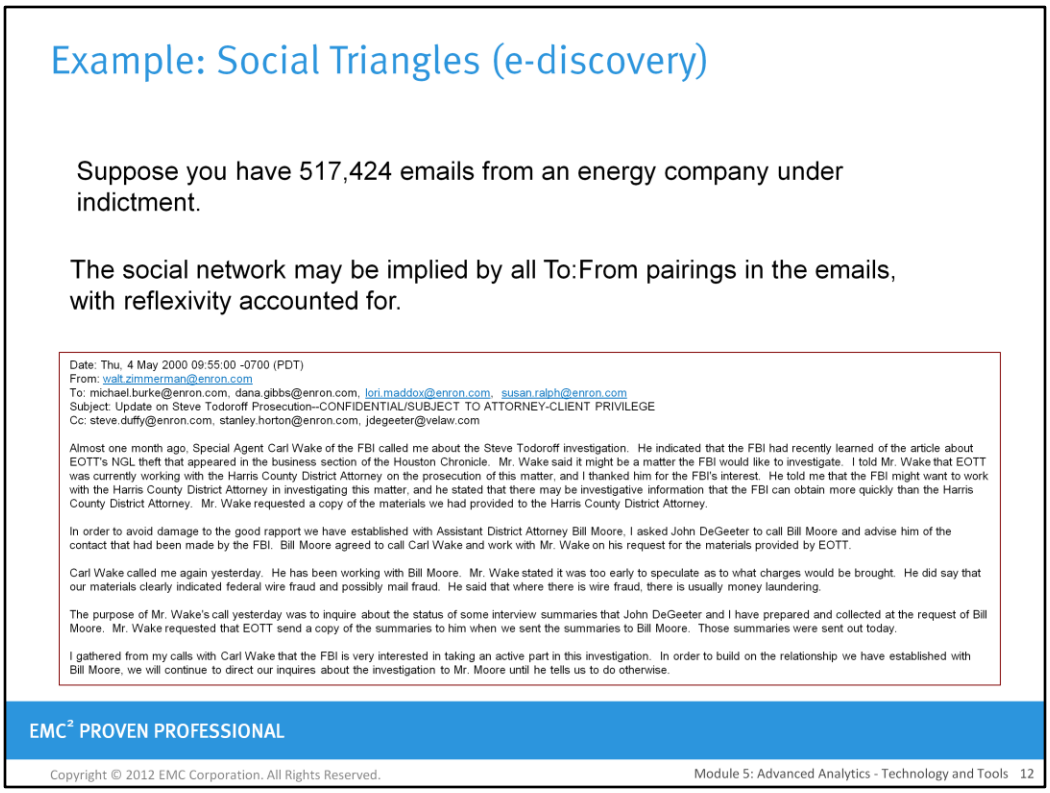
For example, you could imagine a combiner function that would take output from multiple mappers and further recombine it before handing it off to the Reducer job. **Combiners work well when Mapper output is voluminous: adding a Combiner step may decrease network traffic resulting in better performance.**



The "Word Count" example is the "hello, world" of the Data Analytics world. In this case, we have a millions of documents and hundreds of machines. We want to count the number of times the word "beach" appears in these documents.

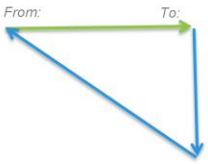
Key to understanding this problem is that the Mappers don't attempt to actually aggregate the count of the number of times the word "beach" appears in a single document. Instead, they simply output a key value pairs consisting of <beach,1>.

It's up to the Reducers to aggregate the final results and output the single key/value pair <beach, nnn>.



e-Discovery can involve processing a huge number of documents. Consider the above example where we need to process over ½ million email messages, and we wish to determine the “social network” in the company, where “social network” is a particular group of people who communicate.

Social Triangle: First Directed Edge



Mapper1

Maps two regular expression searches:
To: Michael, Dan, Lori, Susan
From: Walt

Emits the outbound directed edge of the social graph:
<Key, Value> = <Walt, [Michael, Dan, Lori, Susan]>

Reducer1

Gets the output from the mapper with different values
<Key, Value> = <Walt, [Michael, Dan, Lori, Susan]>
<Key, Value> = <Walt, [Lori, Susan, Jeff, Ken]>

Unions the values for the second directed edge:
<Key, Value> = <Walt, [Dan, Jeff, Ken, Lori, Michael, Susan]>

Data is reduced by about a third.

EMC² PROVEN PROFESSIONAL

We build this job as a sequence of MapReduce passes through the data (we can assume that the modified data is written back into HDFS).

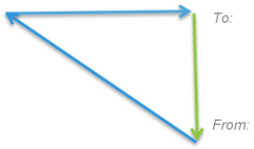
The output from Mapper1 is a key/value pair where the key is the sender of the message and the value is a list of the recipients.

The Reducer takes as input the list of sender/recipients and creates a single record for each sender with the aggregated list of all recipients. These are all the people to whom Walt sent email.

<Walt, [Michael, Dan, Lori, Susan]> + <Walt, [Michael, Dan, Lori, Susan]> → <Walt, [Dan, Jeff, Ken, Lori, Michael, Susan]).

Social Triangle: Second Directed Edge

Mapper2



Reverses the previous Map:
To: Michael, Dan, Lori, Susan
From: Walt
Emits the inbound directed edge of the social graph:
<Key, Value> = <Susan, Walt>; <Lori, Walt>; <Dan, Walt>; etc

Reducer2

Gets the output from the mapper with different values
<Key, Value> = <Susan, Walt>
<Key, Value> = <Susan, Jeff>

Unions the values for the third directed edge:
<Key, Value> = <Susan, [Jeff, Ken, Walt]>

Data again reduced by about a third.

EMC² PROVEN PROFESSIONAL

Mapper2 does the opposite.

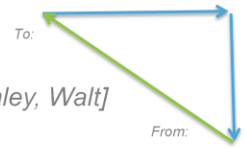
For each email recipient, it creates a key/value pair of the form <recipient/participant>.

Reducer2 is similar to Reducer1: for each recipient it creates a single output record of the form <recipient/list of senders>. These represent people from whom the person received mail.

Susan received email from [Jeff, Ken, Walt].

Social Triangle: Third Directed Edge

Mapper3



Join [inbound] and [outbound] lists by Key
Walt, [Jeff, Ken, Lori, Susan], [Jeff, Lori, Stanley, Walt]

Emits <Person, Person> pair with level of association:
<Key, Value> = <Walt:Susan, reciprocal>; <Walt:Lori, directed>, etc

Reducer3

Reducer unions the output of the mappers and presents rules:
<Key, Value> = <Walt:Susan, reciprocal>
<Key, Value> <Walt:Lori, directed>

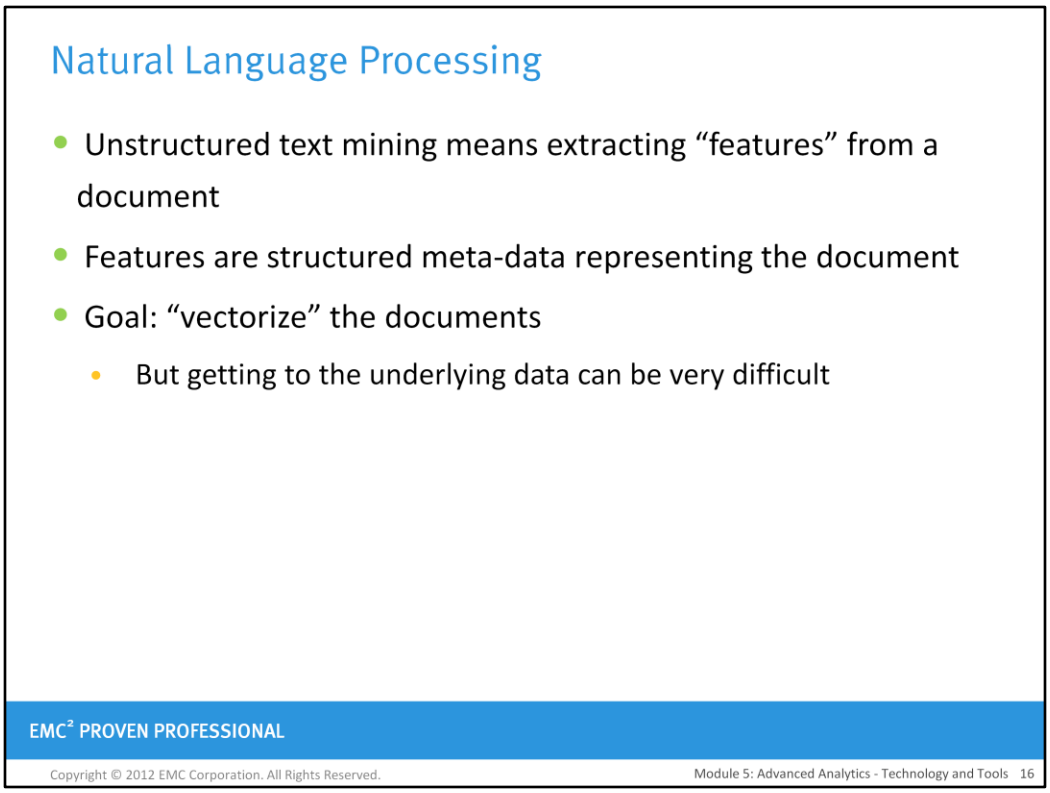
The third reducer can shape the data any way that serves the business objective.

EMC² PROVEN PROFESSIONAL

The third time is the charm.

Mapper3 takes the combination of inbound [mail sent to] and outbound [mail sent from] and outputs a key value pair of the form <Sender:Recipient, relationship.

In this instance the relationships are defined as reciprocal (Sender sent/received mail to/from the recipient) or directed (person sent mail to the recipient)

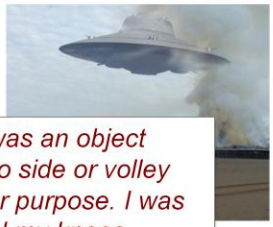


Unstructured text mining means reading a document and extracting various “features” from it.

These features are structured meta-data representing the document, like sentiment, topic or time of composition.

Ultimately, the documents are “vectorized”, which means representing the text as mathematical objects. But getting to the underlying data can be very difficult at times.

Example: UFOs Attack



July 15th, 2010. Raytown, Missouri

When I fist noticed it, I wanted to freak out. There it was an object floating in on a direct path, It didn't move side to side or volley up and down. It moved as if though it had a mission or purpose. I was nervous, and scared, So afraid in fact that I could feel my knees buckling. I guess because I didn't know what to expect and I wanted to act non aggressive. I though that I was either going to be taken, blasted into nothing, or…

Q: What is the witness describing?

A: An encounter with a UFO.

Q: What is the emotional state of the witness?

A: Frightened, ready to flee.

Source: http://www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada

EMC² PROVEN PROFESSIONAL

Observe this particular account of an observer to what is described as a UFO attack.

Example: UFOs Attack

If we really are on the cusp of a major alien invasion, eyewitness testimony is the key to our survival as a species.

*Strangely, the computer finds this account **unreliable!***

When I fist noticed it, I wanted to freak out ject floatin in on a direct path, It didn't move side to side or volley up n. It moved as if though it had a mission or purpose. I was and scared, So afraid in fact that I could feel my knees buckling. I guess because I didn't kno d I wanted to h that I was either going to be taken, blasted into nothing, or…

Machine error

Typo

Turn of phrase

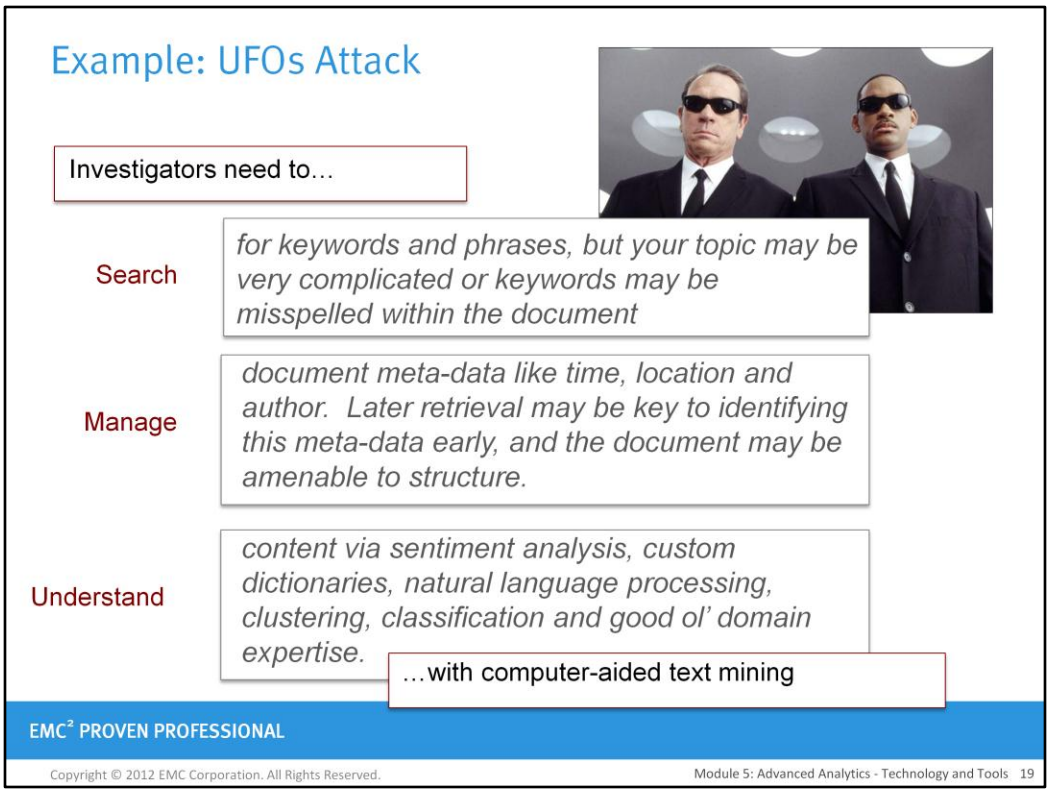
Ambiguous meaning

"UFO" keyword missing

Source: http://www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada

EMC² PROVEN PROFESSIONAL

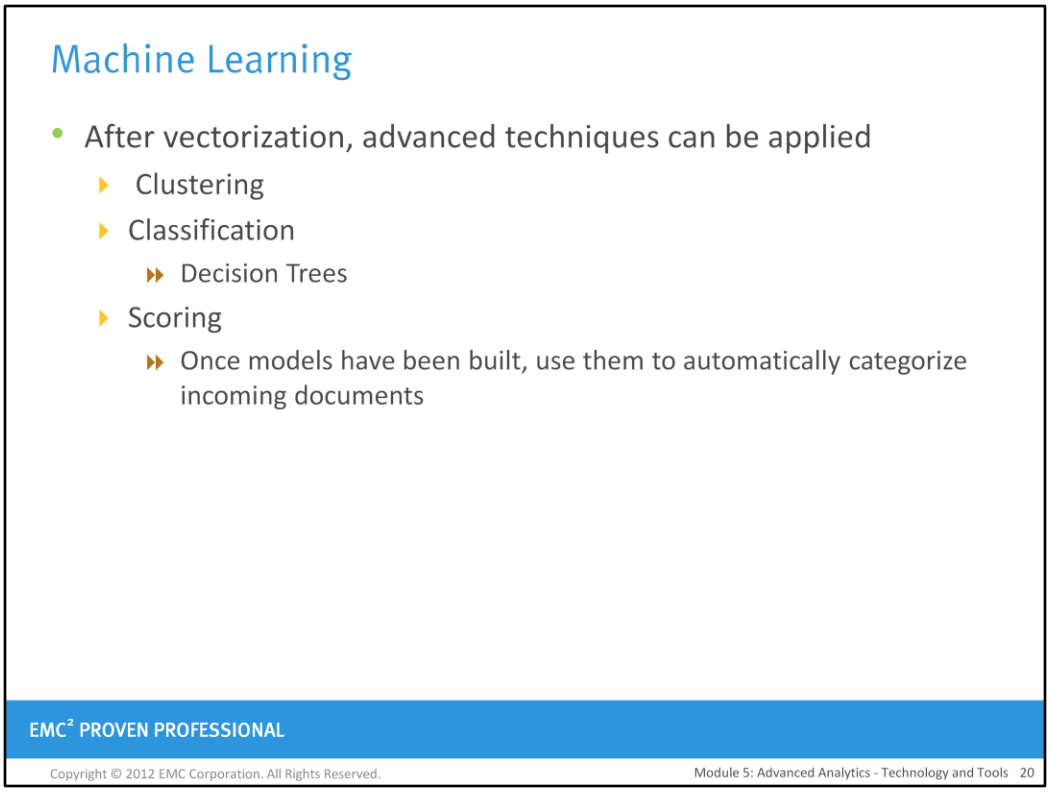
Yes, survival as a species does tend to concentrate the mind wonderfully. But, an analysis of the text by a computer indicates that the text is unreliable! Let us catalog the problems: machine error, a typo, ambiguity, a "turn of phrase", and a missing term ("UFO").



Consider what an investigator needs to do in this situation.

We have a large search problem, made more difficult because of misspellings, convoluted prose, clichés, etc. In addition, we have to manage the document meta-data (time, local, author, weather, distance from military base, and so forth).

Finally, the investigator needs to understand the content. This is achieved by sentiment analysis, custom dictionaries, clustering algorithms, classification algorithms, and finally, knowledge and expertise within the given domain (consider the meaning of "crash" when spoken by a pilot and by a storage administrator).



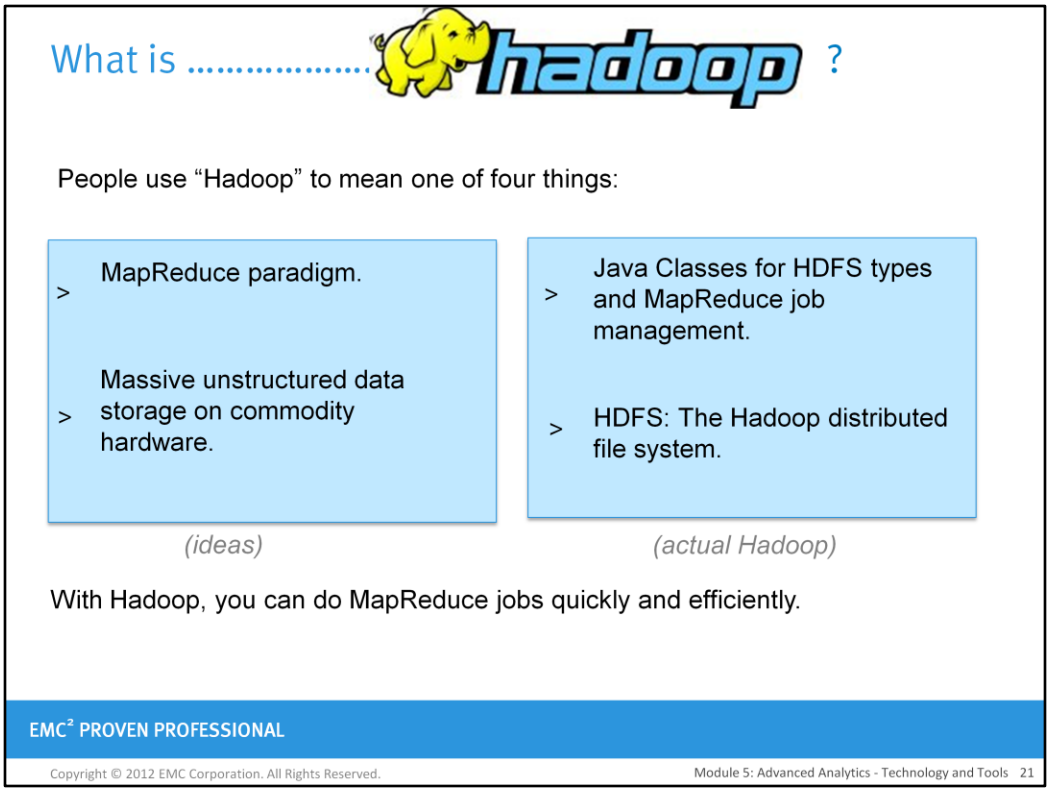
Once the texts have been vectorized they can be subjected to many advanced techniques. **A vector is really a key followed by a list of numerical values**.

These include **clustering**, which mean find "clouds" in the data, such as topics, and use these to guide discovery. Among the techniques applied here are **k-means clustering**, where we attempt to find "k" clouds, and agglomerative clustering, where single items are aggregated into clusters.

Classification identifies which documents fall into a particular category, such as "red", "blue" or "green."

Decision trees construct a series of yes/no decisions such that a document can be assigned to a particular category.

Last is **scoring**: once models have been built, we can use them to automatically categorize incoming documents.



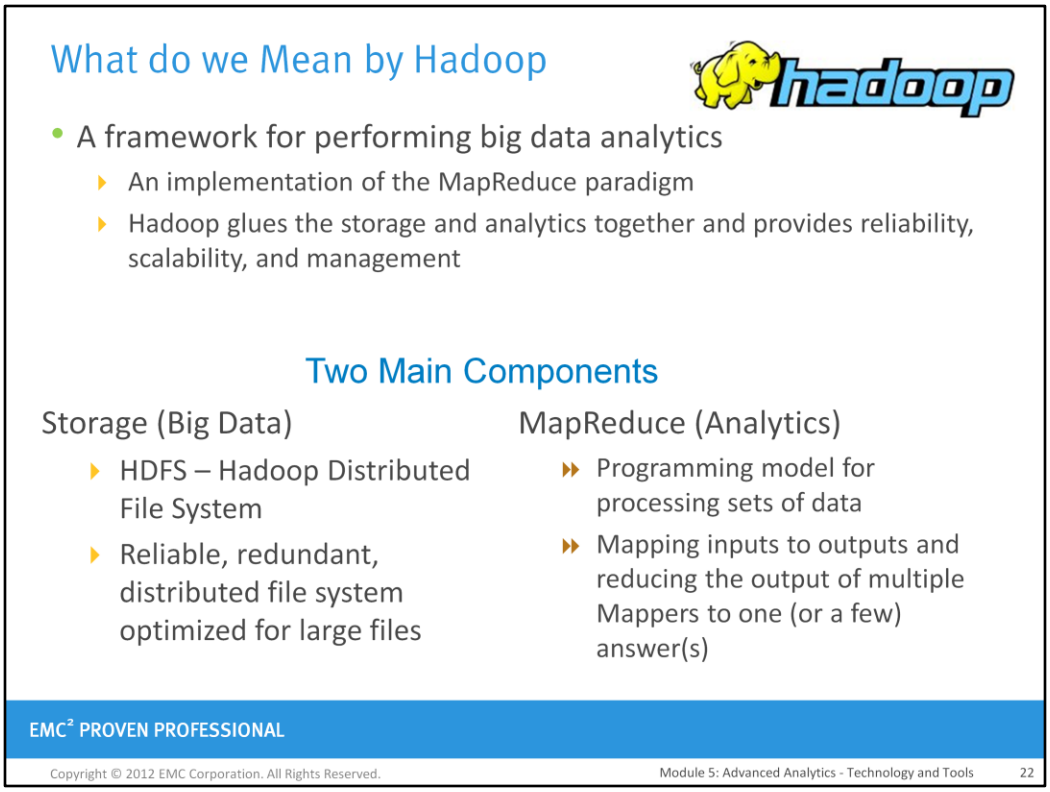
Unfortunately, people may use the word "Hadoop" to mean multiple things. They may use it to describe the MapReduce paradigm, or they may use if to describe massive unstructured data storage using commodity hardware (although commodity doesn't mean inexpensive).

On the other hand, they may be referring to the Java classes provided by Hadoop that support HDFS file types or provide MapReduce job management.

Or they may be referring to HDFS: the Hadoop distributed file system. And they might mean both HDFS and MapReduce.

The point is that **Hadoop enables the Data Scientist to create MapReduce jobs quickly and efficiently.** As we shall see, one can utilize Hadoop at multiple levels: writing MapReduce modules in Java, leveraging streaming mode to write such functions in one of several scripting languages, or utilizing a higher level interface such as Pig or Hive.

The Web site http://hadoop.apache.org/ provides a solid foundation for unstructured data mining and management.



So what exactly is Hadoop anyway?

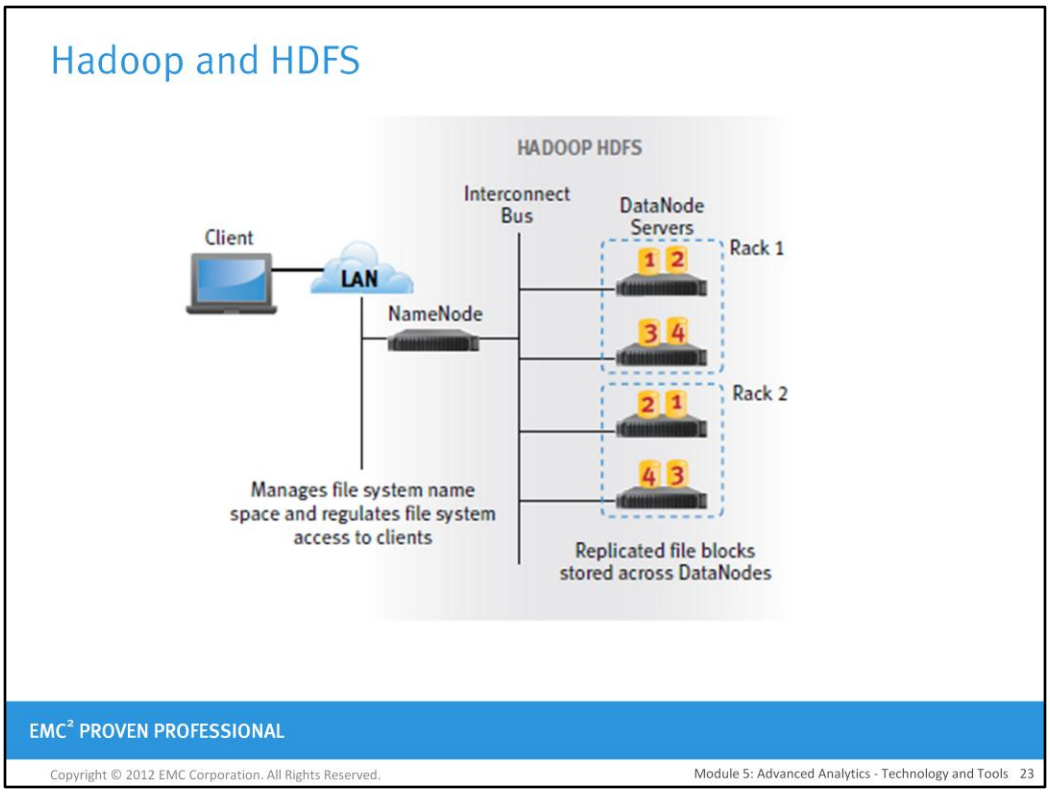
The quick answer is that Hadoop is a framework for performing Big Data Analytics, and as such is an ***implementation*** of the MapReduce programming model.

Hadoop is comprised of two main components, HDFS for storing big data and **MapReduce** for big data analytics.

The storage function consists of HDFS (Hadoop Distributed File System) that provides a reliable, redundant, distributed file system optimized for large files.

The analytics functions are provided by MapReduce that consists of a Java API as well as software to implement the services that Hadoop needs to function.

Hadoop glues the storage and analytics together in a framework that provides reliability, scalability, and management of the data.



Let's look a little deeper at the HDFS.

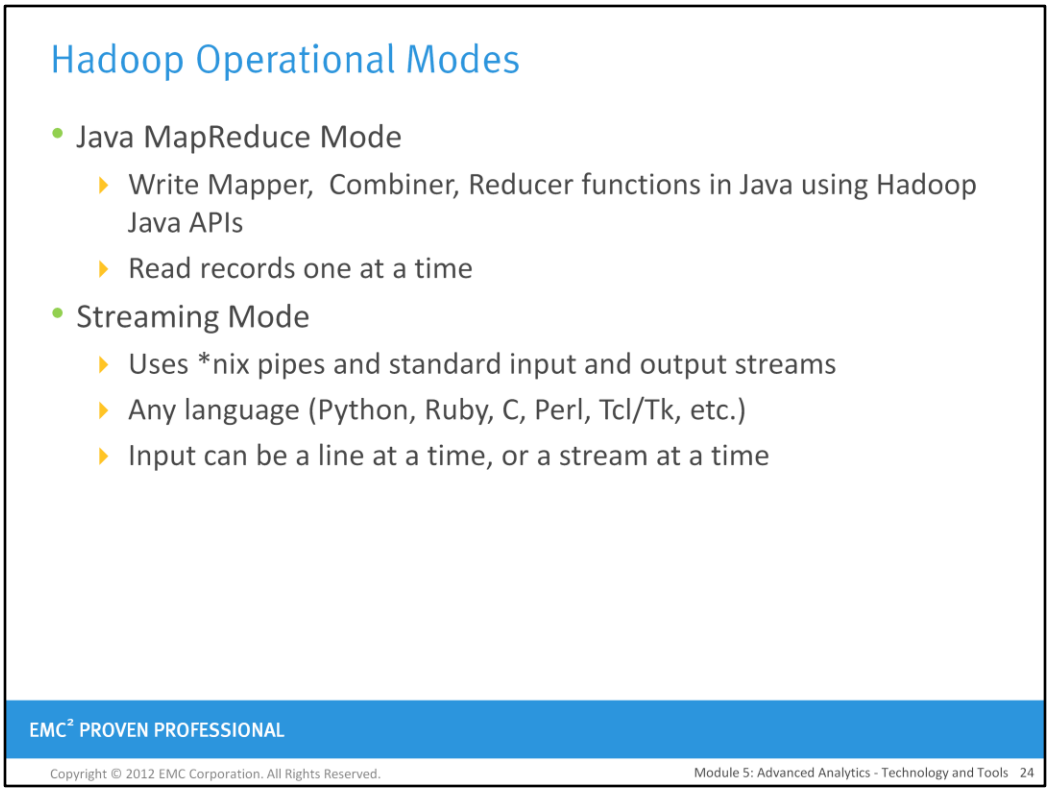
Between MapReduce and HDFS, **Hadoop supports four different node types** (a node is a particular machine on the network).

The NameNode and the DataNode are part of the HDFS implementation. Apache Hadoop has one NameNode and multiple DataNodes (there may be a secondary NameNode as well, but we won't consider that here).

The **NameNode service in Hadoop acts as a regulator/resolver between a client and the various DataNode servers**. The NameNode manages that name space by determining which DataNode contains the data requested by the client and redirecting the client to that particular datanode. DataNodes in HDFS are (oddly enough) where the data is actually stored.

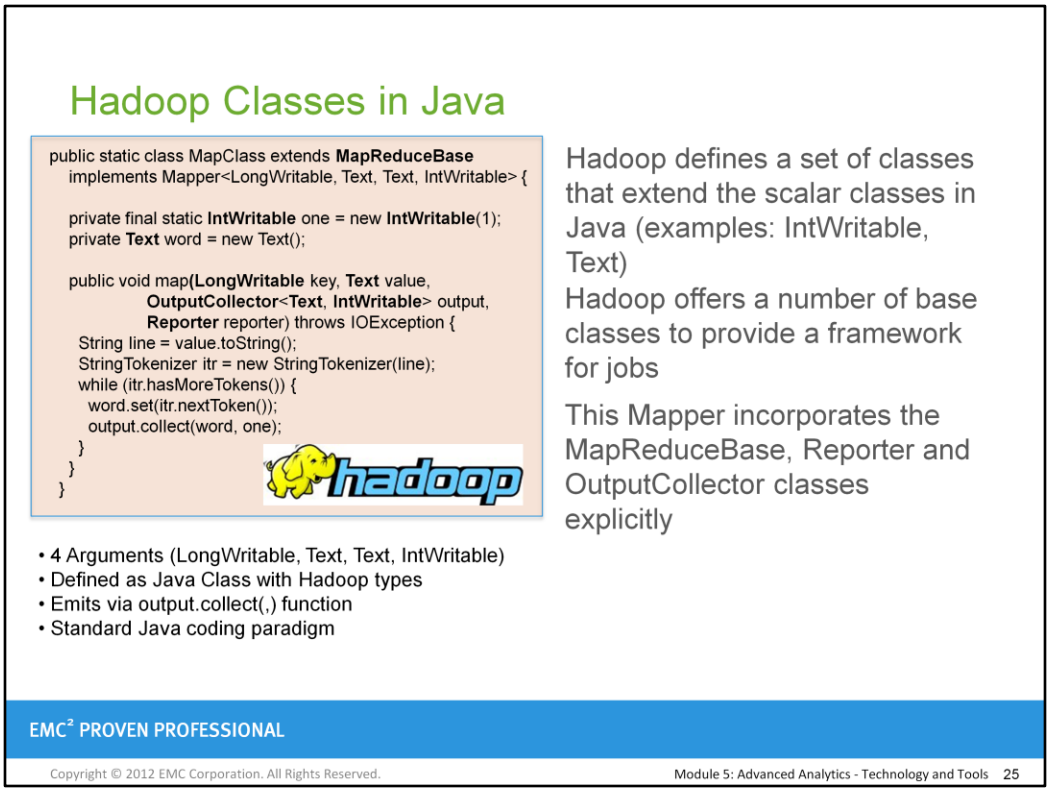
Hadoop is "rack aware": that is, the NameNode and the Jobtracker node utilize a data structure that determines what DataNode is preferred based on the "network distance" between them. Nodes that are "closer" are preferred (same rack, different rack, same datacenter).

