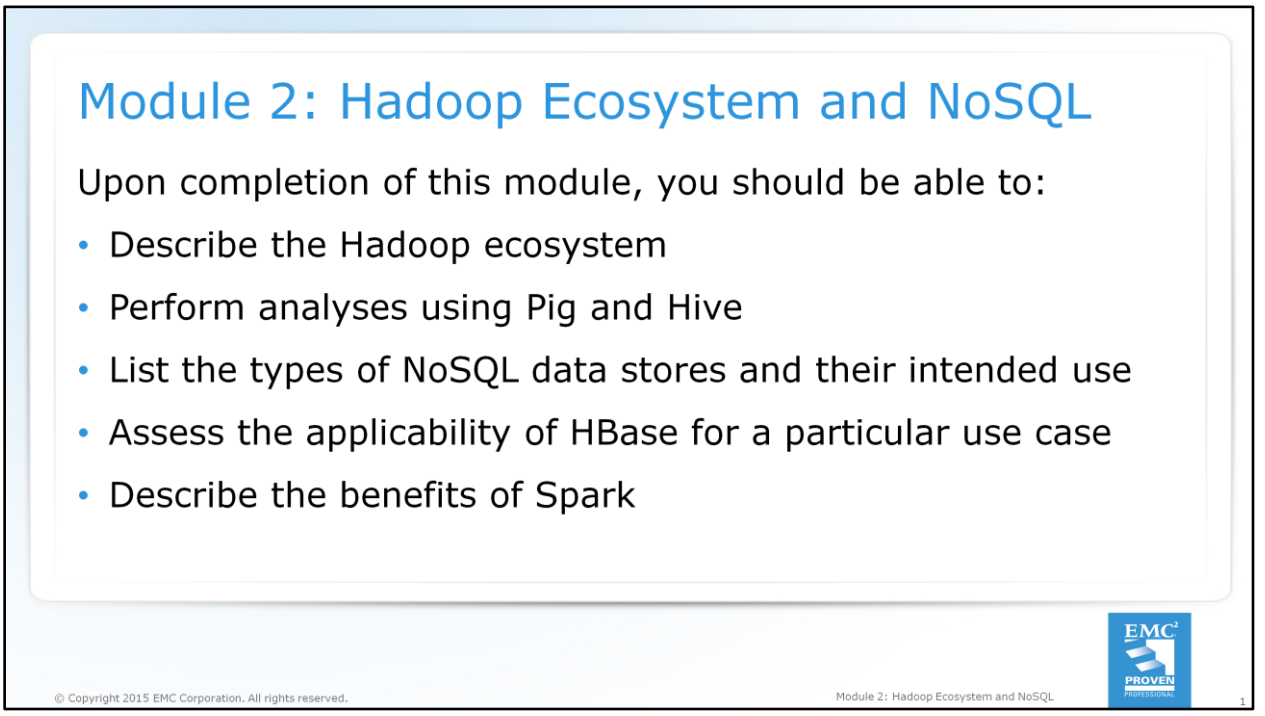
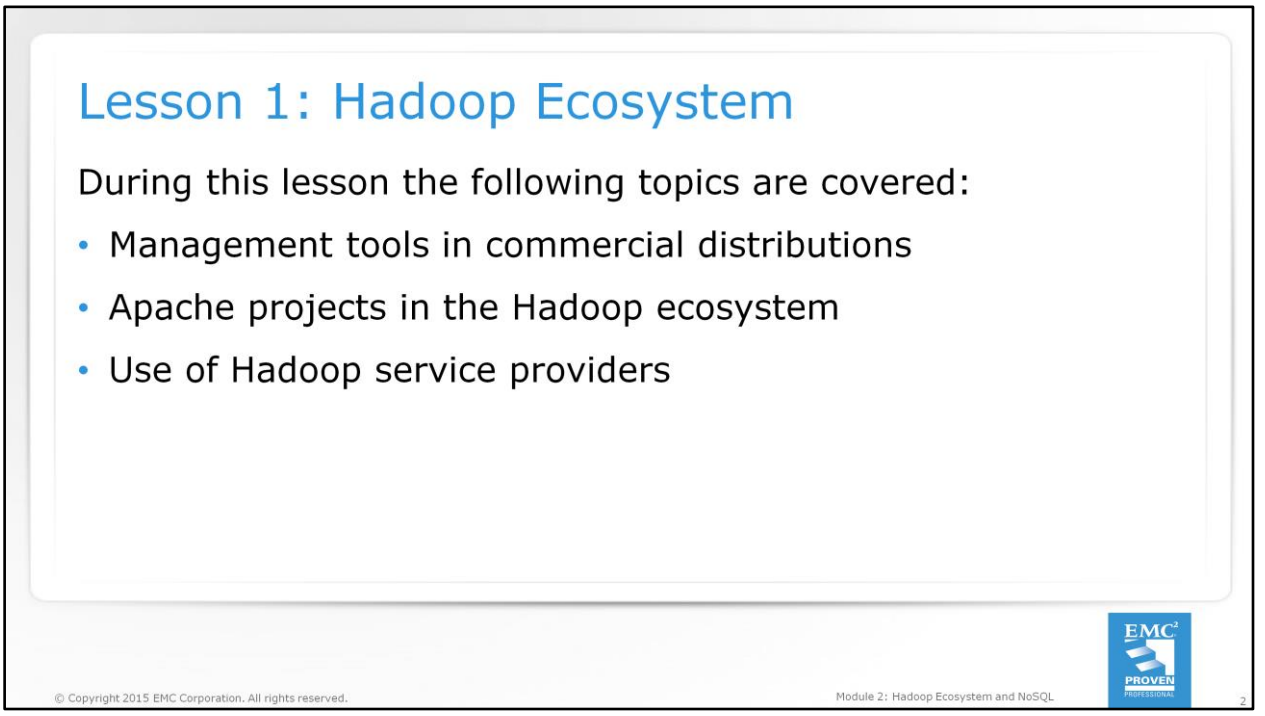


This module provides an overview of the Hadoop ecosystem and NoSQL. This module also presents the essential operations and commands to properly utilize Pig, Hive, and HBase. Finally, this module ends with an overview of Spark, a recent addition to the Hadoop ecosystem.

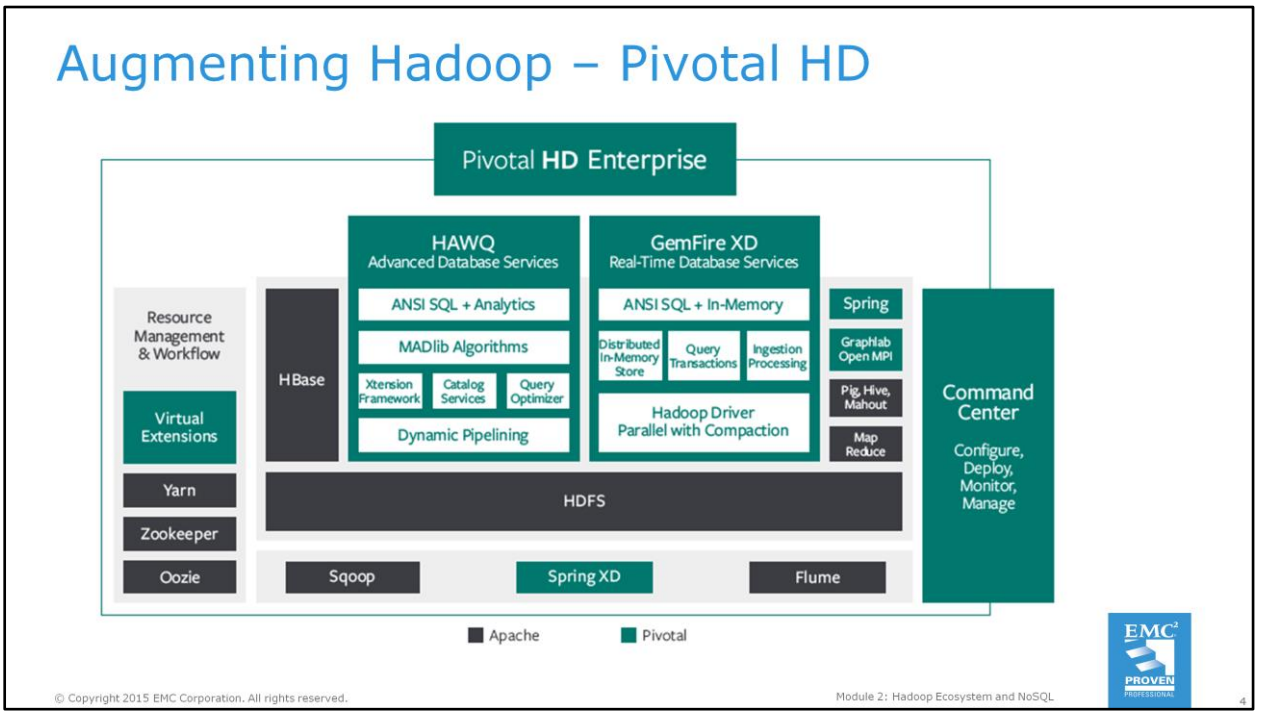


This lesson discusses the options to simplify the installation, maintenance, and use of Apache Hadoop.

Benefits of Commercial Distributions

- Pre-packaged open source and commercial tools
- Simplified installation and deployment
- Hardware options
- Cluster management and monitoring tools
- Customer service support options
- Example distributions:
  - Cloudera
  - Pivotal

There are several commercial distribution options from companies such as Cloudera and Pivotal. Such distributions package together Apache Hadoop, other related Apache tools, and additional utilities to simplify the installation, maintenance, and operation of Hadoop cluster environments. The pre-package distributions often are included with hardware and support options.



To illustrate the additional features and functionality provided in commercial distributions, consider Pivotal HD which includes the following:

Command Center is a robust cluster management tool that allows users to install, configure, monitor and manage Hadoop components and services through a Web graphical interface. It provides a comprehensive dashboard with instant views of the health of the cluster and key performance metrics. Users can also view live and historical information about the host, application and job-level metrics across the entire Pivotal HD cluster.

Graphlab OpenMPI is a highly used and mature graph-based, high performing, distributed computation framework for data scientists for graph analytics such as page rank.

Hadoop Virtualization Extensions (HVE) are plug-ins that enable Hadoop virtual ware. Pivotal HD is the first Hadoop distribution to include HVE plug-ins, enabling easy deployment of Hadoop in enterprise environments.

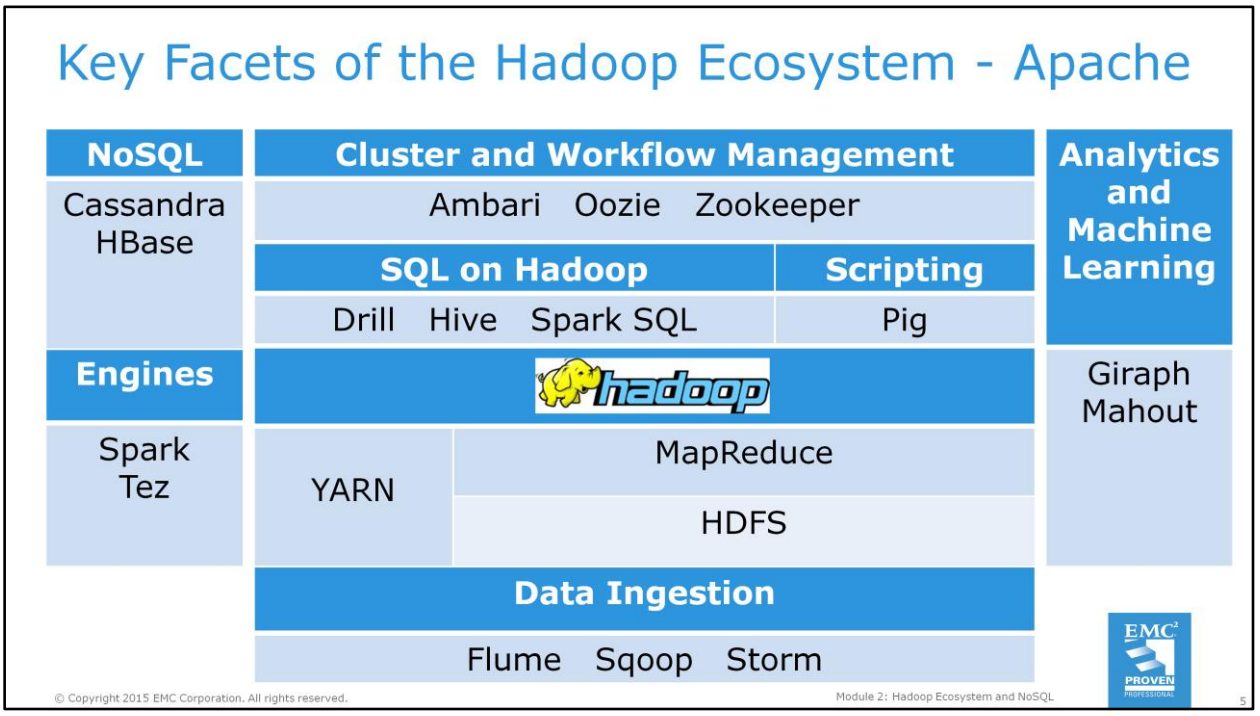
Spring for Apache Hadoop simplifies developing Big Data applications by providing a unified configuration model and easy-to-use APIs for using HDFS, MapReduce, Pig, and Hive.