The data itself is replicated across racks: this means that a failure in one rack will not halt data access at the expense of possibly slower response. Since HDFS isn't suitable for near real-time access, this is acceptable in the majority of cases.



Hadoop provides two operational modes, where each mode supports a type of interaction with MapReduce and HDFS. Since Hadoop is written in Java and provides Java classes and APIs to access them, **Java MapReduce mode** writes the mapper, combiner and reducer functions in Java. In the Java MapReduce mode, input data is made available to each function a record at a time.

In contrast, **streaming mode** supports standard *nix streams (stdin, stdout) and the *nix pipe mechanisms. This means that all the MapReduce functions can be written in any programming or scripting language you desire (C, Ruby, Python, Perl, etc.) Although input can be read a line at a time, some languages support the “slurping” of the entire input stream into memory (Perl is one example).



Here's an example of defining a Map class *MapClass* that inherits from the MapReduceBase class and implements the Mapper interface. In this case, it's a *map* function that takes as input a key value, a text value, an OutPutCollect, and a reporter.

Hadoop defines Java classes that replace/enhance the standard Java classes: for example, IntWritable instead of Int, Text instead of String.

This is done to optimize i/o operations on the data. Hadoop also defines a series of base classes and interfaces that enables the developer to write Mapper and Reducer classes and focus more closely on the analysis code than the underling technical implementations.

In this example, the MapClass class utilizes the MapReduceBase, the Mapper interface, and an OutputCollector class to perform its functions. Programming Hadoop using Java means that the developer has to learn the appropriate use of these classes and their associated methods.

Example: Hadoop Streaming Mode

	Script to invoke Hadoop
1	`Hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-streaming.jar \`
2	`-input input/ \ # relative to HDFS`
3	`-output output \ # relative to HDFS`
4	`-mapper mapper.py \`
5	`-reducer reducer.py \`
6	`-file scripts/mapper.py \`
7	`-file scripts/reducer.py`

EMC² PROVEN PROFESSIONAL

This example shows a Hadoop invocation that uses the streaming interface. Let's go through each numbered line. Our Mapper and Reducer function are written Python.

1. Hadoop implements the streaming function as a .jar (java archive) file. This line describes the jar file that implements the streaming framework
2. The –input argument defines the location of the input data in HDFS
3. Likewise, the –output switch defines the location for the output data in HDFS
4. The –mapper switch defines the program to run that implements the Map function. This can be a Unix pipeline command, or a single command
5. The –reducer argument names the program to run that implements the reduce function
6. When running using fully-distributed HDFS, the *–file* arguments denote which programs must be copied to the DataNodes for execution.
7. A separate file argument is required for each program

Simple Example of a Mapper: mapper.py

Code	Comments
import sys	
for line in sys.stdin	# input comes from STDIN
line = line.strip()	# strip leading/trailing whitespace
words = line.split()	# split line based on whitespace
for word in words: print '%s\t %s %(word, 1)	# for each word in the collection words # write word and count of "1", tab delimited

EMC² PROVEN PROFESSIONAL

Here are our two functions, written in Python, that perform the map and reduce functions for our wordcount example.

The Mapper function simply reads the data from stdin (the standard input unit) a line at a time; splits the line into tokens (words) based on whitespace (blank, tab, etc.), and then writes each word to the standard output device (stdout) with a count of one.

This isn't the most robust word count program (professional programmers will know not to use this code), but it's sufficient as an example.

Simple Example of a Reducer: reducer.py

Code	Code, continued
cur_count = 0	if cur_word == word:
cur_word = None	cur_count += count
for line in sys.stdin	else:
line = line.strip()	if cur_word:
word, count = line.split("\t",1)	print '%s\t%s' (cur_word, cur_count)
try	cur_count, cur_word = (count, word)
count = int(count)	if cur_word == word:
except ValueError:	print '%s\t%s' %(cur_word, cur_count)
continue	

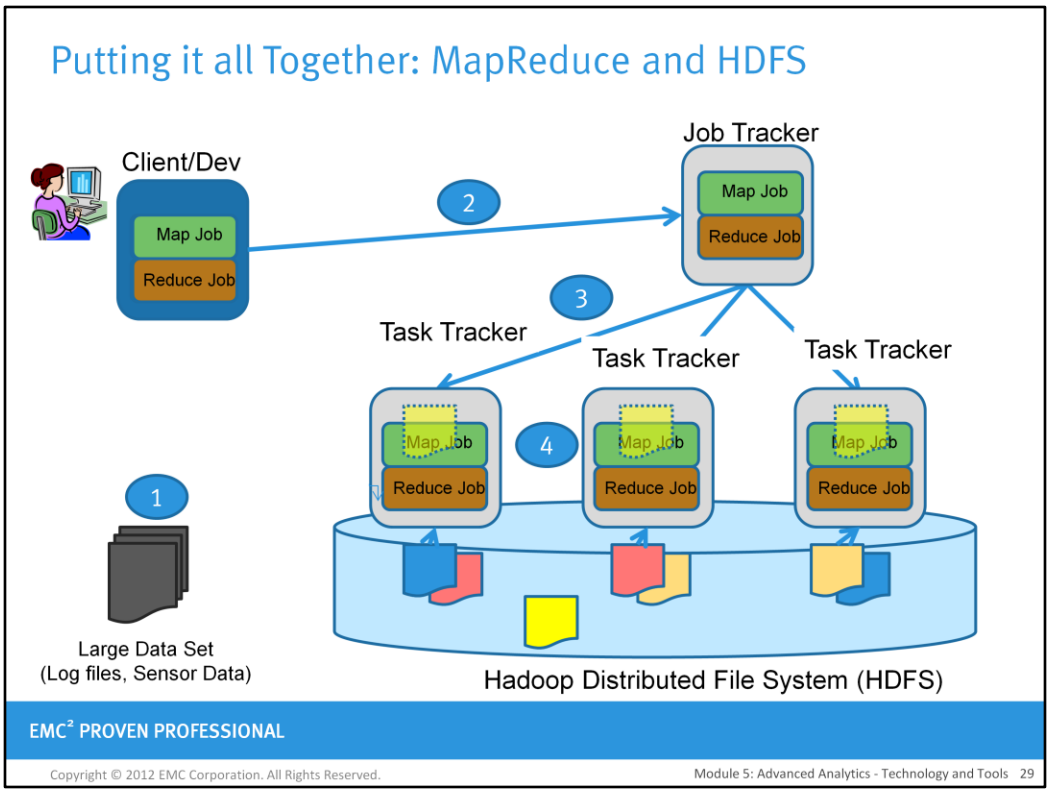
EMC² PROVEN PROFESSIONAL

The reduce function, reducer.py, is only a little more complicated. It reads a tab-delimited string from the standard input, and aggregates the total values for each word (Hadoop guarantees that the input to a reducer is sorted). For each different word, it outputs the word and the aggregated count. So input of "soft 1; soft 1; soft 1" would be output as "soft 3".

Another benefit of using streaming mode is that test harnesses are easily constructed. The following *nix shell script tests the execution of mapper.py and reducer.py

```
cat test.input | mapper.py | sort | reducer.py
```

Hadoop automatically adds the "sort/shuffle" processing in its framework.



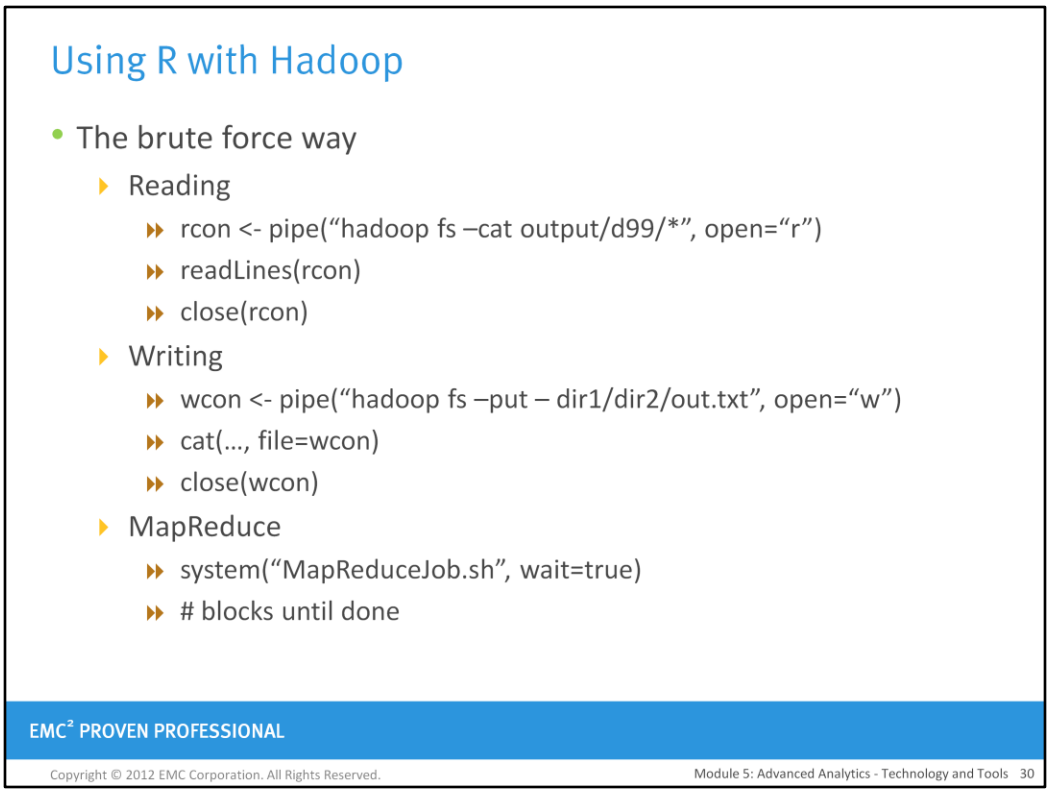
The MapReduce function within Hadoop depend on two different nodes: the JobTracker and the TaskTracker.

The JobTracker node exists for each MapReduce implementation. JobTracker nodes are responsible for distributing the Mapper and Reducer functions to available TaskTrackers and monitoring the results, while TaskTracker nodes actually run the jobs and communicate results back to the JobTracker. That communication between nodes is often through files and directories in HDFS so internode (network) communication is minimized.

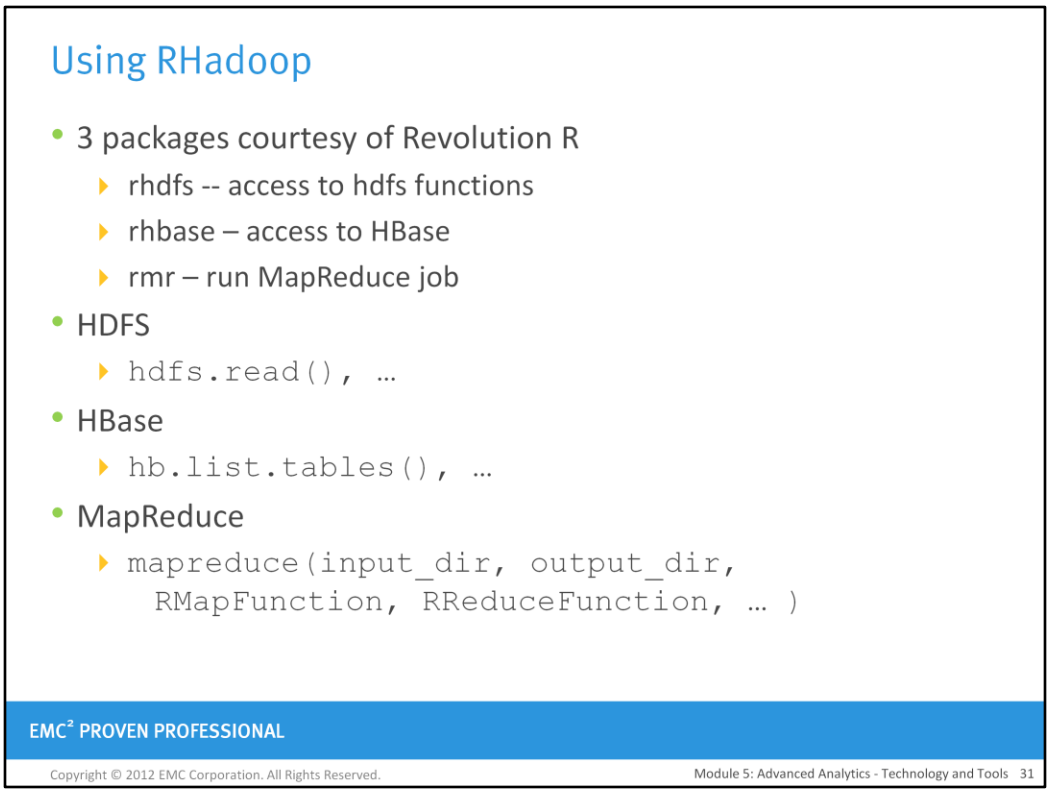
Let's consider the above example. Initially **(1)** , we have a very large data set containing log files, sensor data or whatnot. HDFS stores replicas of that data (represented here by the blue, yellow and beige icons) across DataNodes.

In **Step 2,** the client defines and executes a map job and a reduce job on a particular data set, and sends them both to the Jobtracker, where in **Step 3**, the jobs are in turn distributed to the TaskTrackers nodes. The TaskTracker runs the mapper, and the mapper produces output that itself is stored in the HDFS file system. Lastly, in **Step 4**, the reduce job runs across the mapped data in order to produce the result.

We've deliberately skipped much of the complexity involved in the MapReduce implementation, specifically the steps that provide the "sorted by key" guarantee the MapReduce framework offers to its reducers. Hadoop provides a Web-based GUI for the Namenode, Jobtracker and Tasktracker nodes: we'll see more of this in the lab associated with this lesson.



You can access data stored in Hadoop HDFS as well as running MapReduce jobs by using standard R functions. Reading and writing are supported via the pipe() command that either receives information from R or transmits data to R. A MapReduce job can be run synchronously or asynchronously as a shell script: both stdout and stderr can be redirected and captured if required.



Revolution Analytics has released a set of R packages callef RHadoop that allow direct access to Hadoop HDFS, HBase, and MapReduce. The HBase and HDFS interfaces provide a number of access functions: the goal of the project was to mirror the Java interfaces as much as possible.

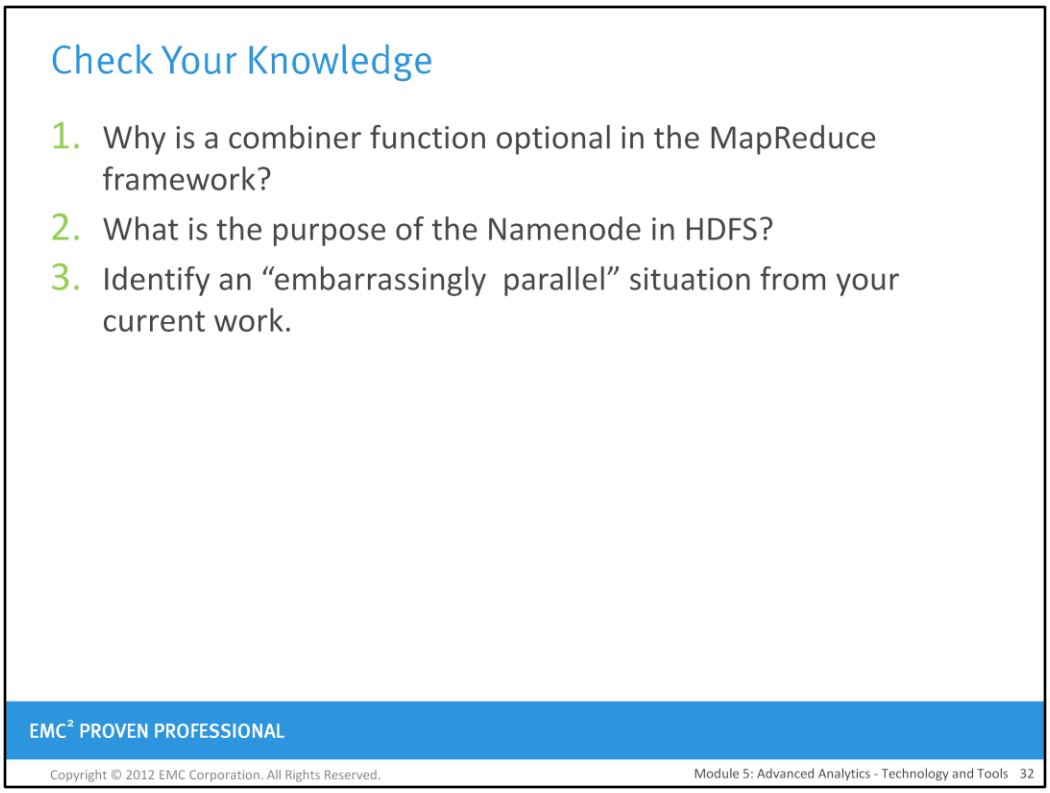
`mapreduce()` is the exception. The input and output directories are specified as would be done with an Hadoop MapReduce job; however, the Map function and the Reduce function are both written in R.

The rhdfs functions are:

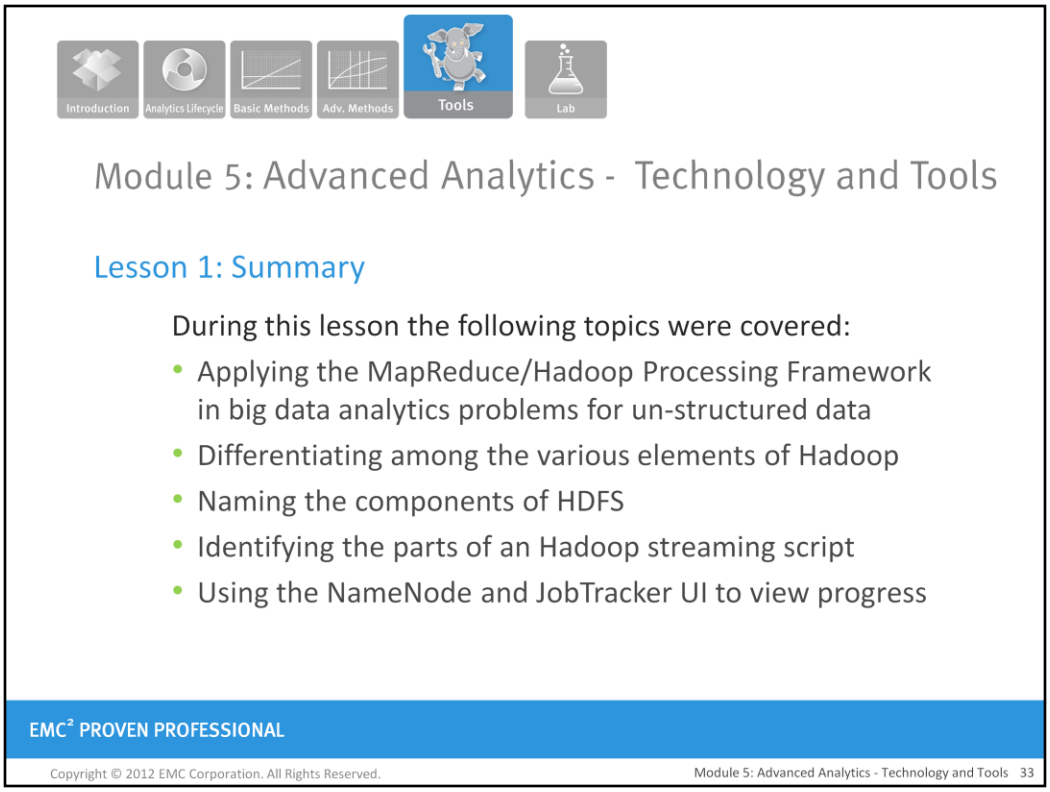
- **File Manipulations** -hdfs.copy, hdfs.move, hdfs.rename, hdfs.delete, hdfs.rm, hdfs.del, hdfs.chown, hdfs.put, hdfs.get
- **File Read/Write** -hdfs.file, hdfs.write, hdfs.close, hdfs.flush, hdfs.read, hdfs.seek, hdfs.tell, hdfs.line.reader, hdfs.read.text.file
- **Directory** -hdfs.dircreate, hdfs.mkdir
- **Utility** -hdfs.ls, hdfs.list.files, hdfs.file.info, hdfs.exists
- **Initialization** –hdfs.init, hdfs.defaults

The HBase functions are:

- **Table Manipulation** –hb.new.table, hb.delete.table, hb.describe.table, hb.set.table.mode, hb.regions.table
- **Row Read/Write** -hb.insert, hb.get, hb.delete, hb.insert.data.frame, hb.get.data.frame, hb.scan
- **Utility** -hb.list.tables
- **Initialization** -hb.defaults, hb.init

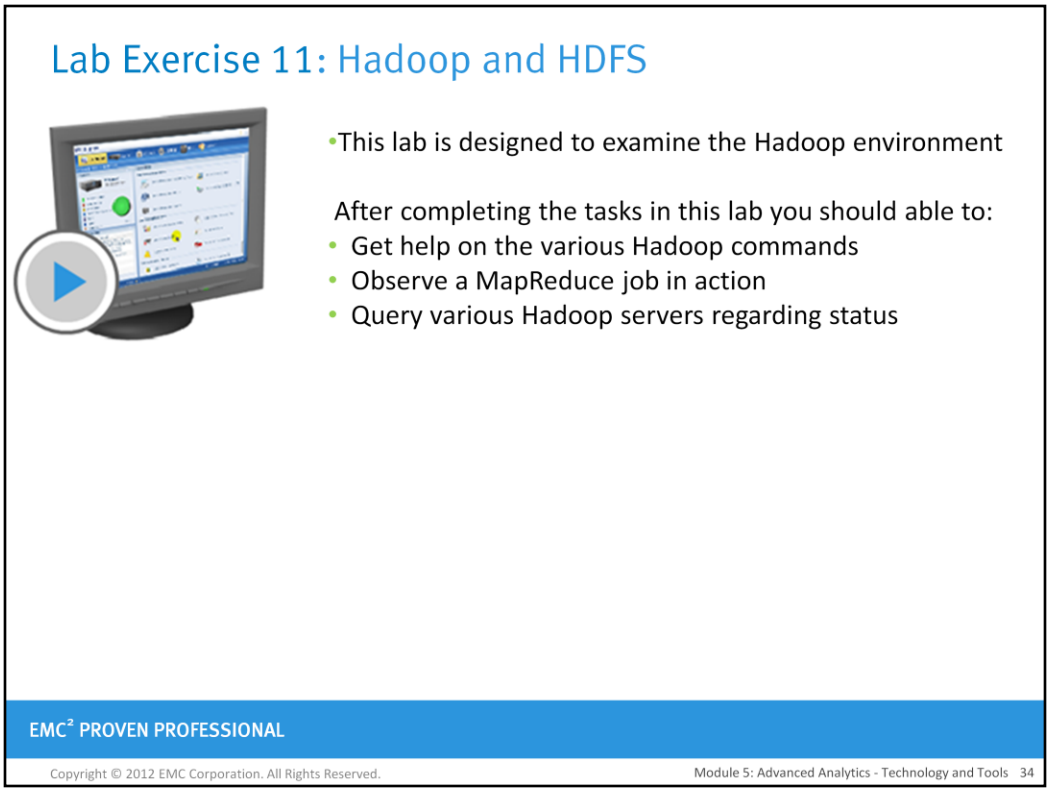


Please take a moment to answer these questions. Record your answers here.



Lesson 1 introduced the idea behind MapReduce processing, and then described how Hadoop implements this algorithm.

This lesson covered Data Management: the processing and development of frameworks to work on unstructured data in the terabyte range, and presented extensions to Hadoop that leverage its capabilities.

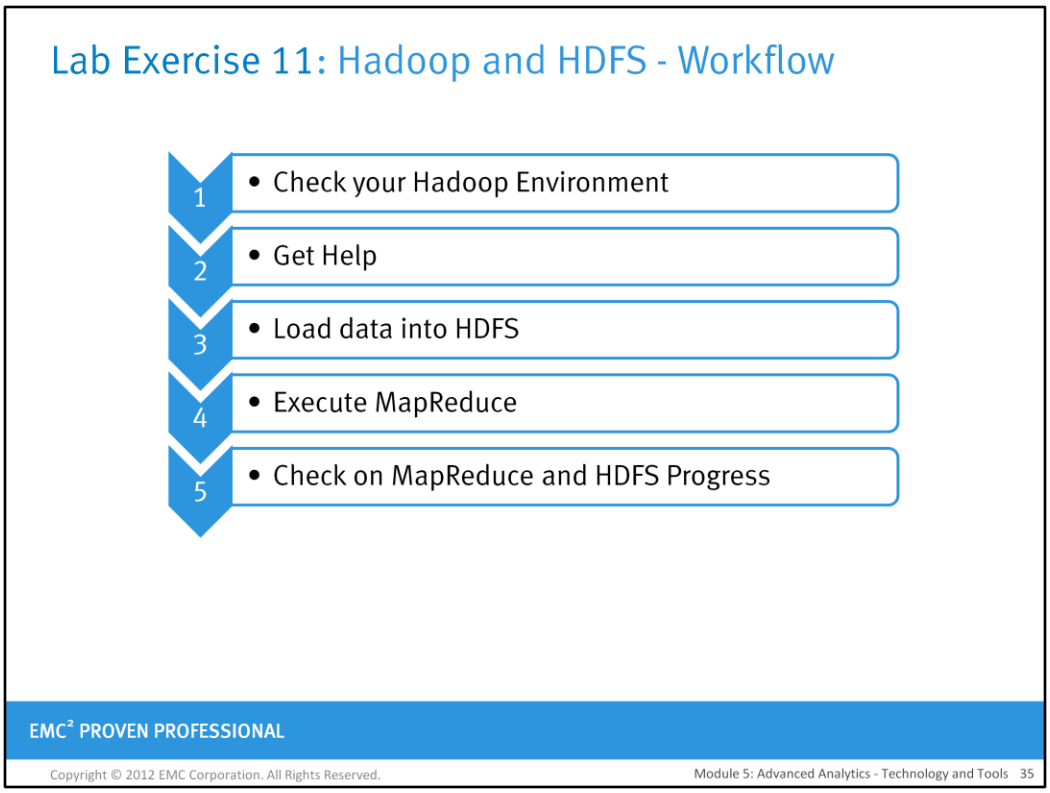


This lab is an introduction to Hadoop and HDFS, and is organized as a tutorial. This lab utilizes the Linux command-line environment (shell) to execute Hadoop programs.

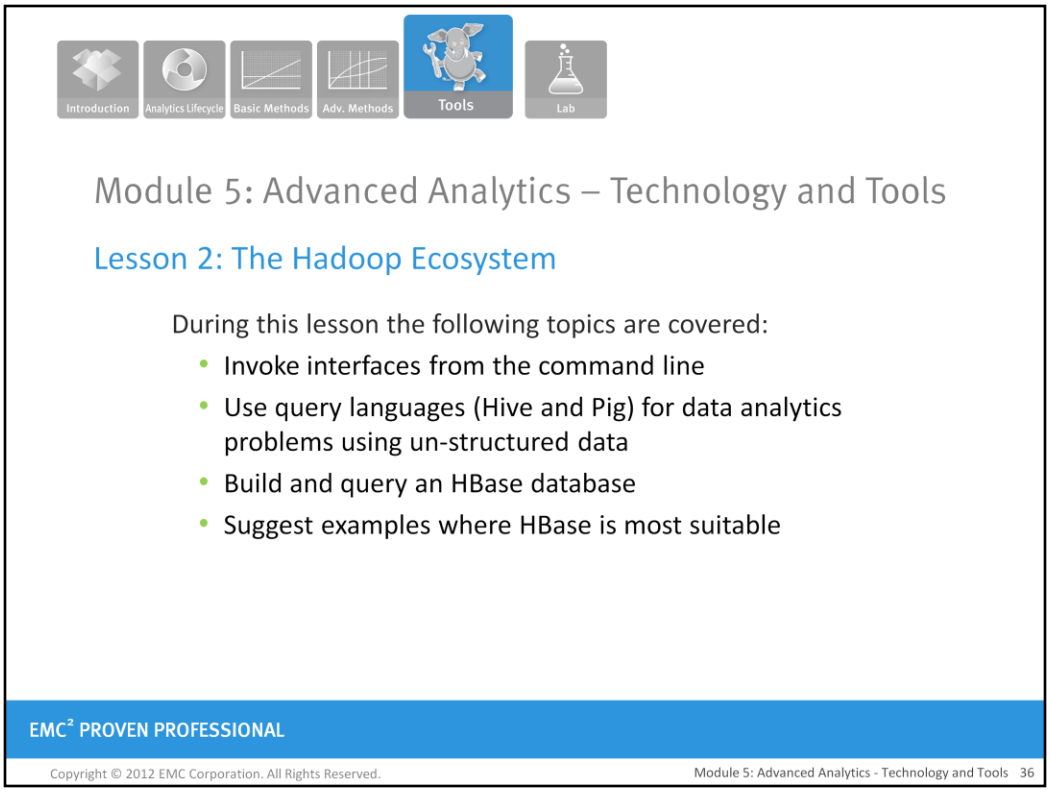
Tasks you will be completing in this lab include

- Using the Hadoop interface
- Working with the Hadoop environment
- Accessing HDFS using Hadoop
- Running a Hadoop job and observe your progress
- Examining the NameNode, Jobtracker and Tasktracker Uis

Instructions for the lab are contained in your lab guide.

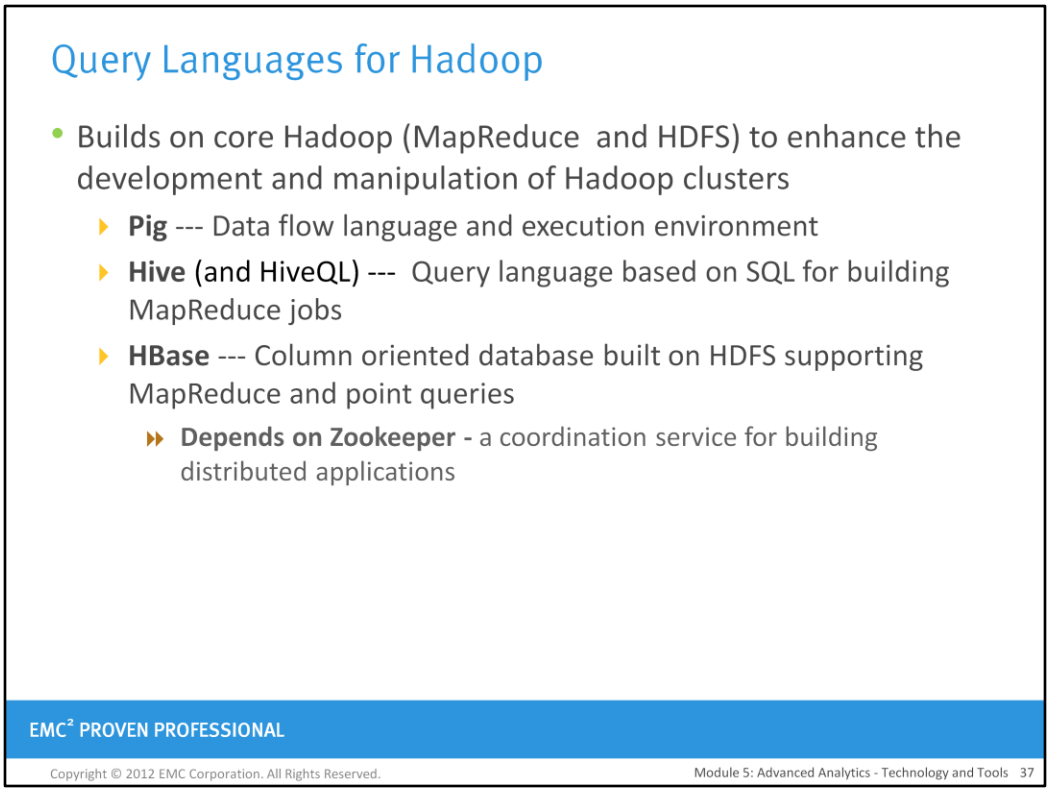


This slide represents the workflow that you will be performing as part of your lab. Please take a moment to review each step.