Pivotal HAWQ is a powerful SQL query optimizer and processor that is optimized to run analytical queries and mixed query workloads in massively parallel, distributed environments.

Pivotal Xtension Framework (PXF) is an external table interface in HAWQ, which enables access to data stored within the Hadoop ecosystem. External tables can be used to load data into HAWQ from Hadoop and/or also query Hadoop data without materializing it into HAWQ.

GemFire XD enables the creation of low latency, scale out OLTP applications integrated out of the box with a big data store (HDFS). This provides sub-second response to applications, while allowing the data to be analyzed in the back end via HAWQ or Map Reduce in near real time.

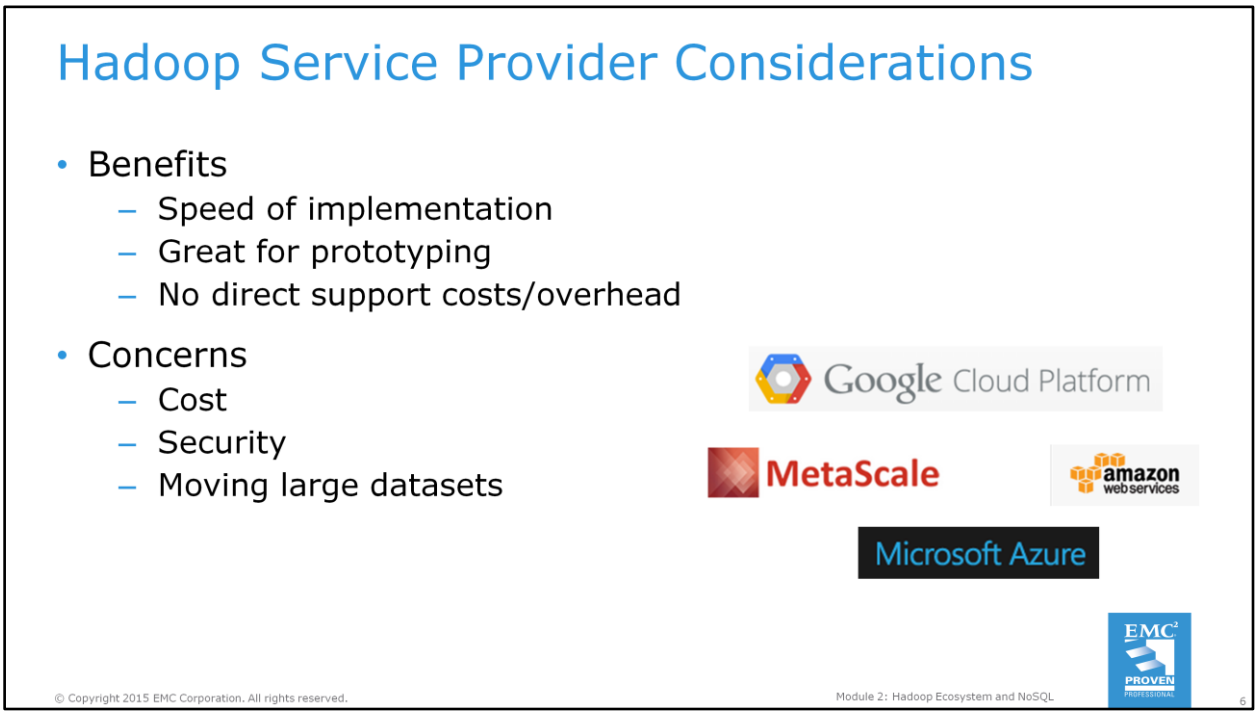
Reference: http://www.pivotal.io/big-data/pivotal-hd



This diagram provides several key categories for the Hadoop ecosystem members. At the center is Hadoop, which includes HDFS, MapReduce, and YARN. For the developers or analysts with experience in SQL, tools such as Drill, Hive, and Spark may be used to analyze HDFS data files. Pig is a good choice for those users who are comfortable with scripting languages. For more complex data mining and analytical activities, tools such as Giraph and Mahout provide routines for graph and clustering analysis, for example.

More information on these tools as well as other Apache projects can be found at the following web sites:

http://www.apache.org
http://incubator.apache.org



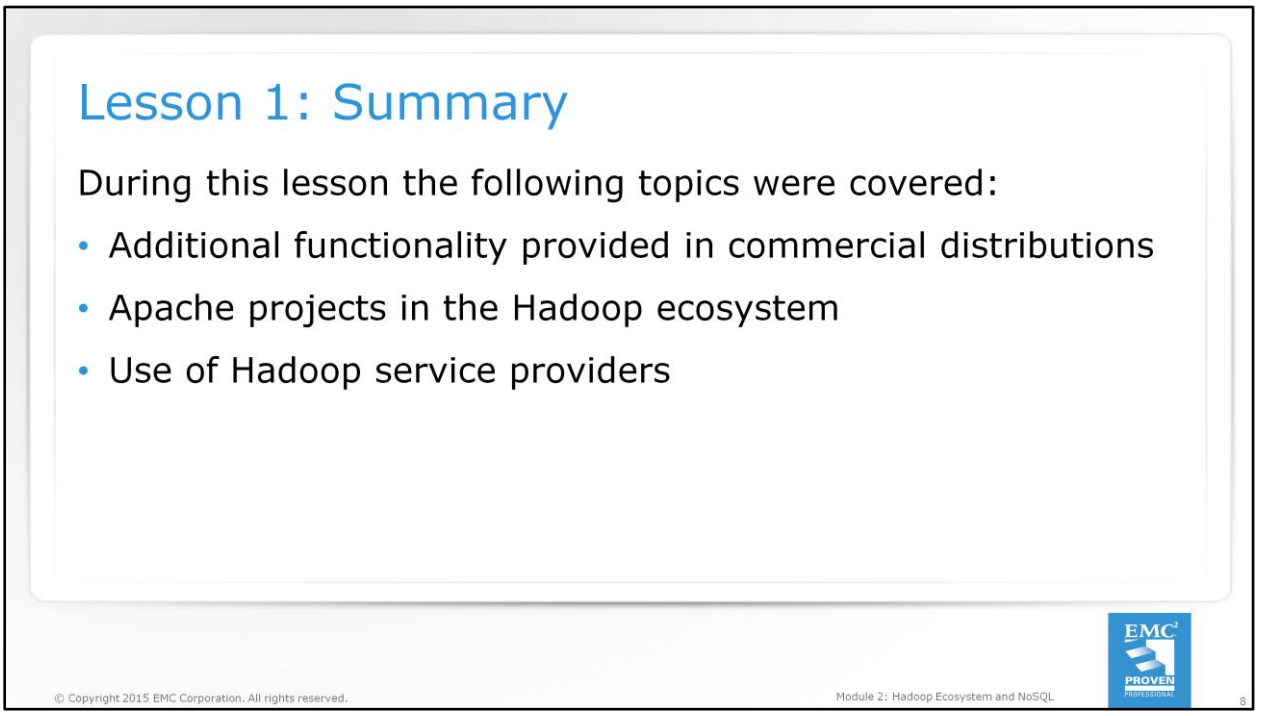
The clear advantage of using Hadoop from a Software as a Service (SaaS) provider is the speed at which a cluster can be provisioned and development work can begin, especially for prototyping and proof of concepts. With SaaS, there are no upfront costs for hardware, installation, and support. Concerns with using an external software provider are cost, security, and network issues with moving large datasets.

References for possible Hadoop service providers are listed below:
https://cloud.google.com/solutions/hadoop
http://www.metascale.com
http://aws.amazon.com/elasticmapreduce
http://azure.microsoft.com/en-us/services/hdinsight

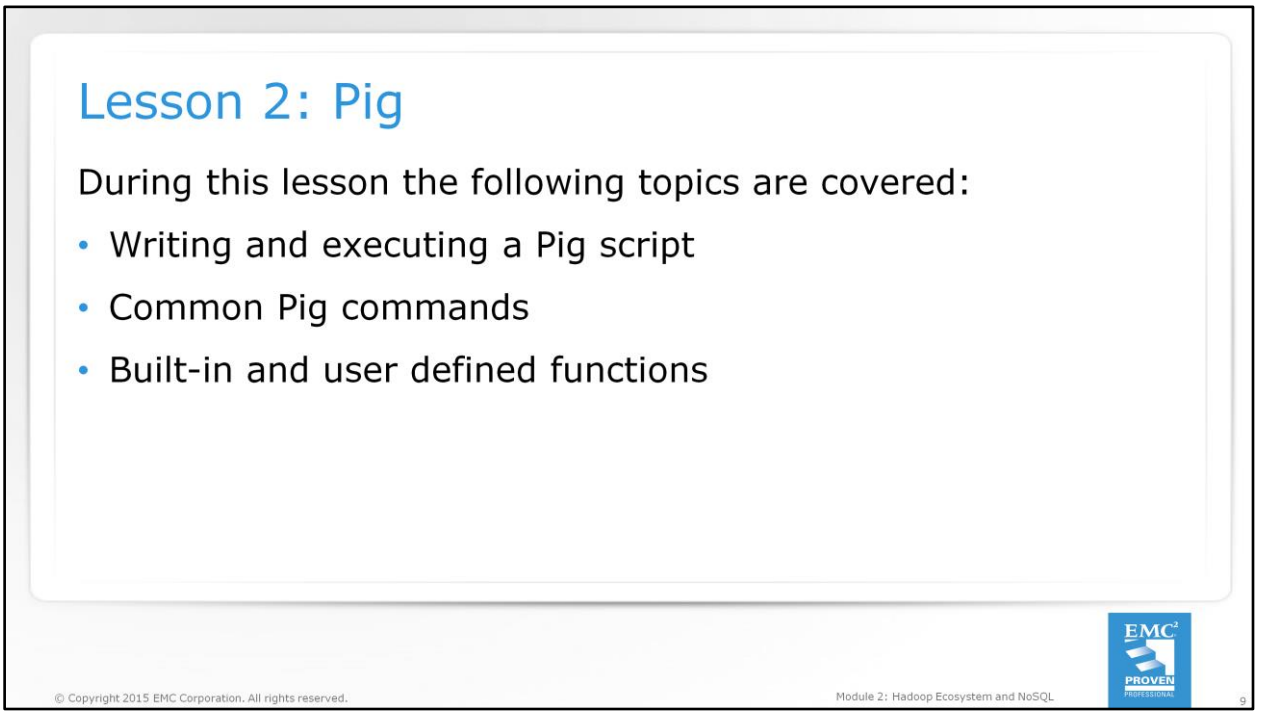
Preview of Upcoming Module 2 Lessons

- Pig – high level code translated into MapReduce jobs
  - Implementing User Defined Functions
- Hive – query language on top of Hadoop
  - Define and load tables
  - Sample queries
- NoSQL (Not Only SQL)
  - Various types and applications
- HBase – a distributed, scalable, big data store on top of Hadoop
  - Design and architecture
- Spark – execution engine supporting in-memory computing
  - YARN application

The remaining Module 2 lessons covers Pig, Hive, NoSQL, HBase and Spark, and how they can be applied to in a Hadoop environment.