Hadoop by itself was developed as an open source implementations of Google, Inc.'s Google File System (GFS), MapReduce framework, and BigTable system. Additionally, Hadoop has spawned a number of associated projects, all of which depend on Hadoop's MapReduce capabilities and HDFS.

You will examine several of these in this lesson.



These interfaces build on core Hadoop to support different styles of interaction with the data stored in HDFS. All are layered over HDFS and MapReduce.

Pig is a data flow language and execution environment on top of Hadoop.

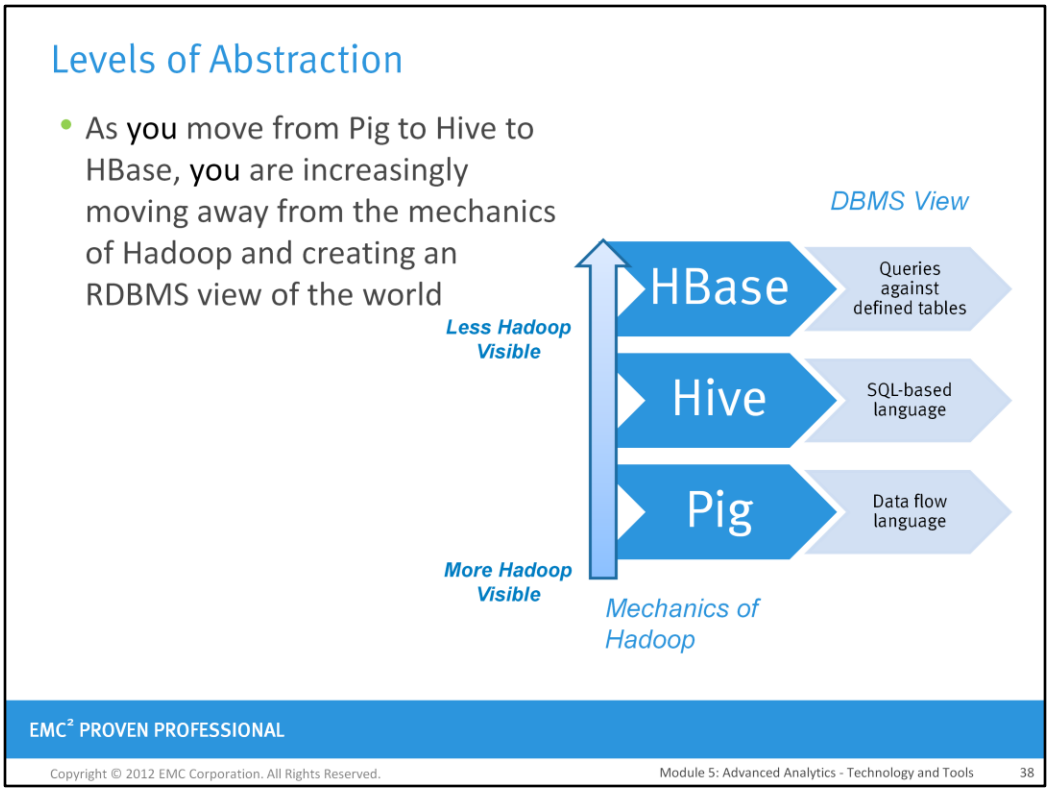
Hive (HiveQL) is an SQL-like query language for building MapReduce jobs.

HBase is a column-oriented database running over HDFS and supporting MapReduce and point queries.

HBase depends on **Zookeeper**, a coordination service for building distributed applications. We won't be discussion Zookeeper in this course except to say that it exists.

Mahout is a project to build scalable, machine learning systems by packaging machine learning algorithms as an executable library. You can specify input and output sources in HDFS to the algorithms, for example: each algorithm can take different parameters.

Mahout is still in early development (0.7 is the latest release). More information is available at http://mahout.apache.org/



In Pig and Hive, the presence of HDFS is very noticeable.

Pig, for example, directly supports most of the Hadoop file system commands.

Likewise, Hive can access data whether it's local or stored in an HDFS.

In either case, data can usually be specified via an HDFS URL (hdfs://<namenode>/path>). In the case of HBase, however, Hadoop is mostly hidden in the HBase framework, and HBase provides data to the client via a programmatic interface (usually Java).

Via these interfaces, a Data Scientist can focus on manipulating large datasets without concerning themselves with the inner working of Hadoop. Of course, a Data Scientist must be aware of the constraints associated with using Hadoop for data storage, but doesn't need to know the exact Hadoop command to check the file system.

What is Pig?

- Data flow language and execution environment for Hadoop

Two Main Elements

- A **data flow language** (Pig Latin)
- Two modes execution environment:
  - Local --- access a local file system
  - MapReduce when you're interested in the Hadoop environment
- When NOT to use Pig
  - If you only want to touch a small portion of the dataset (Pig eats it all)
  - If you do NOT want to use batch processing
    - Pig ONLY supports batch processing

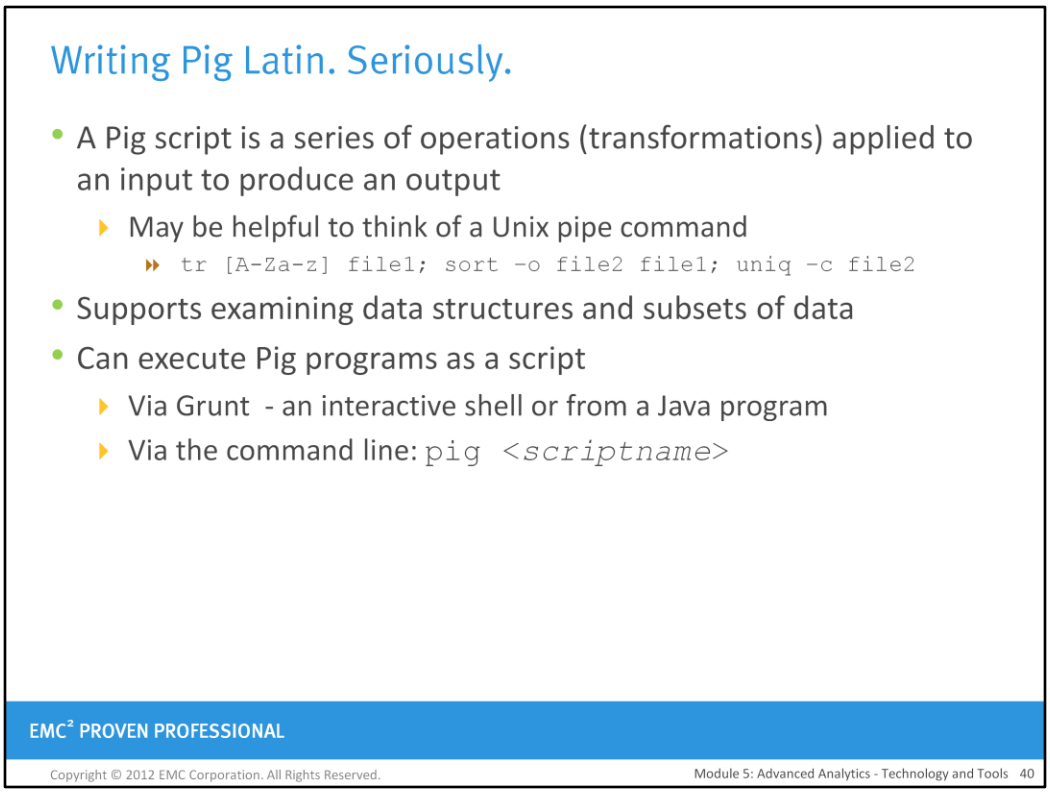
EMC² PROVEN PROFESSIONAL

Pig is a data flow language and an execution environment to access the MapReduce functionality of Hadoop (as well as HDFS).

Pig consists of two main elements:

1. A data flow language called Pig Latin (ig-pay atin-lay) and
2. An execution environment, either as a standalone system or one using HDFS for data storage.

A word of caution is in order: If you only want to touch a small portion of a given dataset, then Pig is not for you, since it only knows how to read all the data presented to it. Pig only supports batch processing of data, so if you need an interactive environment, Pig isn't for you.



One can think of a data flow programming model as a series of transforms or filters applied to an input stream.

In **Pig, each transform is defined as new input data source** (you'll see an example of this in the next slide). Descriptions of these transforms can be provided to Pig via a Pig Latin script, or interactively using Grunt, Pig's command-line interface.

Grunt also provides commands to query intermediate steps in the process. EXPLAIN shows the related MapReduce plan; DUMP lists out a dataset, and DESCRIBE describes the schema structure for that particular dataset.

Someone described Pig in this way: "You can process terabytes of data by issuing a half-dozen lines of Pig Latin from the console." Not a bad return on investment. Just make sure they are the right half-dozen lines.

Deconstructing Pig

	-- max_temp.pig -- Finds the max temperature by year
1	`records = LOAD 'data/samples.txt' AS (year:chararray, temperature:int, quality:int) ;`
2	`filtered_records = FILTER records BY temperature != 9999 AND (quality == 0 OR quality == 4 OR quality == 5 OR quality == 9);`
3	`grouped_records = GROUP filtered_records BY year;`
4	`max_temp = FOREACH grouped_records GENERATE group, MAX(filtered_records.temperature)`
5	`DUMP max_temp ;`

EMC² PROVEN PROFESSIONAL

This simple five line Pig program computes the maximum temperature per year gathered from a set of multi-year, multi-observations file of temperatures.

Line 1 - the data is read into a local variable named "records." Note that the code implies that records contains all the data, although this an abstraction.

Line 2 - the data set is filtered by removing missing values for temperature and by ensuring that *quality* takes on the values 0, 1, 4, 5, or 9.

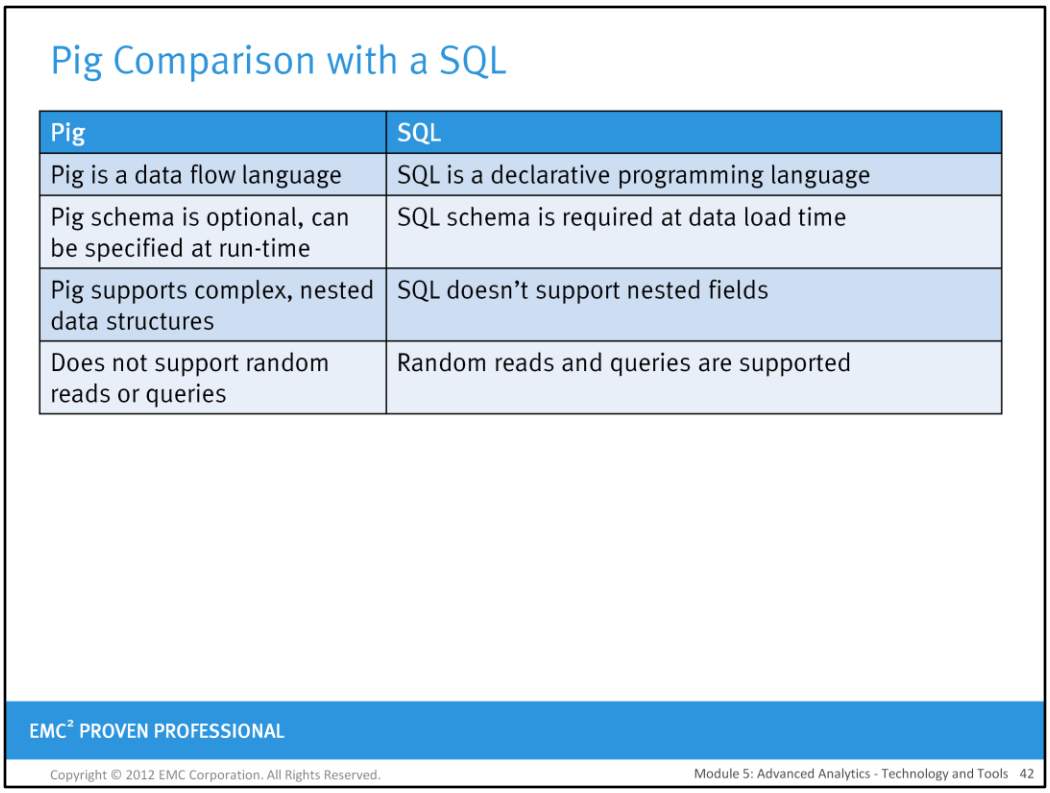
Line 3 - the filtered records are grouped by year (effectively sorting by year).

Line 4 - choose the maximum temperature value for each group.

Line 5 - dumps the final values stored in the variable max_temp.

Observe that the argument to the LOAD function could be a local file (file:///) or an HDFS URI (hdfs://<NameNode>/…). In the default case, since we're running HDFS, it will be found in HDFS.

The results of the LOAD command is a table that consists of a set of tuples (collections of variables). In line 3 we created a single record consisting of the year and a *bag* (an unordered collection of tuples), that contains a tuple for every observation made for that year (the year is actually repeated in each record, so the data looks like `<1949/ {(1949, 111,1), (1949, 78,1) …}>` This is still a representation of the <key/value> pair that we saw earlier, but in this case the value pair is a structured data type (a list consisting of multiple 3-tuple). Line 4 aggregates the data by the grouping variable and the maximum temperature for that year.



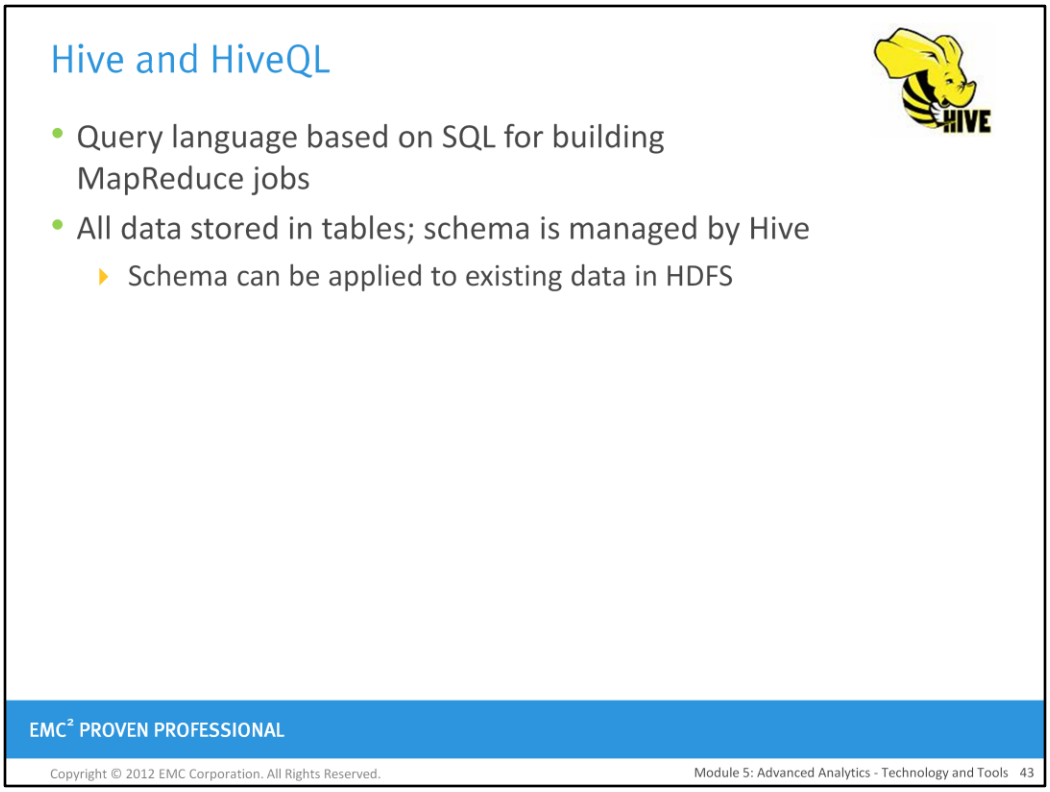
Pig acts as a front end to Hadoop; each command in Pig could theoretically become either a Map or a Reduce task (guess which ones are which in the preceding example).

PIG implements a data flow language where data flows through a series of transformations that alter the data in *some* way, and each step corresponds to a particular action. SQL, on the other hand, is a declarative programming language, and each SQL command represents a set of constraints that define the final output of the query.

For example, consider this SQL equivalent of our Pig example:

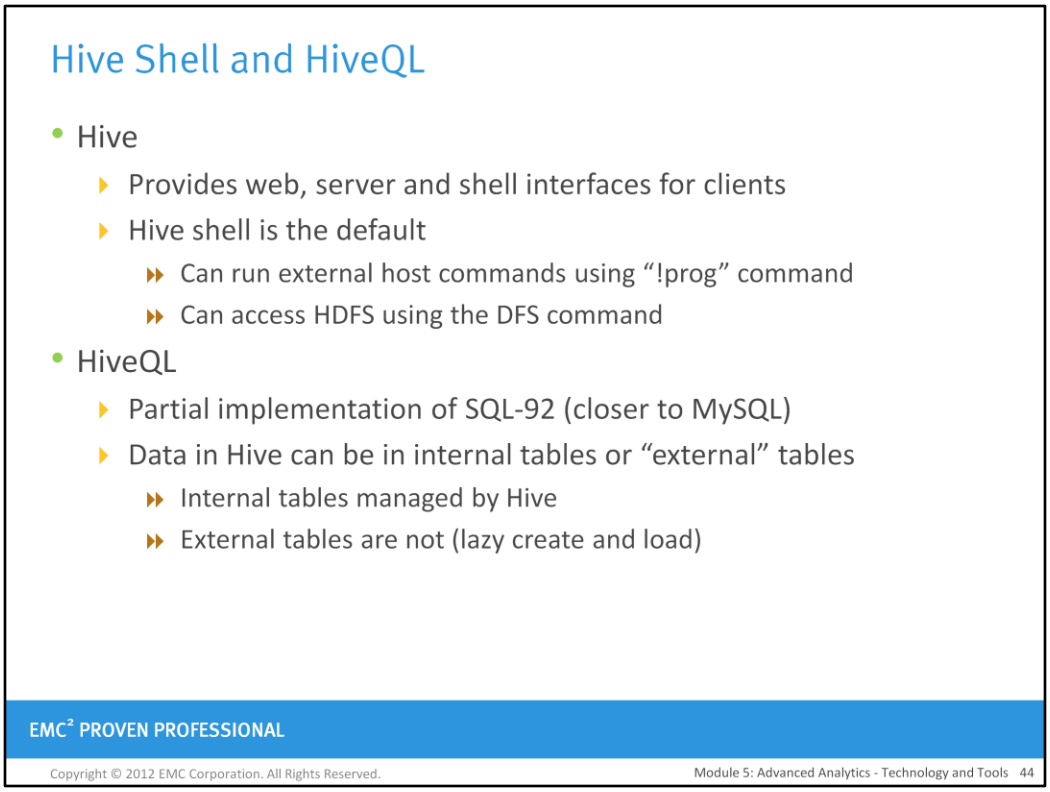
```
SELECT year, MAX(temperature) FROM (SELECT year, temperature quality
FROM data WHERE temperature <> 9999 AND quality NOT IN (0,1,4,5,9)
ORDER BY year GROUP BY year)
```

Unlike SQL, Pig doesn't require a schema to be present; if one is, it can be declared at run-time instead of when the RDBMS is created. SQL expects each row to represent a set of single-valued variables; Pig, on the other hand, can deal with arbitrarily nested data structures (bags within bags). SQL, however, supports random reads and queries; Pig reads the entire data set offered to it.



The Hive system is aimed at the Data Scientist with strong SQL skills. Think of Hive as occupying a space between Pig and an DBMS (although that DBMS doesn't have to be a Relational DBMS [RDBMS]).

In Hive, all data is stored in tables. The schema for each table is managed by Hive itself. Tables can be populated via the Hive interface, or a Hive schema can be applied to existing data stored in HDFS.



The hive program provides different functions depending on which commands are provided. The simplest invocation is simply "hive", that brings up the Hive shell. From there, you can enter Hive SQL commands interactively, or these commands could be combined into a single script file.

hive hwi is the Hive web interface whereby one can browse existing database schemas and create sessions for issuing database queries and Hive commands. The interface is available at http://<*hivehost*>:9999/hwi.

Hive hiveserver will start Hive as a server listening on port 10000 that provides a Thrift and a JDBC/ODBC interface to Hive databases.

Data for Hive can be stored in Hive's internal tables (managed tables) or can be retrieved from data in the filesystem (HDFS). An example of creation of an external table is

CREATE EXTERNAL TABLE my_ext_data (dummy STRING) LOCATION '/opt/externalTable' ;

LOAD DATA INPATH '/opt/externalTable' INTO TABLE my_ext_data ;

The existence of this data isn't checked when these statements are executed, nor is data loading in Hive's datastore. Hence, the notion of "lazy create and lazy load."

Temperature Example: Hive

	Example Hive Code
1	`CREATE TABLE records (year STRING, temperature INT, quality INT) ROW FORMAT DELIMITED FIELDS TERMINATED by '\t' ;`
2	`LOAD DATA LOCAL 'data/samples.txt' OVERWRITE INTO TABLE records ;`
3	`SELECT year, MAX(temperature) FROM records WHERE temperature != 9999 AND (quality == 0 OR quality == 1 OR quality = 4 OR quality == 5 OR quality == 9) GROUP BY year;`

EMC² PROVEN PROFESSIONAL

Let's go back to our example of calculating the maximum temperature for a given year from thousand and thousands of weather observations from hundreds of weather stations.

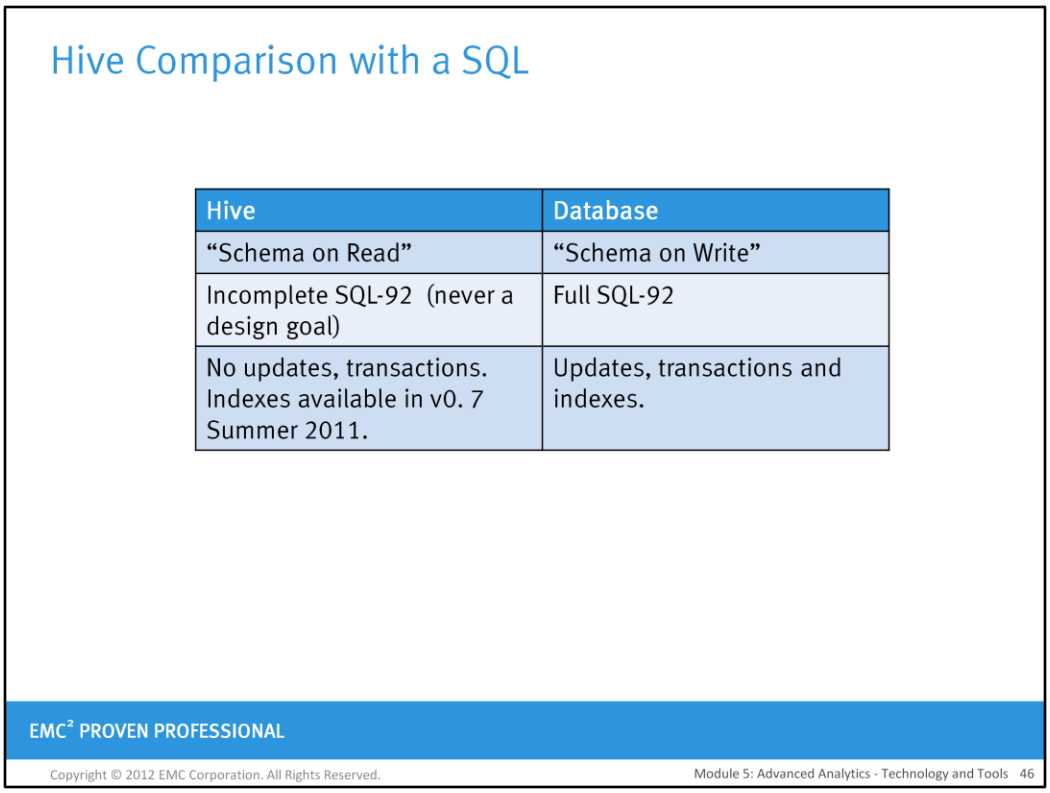
Line 1 defines your table and states that our input consists of tab-delimited fields.

In line 2, you encounter our old favorite LOAD DATA again with a slightly different syntax.

Line 3 looks like a standard SQL query that produces a relation consisting of a year and the max temperature for that year. The ROW FORMAT clause is a Hive-specific addition.

Hive maintains its own set of tables; these tables could exist on a local file system or in HDFS as /usr/hive/XXXXX. A directory in the filesystem corresponds to a particular table.

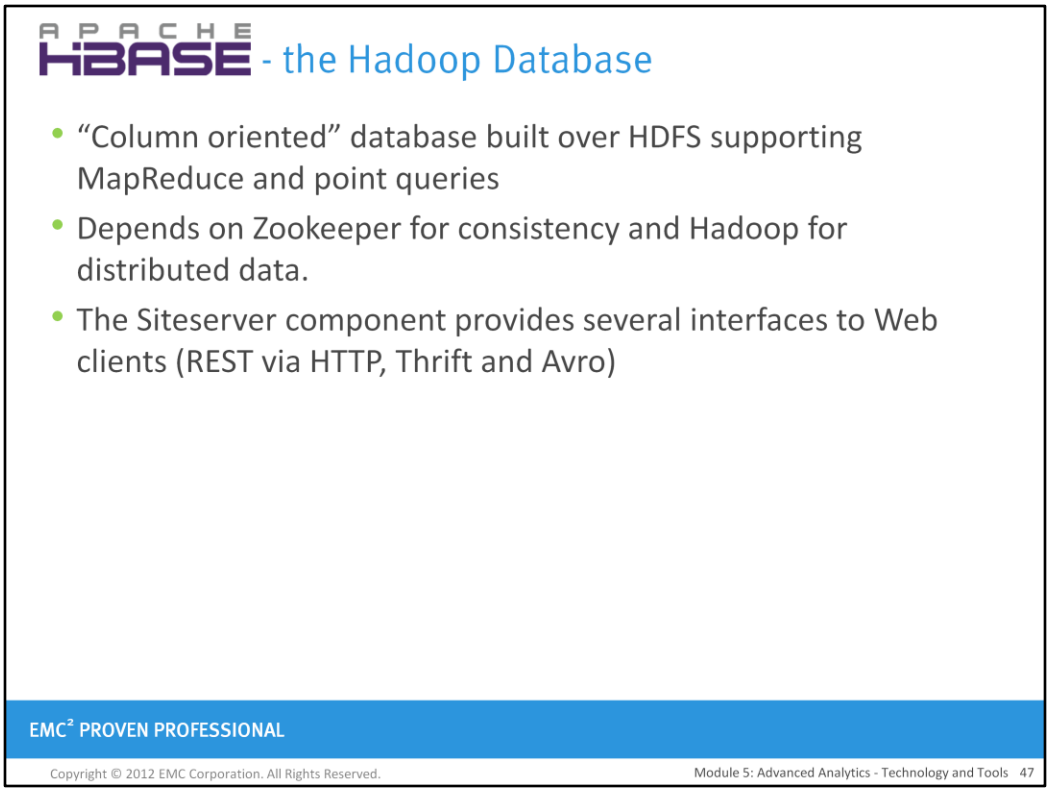
Hive does not implement the full SQL-92 standard, and additionally provides certain clauses that don't appear in standard SQL ("ROW FORMAT …" is one such example).



In most traditional DBMS, the database description (or schema) is read and applied when the data is loaded. If the data doesn't match the schema (specifically, the table into which it is read), then the load fails. This is often called "Schema on Write."

Hive, on the other hand, doesn't attempt to apply the schema until the data is actually read when someone issues a query. This results in fast loads as well as supporting multiple schemas for the same data (only defining as many variables as needed for your particular analysis). In addition, the actual format of the data may not be known because queries against the data haven't been defined.

Updates and transactions aren't supplied with Hive. Indexes are available as of Summer 2011 in Hive 0.7. If concurrent access to tables is desired, then the application must roll its own. That said, the Hive project is working towards integration with the HBase project that does provide row updates. See the Hive project page at <http://hive.apache.org/> for further details.

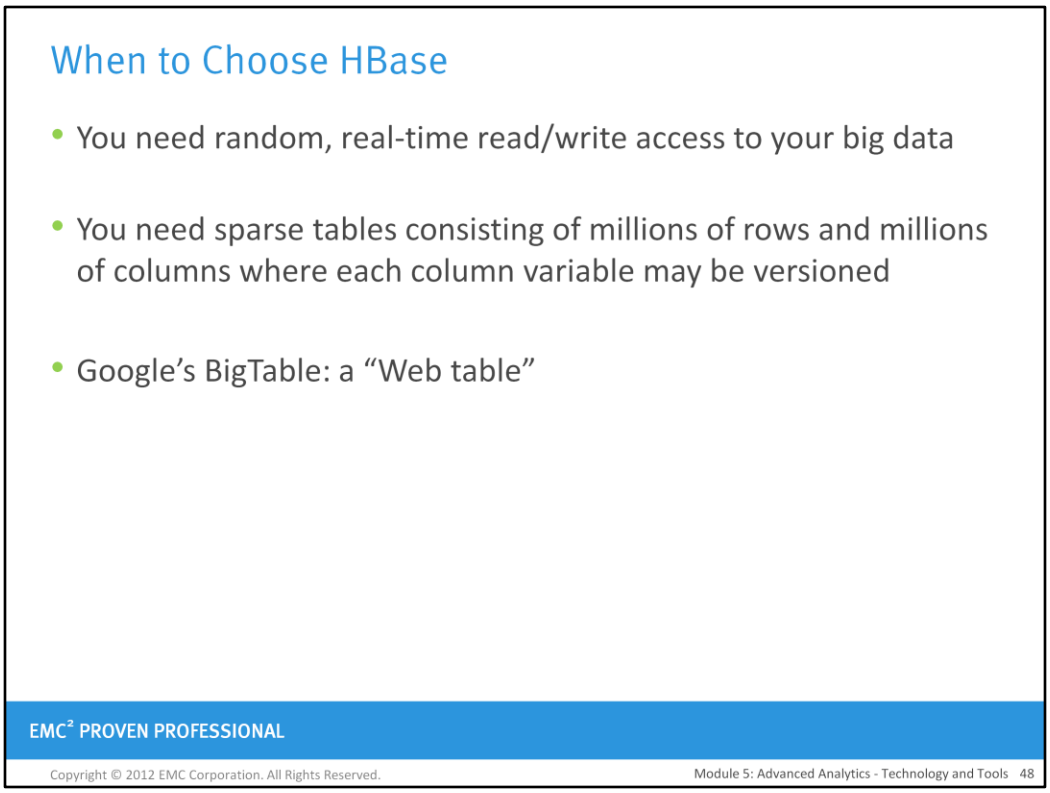


HBase represents a further layer of abstraction on Hadoop. HBase has been described as "a distributed column-oriented database [data storage system]" built of top of HDFS.

Note that HBase is described as managing *structured* data. Each record in the table can be described as a key (treated as a byte stream) and a set of variables, each of which may be versioned. It's not a structure in the same sense as an RDBMS is structured.

HBase is a more complex system than what we have seen previously. HBase uses additional Apache Foundation open source frameworks: Zookeeper is used as a co-ordination system to maintain consistency, Hadoop for MapReduce and HDFS, and Oozie for workflow management. As a Data Scientist, you probably won't be concerned overmuch with implementation, but it is useful to at least know the names of all the moving parts.

HBase can be run from the command line, but also supports REST (Representational State Transfer – think HTTP) and Thrift and Avro interfaces via the Siteserver daemon. Thrift and Avro both provide an interface to send and receive serialized data (objects where the data is "flattened" into a byte stream).