This lesson described the benefits of the commercial Hadoop distributions that provide additional management software and tools, analytical capabilities, and extensions to improve data access. These commercial offerings often include additional Apache projects such as Hive and Pig. Also, the pros and cons of using a Hadoop service provider were presented.



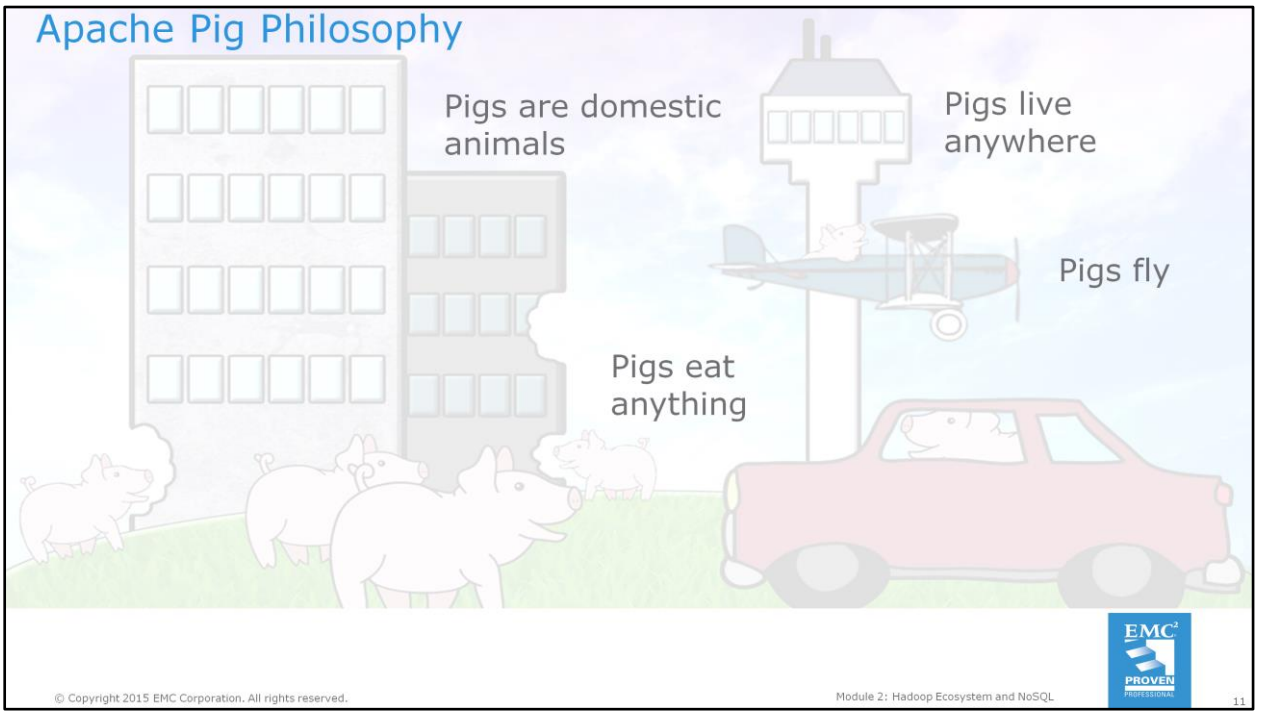
This lesson focuses on the use of Pig to perform analytics in a Hadoop environment.

Overview of Pig

- Pig is an Apache Project
- Pig includes:
  - Environment
  - Language (Pig Latin)
- Pig Latin can be translated into MapReduce code
- Motivation behind Pig
  - Programming MapReduce can be difficult and time consuming
  - Make the power of MapReduce accessible to more developers

Pig is an Apache project that provides a scripting language known as Pig Latin. Pig development began at Yahoo to simplify the effort to code MapReduce jobs.

Image: "Welcome to Apache Pig!" *pig.apache.org,* http://pig.apache.org/



There are four tenants to the Apache Pig Philosophy.

Pigs Eat Anything: Pig processes structured, relational, nested, or unstructured data.

Pigs Live Anywhere: Although Pig was developed first on Hadoop, Pig is not intended to be solely Hadoop dependent.

Pigs Are Domestic Animals: Pig is designed to be easily commanded, like a pet or farm animal, by its users. Pig provides an optimizer to enable performance tuning. Also, Pig accommodates the addition of user-defined functions.

Pigs Fly: Pig needs to quickly process data. New features and functionality will not be added to Pig that will prevent Pig from flying.

Reference: "Apache Pig Philosophy," *pig.apache.org,* February 24, 2015, http://pig.apache.org/philosophy.html

Executing Pig Statements

- From a Pig script

```
$ pig mycode.pig
```

- Interactively

```
$ pig
grunt> LOAD 'input/lab1/Cristo_ch1.txt' as body;
grunt> <enter additional statements>
grunt> quit
```


Two options for executing pig statements are from a script file or from the interactive Pig environment. When in the Pig environment, the user enters Pig Latin statements at the grunt command prompt.

Word Count in Pig Latin

- Just 5 lines of code to perform a word count

```
doc = LOAD 'input/lab1/Cristo_ch1.txt' as body;
words = FOREACH doc GENERATE FLATTEN(TOKENIZE((chararray)$0))
                                                    AS word;
word_grp = GROUP words BY word;
word_count = FOREACH word_grp GENERATE group, COUNT(words);
DUMP word_count;
```


A word count can be performed in as little as five lines of Pig Latin code.

1. The LOAD statement specifies the text file to be processed.
2. The 2nd statement sets the requirement to split the text into words (TOKENIZE).
3. Next, the need to group identical words together is specified.
4. For each word grouping (word_grp), the counting methodology is stated.
5. Finally, the results are written (DUMP) to the screen.

It is important to note that as the first four lines of code are run, no data is processed. From these first few lines of code, the Pig environment constructs an execution plan. This plan is submitted as a series of MapReduce jobs, whenever a DUMP command is processed.

So, nothing is truly stored in the apparent variables, known as **relations** in Pig. For example, if the DUMP command was processed a second time, the same MapReduce jobs would need to be submitted. In the next few slides, relations and the associated Pig Latin terminology will examined.

Pig Latin Statements Process Relations

- A relation is a collection (bag) of tuples — R = {t0, t1, … tn}
- A tuple is an ordered set of fields — ti = (f0, f1, … fm)
- A field is a piece of data — f0 = 'the' f1=201

```
words = FOREACH doc GENERATE FLATTEN(TOKENIZE((chararray)$0))
                                                    AS word;
DESCRIBE words;
words: {word: chararray}      ←  a bag of one-tuples
DUMP words;
(On) (the) (24th) (of) (February) (1810) …
```


A relation is a collection or bag of tuples. A common tuple is key/value pairs. In the word count example, a tuple consists of a word for the key and the word's count for the value such as <'the', 380>.

The DESCRIBE operator displays a relation's schema, if one exists. In this example, the relation, words, is a bag of one-tuples, where each tuple is a word in the first chapter from the Count of Monte Cristo.

References:

"Relations, Bags, Tuples, Fields," Pig Latin Basics, *pig.apache.org,* July 5, 2014, http://pig.apache.org/docs/r0.13.0/basic.html#relations

A More Complex Relation

- *word_grp* is a bag of ordered pairs (f0, f1)
  - f0 (group) is a unique word
  - f1 (word) is a bag of words

```
word_grp = GROUP words BY word;
DESCRIBE word_grp;
word_grp: {group: chararray,words: {(word: chararray)}}
DUMP word_grp;
…
(On   {(On),(On)})
…
(the {(the),(the),…, (the) })
…
```


The schema of word_grp relation is somewhat more complex. Although word_grp is still a bag of tuples, in this case ordered pairs, each order pair consists of a unique word and a bag of the same word. The word appears in the bag for each occurrence in the text.

Although the schema of the word_count relation is not that interesting, it is included for completeness. word_count is a bag of ordered pairs where the first field is the unique word and the second field is the count of the occurrences of the word. As seen below, the data type for the second field is a long.

word_count = FOREACH word_grp GENERATE group, COUNT(words);

DESCRIBE word_count;

word_count: {group: chararray,long}

DUMP word_count;

.
.
.
(On , 2)
.
.
.
(the, 201)
.
.
.

Debugging Code with the Illustrate Operator

```
illustrate word_count;
```

```
-------------------------------------------------------------
| doc     | body:bytearray                                  |
-------------------------------------------------------------
|         | "Yes, if you have nothing more to say to me." |
-------------------------------------------------------------
-------------------------------------
| words     | word:chararray        |
-------------------------------------
|           | Yes                   |
|           | if                    |
|           | you                   |
|           | have                  |
|           | nothing               |
|           | more                  |
|           | to                    |
|           | say                   |
|           | to                    |
|           | me.                   |
-------------------------------------
```


In this example, the illustrate operator runs the word count Pig Latin code on a random subset of the input file. As seen on this slide and the next slide, the output from the illustrate statement provides the resulting structures and tuples for the various Pig identifiers, doc, words, word_grp, and word_count.

Additional Output from *illustrate word_count*;

word_grp	group:chararray	words:bag{:tuple(word:chararray)}
	if	{(if)}
	to	{(to), (to)}
	Yes	{(Yes)}
	me.	{(me.)}
	say	{(say)}
	you	{(you)}
	have	{(have)}
	more	{(more)}
	nothing	{(nothing)}

word_count	group:chararray	:long
	if	1
	to	2
	Yes	1
	me.	1
	say	1
	you	1
	have	1
	more	1
	nothing	1

Finally, the illustrate output provides the structure and sample content of word_grp and word_count. To complete the discussion of the word count code, Pig Latin operators and built-in functions will be examined next.

Examples of Pig Operators

- Typical Operators
  - Arithmetic: +, - , *, /
  - Boolean: AND, OR, NOT, IN
  - Comparison: ==, !=, >=, >, <, <=
- Pig-centric Operators
  - Flatten() : de-nests tuples and bags
  - Relational: FILTER, FOREACH, GROUP, JOIN

Pig Latin provides the standard set of arithmetic, Boolean, and comparison operators that one would expect to see in any high-level programming language. As seen in the Pig word count example, Pig Latin also includes several operators, such flatten() and filter, specifically designed to work with Pig's tuple and bag structures.

An extensive list of Pig Latin operators and their usage can be found at: http://pig.apache.org/docs/r0.14.0/basic.html.

Built-in Pig Functions

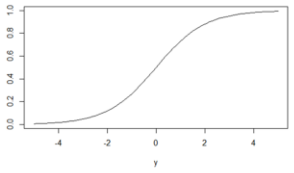
Evaluation	Load/Store	Math	String	Date/Time
AVG	BinStorage()	ABS	INDEXOF	AddDuration
CONCAT	JsonLoader	CEIL	LAST_INDEX_OF	CurrentTime
COUNT	JsonStorage	COS, ACOS	LCFIRST	DaysBetween
COUNT_STAR	PigDump	EXP	LOWER	GetDay
DIFF	PigStorage	FLOOR	REGEX_EXTRACT	GetHour
IsEmpty	TextLoader	LOG, LOG10	REPLACE	GetMinute
MAX	HBaseStorage	RANDOM	STRSPLIT	GetMonth
MIN		ROUND	SUBSTRING	GetWeek
SIZE		SIN, ASIN	TRIM	GetWeekYear
SUM		SQRT	UCFIRST	GetYear
TOKENIZE		TAN, ATAN	UPPER	MinutesBetween

Pig provides several built-in functions. The evaluation, math, string and date/time functions are typical functions provided in most programming languages. The load/store functions provide specific functionality to move data into and out of the Pig environment. The word count example used the COUNT and TOKENIZE functions.

Using a User-Defined Function (UDF)

- Python Script for the Logistic Function

$$f(y) = \frac{e^y}{1+e^y} \quad \text{for} -\infty < y < \infty$$



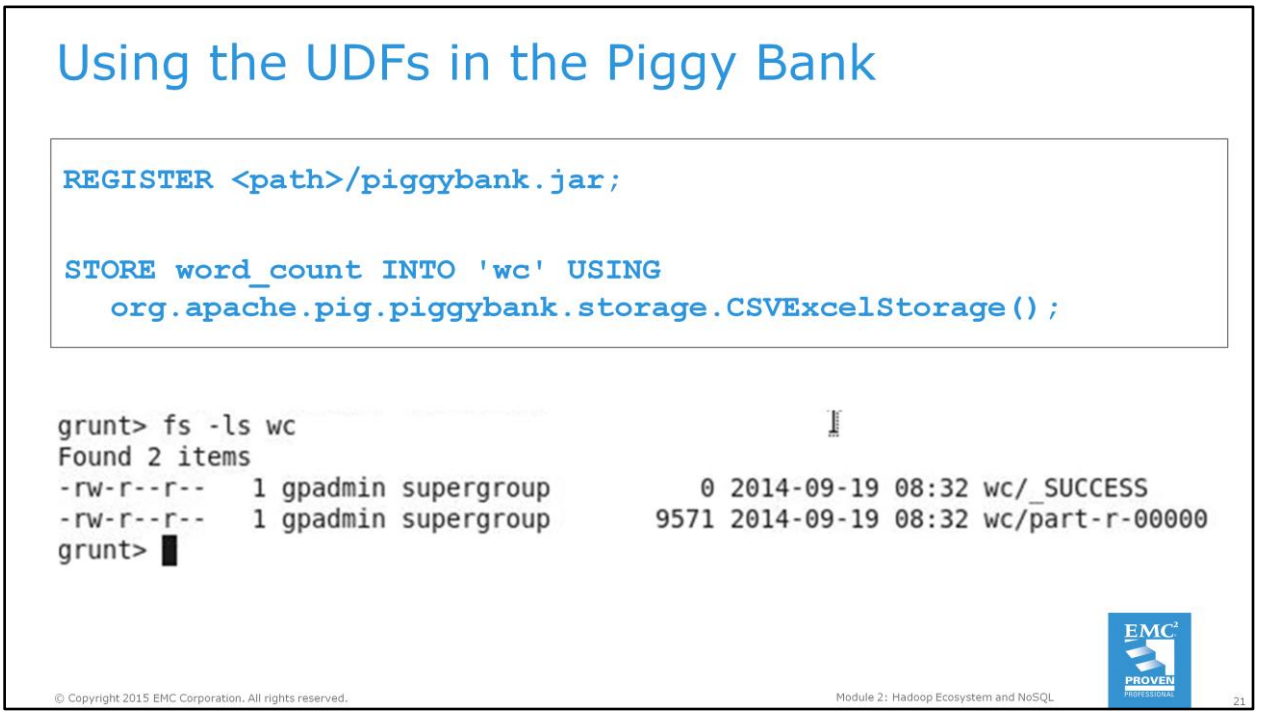
```
import math

@outputSchema("as:double")
def logistic(y):
    if y is None: return None
    return (1/(1+math.exp(0-y))
```

- Pig Statements

```
register 'logistic.py' using jython as udf;
Z = FOREACH Y GENERATE $0, udf.logistic($0);
DUMP Z;
(0,0.5)
(1,0.7310585786300049)
(2,0.8807970779778823)
```


A user-defined function (UDF) enables the creation of additional functionality beyond what is provided by the built-in functions. Furthermore a UDF can be written in Java or Jython, an implementation of Python written in Java. Some experimental languages include JavaScript, Ruby, Groovy, and Python.

In this example, the logistic function is expressed in a python script, `logistic.py`. After registering this script in the Pig environment, this function can be applied to the contents of a relation such as Y in this example. The output provides the corresponding tuples (y, f(y)) for contents of relation, Y.



Before writing a UDF, it is advisable to see if the function has already been written by another Pig user and deposited into the Piggy Bank. Over time, many of the functions in the piggy bank are included as built-in functions.

After registering the piggybank.jar file, the provided example writes the word_count relation to the wc folder in HDFS by taking advantage of the CSVExcelStorage() function which will create a comma separated file from the (word, count) tuples. Note: entering `fs –ls wc` at the grunt prompt allows the user to access the contents of HDFS in a similar manner as at the $ prompt.

Detailed instructions on downloading the source code and building the piggybank.jar file are located at the following sites:
https://cwiki.apache.org/confluence/display/PIG/PiggyBank
http://pig.apache.org/docs/r0.14.0/udf.html#piggybank

Performing an Inner Join

```
S = LOAD 'data1' as (s1: int, s2: int);
T = LOAD 'data2' as (t1: int, t2: chararray);
U = JOIN S by s1, T by t1;
V = FOREACH U GENERATE S::s1 as s1, S::s2 as s2, T::t2 as t2;
```

DUMP S;	DUMP T;	DUMP U;	DUMP V;
(1,4)	(1,A)	(1,4,1,A)	(1,4,A)
(2,5)	(3,B)	(3,7,3,C)	(3,7,C)
(3,6)	(3,C)	(3,7,3,B)	(3,7,B)
(3,7)		(3,6,3,C)	(3,6,C)
		(3,6,3,B)	(3,6,B)

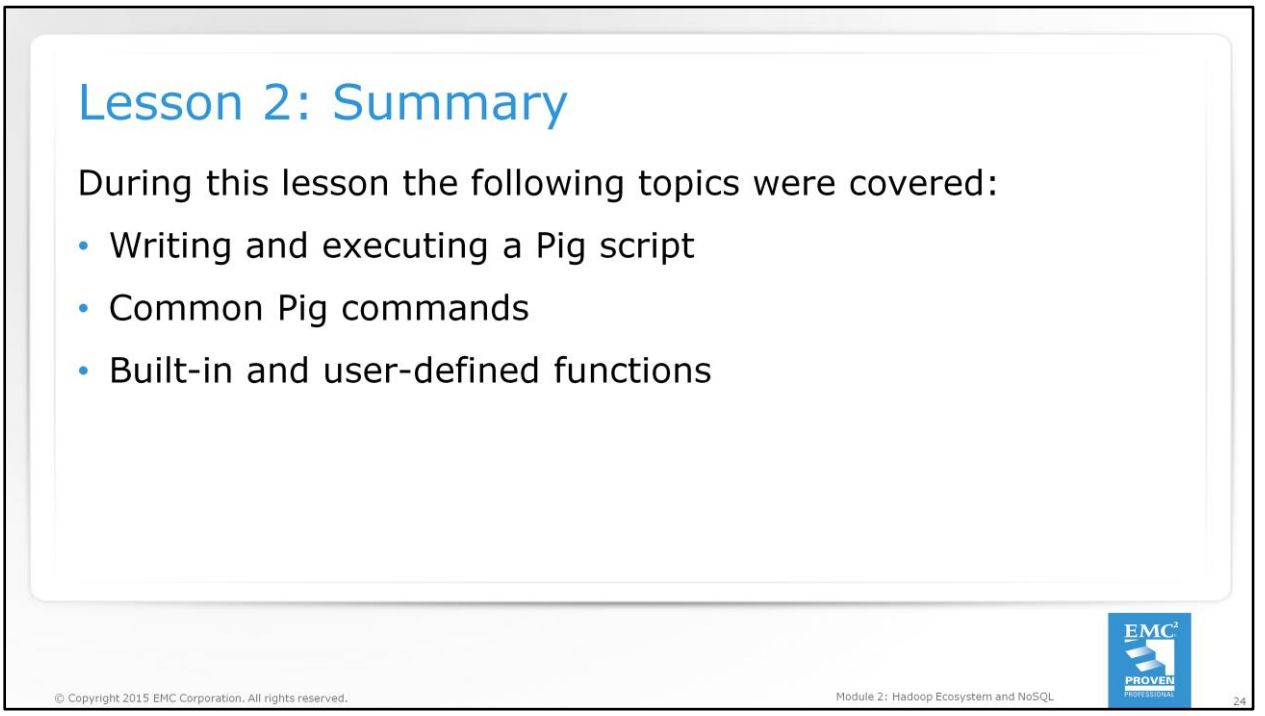
After loading datasets into relations S and T, an inner join is performed and the results are represented by relation U. As the DUMP U statement illustrates, all fields from S and T are included in U, but U still retains the knowledge of the source relations.

Performing an Outer Join

```
S = LOAD 'data1' as (s1: int, s2: int);
T = LOAD 'data2' as (t1: int, t2: chararray);
W = JOIN S by s1 LEFT OUTER, T by t1;
X = FILTER W by T::t1 IS NULL;
```

DUMP S;	DUMP T;	DUMP W;	DUMP X;
(1,4)	(1,A)	(1,4,1,A)	(2,5,,)
(2,5)	(3,B)	(2,5,,)	
(3,6)	(3,C)	(3,7,3,C)	
(3,7)		(3,7,3,B)	
		(3,6,3,C)	
		(3,6,3,B)	

In this example, a left outer join is performed to include all tuples in S that may not have a corresponding tuple in T. Those particular tuples are filtered out from the join results with the use of the keyword, NULL.



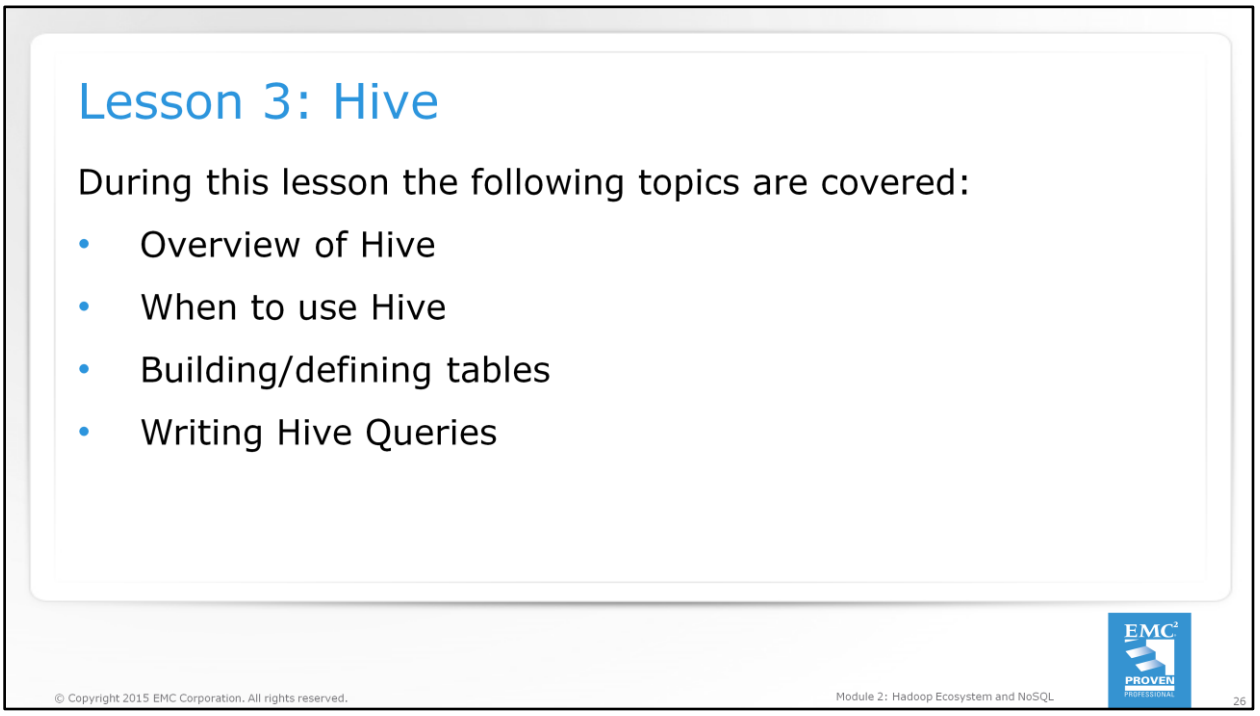
This lesson provided an overview of Apache Pig and how a simple word count program can be written using Pig Latin operators and commands. Pig Latin provides several built-in functions as well as the option of using a function from the Piggy Bank or writing a user-defined function.

Pig Lab

In this lab, you will build a training dataset for a classification model. The tasks include:

- Load HDFS files into relations
- Join and restructure the contents of the files
- Summarize the data set
- Export results to a CSV file

The pig lab addresses several common operators and statements as well as the use of the Piggy Bank.



This lesson focuses on the use of Hive to perform SQL-like analytics in Hadoop.