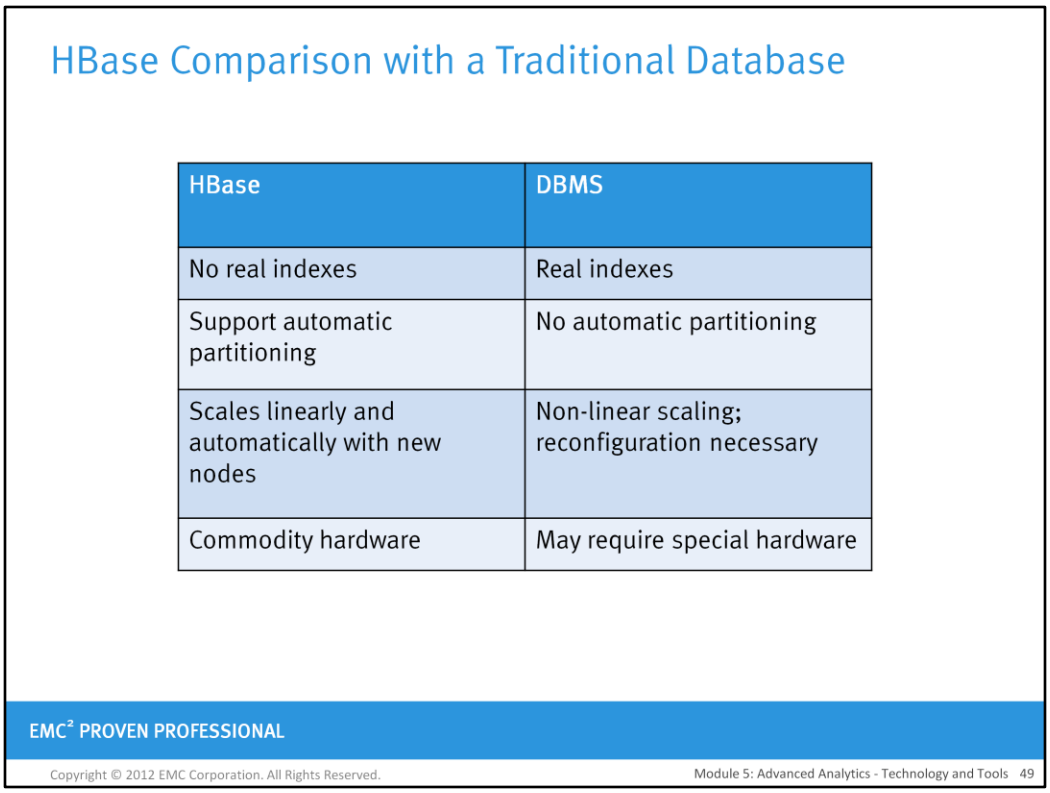


HBase has been described like this: "[its] forte is when real-time read-write random-access to very large datasets is required." HBase is an open source version of Google's BigTable, and it's instructive to read the definition of BigTable by the original authors:

> *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.*

Note that HBase is described as managing *structured* data. Each record in the table can be described as a key (treated as a byte stream) and a set of variables, each of which may be versioned. It's not a structure in the same sense as an RDBMS is structured, but it does have structure nonetheless. And because HBase is not constrained in the same was as an RDBMS system is constrained, HBase designs can take advantage of the physical layout of the table on disk to increase performance.

It's useful to recall that Google's BigTable was designed to store information about Web URLs (Web documents). Fields in this table could be versioned (new versions of an HTML page, for example) and the table could be updated frequently as web-crawlers discovered new data. One design decision to speed access to URLs from the same site was to reverse the order of the URL: instead of media.google.com, the URL would be stored as com.google.media, ensuring that other Google URLs would be "reasonably" close.



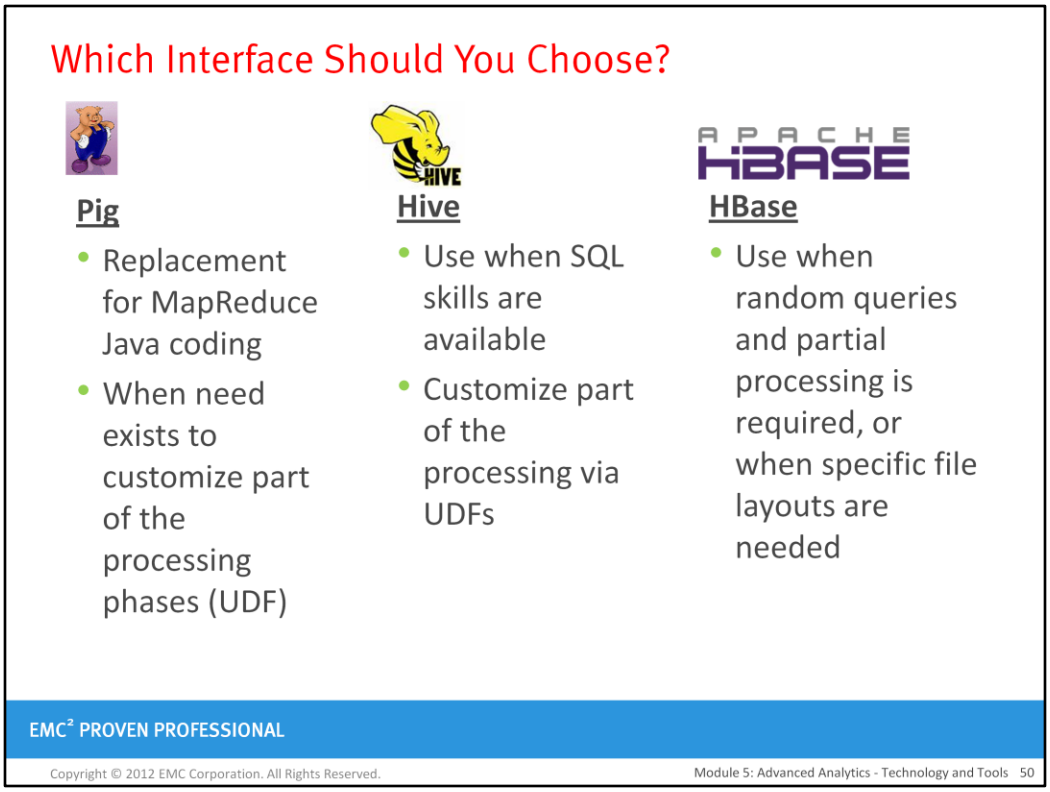
Although HBase may look like a traditional DBMS, it isn't.

HBase is a "distributed, column-oriented data storage system that can scale tall (billions of rows), wide (billions of columns), and can be horizontally partitioned and replicated across thousands of commodity servers automatically."

The HBase table schemas mirror physical storage for efficiency; a RDBMS doesn't. (the RDBMS schema is a logical description of the data, and implies no specific physical structuring.)

Most RDBMS systems require that data must be consistent after each transaction (ACID prosperities). DBMS systems like HBase don't suffer from these constraints, and implement eventual consistency. This means that for some systems you cannot write a value into the database and immediately read it back in. Strange, but true.

Another of HBase's strengths is in its wide open view of data – HBASE will accept almost anything it can cram into an HBase table.



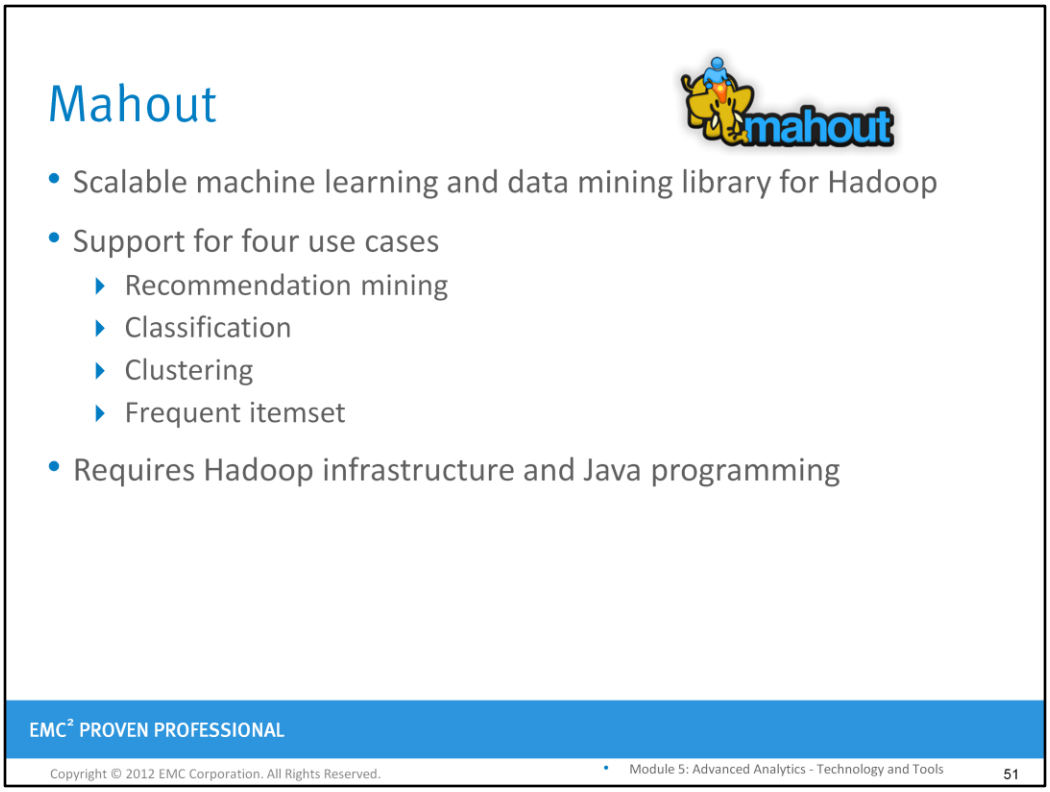
If you are Java-savvy, or you have scripting skills (shell plus Ruby/Python/Perl/Tcl/Tk), you might simply want to use the existing Hadoop framework for MapReduce tasks.

If your talent lies in other areas, such as database administration or functional programming, you might wish to choose a different method of accessing your data.

Pig provides an abstraction above that of MapReduce for HDFS, and makes the interface simpler to use. Both Pig and Hive support UDFs (User Defined Functions).

Hive provides an SQL-like interface to data that may be stored in HDFS, but Hive tables don't meet the definition of a RDBMS.

HBase, as the "Hadoop Database," leverages Hadoop/HDFS for data storage and the Zookeeper system for co-ordination. Since there is no fixed schema *per se,* attributes (columns) can be added to a dataset without requiring programs to change to address the extra data; attribute values may be versioned to record changes to a particular value. Bulk loading can be accomplished by having MapReduce write files in HBase internal format directly into HDFS with an order of magnitude increase in populating the database.

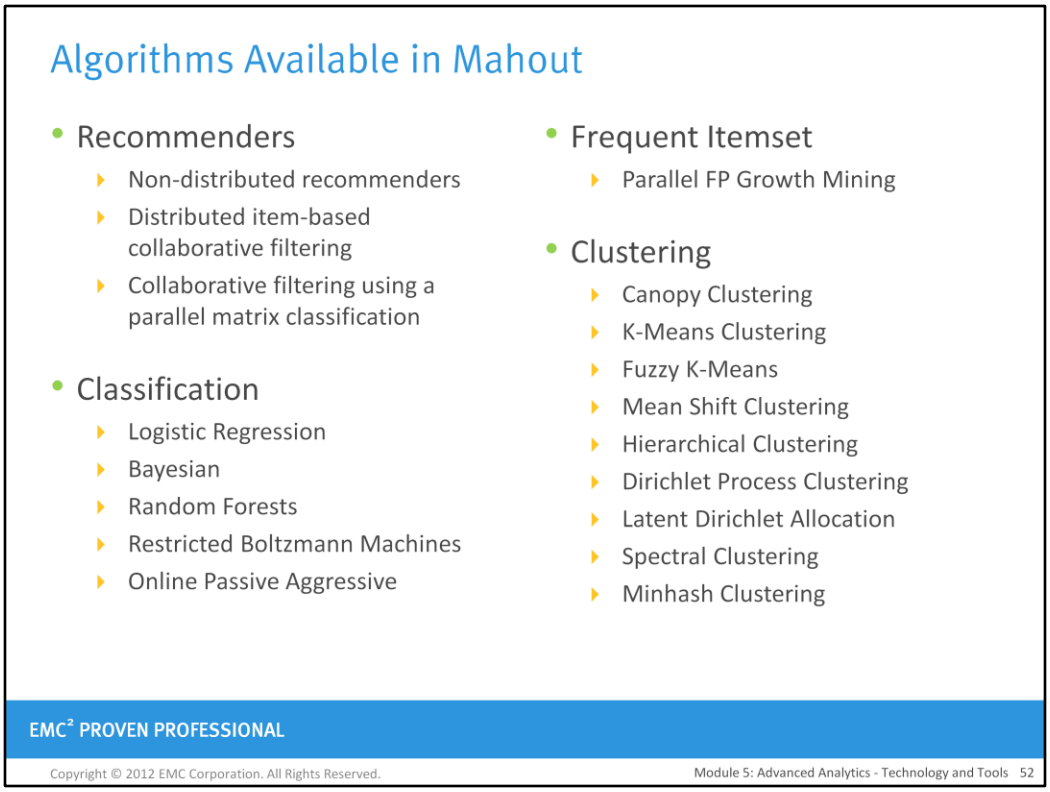


Mahout is a set of machine learning algorithms that leverages Hadoop to provide both data storage and the MapReduce implementation.

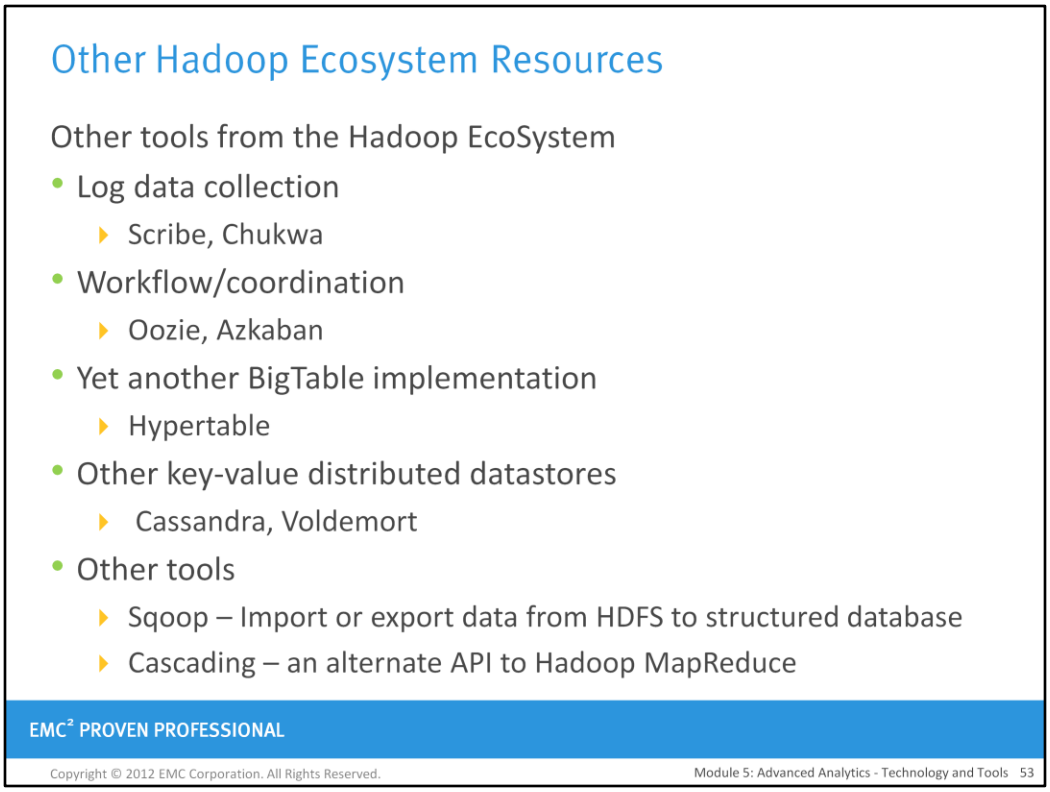
The *mahout* command is itself a script that wraps the Hadoop command and executes a requested algorithm from the Mahout job jar file (jar files are **J**ava **AR**chives, and are very similar to Linux tar files [tape archives]). Parameters are passed from the command line to the class instance.

Mahout mainly supports four use cases:

- ***Recommendation mining*** takes users' behavior and tries to find items users might like. An example of this is LinkedIn's "People You Might Know" (PYMK).
- ***Classification*** learns from existing categorized documents what documents of a specific category look like and is able to assign unlabelled documents to the (hopefully) correct category.
- ***Clustering*** takes documents and groups them into collections of topically related documents based on word occurrences.
- ***Frequent itemset mining*** takes a set of item groups (for example, terms in a query session, shopping cart content) and identifies which individual items usually appear together.
If you plan on using Mahout, remember that these distributions (Hadoop and Mahout) anticipate running on a *nix machine, although a Cygwin environment on Windows will work as well (or rewriting the command scripts in another language, say as a batch file on Windows). It goes without saying that a compatible working version of Hadoop is required. Lastly, Mahout requires that you program in Java: no other interface outside of the command line is supported.



Mahout provides a number of different algorithms to support its use cases. Some of these algorithms are listed. We've only included those algorithms that are curently implemented (at the time of this version of the course) and don't have problem reports issued against them. Other algorithms are in various stages of development: check the website at <http://mahout.apache.org/> for more details.



We have already been introduced to HBase, Hive, Pig, and Zookeeper. Other tools/system based on Hadoop HDFS and MapReduce include:

Howl –- mixture of Hive and OWL (another interface to HDFS data).

Oozie –- a workflow/coordination system to manage Apache Hadoop(TM) jobs.

Zookeeper –- a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

Chukwa: –- a Hadoop subproject devoted to large-scale log collection and analysis.

Cascading -- an alternative API to Hadoop MapReduce.

Scribe –- a server designed for the real-time streaming of log data.

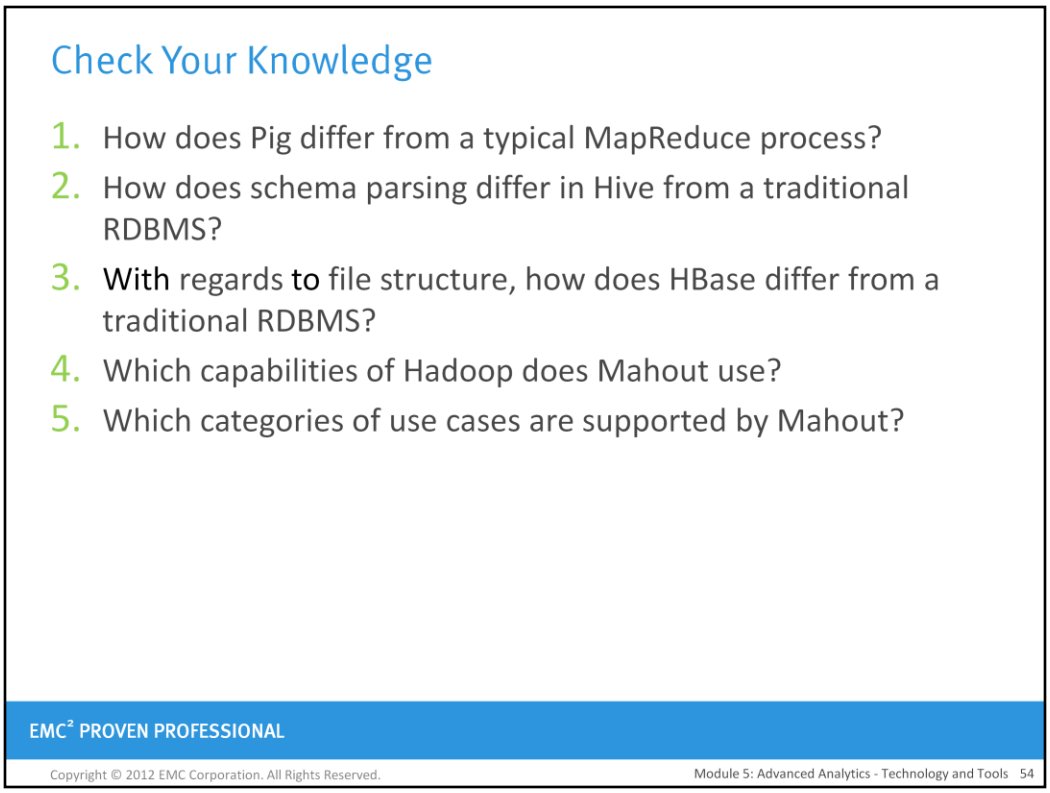
Cassandra –- another database system, noted for its elasticity, durability and availability.

Hypertable –- another BigTable implementation that runs over Hadoop.

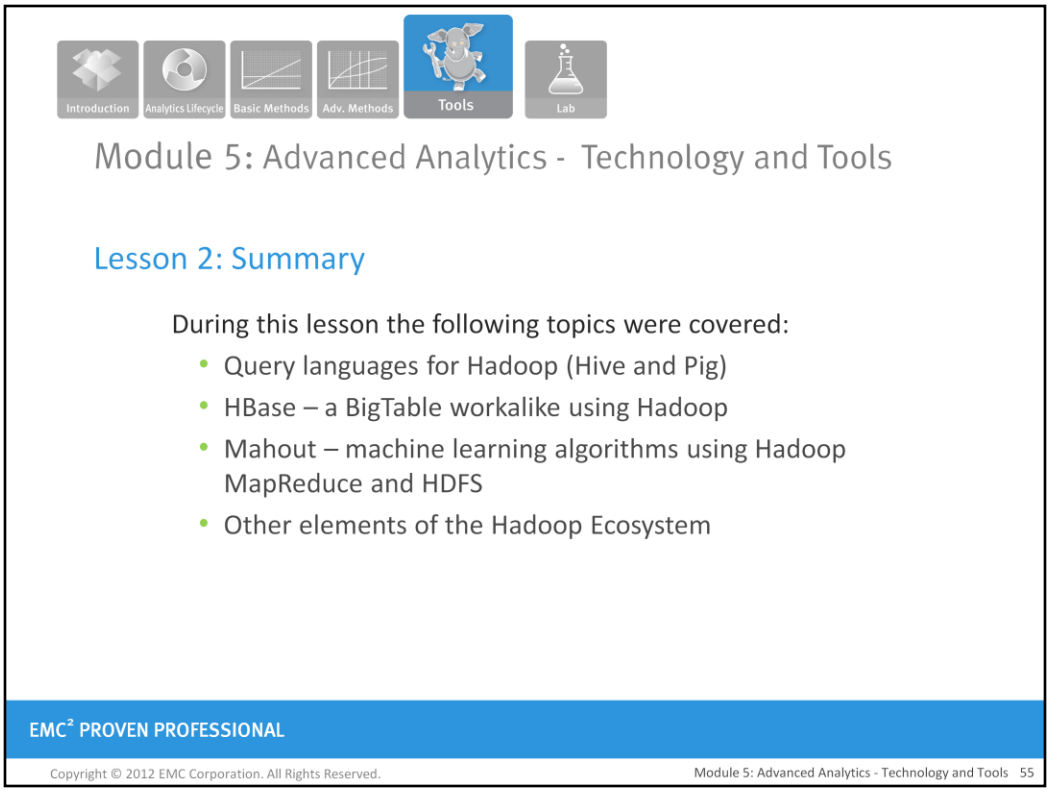
Voldemort –- a distributed key-value storage system, used at LinkedIn.

Azkaban –- workflow scheduler, developed at LinkedIn.

Sqoop –- a tool for importing and exporting data from Hadoop to structured databases.



Please take a moment to answer these questions. Write you answer in the space below.

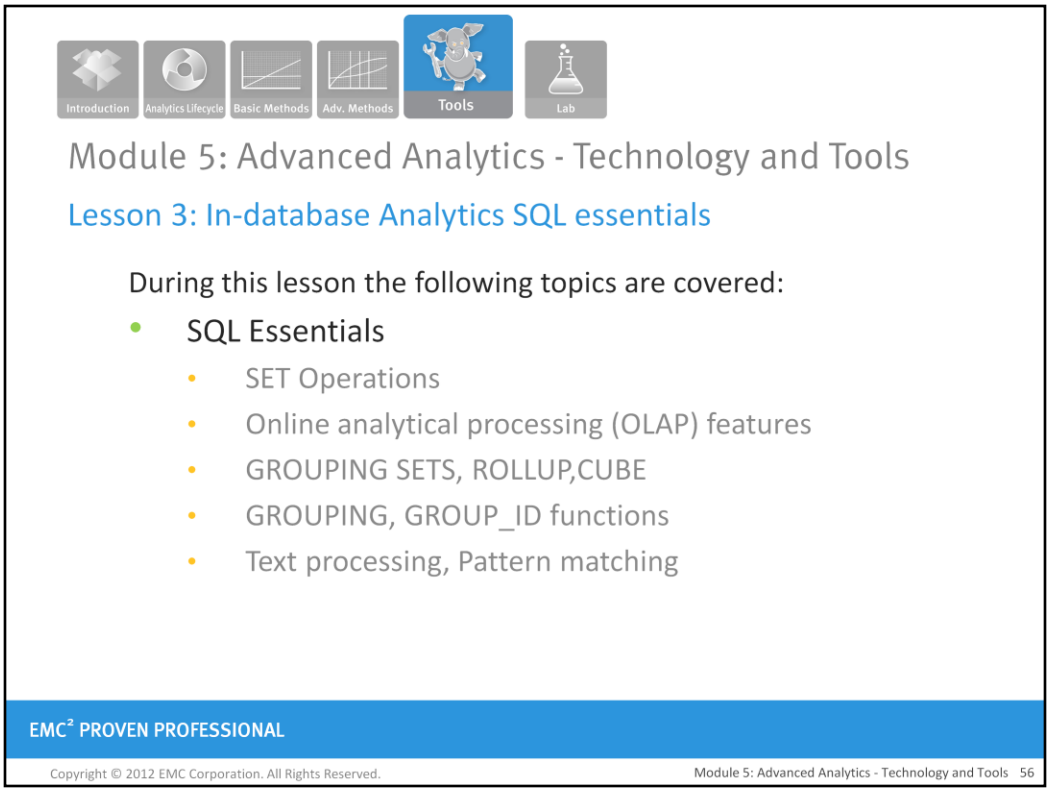


Lesson 2 covered:

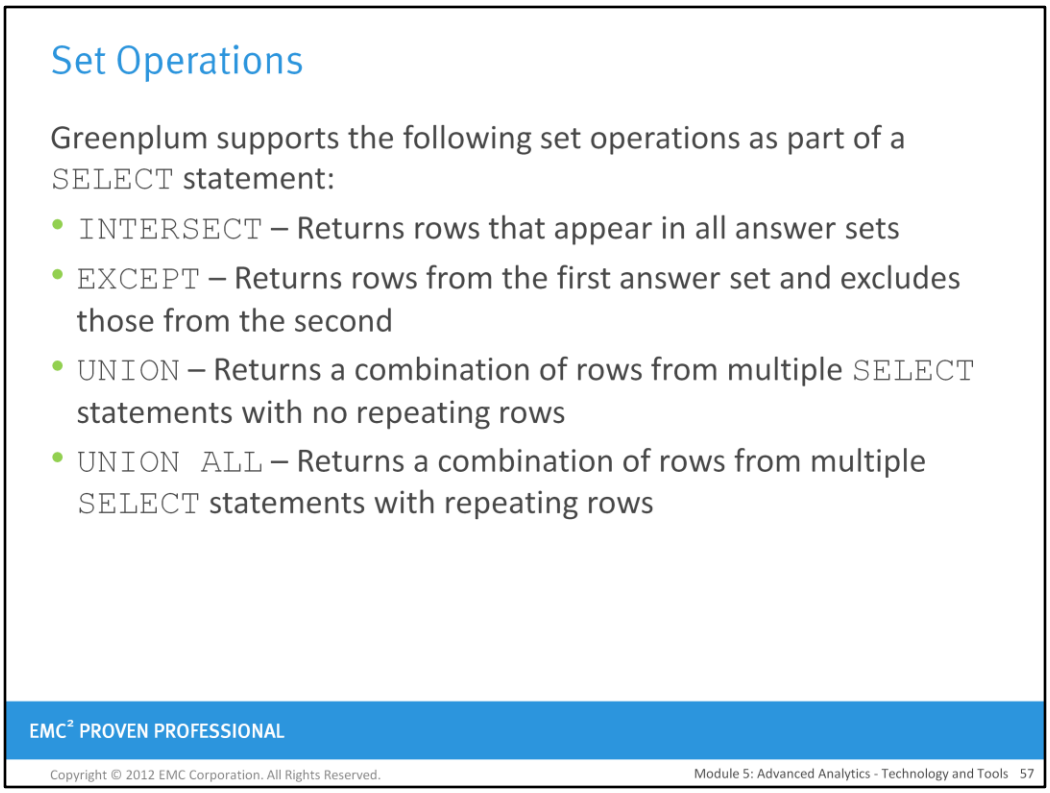
Hive and Pig – Hadoop query languages

HBase – a BigTable workalike using Hadoop

Mahout – machine learning algorithms and Hadoop MapReduce



These topics are covered in this lesson.



Set Operations

Set operators:

- Manipulate the results sets of two or more queries by combining the results of individual queries into a single results set.
- Do not perform row level filtering.

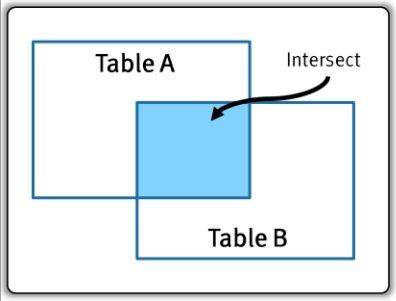
Set operations supported by Greenplum are:

- `INTERSECT` which returns rows that appear in all answer sets generated by individual `SELECT` statements.
- `EXCEPT` returns all rows from the first `SELECT` except for those which also selected by the second `SELECT`. This operation is the same as the `MINUS` operation.
- `UNION` combines the results of two or more `SELECT` statements. There will be no repeating rows.
- `UNION ALL` combines all the results of two or more `SELECT` statements. There may be repeating rows.

Set Operations – INTERSECT

INTERSECT:

- Returns only the rows that appear in both SQL queries
- Removes duplicate rows




```
SELECT   t.transid,
         c.custname
FROM     facts.transaction t
    JOIN dimensions.customer c
      ON c.customerid = t.customerid

INTERSECT

SELECT t.transid,
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```

EMC² PROVEN PROFESSIONAL

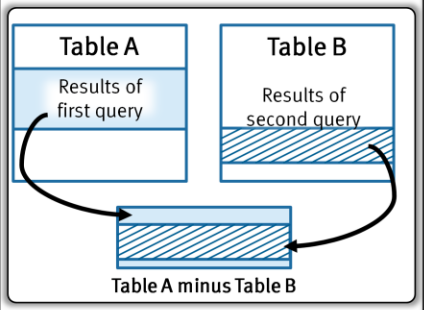
Set Operations – INTERSECT

A set operation takes the results of two queries and returns only the results that appear in both result sets. Duplicate rows are removed from the final set returned.

Set Operations – EXCEPT

EXCEPT:

- Returns all rows from the first SELECT statement
- Omits all rows that appear in the second SELECT statement

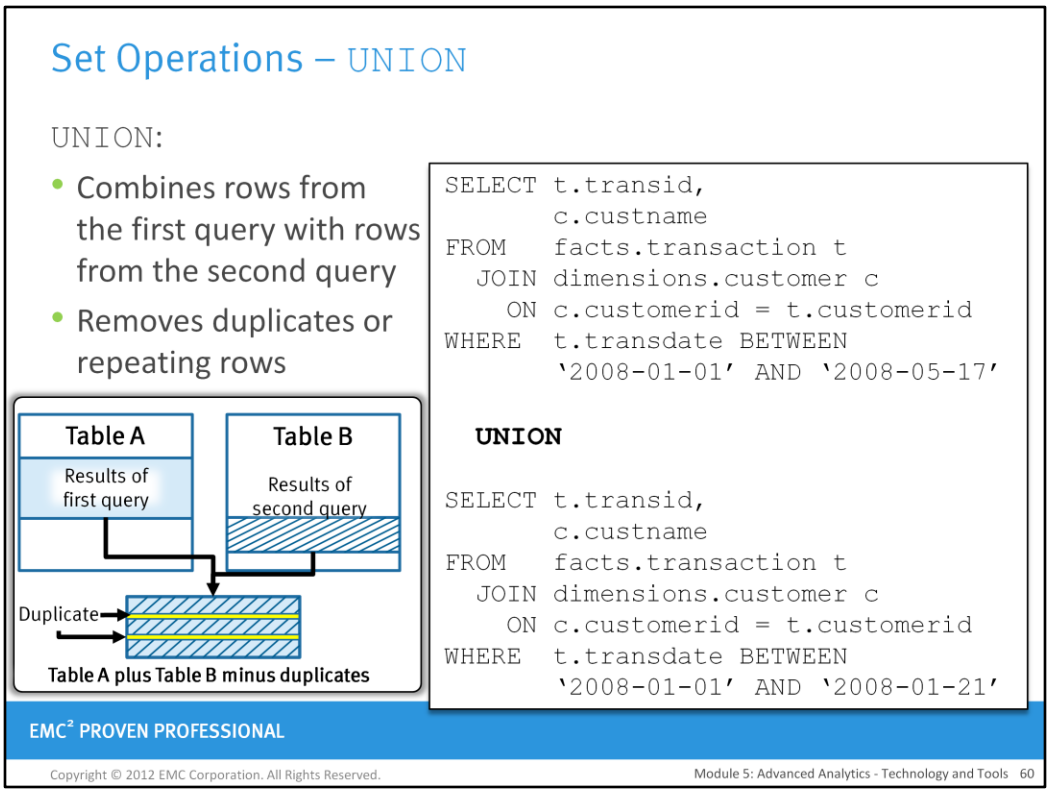



```
SELECT   t.transid,
         c.custname
FROM     facts.transaction t
    JOIN dimensions.customer c
      ON c.customerid = t.customerid

  EXCEPT

SELECT t.transid,
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```

EMC² PROVEN PROFESSIONAL

Set Operations – EXCEPT

The EXCEPT set operation takes the distinct rows of the first query and returns all of the rows that do not appear in the result set of the second query.