Overview of Hive

- An Apache project
- Defines queryable tables from HDFS files
- Hive Query Language (Hive QL)
  - Very similar to ANSI-SQL
  - Handles large distributed data sets
  - Hive QL runs MapReduce jobs
  - Batch oriented
- Based on the early efforts by Facebook
  - Migrated from a commercial RDBMS
  - Supported ad-hoc analyses and specific products

Similar to Pig, the intent of Hive is to simplify the analysis of data in a Hadoop environment. Hive does assume that data is already in some table-like structure for which the typical SQL commands can be applied. Similar to the reasons Pig was developed, Facebook wanted to take advantage of the power of Hadoop, but did not want to be burdened by having to deal with writing MapReduce jobs. Since the existing data warehouse was already using a RDBMS and its employees were already proficient in SQL, it was natural for Facebook to develop a SQL-like language, Hive QL, that could be translated into a series of MapReduce jobs.

Ref: https://www.facebook.com/note.php?note_id=89508453919

When to Use Hive

- Data easily fits a table structure
- Need to address large datasets and scalability issues
- Data is already in HDFS
  - Note: non-HDFS files can be loaded into a Hive table
- Developers are proficient with SQL programming and queries
- Desire to partition datasets based on time
  - e.g. Daily updates are added to the Hive table
- Batch processing is acceptable

Hive is ideal for those uses where datasets can be easily described as table structures that one would see in a relational database. Typically, the data already resides in HDFS, but it can be loaded from other data sources including a relational database. Once in the Hadoop environment, the data can be efficiently processed using familiar SQL. Hive includes the use of partitions which aid the addition of new data to existing tables. Since Hive utilizes MapReduce batch processing, Hive is not intended for real-time processing and applications. Real-time data stores will be address later in the NoSQL and HBase lessons.

Executing Hive QL Statements

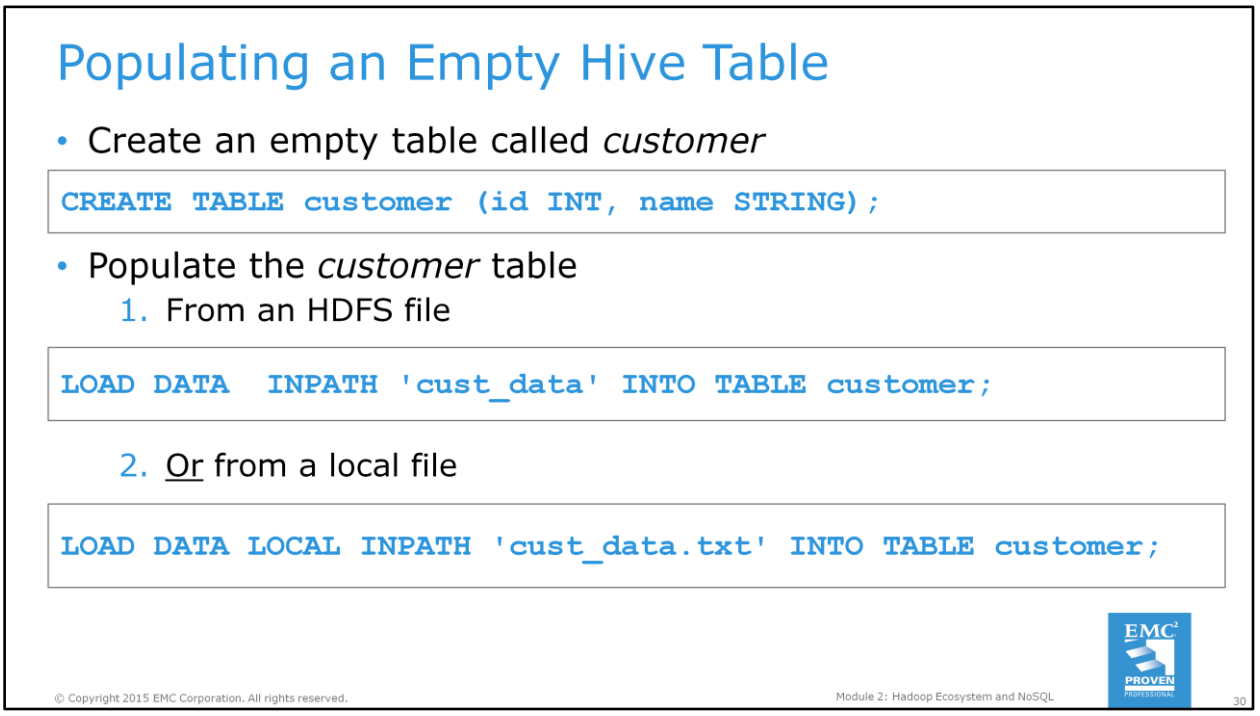
- From a Hive script

```
$ hive -f mycode.sql
```

- Interactively

```
$ hive
hive> select * from my_table;
hive> <enter additional statements>
hive> quit;
```

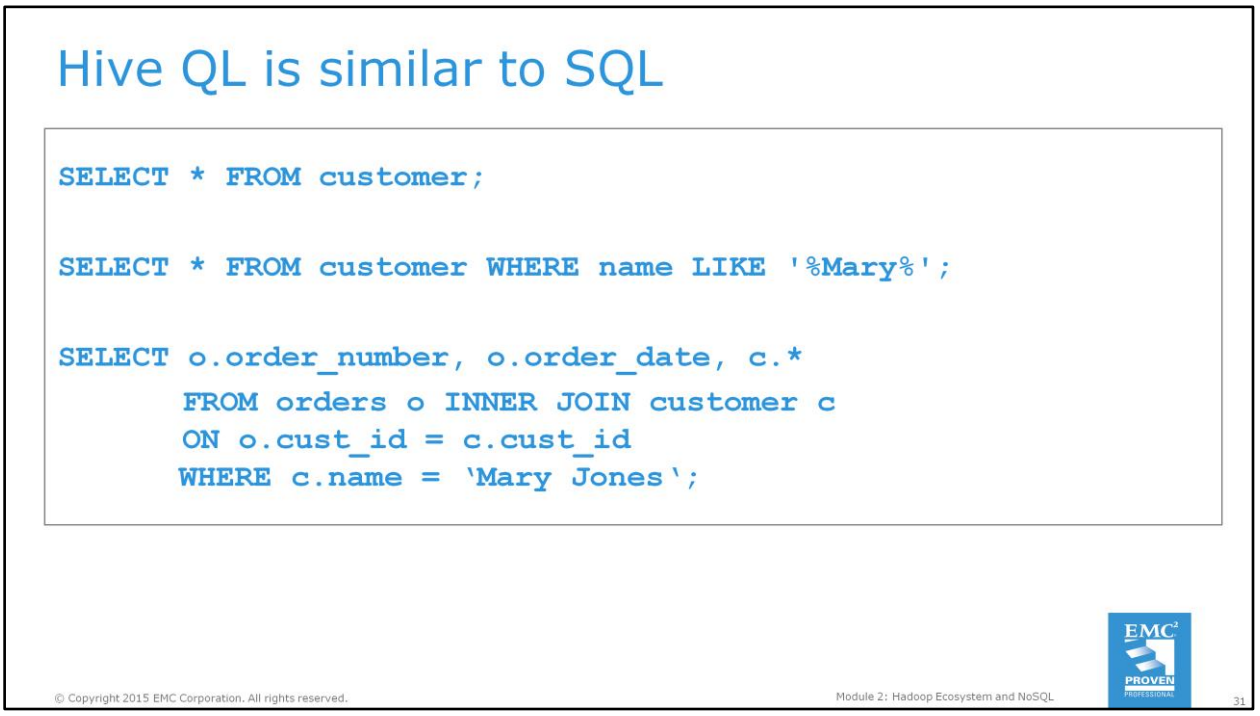
Two options for executing Hive QL statements are either from a script file or from within the interactive Hive environment. When in the Hive environment, the user enters Hive QL statements at the hive command prompt.



One of the first steps is to create Hive tables to store the data. The provided example creates a table with two columns to store the id number and the name of a customer. Using the LOAD command, the customer table can be populated from an HDFS file or from a local file. The default delimiter is the ASCII string Control-A (^A) character (\001).

Reference:

"Hive Data Manipulation Language," LanguageManual DML, *cwiki.apache.org*, Oct 22, 2014, https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML#LanguageManualDML-Loadingfilesintotables



Anyone familiar with SQL will be very comfortable with Hive QL. Originally, Hive was not intended for updates or deletes of table records. In such cases, Hive was well suited for Write Once Read Many (WORM) times use cases. However, starting with Hive 0.13.0, insert, update, and delete queries can be written to modify the contents of Hive tables. The current default is that a table may not be modified by Hive.

Text Analytics With Hive

```
SELECT CONTEXT_NGRAMS(SENTENCES(LOWER(comment)),
   array("i", "like", null, null), 100) FROM feedback;
[ {"ngram":["emc's","service"],"estfrequency":789.0},
   {"ngram":["the","performance"],"estfrequency":324.0}
   …]
```

Breakdown of `CONTEXT_NGRAMS(SENTENCES(LOWER(comment)…`

- `LOWER()`: converts a string to lower case
- `SENTENCES()`: converts the string to an array of words
- `CONTEXT_NGRAMS()`: in this example, finds the top 100 word pairs that follow the phrase "i like"

This example illustrates some of the more complex processing that can be accomplished in Hive. This SELECT query finds the top 100 four word phrases that start with "i like". An n-gram is an ordered list of n items, such as words.

More details on conducting analytics in Hive can be found at: https://cwiki.apache.org/confluence/display/Hive/StatisticsAndDataMining.

Essential Hive Database Design Considerations

- Table Structures
  - Normalized vs. denormalized
  - Accommodation for new columns
  - Partitions
  - Buckets
- Refresh frequency
- Number of Databases
- Audience

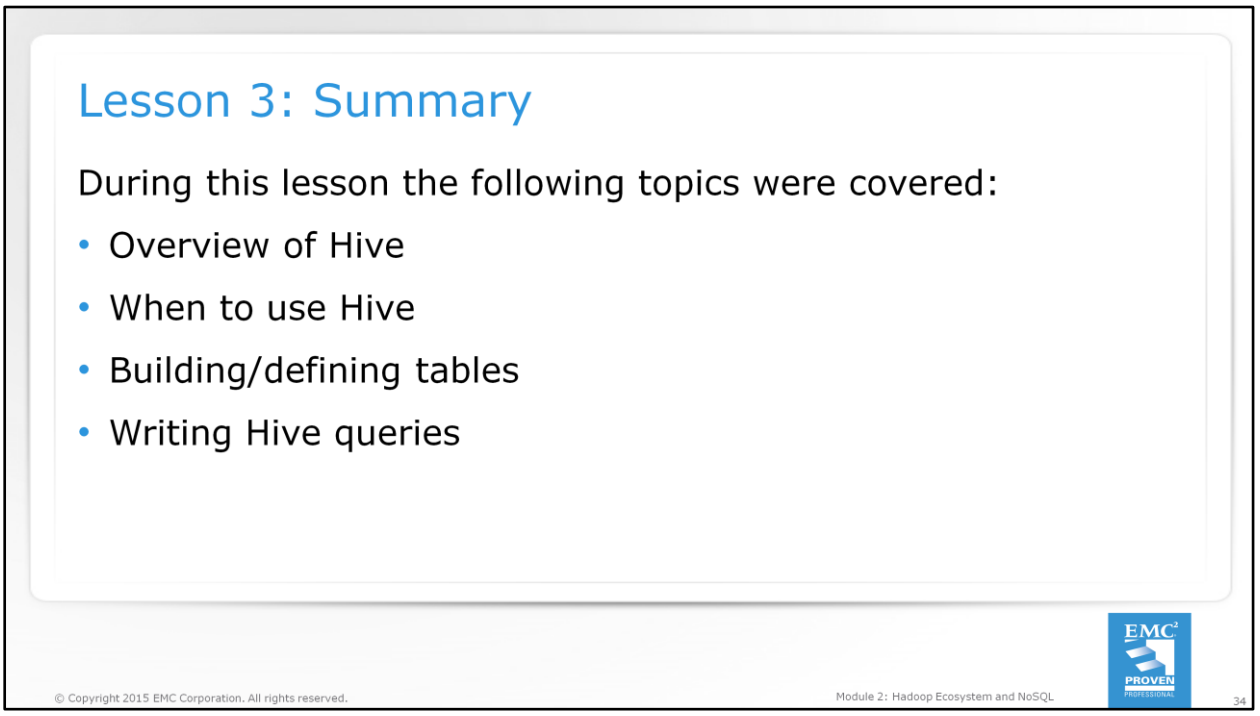
Designing a Hive data warehouse requires many of the same considerations as designing a typical RDBMS database. Should the design be highly normalized or should several of the tables be joined in advance to build one larger table. Related to this consideration, it is important to understand how new columns will be added to the existing table design as well as the update strategy/frequency for the Hive tables.

Partitions define how the table records are stored. For example, if new log file data is added daily to a Hive table, the table can be partitioned based on a date which will simplify the append operation and reduce the amount of data that needs to be examined when a certain data range is to be processed.

Applied to partitions or tables, buckets organize records based on specified columns. This organization can reduce the time required to perform joins on these columns and to query on these columns in the where clause. Properly bucketing of fields can lead to efficient joins.

Reference:

"LanguageManual DDL Bucketed Tables," LanguageManual DDL, *cwiki.apache.org,* Sept 28, 2013, https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL+BucketedTables



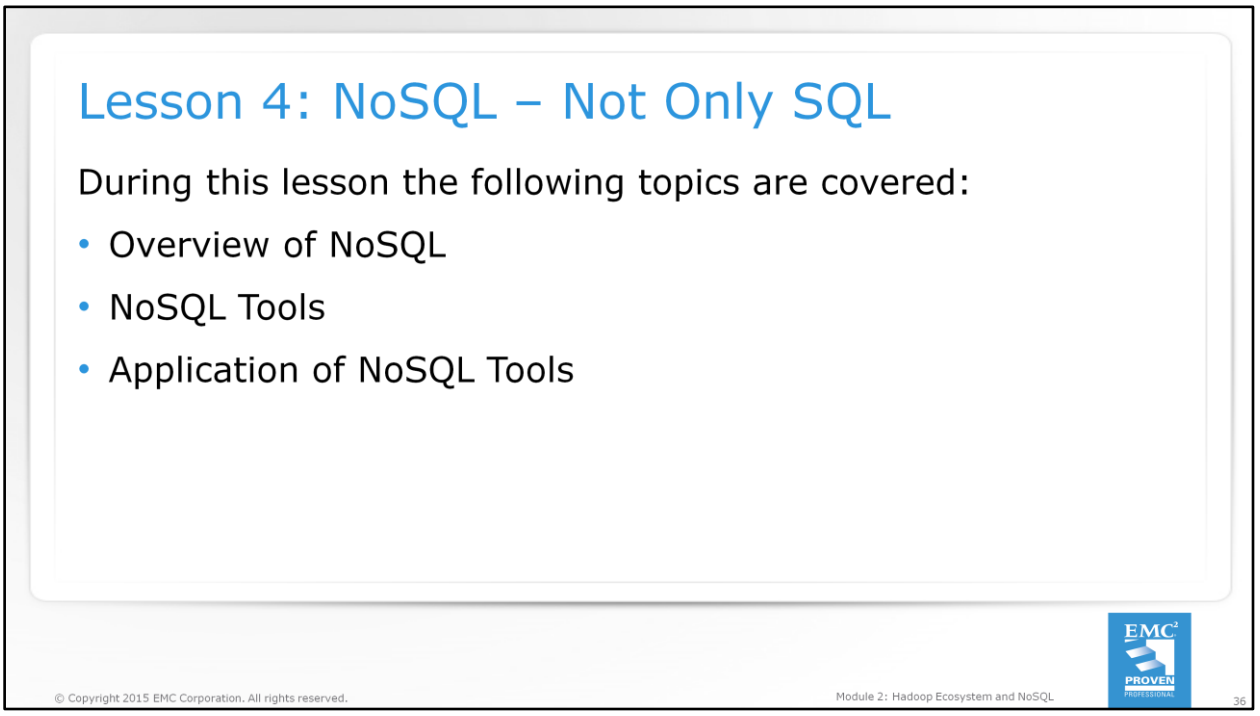
This lesson described the use of Apache Hive to analyze HDFS data sets that can be easily described by tables such as might be seen in a relational database. Hive QL enables the writing of SQL-like commands to query these tables individually or joined together.

Hive Lab

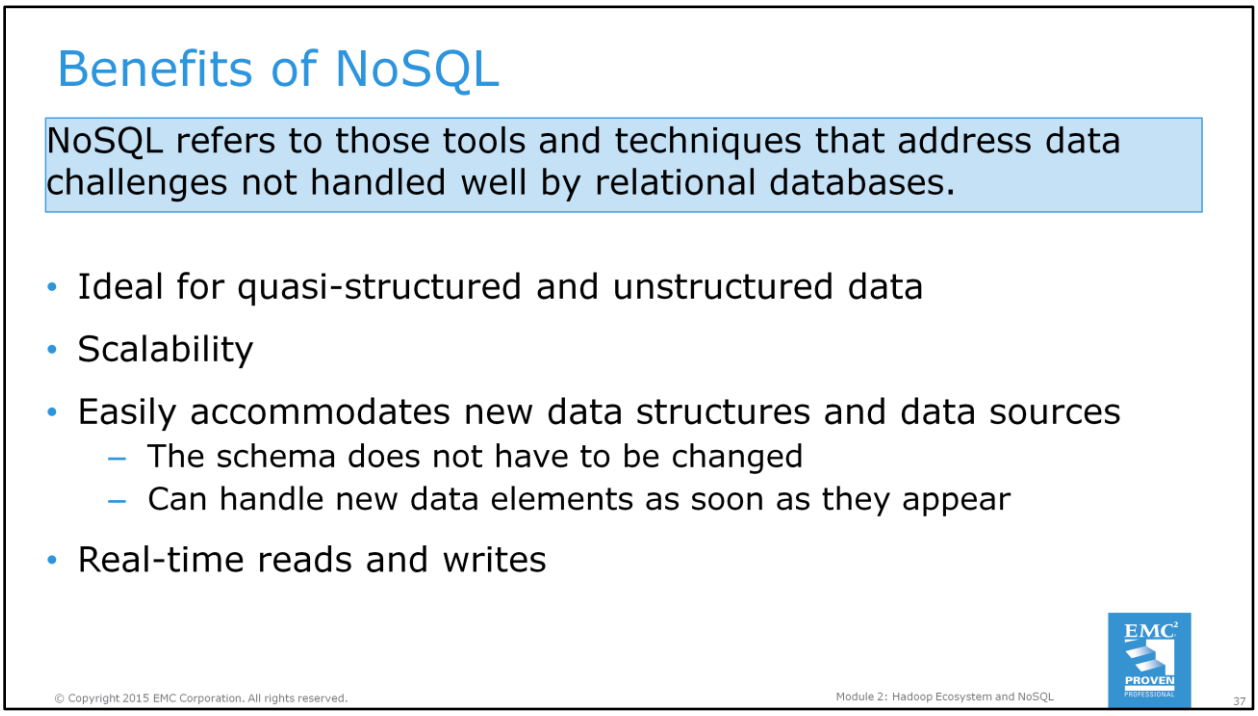
On the existing HDFS data, from previous lab:

- Create and populate Hive tables
- Perform an inner join
- Write an outer join
  - Place results into a Hive table
  - Export results to a csv file

This lab covers the essential tasks of loading data into a Hive table, performing JOIN operations, and exporting query results to a local file system.



This lesson focuses on the various NoSQL tools and their intended applications.

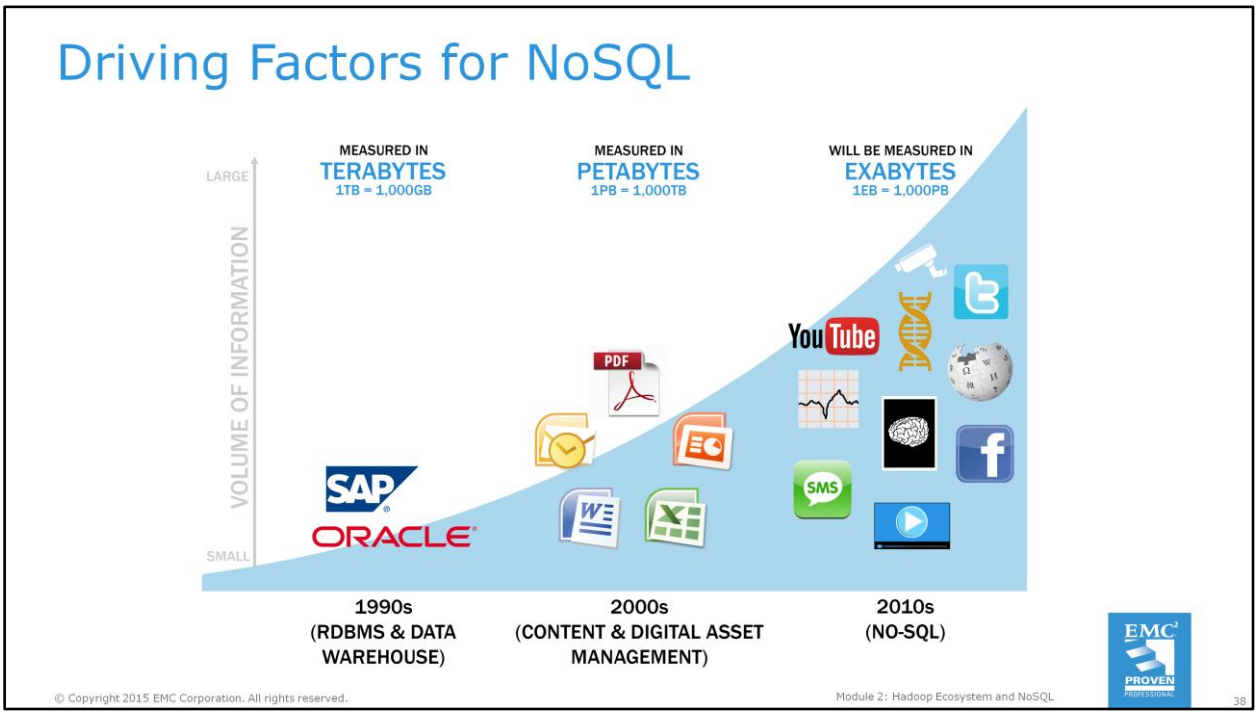


The term, NoSQL (Not only SQL), is applied to a wide range of tools and techniques. Relational databases with SQL have been very popular for over 30 years and will continue to play a prominent role in many enterprises in the near future. While relational databases are well suited to handle structured data, relational databases often have problems dealing with quasi-structured and unstructured data. Many NoSQL tools handle such unstructured data in a scalable fashion as the data volumes grow. Additionally, NoSQL tools can easily accommodate new data structures and data sources. For transactional applications, many NoSQL tools provide real-time read and write operations.

However, the power of NoSQL may come at a cost. Although not limited to merely NoSQL implementations, it is important to properly design an application when using NoSQL tools. For example, a bitcoin exchange was forced to shutdown after hackers exploited a flaw that allowed more bitcoins to be transferred from an account than were in the account by simultaneously submitting multiple transfer requests.

References:
"Flexcoin is shutting down," *Flexcoin: the bit coin bank*, March 3, 2014, http://flexcoin.com/

Ermin Gun Sirer, "NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex," *Hacking, Distributed,* Apr 6, 2014, http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/



The growth of big data has fostered the need for new tools such as Apache Hadoop to store and process data in a distributed manner. Additionally, the number of users as well as real-time processing requirements have led to the creation of several NoSQL tools. At the turn of the century, the number of concurrent users for any business application was typically in the hundreds. Now, sites such as Facebook and Twitter have to deal with millions of users concurrently.