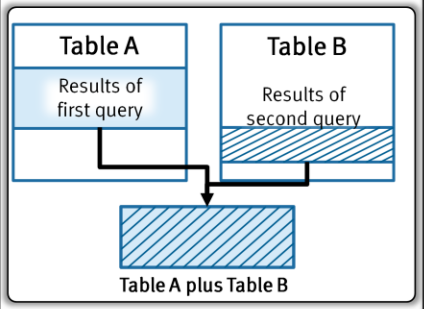
Set Operations – `UNION`

A union operation combines the results of the SELECT statement from the first table with the results from the query on the second table. The result set does not contain any repeating rows.

Set Operations – UNION ALL

UNION ALL:

- Combines rows from the first query with rows from the second query
- Does not remove duplicates rows

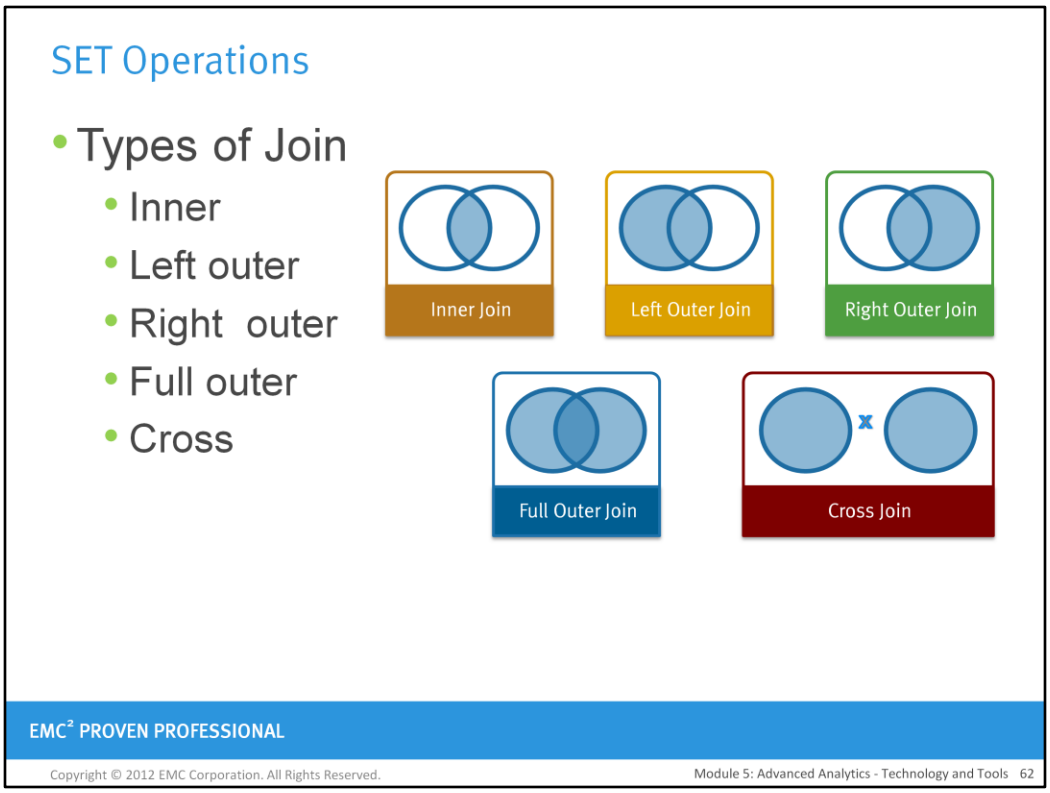



```
SELECT t.transid,
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-05-17'

  UNION ALL

SELECT t.transid,
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```

EMC² PROVEN PROFESSIONAL

Set Operations – `UNION ALL`

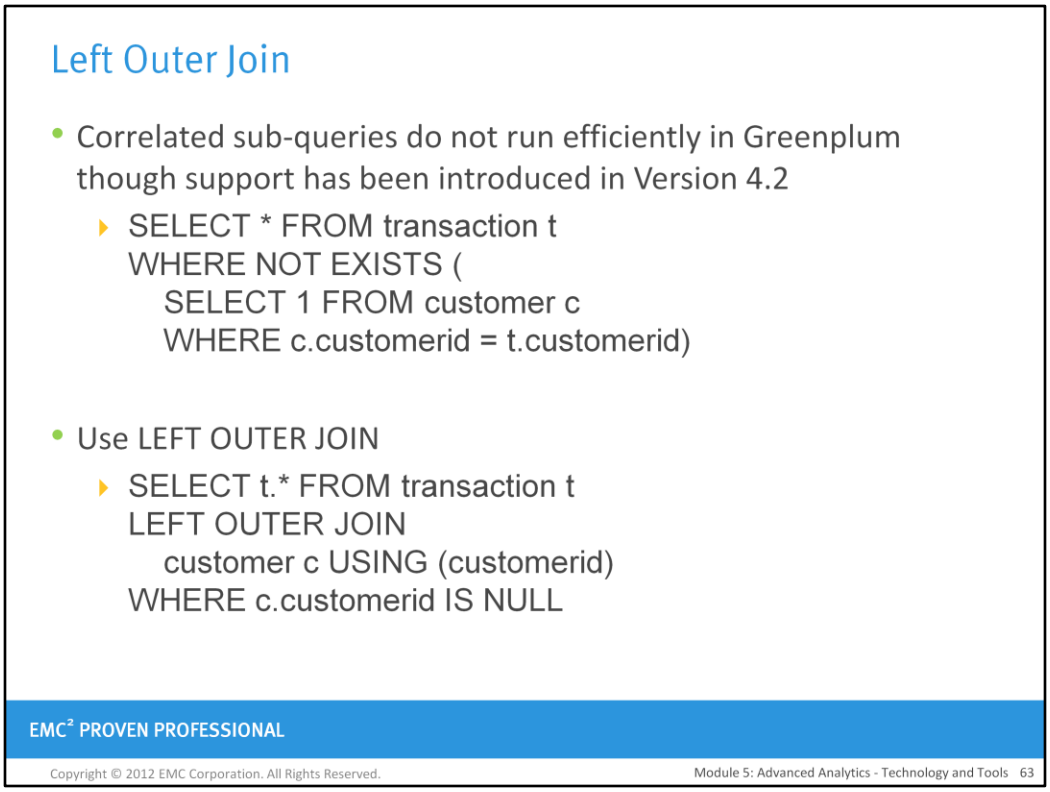
The UNION ALL set operation is like the UNION operation but it does not remove duplicate or repeating rows.



The type of SET operations you can perform are:

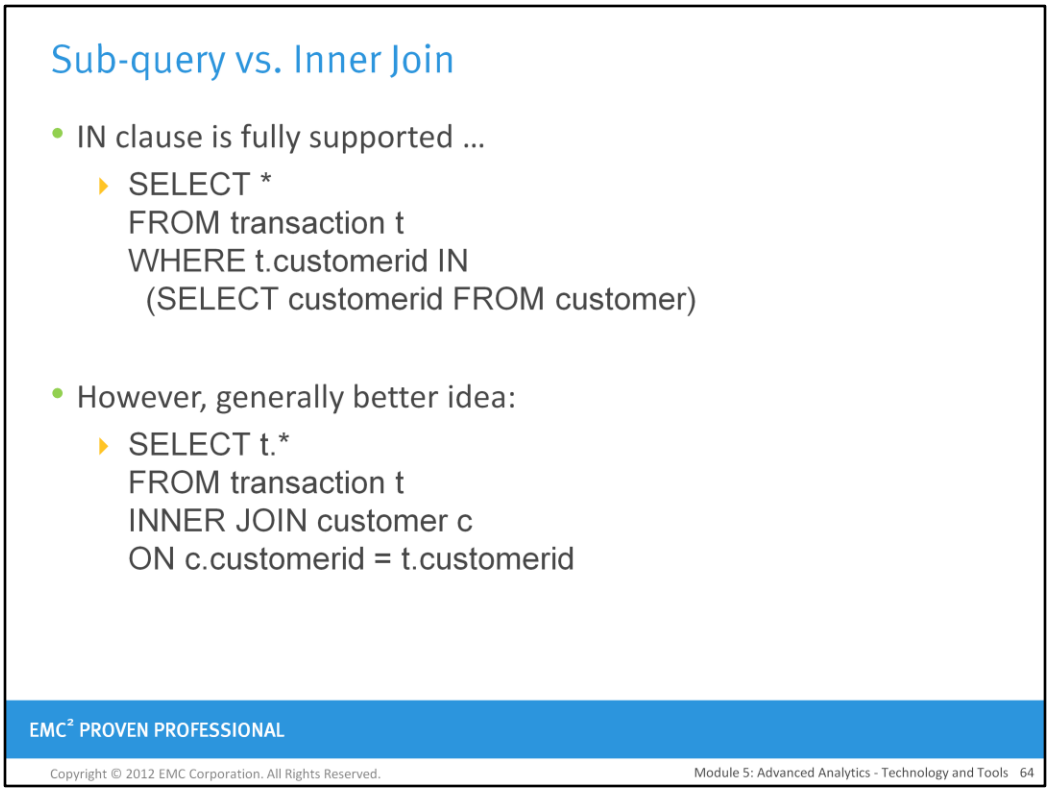
- Inner join – The inner join is possibly the most common type of join. The resulting data set is obtained by combining two tables on a common column. Each row of the left table is compared against each row of the right table. All matching rows are returned as part of the result set. An equijoin is an inner join that uses only equality comparisons in the join predicate.
- Left outer join – Left outer join returns all of the rows from the left table even if there is no matching row in the right table. It also returns matching rows from the right table. Rows in the right table that do not match are not included as part of the result set.
- Right outer join – Right outer join returns all of the rows from the right table even if there is no matching row in the left table. It also returns the matching rows from the left table.
- Full outer join – Full outer join returns all rows from both tables where there is a match and returns NULL for rows that do not have a match.
- Cross join – Cross join returns the Cartesian product of rows from tables in the join. The resulting data set consists of a combination of each row in the left table with each row in the right table. Two tables, each with five rows, will produce a resulting data set that contains twenty-five rows.

.



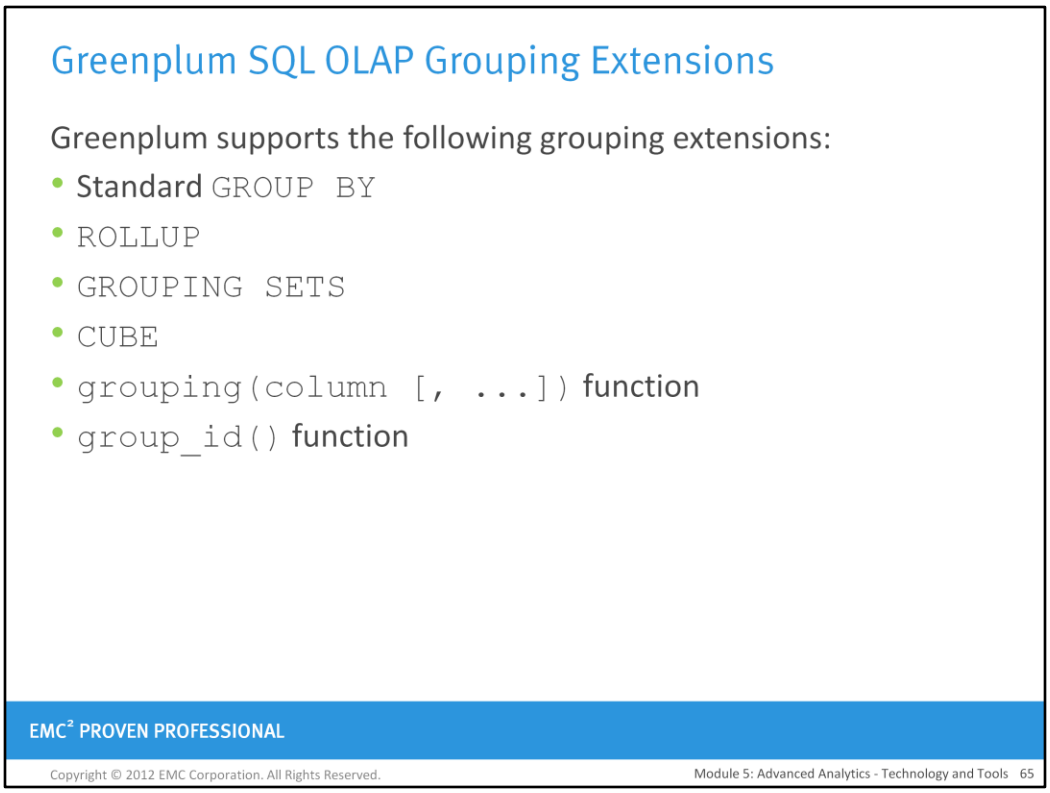
Correlated sub queries are supported in a Greenplum environment 4.2 onwards. The first example shown in the slide is a correlated sub query in which we have a nested correlated sub query using the first variable from the first SELECT statement (on table “transaction”) used for selection with table “Customer” in the query nested with a WHERE clause. In the context of MPP architecture of Greenplum supporting correlated sub queries are not efficient.

The code example shown at the bottom accomplishes the same with a LEFT OUTER JOIN. It is recommended that the multidimensional queries that are traditionally done with sub queries and correlated sub queries be optimally coded with the proper use of the MPP architecture of Greenplum.



Sub queries and nested queries are commonly used for multi-dimensional queries in which we use IN clause with another SELECT statement. Sub-queries such as the one shown above are supported in a Greenplum environment.

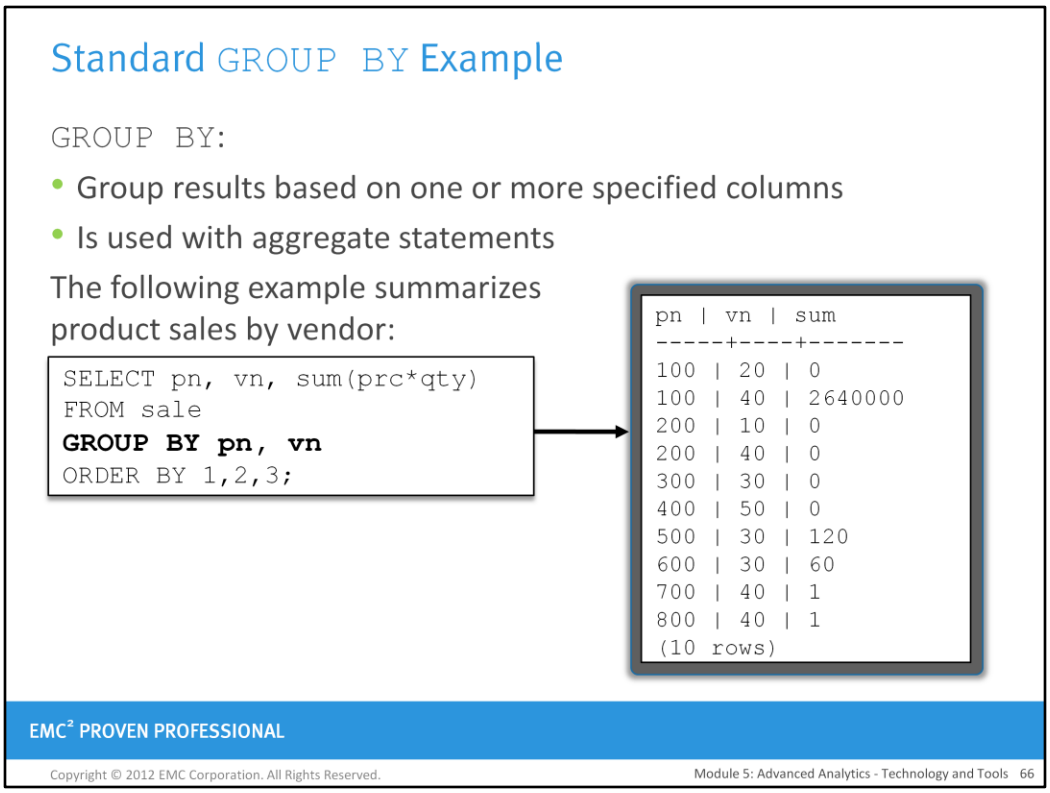
They are supported and valid but it is generally a good idea to use a INNER JOIN to accomplish the same result. The performance query with INNER JOIN is far superior to the nested query in the first example.



Greenplum SQL OLAP Grouping Extensions

Greenplum introduced support for extensions to the standard `GROUP BY` clause, which is fully supported. These clauses can simplify the expression of complex groupings:

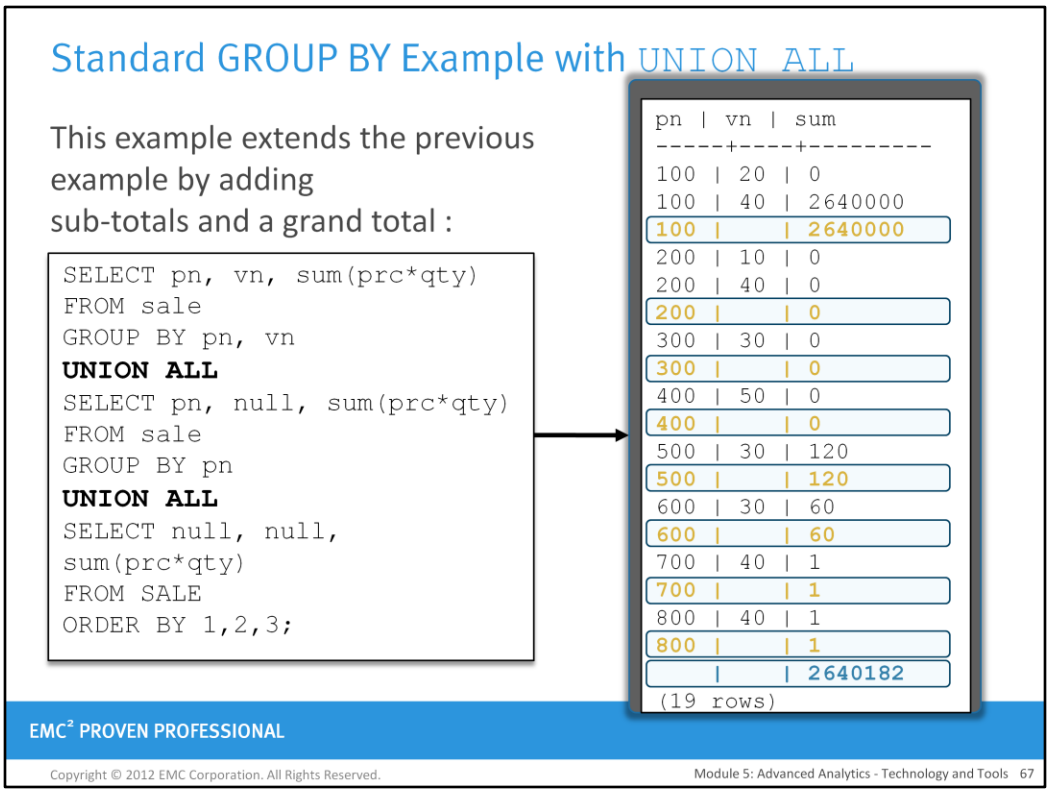
- `ROLLUP` – This extension provides hierarchical grouping.
- `CUBE` – Complete cross-tabular grouping, or all possible grouping combinations, is provided with this extension.
- `GROUPING SETS` – Generalized grouping is provided with the `GROUPING SETS` clause.
- `grouping` function – This clause helps identify super-aggregated rows from regular grouped rows.
- `group_id` function – This clause is used to identify duplicate rows in grouped output.



Standard `GROUP BY` Example

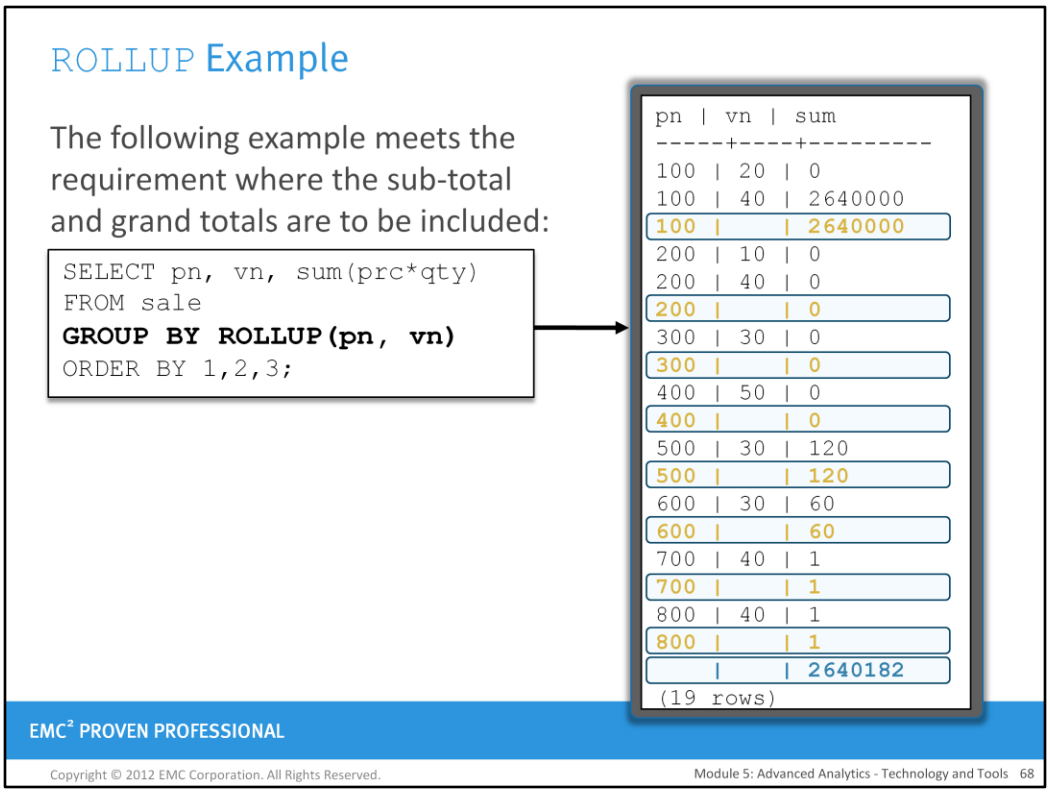
The standard `GROUP BY` clause groups results based on one or more columns specified. It is used in conjunction with aggregate statements, such as `SUM`, `MIN`, or `MAX`. This helps to make the resulting data set more readable.

The slide shows an example of a standard `GROUP BY` clause used to summarize product sales by vendor.



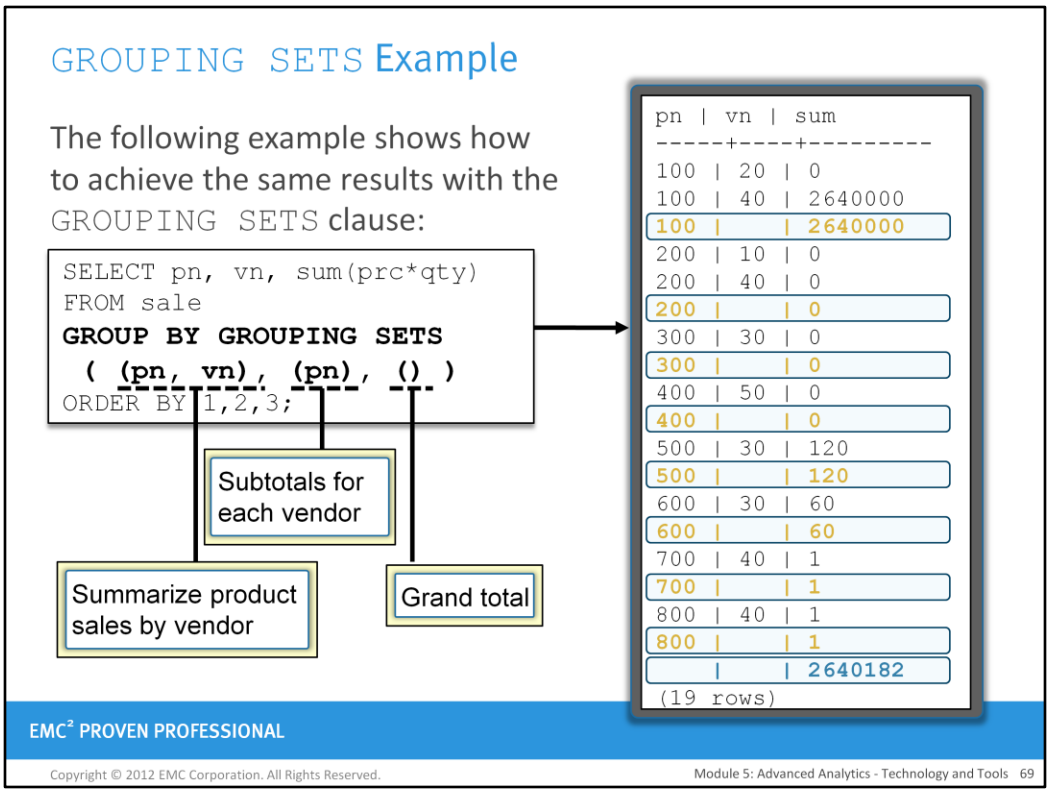
Standard `GROUP BY` Example with `UNION ALL`

In this follow-on example, the requirements for the query have been extended to include sub-totals and a grand total. You would need to use a `UNION ALL` to continue the grouping and provide for the additional requirements.



`ROLLUP` Example

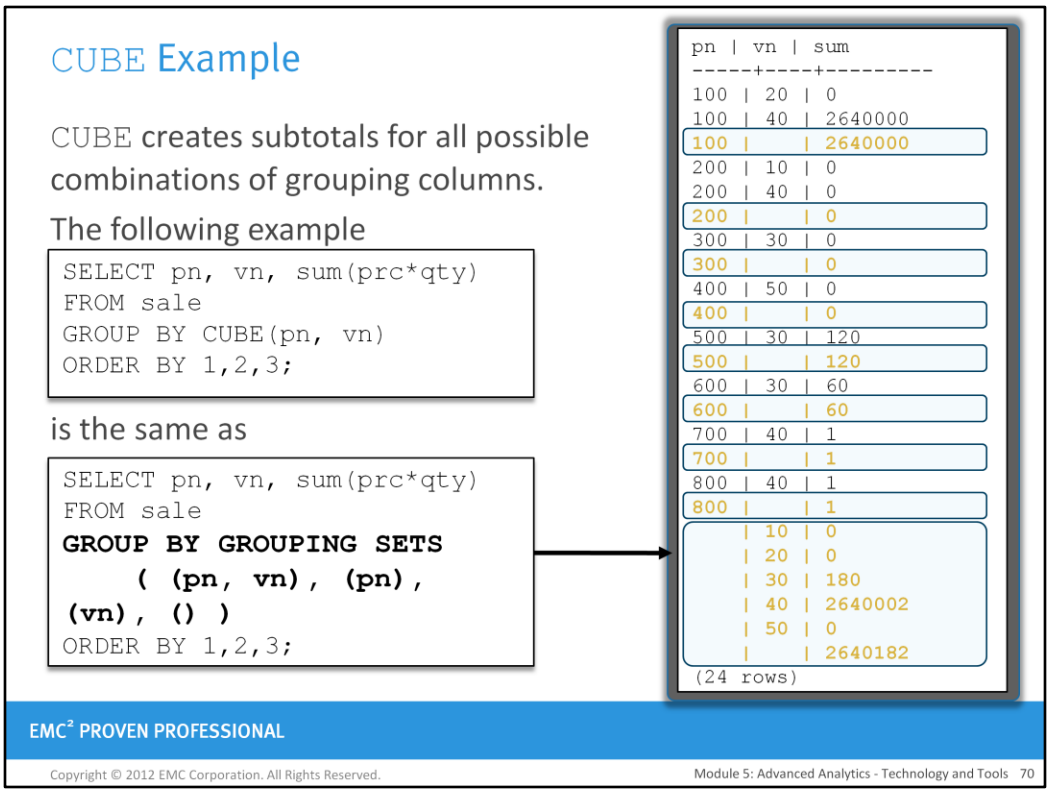
This slide meets the requirements provided in the previous slide, but uses the `ROLLUP` grouping extension. `ROLLUP` allows you to perform hierarchical grouping and helps to reduce the code.



GROUPING SETS Example

The GROUPING SETS extension allows you to specify grouping sets. If you use the GROUPING SETS clause to meet the earlier requirements so that it produced the same output as ROLLUP, it would use the following groups:

- (pn,vn) – This grouping summarizes product sales by vendor.
- (pn) – This grouping provides subtotal sales for each vendor.
- () – This grouping provides the grand total for all sales for all vendors.



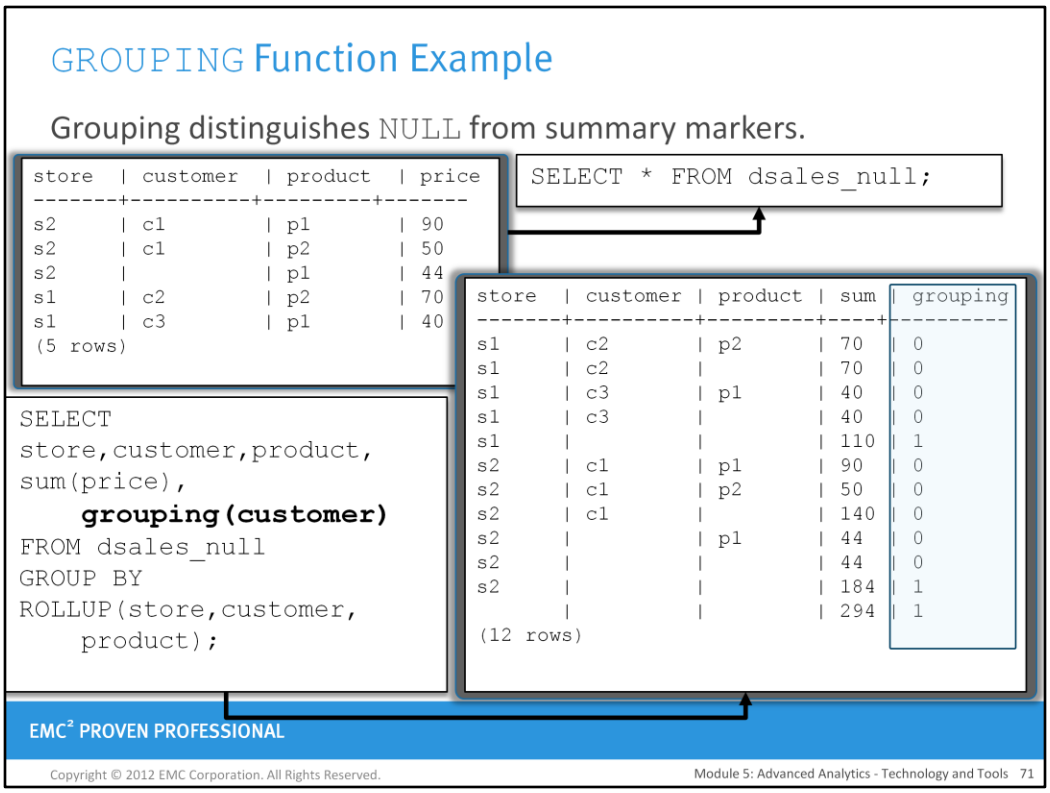
CUBE Example

A CUBE grouping creates subtotals for all of the possible combinations of the given list of grouping columns, or expressions.

In terms of multidimensional analysis, CUBE generates all the subtotals that could be calculated for a data cube with the specified dimensions.

In the example shown on the slide, the additional grouping set of `(vn)` - subtotaling the sales by vendor, is included as part of the cube.

Note that *n* elements of a CUBE translate to 2*n* grouping sets. Consider **using CUBE in any situation requiring cross-tabular reports**. **CUBE is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension**. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product.



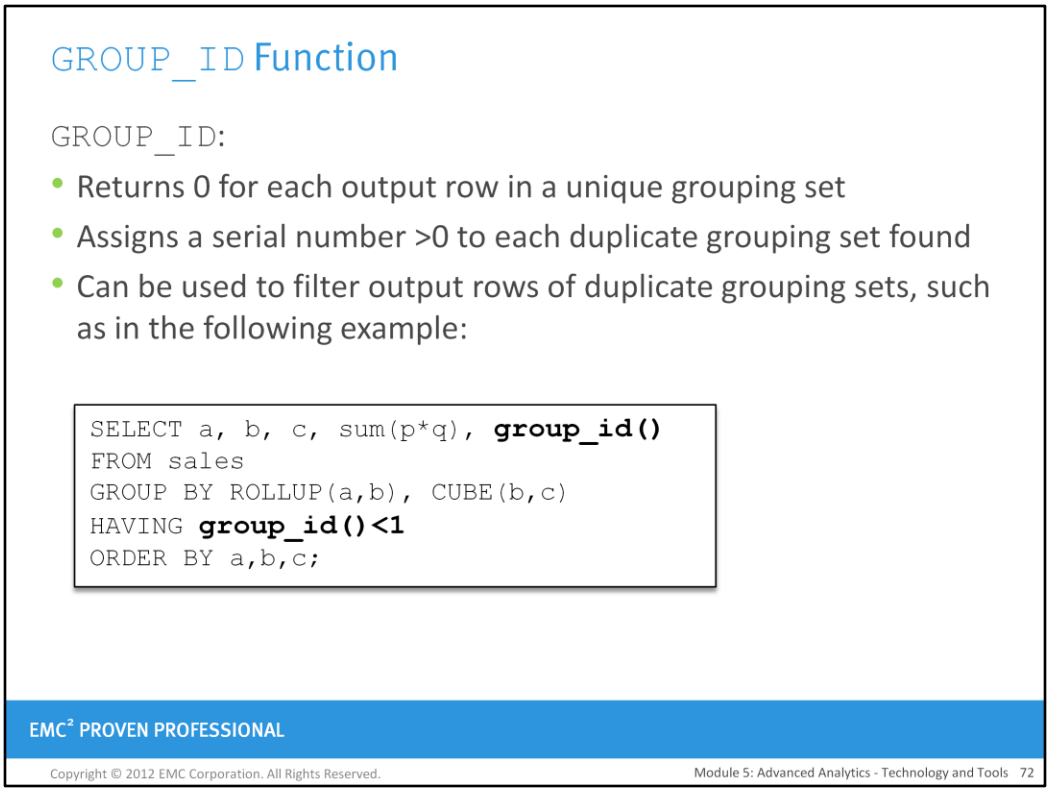
GROUPING Function Example

When you use grouping extensions to calculate summary rows, such as sub-totals and grand totals, the output can become confusing if the data in a grouping column contains NULL values. It is hard to tell if a row is supposed to be a subtotal row or a regular row containing a NULL.

In the example shown on the slide, one of the rows shown where the customer field is NULL. Without the grouping id, you could misinterpret the sum of 44 as a subtotal row for store 2.

The GROUPING function returns a result for each output row, where:

- 1 represents a summary row
- 0 represents grouped rows



`GROUP_ID` Function Is useful when combining grouping extension clauses.

In this example query, the combination of `ROLLUP` and `CUBE` produces:

- 11 grouping sets
- 7 `DISTINCT` grouping sets

The `group_id` function can be used to filter out or identify duplicate grouping sets in the output.

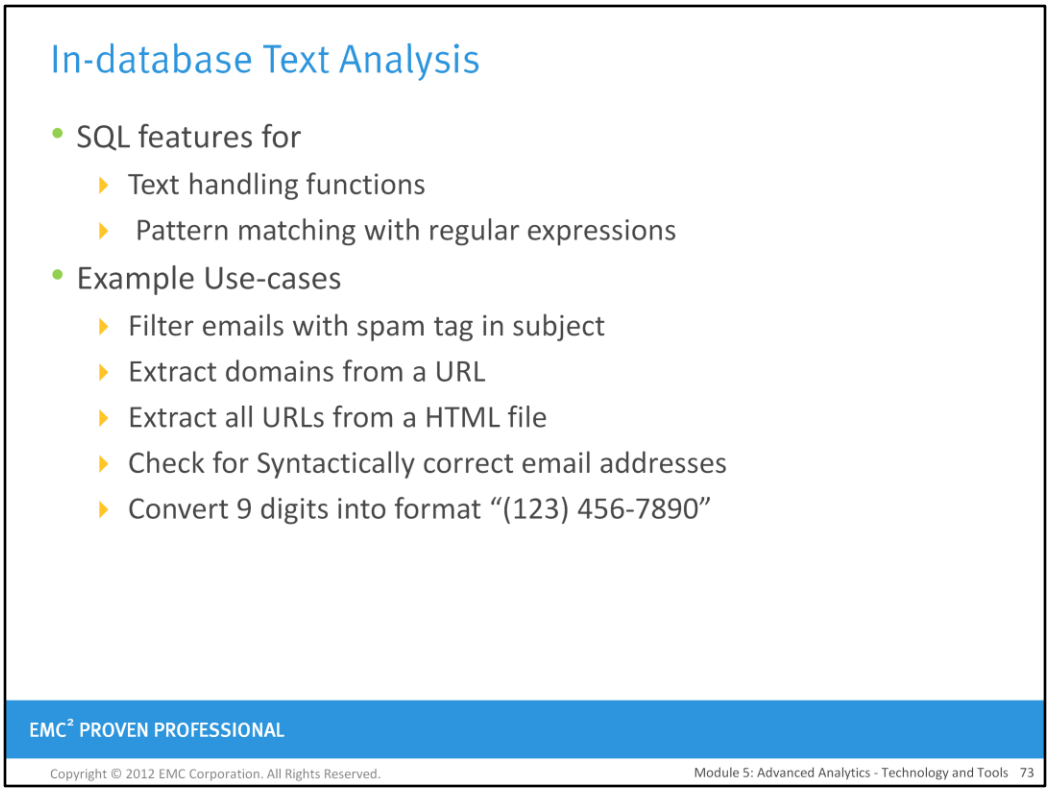
`GROUP BY ROLLUP (a,b), CUBE (b,c)` is the same as

- `GROUP BY GROUPING SETS ( (a,b), (a), () ), GROUPING SETS ( (b,c), (b), (c), () )`
- `GROUP BY GROUPING SETS ((a,b,b,c), (a,b,b), (a,b,c), (a,b), (a,b,c), (a,b), (a,c), (a), (b,c),(b),())`

Where there are 11 total grouping sets but only 7 distinct grouping sets, where the groups are:

- `(a,b,b,c) = (a,b,c) = (a,b,c)`
- `(a,b,b) = (a,b) = (a,b)`
- `(a,c)`
- `(b,c)`
- `(a)`
- `(b)`

- ( )



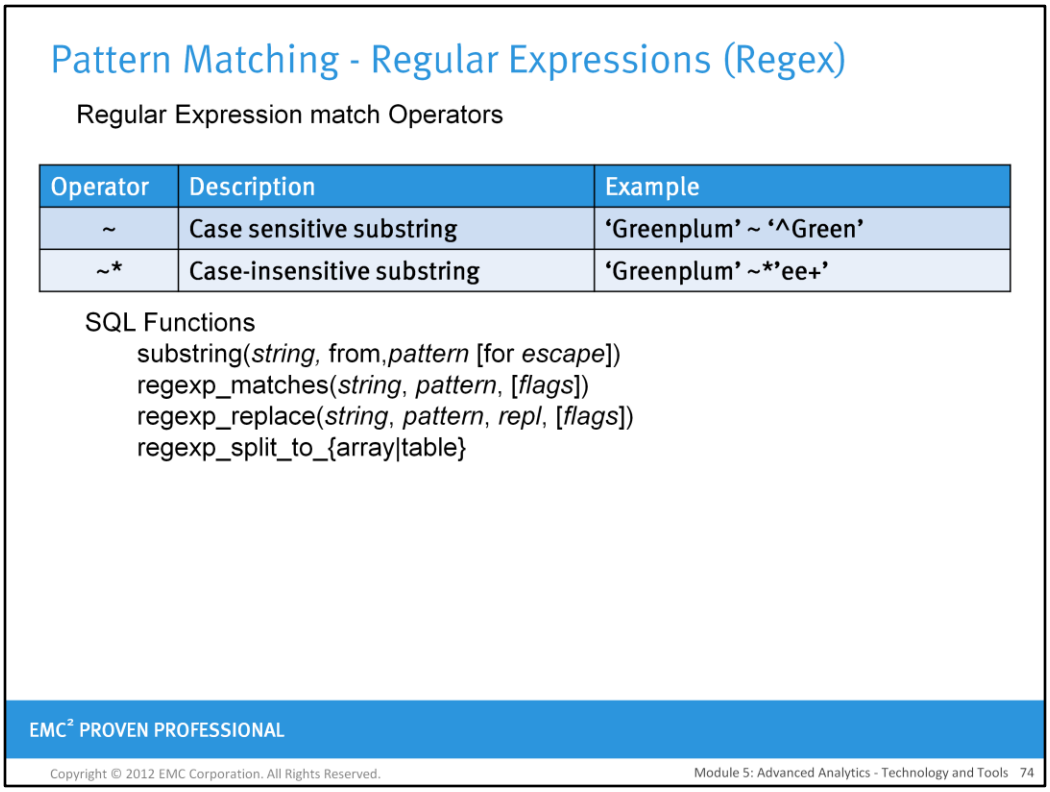
In Lesson 8 of Module 4 we described the techniques for text analysis. It is common practice to store the parsed data from an unstructured source in a database for down stream analysis. With the advent of Hadoop and its ecosystem products, unstructured data is also stored in external tables and accessed by traditional relational databases. We present a brief introduction to text processing in-database.

The topics covered in the next few slides are the in-database methods for handling text and the use of expressions in SQL.