Categories of NoSQL Tools

Category	Use Case	Example Tools
Key/Value stores	Web-site shopping cart	Redis, Voldemort
Columnar stores	E-mail search indexes	Cassandra, HBase
Document stores	Blog site	CouchDB, MongoDB
Graph databases	Social networks	FlockDB, Neo4J

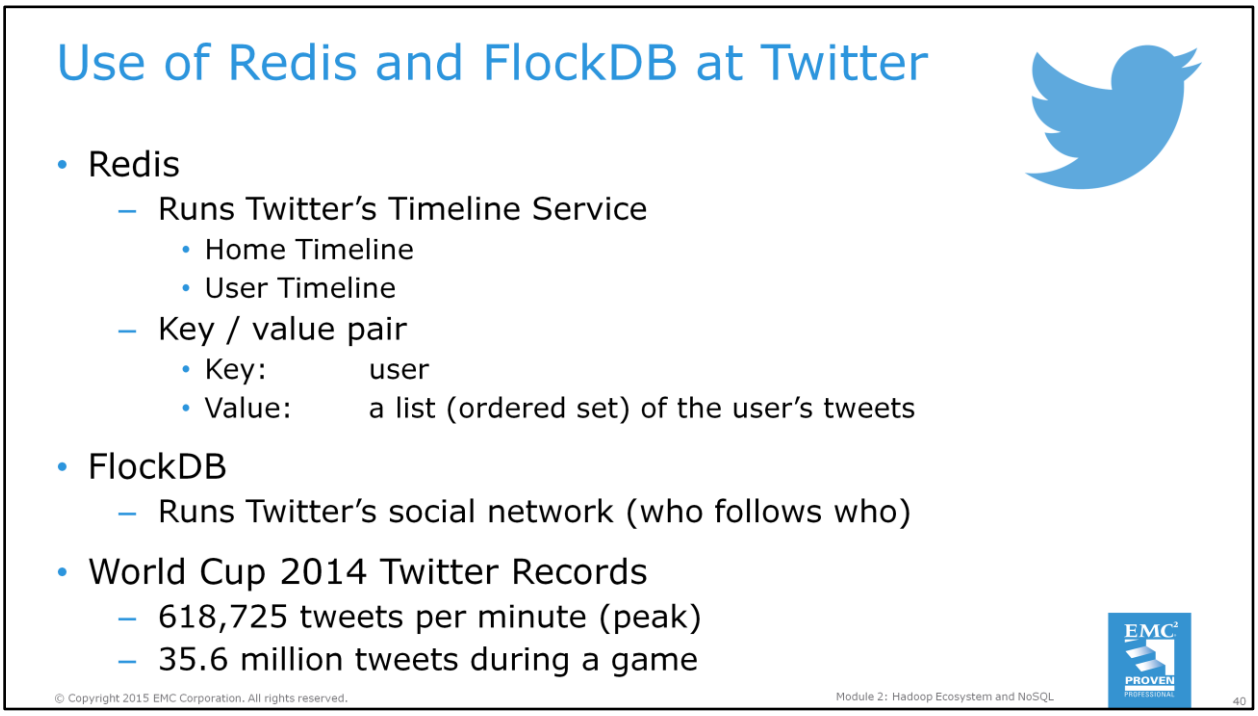
Categories of NoSQL Tools include key/value stores, columnar stores, document stores, and graph databases.

Key/value stores are distributed databases that provide high availability across a distributed system. A use case includes maintaining product catalog information, such as an item name and an HTML document to be displayed for the item. Another use case would be on a retail web site where user identification+session id is the key along with a value consisting of an object that can include user information, such as shipping address and shopping cart contents.
Redis: http://redis.io
Voldemort: http://www.project-voldemort.com/voldemort

Columnar stores are useful for sparse data sets, such as records with hundreds of columns but only a few entries. The key/value concept still applies, but in this case, a key is associated with a collection of columns. In this collection, similar columns are grouped together into column-families. More on columnar stores will be addressed in HBase lesson.
Cassandra: http://cassandra.apache.org
HBase: http://hbase.apache.org

Document stores are useful when the value of the key/value pair is a file, document, etc., and the file is self describing, e.g. JSON or XML. The underlying structure of the documents can be used to query and customize the display of the documents content. Since the structure may vary drastically from document to document, there is not necessarily any overall structure, or schema, to the documents in the store.
CouchDB: http://couchdb.apache.org
MongoDB: http://www.mongodb.org

Graph databases are intended for use cases, such a social networks, where there are items, such as people and web page links, and relationships between these items. While it is possible to store graphs such as trees in a relational database, it often becomes cumbersome to navigate, to scale, and to add new relationships. Graph databases help to overcome these possible obstacles.
FlockDB: https://github.com/twitter/flockdb
Neo4J: http://www.neo4j.org



Accommodating complex key/value pairs, Redis runs Twitter's timeline service which maintains the list of tweets on a user's home timeline as well as the list of tweets from a particular user. An interactive tutorial of Redis can be found at http://try.redis.io/.

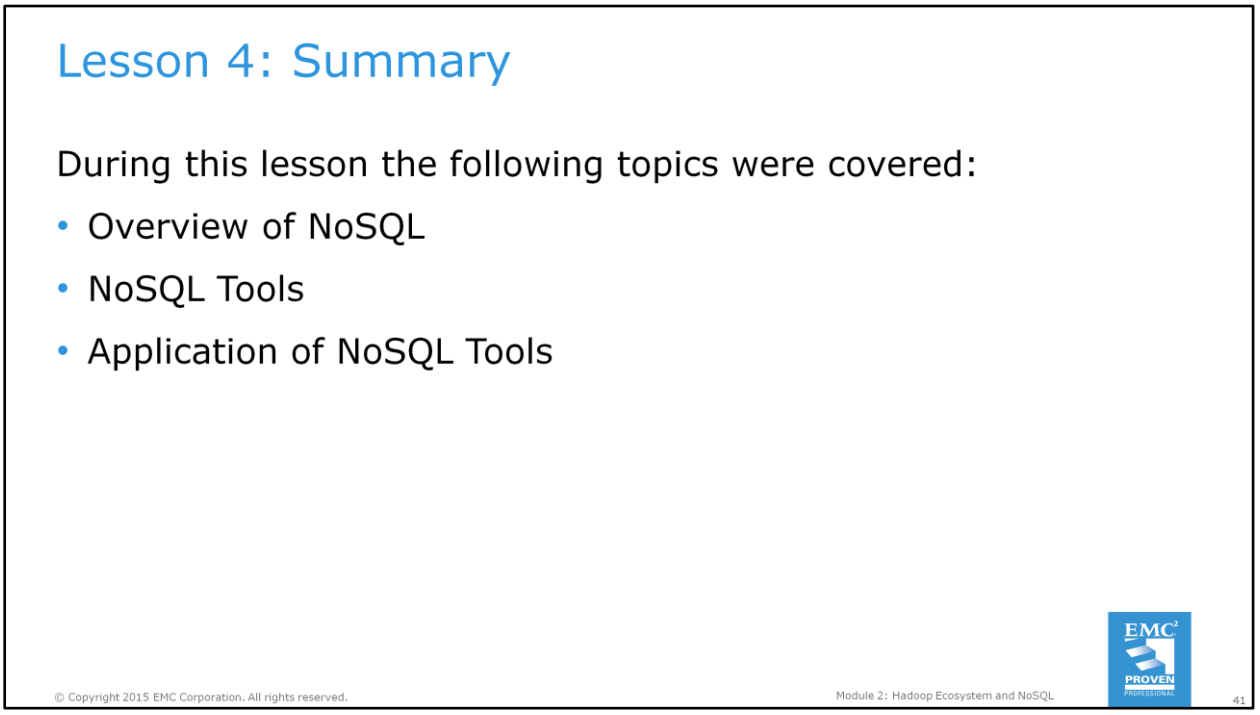
FlockDB manages Twitter's social network of who follows who.

References:
Todd Hoff, "How Twitter Uses Redis to Scale - 105TB RAM, 39MM QPS, 10,000+ Instances," *High Scalability,* Sept 8, 2014, http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html

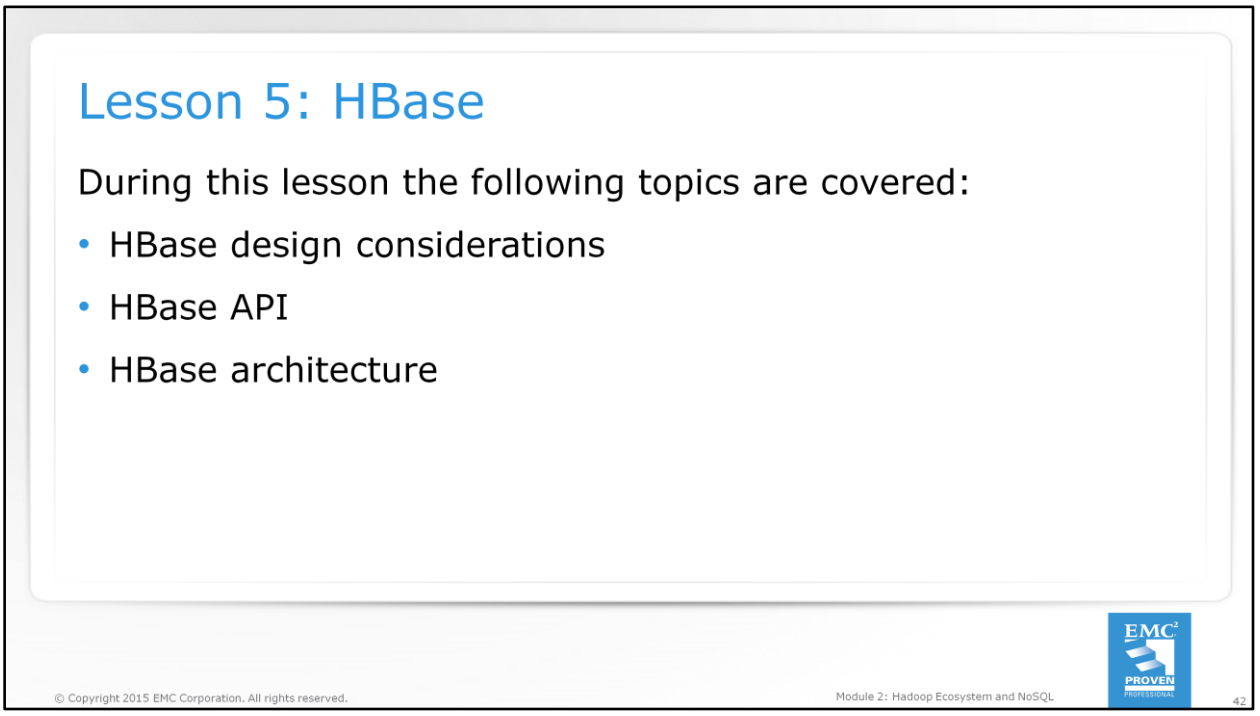
Mark Sweney, "World Cup final breaks Facebook and Twitter records," *theguardian.com,* July 14, 2014, http://www.theguardian.com/media/2014/jul/14/world-cup-final-breaks-facebook-and-twitter-records

More information about FlockDB can be found at the following web sites:

http://www.devwebpro.com/exploration-of-nosql-flockdb
https://github.com/twitter/flockdb



This lesson provided an overview of NoSQL and the conditions that led to its interest. Four types of NoSQL data stores were examined: key/value, columnar, document, and graph. The next lesson will provide considerable detail on the design and use of a HBase data store.



This lesson focuses on the design considerations for an HBase implementation to store and access data. Detail is provided on how the user/client interfaces with HBase tables and the HBase architecture that enables the real-time reads and writes in a Hadoop environment.

Overview of HBase

Apache HBase is a data store
- Distributed
- Versioned
- Defined on top of HDFS

- Based on Google's Bigtable paper
  - "a sparse, distributed, persistent multidimensional sorted map"
- HBase should not be considered a database
  - At least not in the relational database sense

HBase is a distributed, versioned data store that is defined on top of HDFS. Distributed refers to the cluster of nodes running Hadoop/HDFS that provides scalability as data and performance needs grow. Versioned refers to the fact that each row includes a timestamp to easily identify the most current record as well as previous records.

HBase is based on Google's Bigtable paper. Bigtable was designed to provide real-time reads and writes of web-crawler results. Bigtable, refers to just one table with possibly billions of rows and millions of columns. Thus, it is a little misleading to consider HBase a database, at least in the context of a relational database.