We also briefly discussed Regular expressions in Module 4 lesson 8. A regular expression is a character sequence that is an abbreviated definition of a set of strings (a *regular set*). A string is said to match a regular expression if it is a member of the regular set described by the regular expression.

Regular expressions provide the means for matching strings of text and building the functionalities for string manipulation in SQL. The theoretical basis for regular expressions can be explained with Finite State Machines (out of scope for this course). It should be noted that regular expression cannot match subscripts and superscripts or well formed nested parentheses.

We will illustrate some of the SQL syntax with regular expressions.



The "substring" function is primarily used for string pattern matching.

The operator ~ is specified for case sensitive match of the substring and ~* for case insensitive match.

In the first example we are trying to find records with substring "Green" (case sensitive) starting at the beginning specified with character ^

In the second example we are finding match for a pattern "ee" (case in-sensitive) as a preceding term one or more times (Specified with a +)

Refer to http://www.postgresql.org/docs/8.3/static/functions-matching.html for details of the syntax.

Regular Expression Quantifiers

- Quantifier

Expression	Matches
.	Arbitrary character
^ And $	Virtual characters for beginning and end
*	Preceding item zero or more times
+	Preceding item one or more times
?	Preceding item is optional
{*n*}	Preceding item *n* times
a|b	Item *a* or *b*
...	...

EMC² PROVEN PROFESSIONAL

Quantifiers specify how often the preceding Regular Expression should match.

• Try to match the preceding regular expression zero or more times.

- Example: "(ab)c*" matches "ab" followed by zero or more "c"s, i.e., "ab", "abc", "abcc", "abccc" ...

+ Try to match the preceding regular expression one or more times.

- Example: "(ab)c+" matches "ab" followed by one or more "c"s, i.e., "abc", "abcc", "abccc" ...

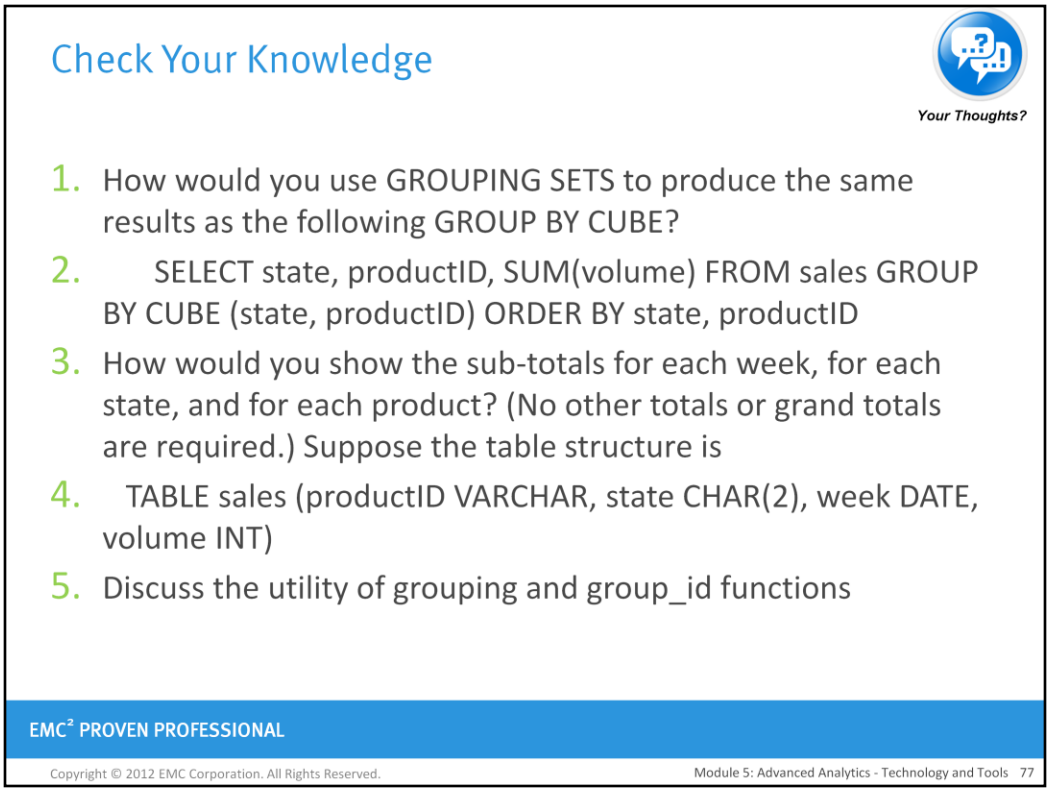
We will look into some examples in the following slide.

Examples

- All mail with at least two + in x_spam_level:
  - SELECT * FROM mail
    WHERE x_spam_level ~ '\\+\\+*'
- All top-level domains of sender's addresses:
  - SELECT substring("from" FROM
    '\\.[[:alnum:]]+$') FROM mail
- Remove [Spam] tag at beginning of subjects:
  - SELECT regex_replace(subject,
    '^((?:Re:[[:space:]]*)*)\\[Spam\\]' || '(.*)', '\\1\\2') FROM mail

EMC² PROVEN PROFESSIONAL

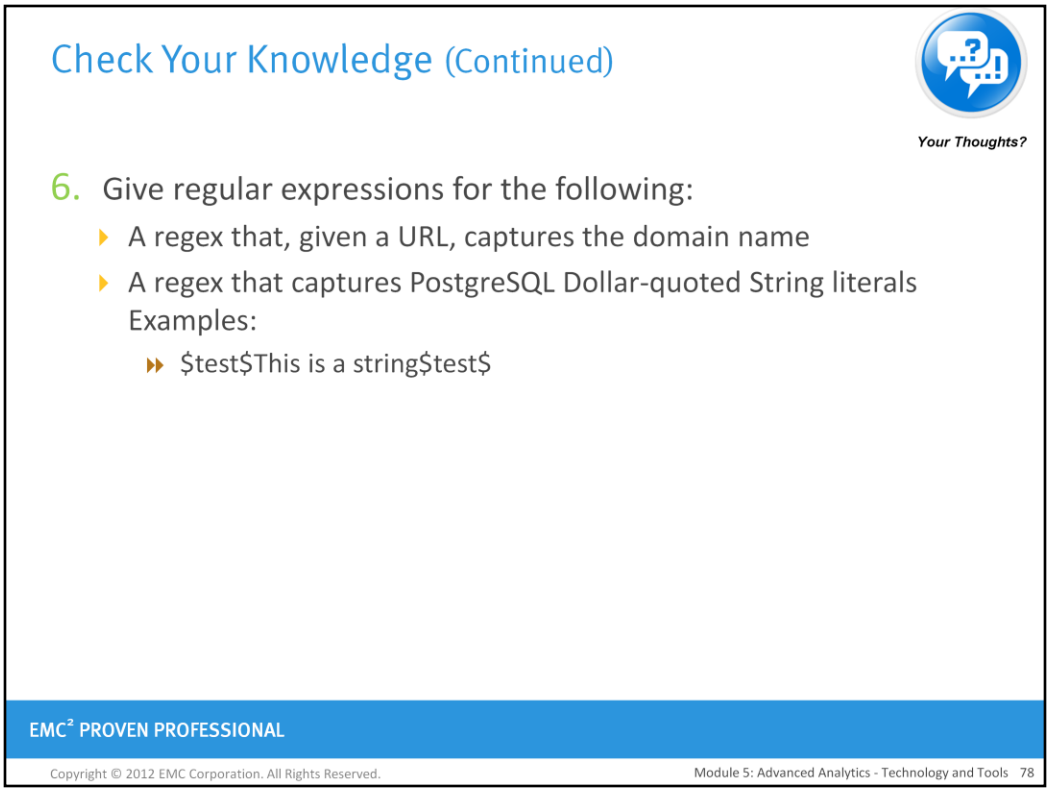
Shown in this slide are some examples of regular expressions used in SQL.



Note your answer/references below:

1. SELECT state, productID, SUM(volume) FROM sales GROUP BY GROUPING SETS ((state, productID), (state), (productID), ()) ORDER BY state, productID

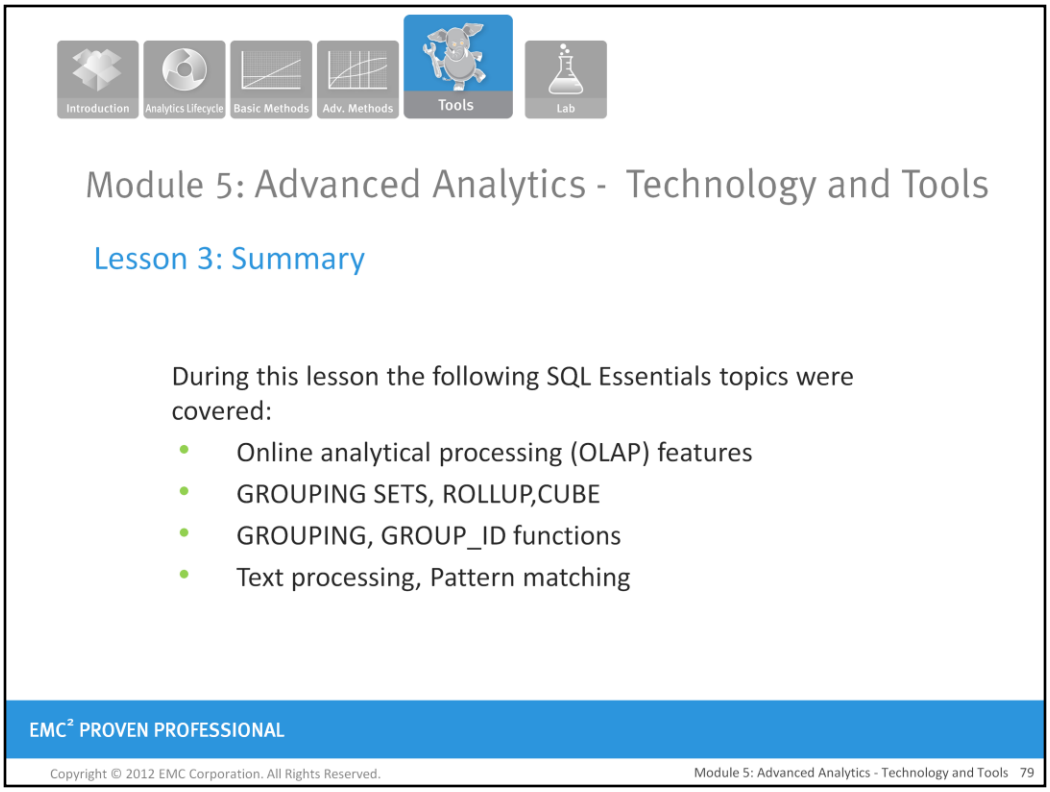
2. SELECT state, productID, week, SUM(volume) FROM sales GROUP BY GROUPING SETS ((state), (productID), (week)) ORDER BY state, productID, week



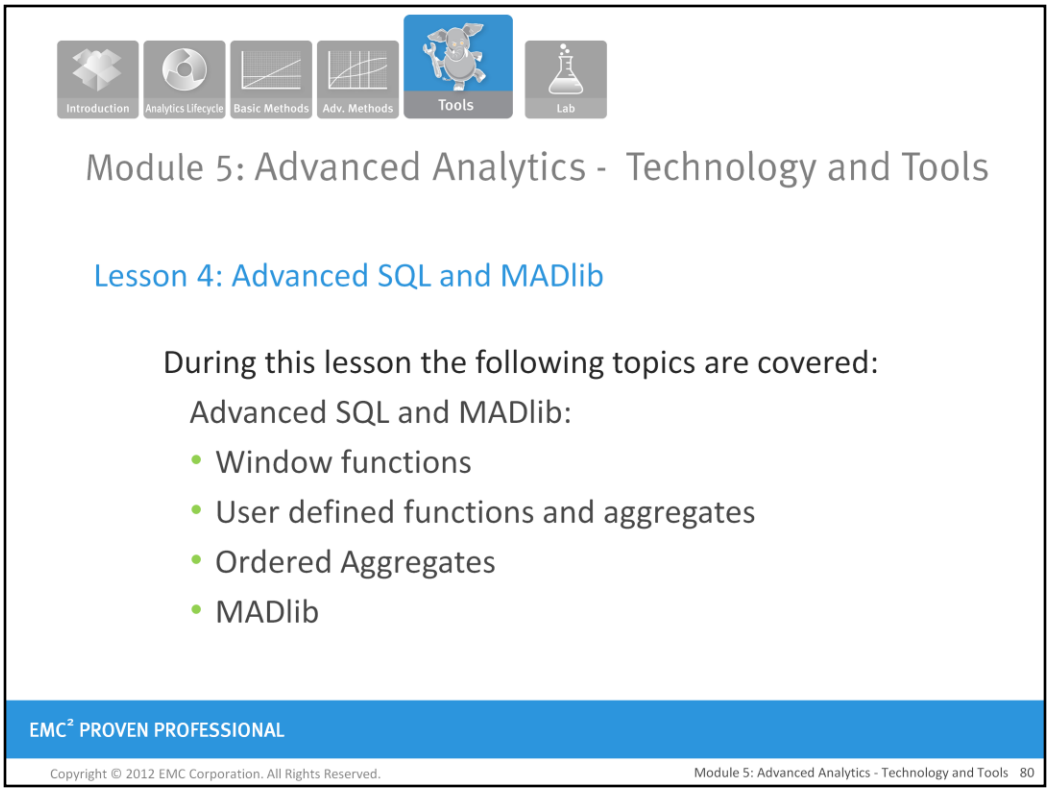
Note your answer/references below:

SELECT substring(c9 from '(?:(?:(?:[^/]|/[^/])*//)|(?=^([^/]|/[^/]|/$)*$))([^/]*)') AS domain

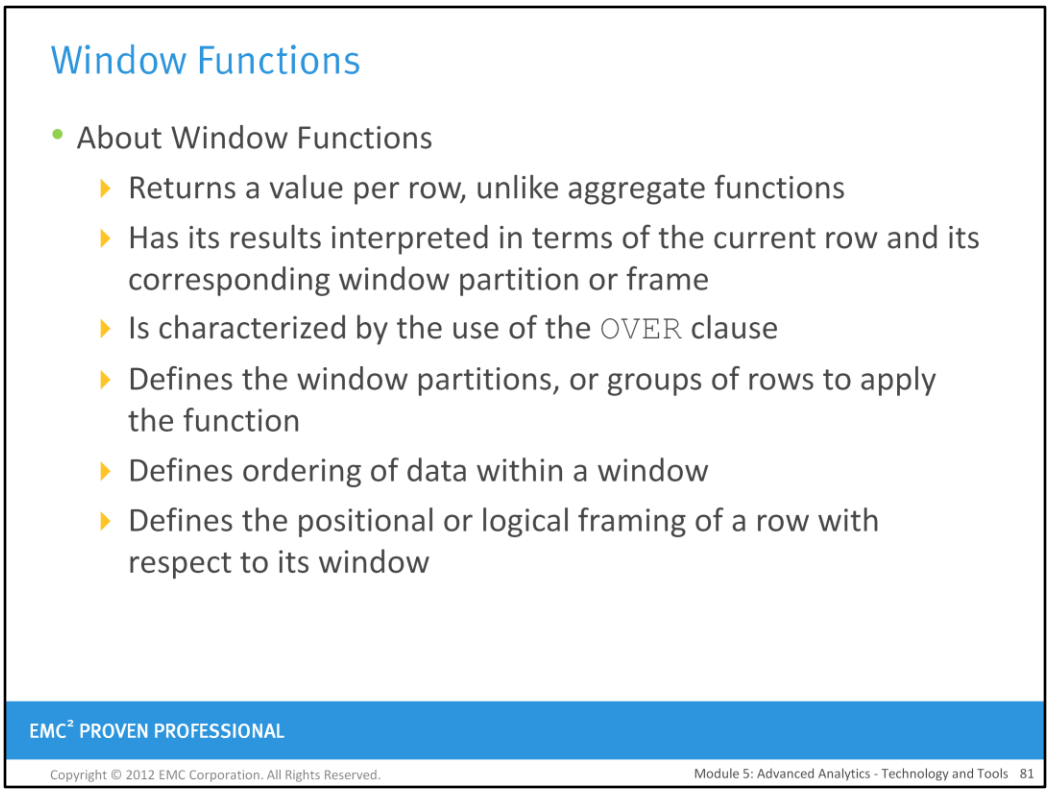
select regexp_matches('$test$This is a string$test$', E'(\\$[a-z]*\\$).*?\\1');



SQL essentials were covered in this lesson.



This lesson covers advanced SQL and MADlib functions



A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

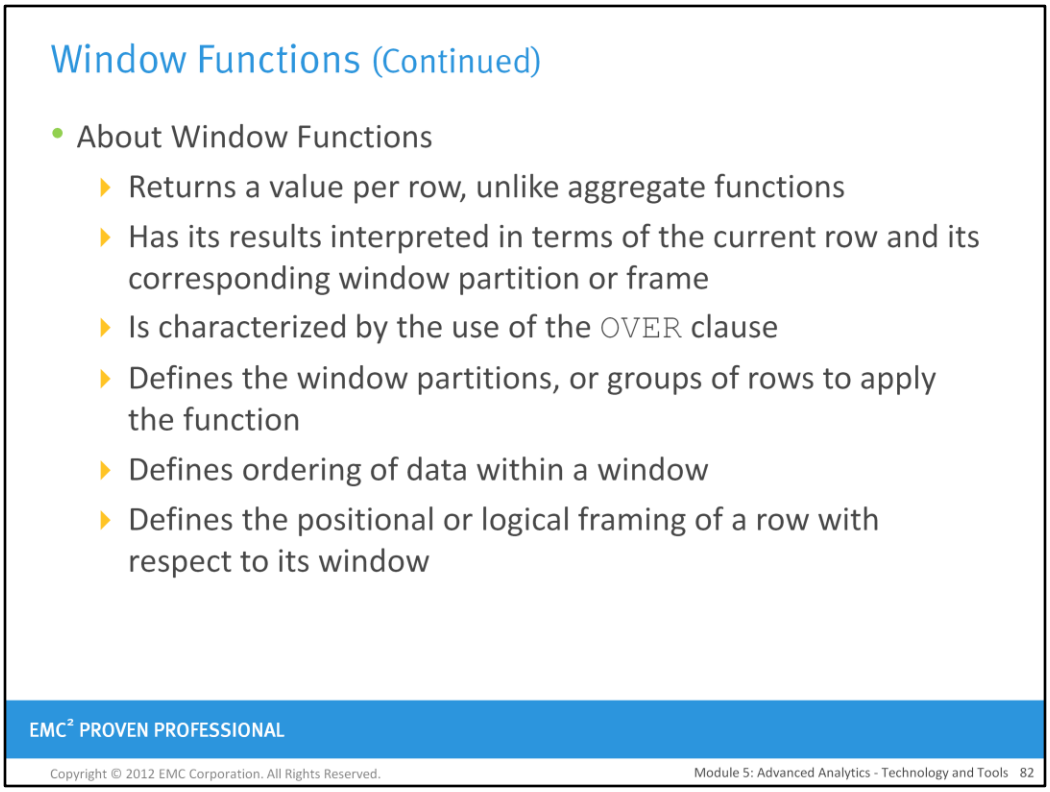
Window functions are a new class of functions introduced in Greenplum.

Window functions allow application developers to more easily compose complex OLAP queries using standard SQL commands. For example:

- Moving averages or sums can be calculated over various intervals.
- Aggregations and ranks can be reset as selected column values change.
- Complex ratios can be expressed in simple terms.

Window functions can only be used in the SELECT list, between the SELECT and FROM keywords of a query.

<Continued>

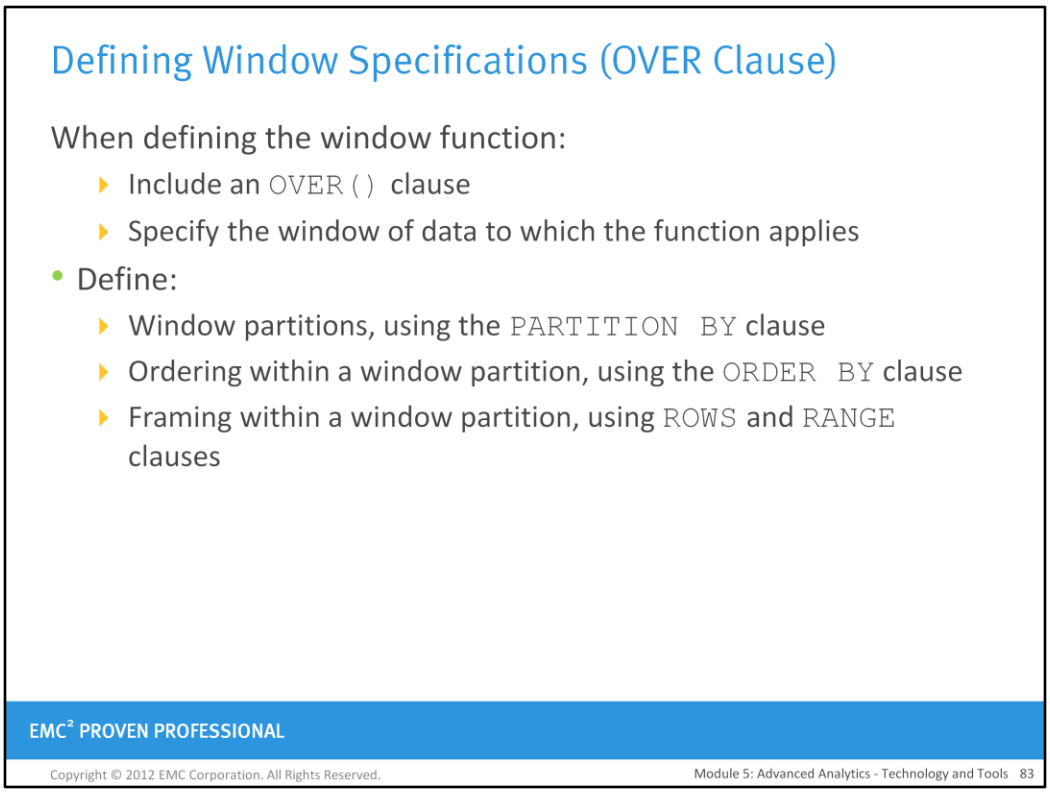


Unlike aggregate functions, which return a result value for each group of rows, window functions return a result value for every row, but that value is calculated with respect to the rows in a particular window partition (grouping) or window frame (row position within the window).

What classifies a function as a window function is the use of an `OVER` clause. The `OVER` clause defines the window of data to which the function will be applied.

There are three characteristics of a window specification:

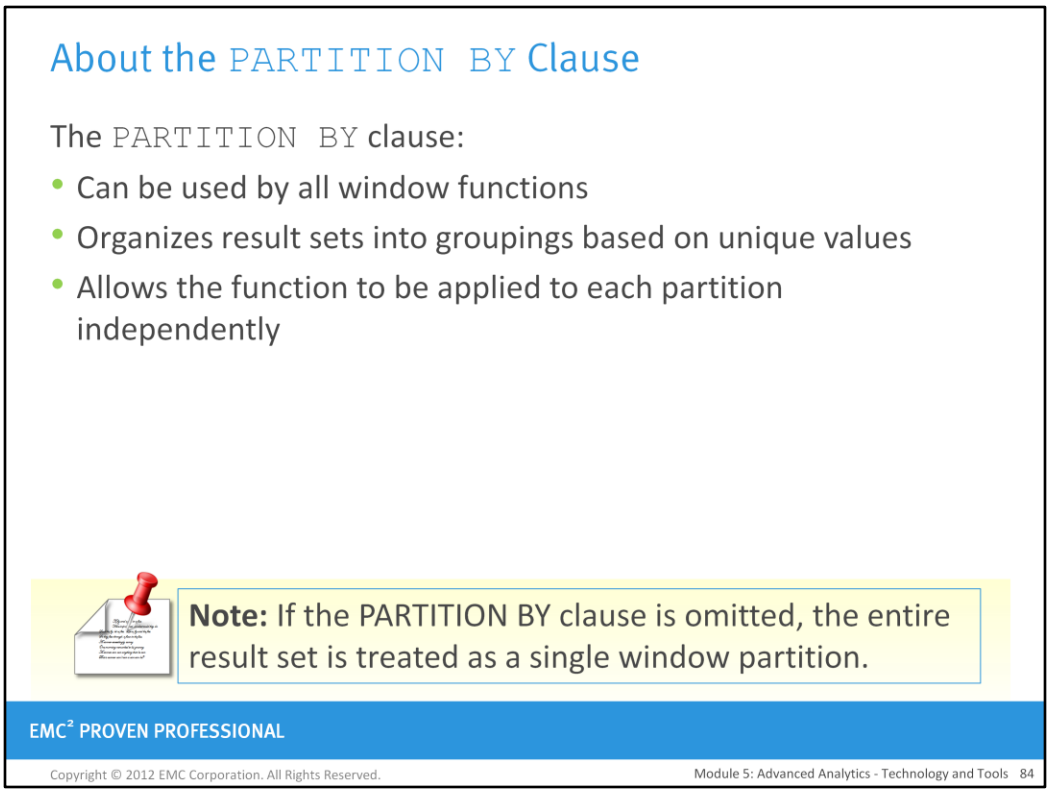
- **Partitions (groupings)** – a window function calculates the results for a row with respect to its partition.
- **Ordering of rows within a window partition** – some window function such as RANK require ordering.
- **Framing** – for ordered result sets, you can define a window frame that analyzes each row with respect to the rows directly above or below it.



All window functions must have an `OVER()` clause. The window function specifies the window of data to which the function applies

It defines:

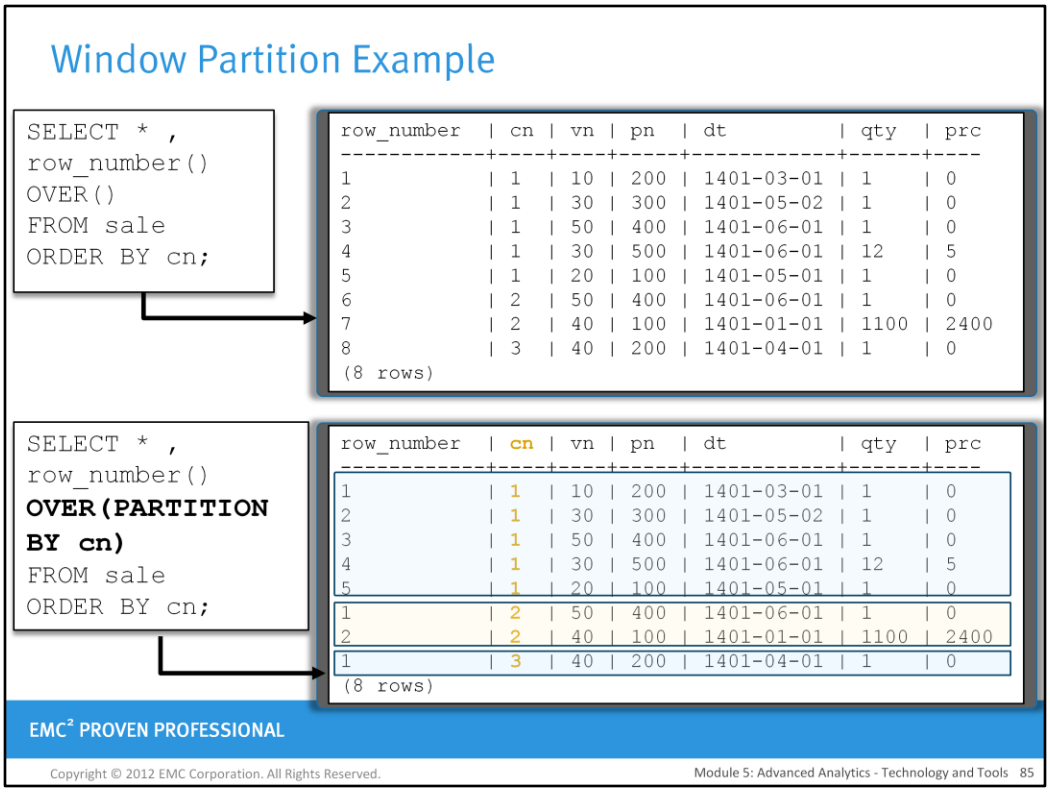
- Window partitions using the `PARTITION BY` clause.
- Ordering within a window partition using the `ORDER BY` clause.
- Framing within a window partition (`ROWS/RANGE` clauses).



About the `PARTITION BY` Clause

The `PARTITION BY` clause:

- Can be used by all window functions. However, it is not a required clause. Windows that do not use the `PARTITION BY` clause present the entire result set as a single window partition.
- Organizes the result set into groupings based on the unique values of the specified expression or column.
- Allows the function to be applied to each partition independently.

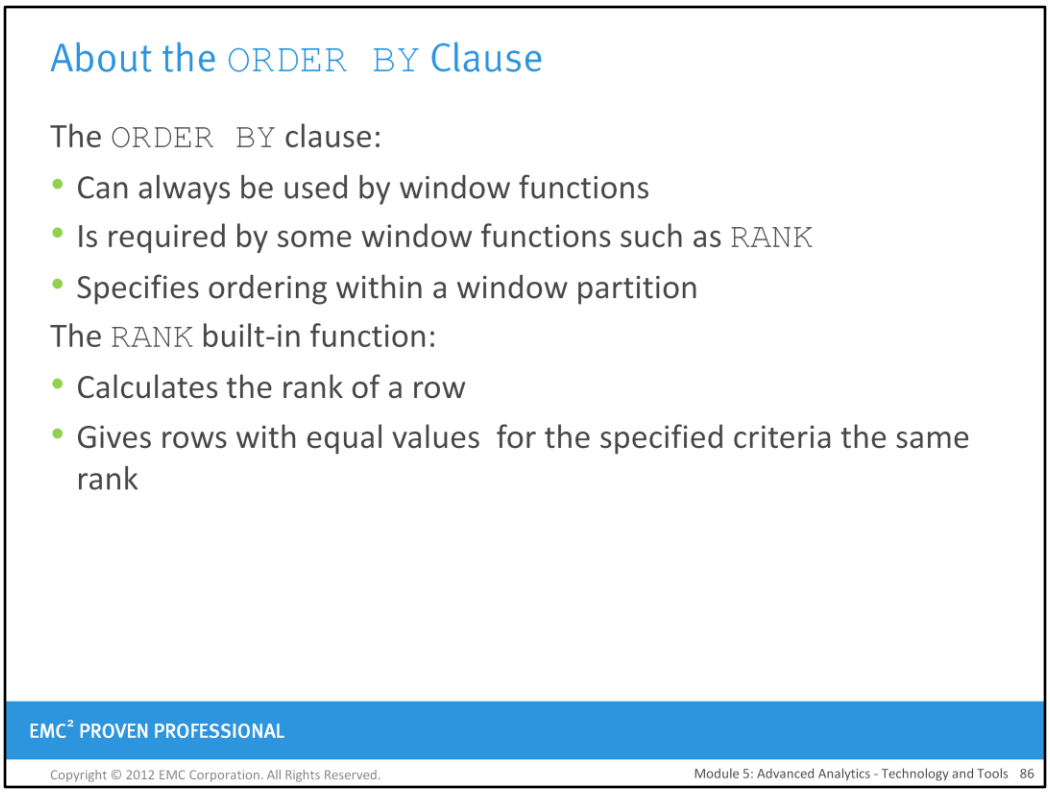


Window Partition Example

The example on the slide uses the `row_number` window function. This function returns a row number for each unique row in the result set.

In the first example, the `OVER` clause does not have a `PARTITION BY`. The entire result set is treated as one window partition.

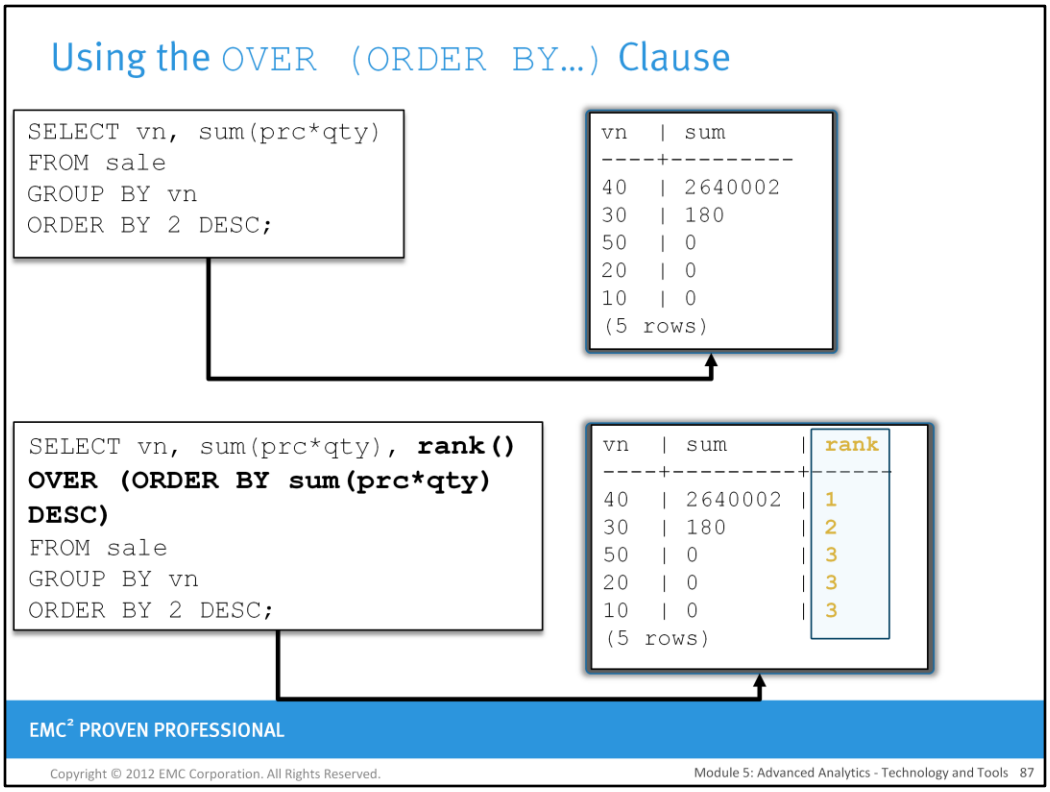
In the second example, the window is partitioned by the customer number. Note that the result of row number is calculated within each window partition.



About the `ORDER BY` Clause

The `ORDER BY` clause is used to order the resulting data set based on an expression or column. It is always allowed in windows functions and is required by some window functions, including `RANK`. The `ORDER BY` clause specifies ordering within a window partition.

The RANK function is a built-in function that calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. In this case, ranks may not be consecutive numbers.

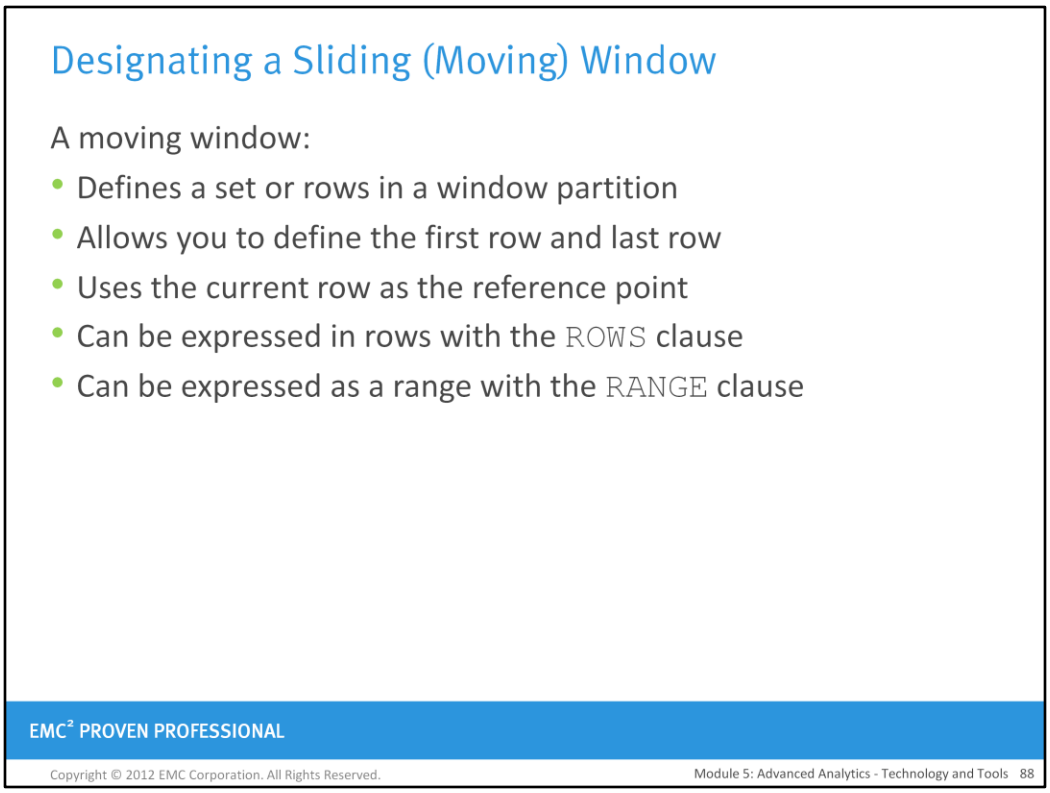


Using the OVER (ORDER BY…) Clause

The slide shows an example of two queries that rank vendors by sales totals.

The first query shows a window function grouped on the vendor column, `vn`.

The second query uses the `RANK` function to output a ranking number for each row. Note that the `PARTITION BY` clause is not used in this query. The entire result is one window partition. Also, do not confuse `ORDER BY` of a window specification with the `ORDER BY` of a query.



A moving or rolling window defines a set of rows within a window partition. When you define a window frame, the window function is computed with respect to the contents of this moving frame, rather than against the fixed content of the entire window partition. Window frames can be row-based, represented by the `ROWS` clause, or value based, represented by a `RANGE`.

When the window frame is row-based, you define the number of rows offset from the current row. If the window frame is range-based, you define the bounds of the window frame in terms of data values offset from the value in the current row.

If you specify only a starting row for the window, the current row is used as the last row in the window.

About Moving Windows (Continued)

A moving window:

- Is defined as part of a window with the ORDER BY clause as follows:

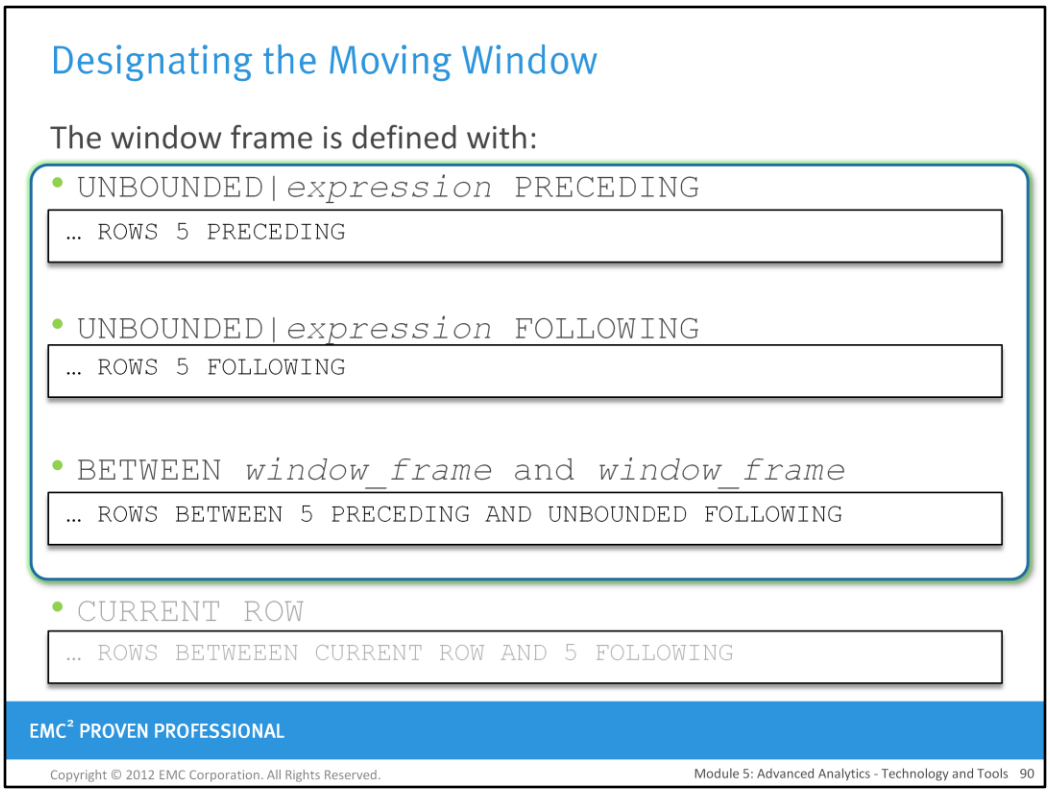
```
WINDOW window_name AS (window_specification)
```

where window_specification can be:

```
[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]
[{RANGE | ROWS}
{ UNBOUNDED PRECEDING
| expression PRECEDING
| CURRENT ROW
| BETWEEN window_frame_bound AND window_frame_bound }]]
```

EMC² PROVEN PROFESSIONAL

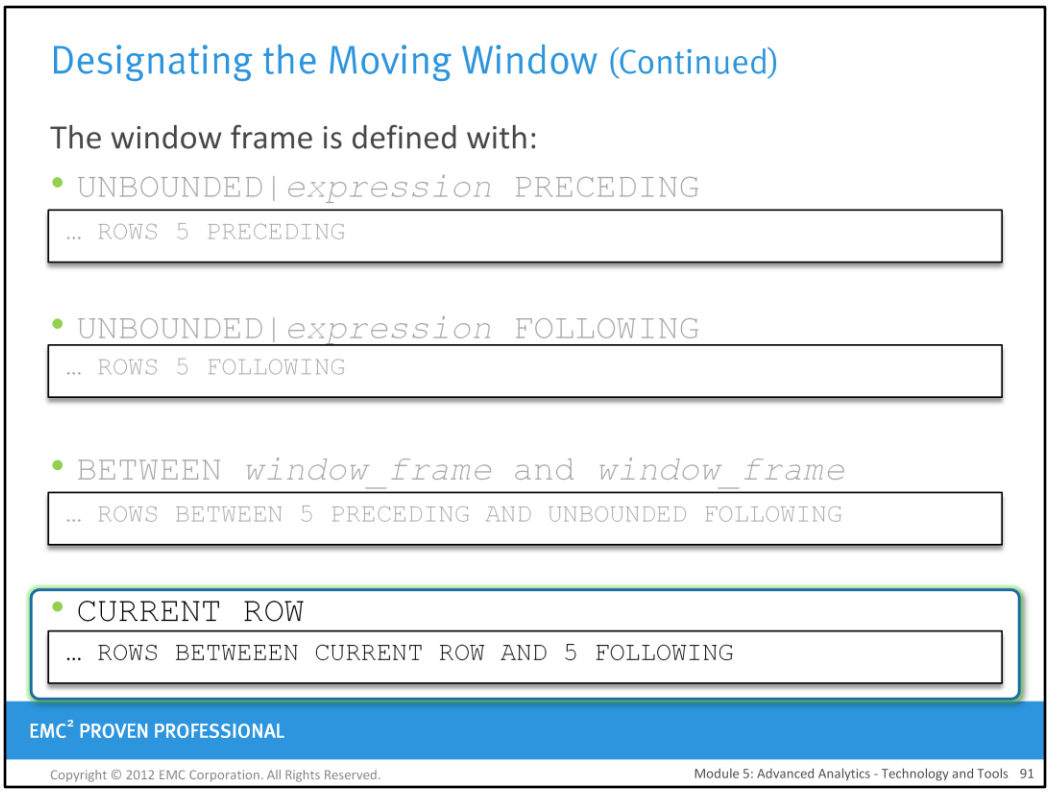
Syntax of a moving window is specified here.



Designating the Moving Window

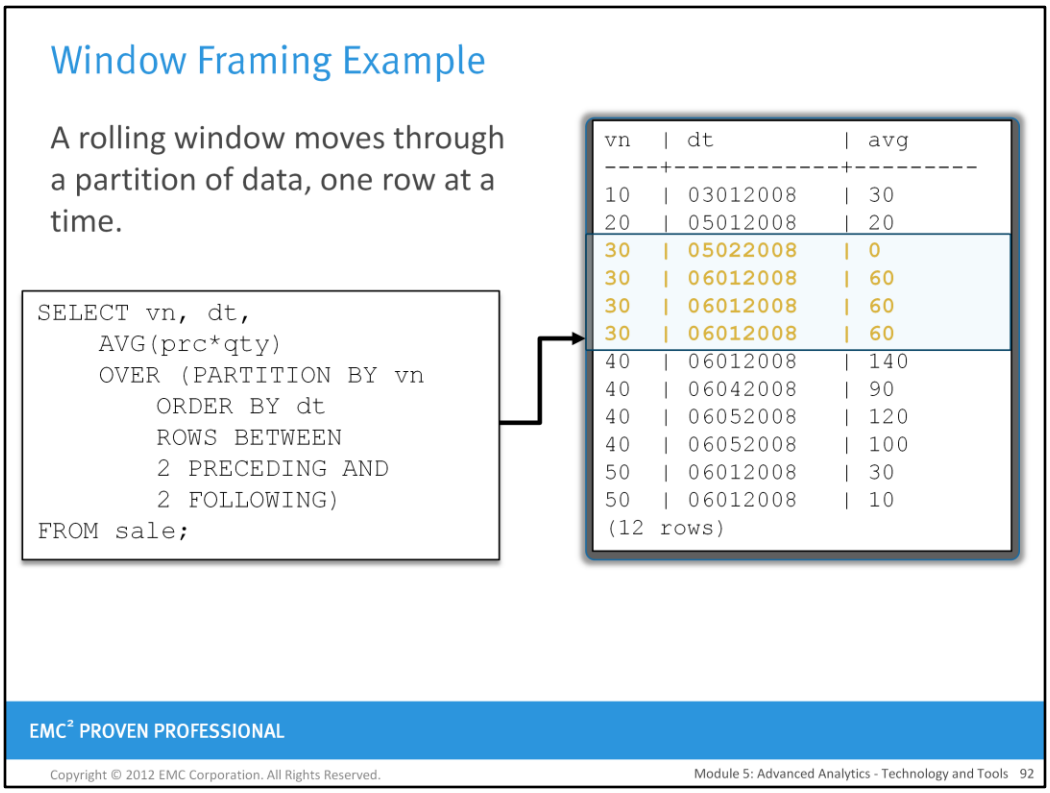
The window frame can be defined as:

- **UNBOUNDED or *expression* PRECEDING** – This clause defines the first row of the window using the current row as a reference point. The starting row is expressed in terms of the number of rows preceding the current row. If you define a ROWS window frame as 5 PRECEDING, the window frame starts at the fifth row preceding the current row. If the definition is for a RANGE window frame, the window starts with the first row whose ordering column value precedes that of the current row by 5. If the term `UNBOUNDED` is used, the first row of the partition acts as the first row of the window.
- **UNBOUNDED or *expression* FOLLOWING** – This clause defines the last rows of the window using the current row as a reference point. Similar to `PRECEDING`, the last row is expressed in terms of the number or rows following the current row. Either an expression or the term `UNBOUNDED` can be used to identify the last rows. If `UNBOUNDED` is used, the last row in the window is the last row in the partition.
- **BETWEEN *window_frame_bound* AND *window_frame_bound*** – This clause defines the first and last rows of the window, using the current row as a reference point. The first and last rows are expressed in terms of the number of rows preceding and following the current row, respectively. You can use other window frame syntax to define the bound. For example, `BETWEEN 5 PRECEDING AND 5 FOLLOWING` defines a window frame where the previous 5 rows and the next 5 rows from the current row are included in the moving window.



Designating the Moving Window (Continued)

- `CURRENT ROW` – This clause references the current row in the partition. If a window frame is defined as `BETWEEN CURRENT ROW AND 5 FOLLOWING`, the window is defined starting with the current row and ending with the next 5 rows.



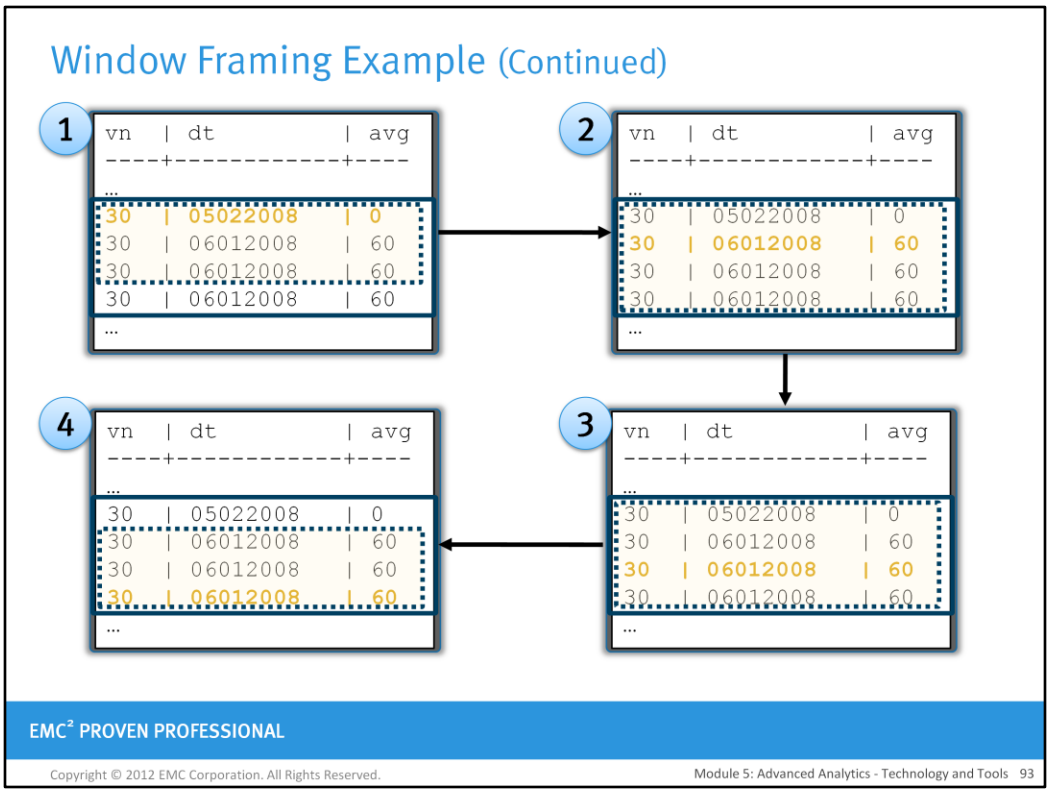
Window Framing Example

While window framing clauses require an `ORDER BY` clause, not all window functions allow framing.

The `ROWS` and `RANGE` clauses specify a positional or logical rolling window that moves through a window partition of data.

In the example shown on the slide, the rolling frame applies to its partition, in this case, `vendor`, and ordering within that partition, `date`.

The example shows positional framing using the `ROWS BETWEEN` clause where the result is interpreted with respect to the `CURRENT ROW` position in the partition. The **focus of the window frame moves from row to row within its partition only.**

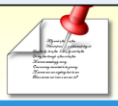


Window Framing Example (Continued)

The focus of the frame moves from the first selectable row in the window partition using the criteria, 2 preceding and 2 following, for the rolling window.

Built-In Window Functions

Built-In Function	Description
`dist()`	Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value.
`rank()`	Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value.
`first_value(`*`expr`*`)`	Returns the first value in an ordered set of values.
`lag(expr [,`*`offset`*`] [,default])`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position. If offset is not specified, the default offset is 1. default sets the value that is returned if the offset goes beyond the scope of the window. If default is not specified, the default value is null.



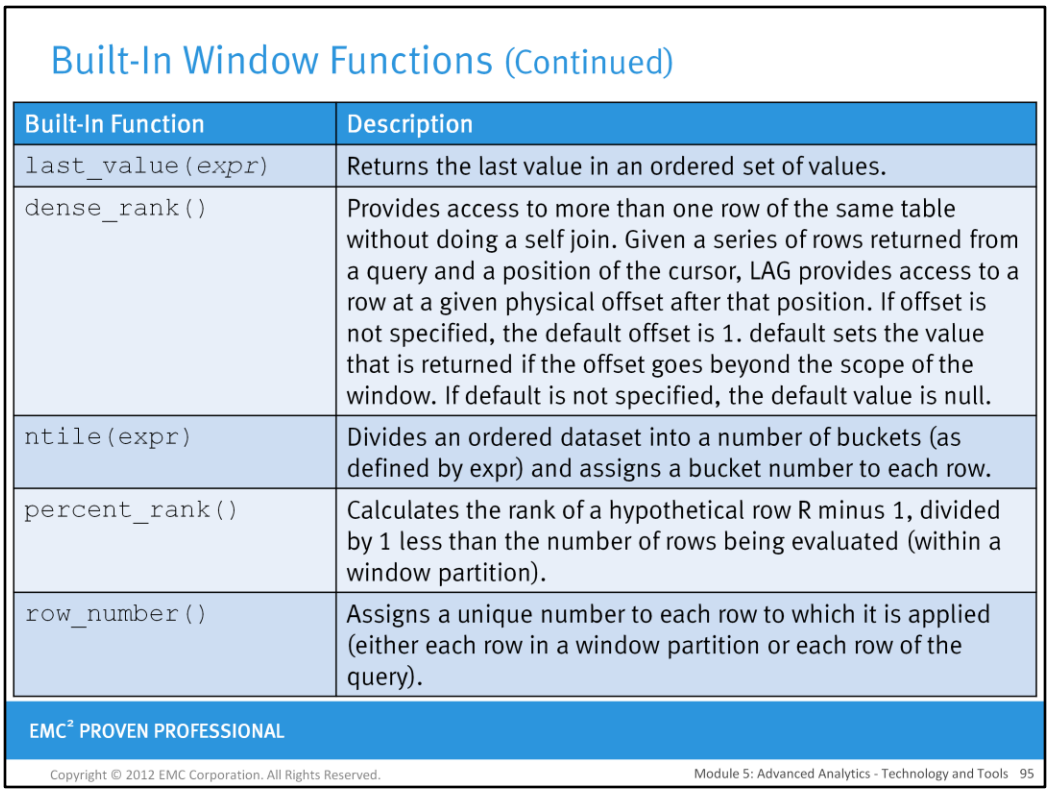
Note: Any aggregate function used with the `OVER` clause can also be used as a window function.

EMC² PROVEN PROFESSIONAL

Built-In Window Functions

The slide shows built-in window functions supported within Greenplum. These built-in functions require an `OVER` clause.

For more detailed information on the functions, refer to the *Greenplum Database Administrator Guide*.



Built-In Window Functions (Continued)

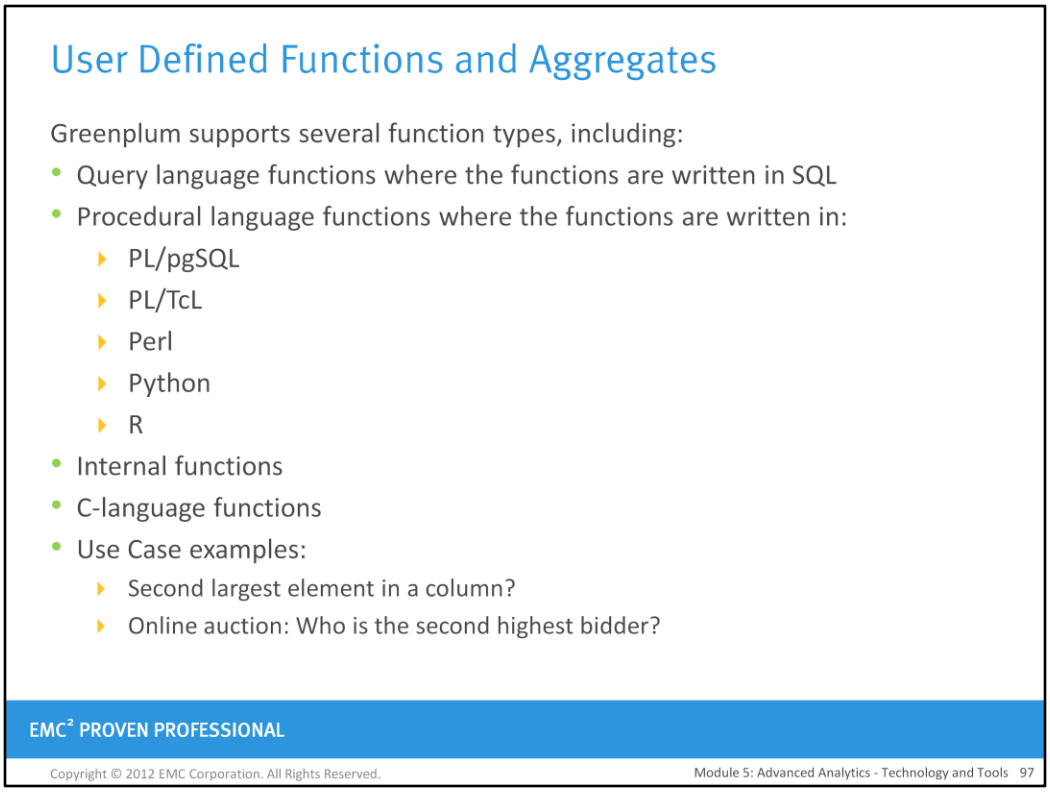
Check Your Knowledge

- Describe how this code will work:

```
SELECT dt, region, revenue,
  count(*) OVER (twdw) AS moving_count,
  avg(revenue) OVER (twdw) AS moving_average
FROM moving_average_data mad
WINDOW twdw AS (PARTITION BY region
   ORDER BY dt RANGE BETWEEN
   '7 days'::interval PRECEDING AND
   '0 days'::interval FOLLOWING)
ORDER BY region, dt
```

EMC² PROVEN PROFESSIONAL

Let us understand the code above. Describe below how this code will work:

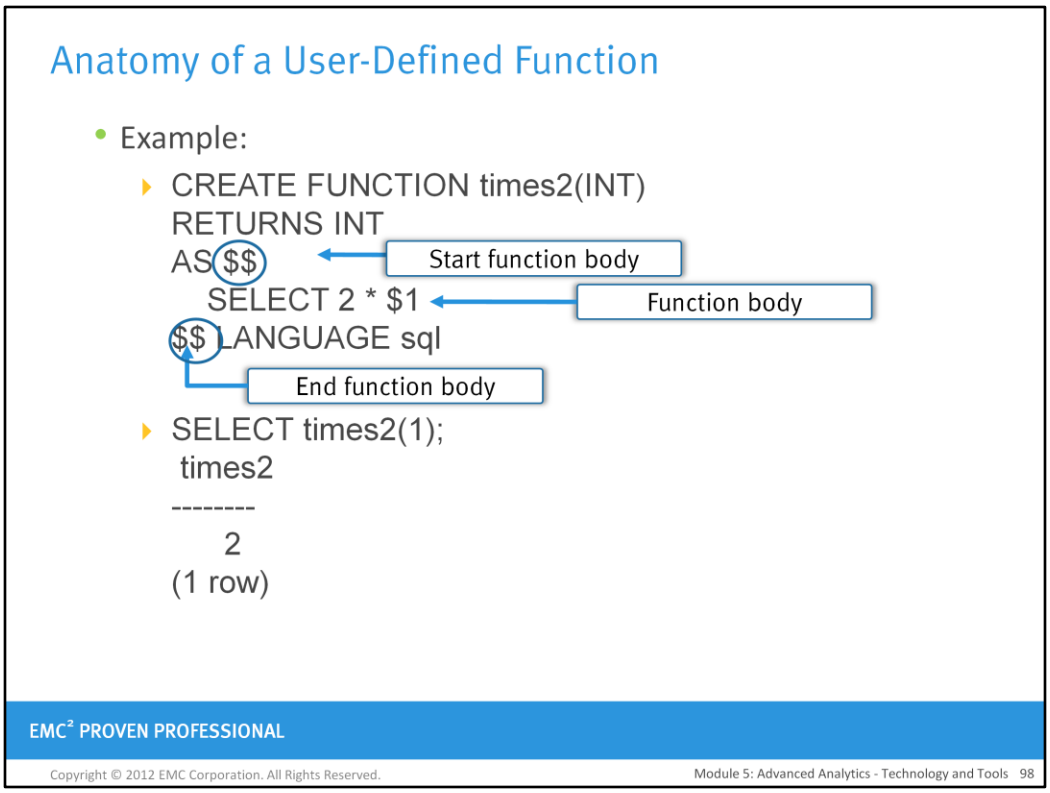


Greenplum supports a variety of methods for developing functions, including:

- Query language support for functions developed in SQL.
- Procedural language support for functions written in languages such as PL/PGSQL, which is a subset of PL/SQL, PL/TcL, Perl, Python, and R, a programming.
- Language for statistical computing and graphics.
- Internal functions.
- C-language functions.

The Data Scientist may need to create a function that could be used in the downstream analysis. Some use case examples are shown in the slide.

Note: Greenplum supports PL/pgSQL, PL/Perl, and PL/Python out of the box. Other languages can be added with the `createlang` utility.



Here we present a simple function that you can create.

Whenever you pass in a parameter, you can identify it as:

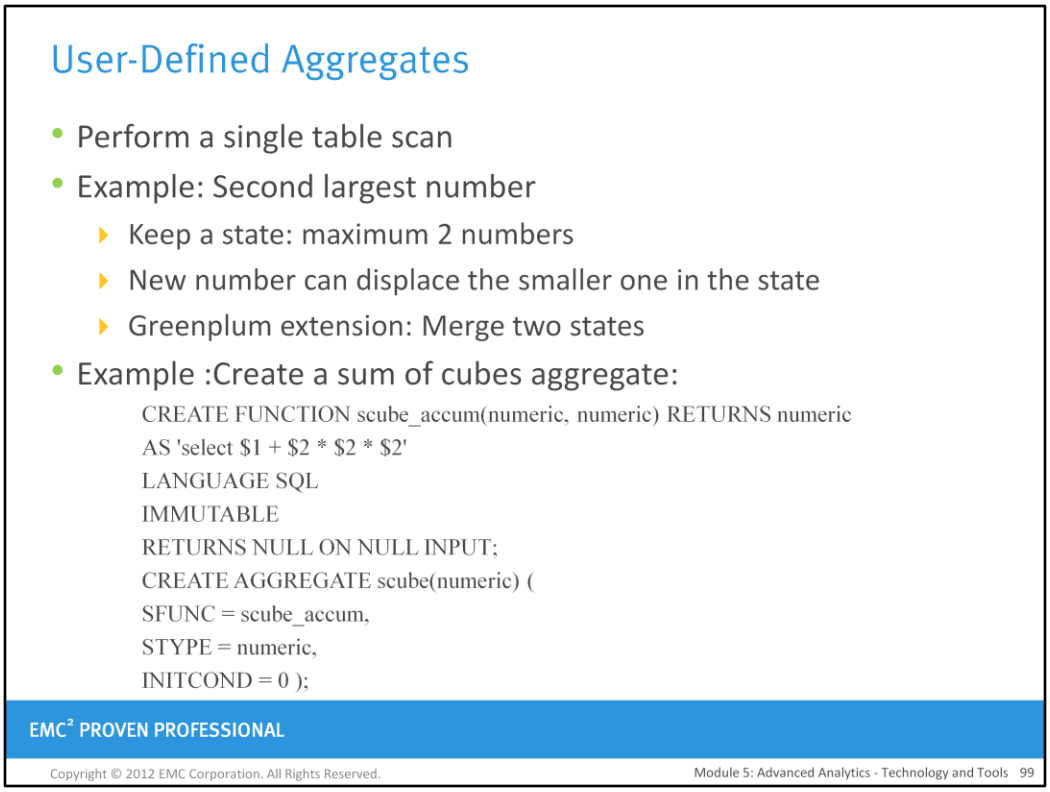
- A base or primitive type, such as `integer`, `char`, or `varchar`

In this example, when you call this function you pass the parameter `INT`. The value is multiplied by 2 and returned as numeric. $1 indicates the first parameter.

Creating, Modifying, and Dropping Functions

Functions that operate on tables must be created in the same schema. If you modify a table, you must have access to a schema. You:

- Create a function with the `CREATE FUNCTION` command. You must have `CREATE` access to the schema to create a function. A function can be created with or without parameters.
- Replace an existing function with the `CREATE OR REPLACE FUNCTION` command. This command either creates a function if one did not exist before, or replaces an existing function. If you are replacing an existing function, you must specify the same number of parameters and the same data types found in the original function. If not, you are actually creating a new function.
- Change a function with the `ALTER FUNCTION` command. You must own the function before you can modify it. If the function is to be created in another schema, you must have `CREATE` privilege on that schema.
- Drop or remove a function with the `DROP FUNCTION` command. Because you can have multiple functions with the same name but different number of parameters and/or parameter types, you must include the appropriate number of parameters and parameter types as part of the command. You must also be the owner of the function to remove the function from the schema.



User defined aggregates performs a single table scan and it keeps state. A state is a maximum of two numbers.

In the example we create a user defined aggregate that returns a maximum of two numbers. We will learn more about it in the Lab.

CREATE AGGREGATE defines a new aggregate function. Some basic and commonly-used aggregate functions such as count, min, max, sum, avg and so on are already provided in the Greenplum Database.

If one defines new types or needs an aggregate function not already provided, then CREATE AGGREGATE can be used to provide the desired features.

An aggregate function is made from one, two or three ordinary functions (all of which must be IMMUTABLE functions): a state transition function *sfunc, an optional preliminary segment-level calculation function prefunc, and an optional final calculation function ffunc. These are used as follows:*

sfunc(internal-state, next-data-values) ---> next-internal-state

prefunc(internal-state, internal-state) ---> next-internal-state

ffunc(internal-state) ---> aggregate-value

In the example shown above we only have the sfunc.

To test this aggregate you can try the following code:

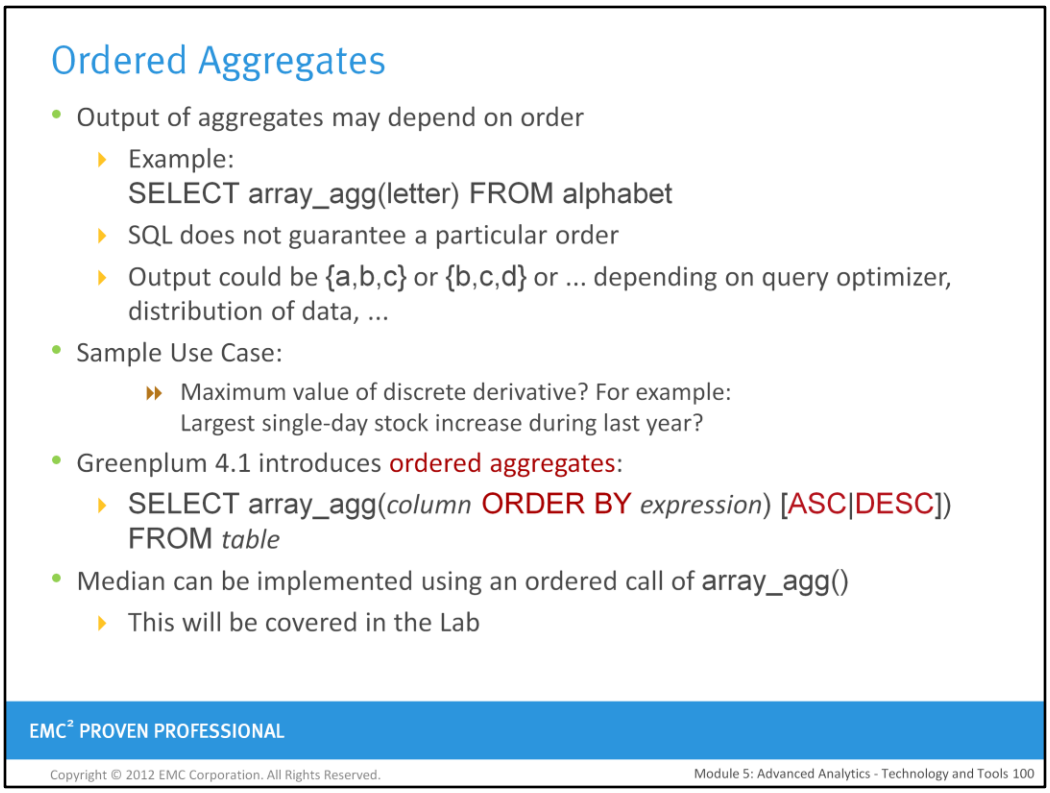
CREATE TABLE x(a INT);

INSERT INTO x VALUES (1),(2),(3);

SELECT scube(a) FROM x;

Correct answer for reference:

SELECT sum(a*a*a) FROM x;



Support has been added for ordered aggregate functions in Greenplum, providing a method for controlling the order in which values are fed to an aggregate function.

In a Greenplum Database, only aggregate functions defined as ORDERED can be called with an ORDER BY clause. This can be followed by other arguments to specify a system-defined ordering.

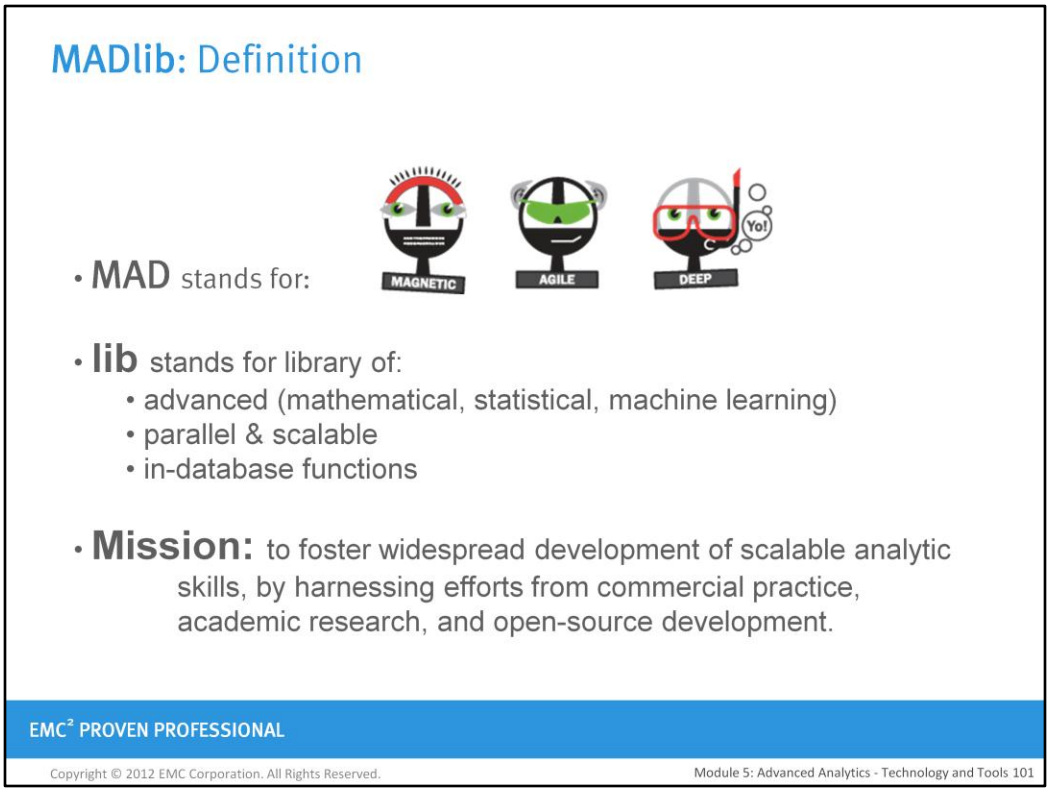
The three built-in ordered aggregates and optional ORDER BY clauses that have been implemented in 4.1, are shown in the following table :

Aggregate Function	**Description**
array_agg(any element)	Concatenates any element into an array. **Example:** SELECT array_agg(anyelement ORDER BY anyelement) FROM table;
string_agg(text)	Concatenates text into a string. **Example:** SELECT string_agg(text ORDER BY text) FROM table;
string_agg(text, delimiter)	Concatenates text into a string delimited by delimiter. **Example:** SELECT string_agg(text, ',' ORDER BY text) FROM table;

The columns in an ORDER BY clause are not necessarily the same as the aggregated column, as shown in the following statement that references a table named product with columns store_id, product_name, and quantity.

SELECT store_id, array_agg(product_name ORDER BY quantity desc) FROM product GROUP BY store_id;

Note: There can only be one aggregated column. Multiple columns can be specified in the ORDER BY clause.



MADlib was first reported at VLDB 2009 in which **MAD Skills**: New Analysis Practices for Big Data was presented

- *http://db.cs.berkeley.edu/papers/vldb09-madskills.pdf*

MADlib: Getting started

1) Visit http://MADlib.net
2) Download the latest release:
3) Click the **MADlib Wiki** link and…
4) Follow the installation guide for PostgreSQL or Greenplum

We have installed MADlib for you in the lab environment and you used MADlib functions in Module 4 labs.

Many functions are implemented as User defined functions and User defined aggregates and some examples are as follows:

```
testdb=# \da madlib.mreg*
                                List of aggregate functions
 Schema |    Name      |                Argument data types                 | Description
--------+--------------+----------------------------------------------------+-------------
 madlib | mregr_coef   | double precision, double precision[]               |
 madlib | mregr_pvalues | double precision, double precision[]              |
 madlib | mregr_r2     | double precision, double precision[]               |
 madlib | mregr_tstats | double precision, double precision[]               |
```

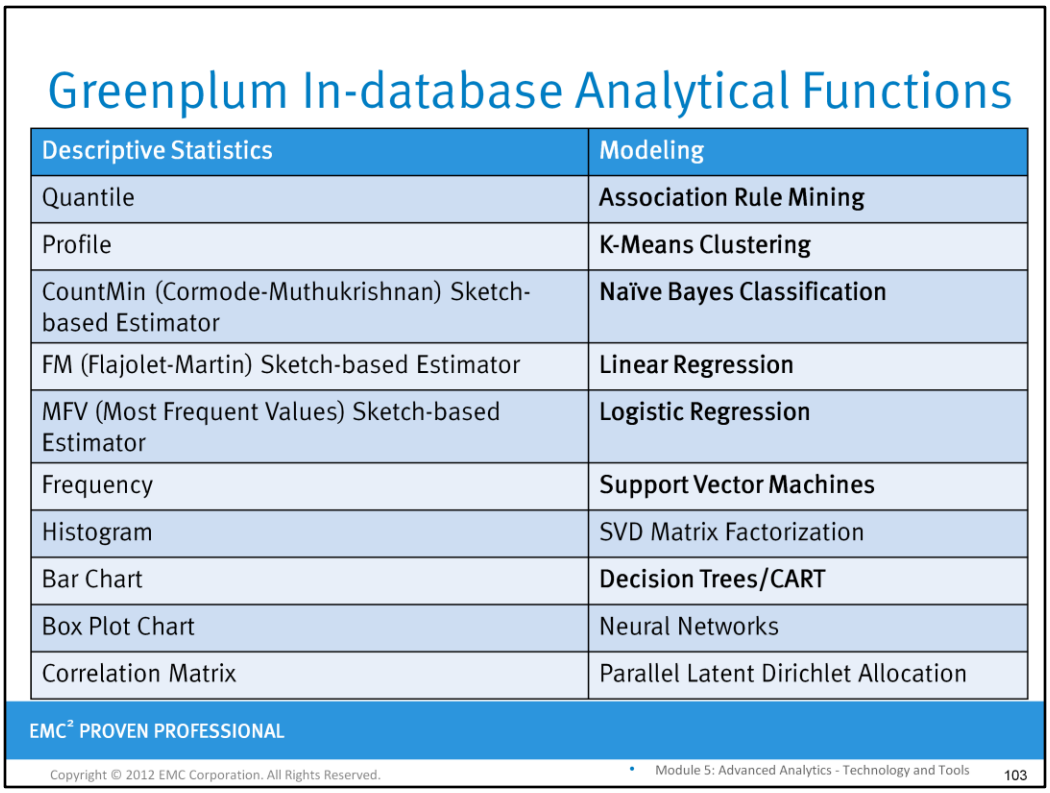
MADlib: Getting Help…

- Check out the **user guide** with examples at: *http://doc.madlib.net*

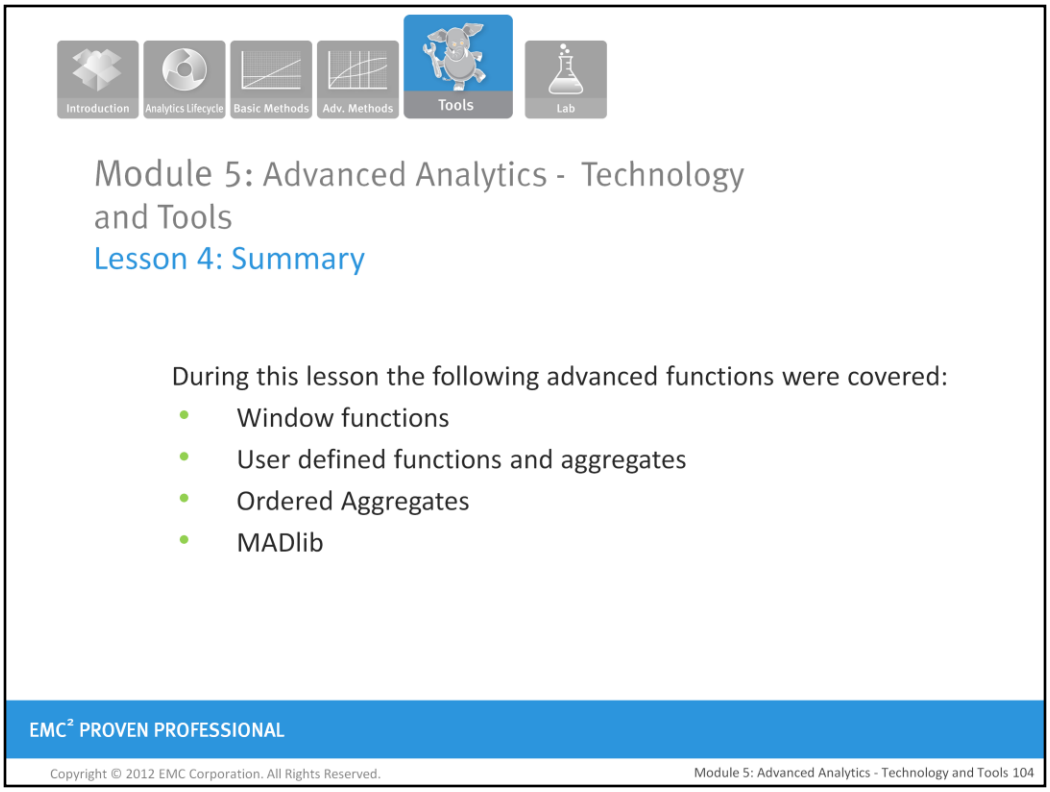
- Need more help?
  Try: http://groups.google.com/group/madlib-user-forum

EMC² PROVEN PROFESSIONAL

The slide provides additional details on obtaining help with MADlib functions.



Listed are the in-database analytic functions available natively in Greenplum and as Madlib functions (MADlib functions in bold). This list keeps expanding with every update and as the user community contributes to the MADlib.



Advanced SQL and MADlib functions were covered in this lesson.

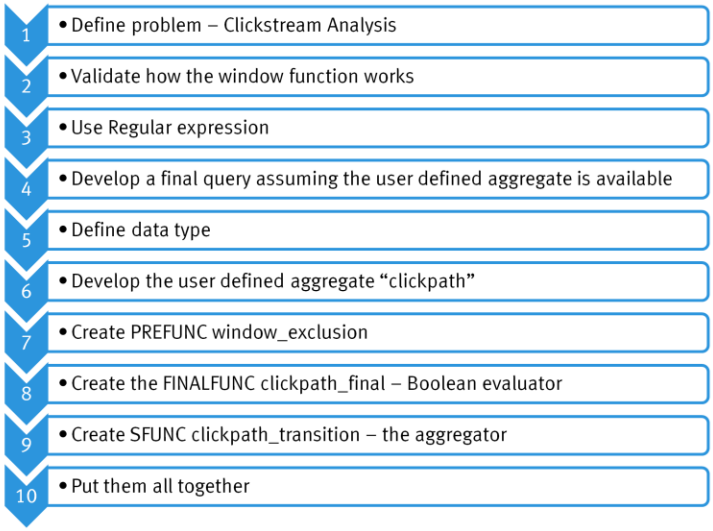
Lab Exercise 12 - In-database Analytics

After completing the tasks in this lab you should able to:

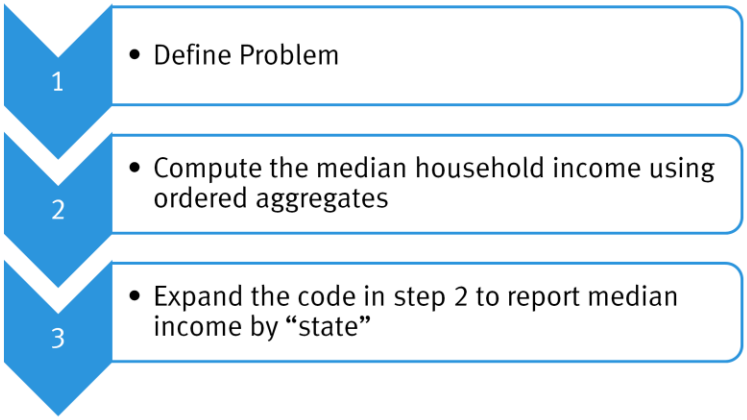
- Use window functions
- Implement user-defined aggregates, user-defined functions
- Use ordered aggregates
- Use Regular expressions in SQL for text filtering
- Use MADlib functions and plot results from MADlib function outputs

EMC² PROVEN PROFESSIONAL

Lab Exercise 12 - In-database Analytics – Part 1 – Clickstream analysis - Workflow

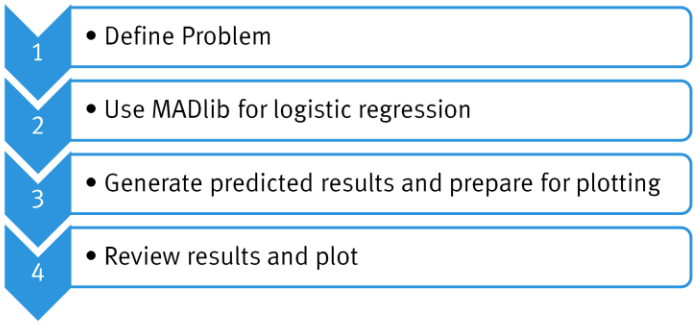


Lab Exercise 12 - In-database Analytics – Part 2 – Compute Median - Workflow

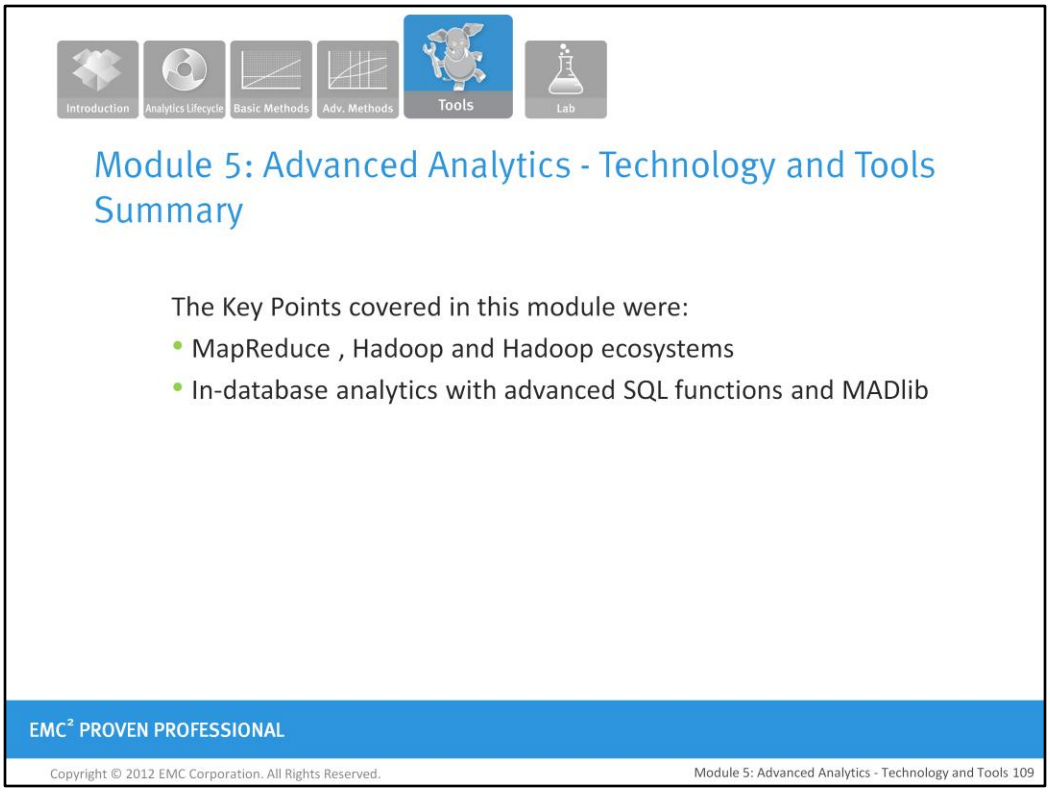


EMC² PROVEN PROFESSIONAL

Lab Exercise 12 - In-database Analytics – Part 3 – Logistic Regression with MADlib- Workflow



EMC² PROVEN PROFESSIONAL



These are the key topics covered in this module.

This slide intentionally left blank.

EMC² PROVEN PROFESSIONAL