Reference:

"Welcome to Apache HBase," *Apache HBase, accessed November 5, 2014,* http://hbase.apache.org

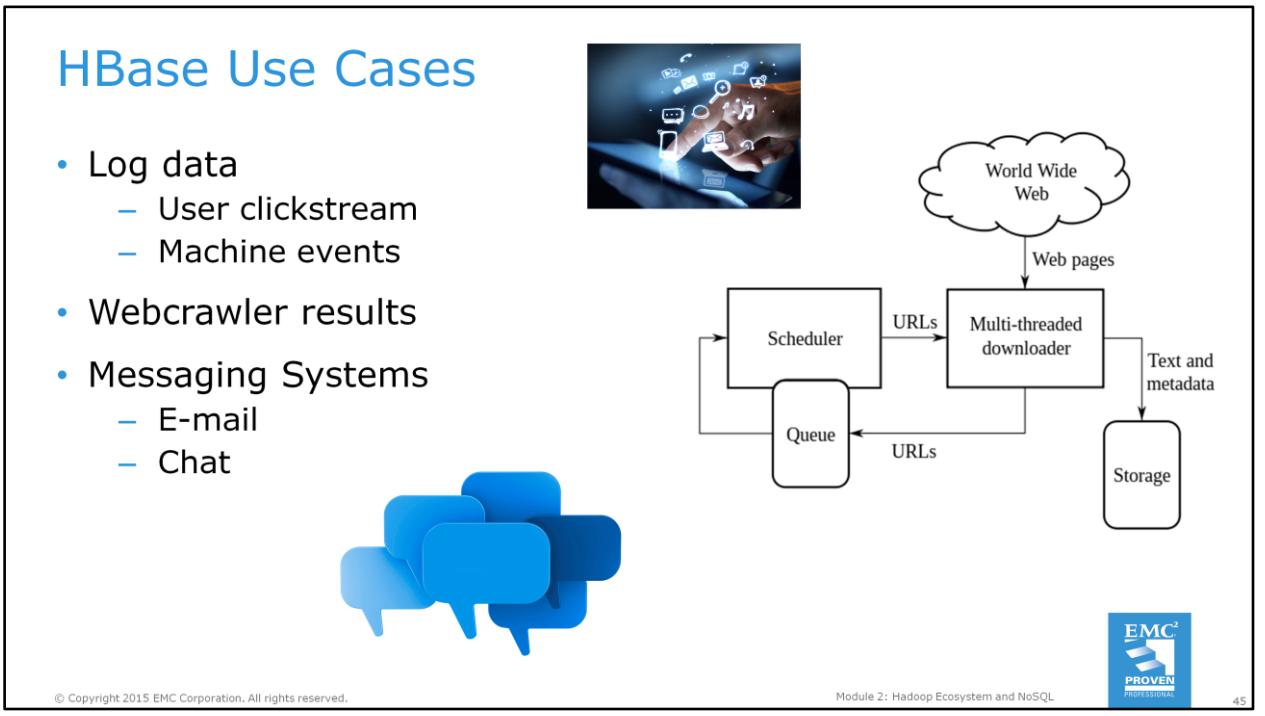
More information about Google's Bigtable can be found at the following web site:
http://research.google.com/archive/bigtable.html

When to Use HBase

- Need to store petabytes to exabytes of data
  - Billions of rows and millions of columns
- Require real-time read/writes
  - Recall Pig and Hive are batch-oriented
  - Only a small subset of the data is required in real-time
- Add new columns in real-time
- Clearly understand how the stored data will be retrieved
  - Will define the key contents of the key/value pair
  - Enable the real-time reads
- Not for general purpose querying

To properly use HBase, it is necessary to understand how the stored data will be used. For example, suppose HBase is to be used as a repository for user e-mails. In this case, the e-mails would be primarily displayed for a particular user at a time. However, there may be secondary considerations such as the e-mails should be retrieved in chronological order with the most recent e-mails displayed first.

As presented in the forthcoming example of HBase's key/value model, the content of the key is the critical aspect of real-time retrieving of data stored in an HBase table. The understanding of how the data needs to be retrieved from HBase will define the contents of the key.



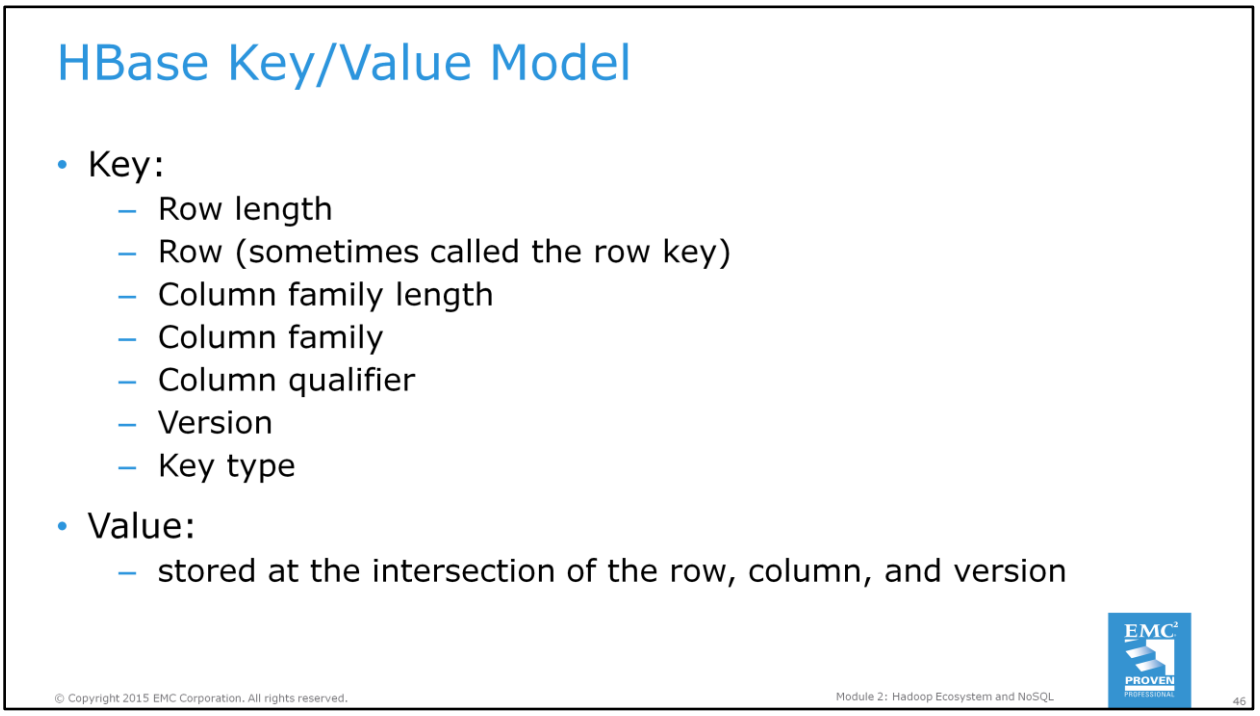
For clickstream data, elements such as user id and session id could be stored along with the page views and a timestamp. For machine event logging, items such as machine name, timestamp, event id, and values, such as error codes, for the event could be stored in an HBase table.

An HBase table could be used to store web crawler results; for example, which web pages provide links to other pages along with the contents of the web pages. This is the example provided in Google's Bigtable paper. For messaging systems, HBase could be used to store users' received and sent messages.

The Apache web site provides some useful illustrations for various use cases as well as on how to build the schema design from normalized to de-normalized.

http://hbase.apache.org/book/schema.casestudies.html

Next, the schema design considerations will be examined in the following examination of HBase's key/value model.



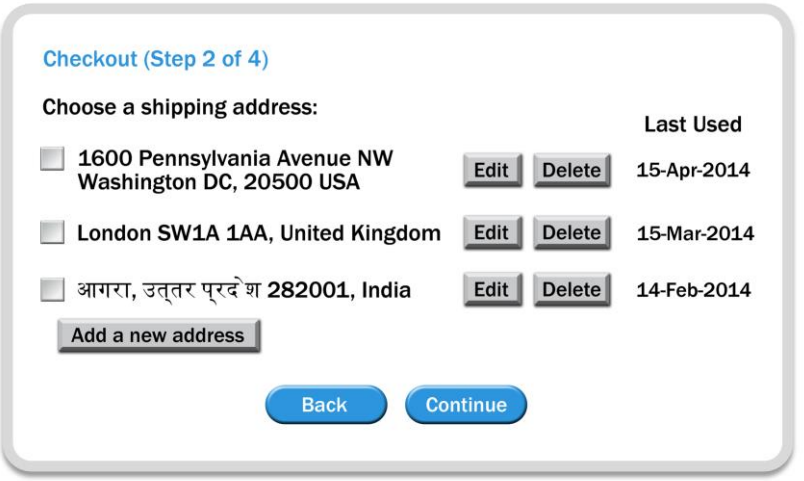
This portion of the lesson covers the meaning of the row and columns in HBase. Version and key type are addressed much later in this lesson; however, a brief overview of these terms is presented here.

A cell is the intersection of a row and a column in a table. The version, sometimes called the timestamp, provides the ability to maintain different values for a cell's contents in HBase.

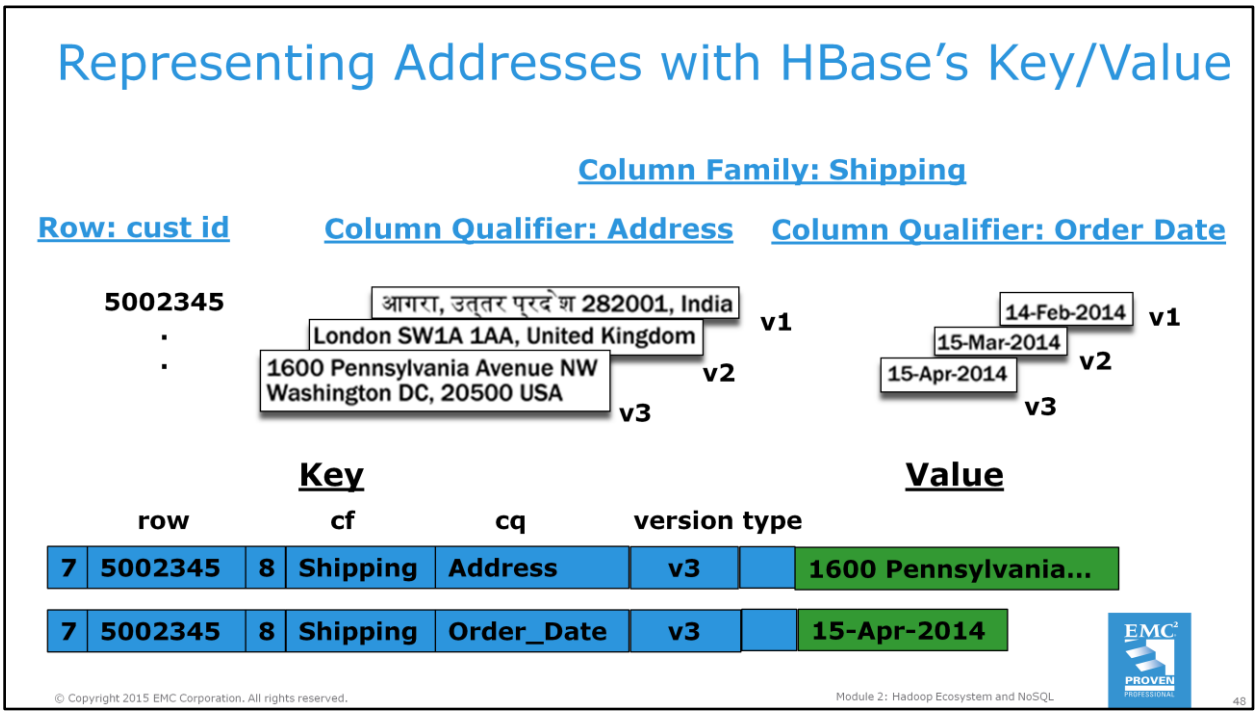
Key type is used to identify whether a particular key corresponds to a write operation to the HBase table or a delete operation from the table. Technically, a delete from an HBase table is accomplished with a write to the table. The key type indicates the purpose of the write.

Using HBase to Provide Customer Addresses

- Scenario: Checkout at an online retailer
- Question: How to use HBase to store the addresses?



Consider the case where an online retailer wants to use HBase to maintain the last three shipping addresses that each customer used at its web site and to serve up these addresses during the checkout process. The user can add a new shipping address, select an existing address, or edit a prior address. In this scenario, the question is how to use HBase to store and maintain customer's shipping addresses.



The shipping address scenario can be implemented in HBase as follows:

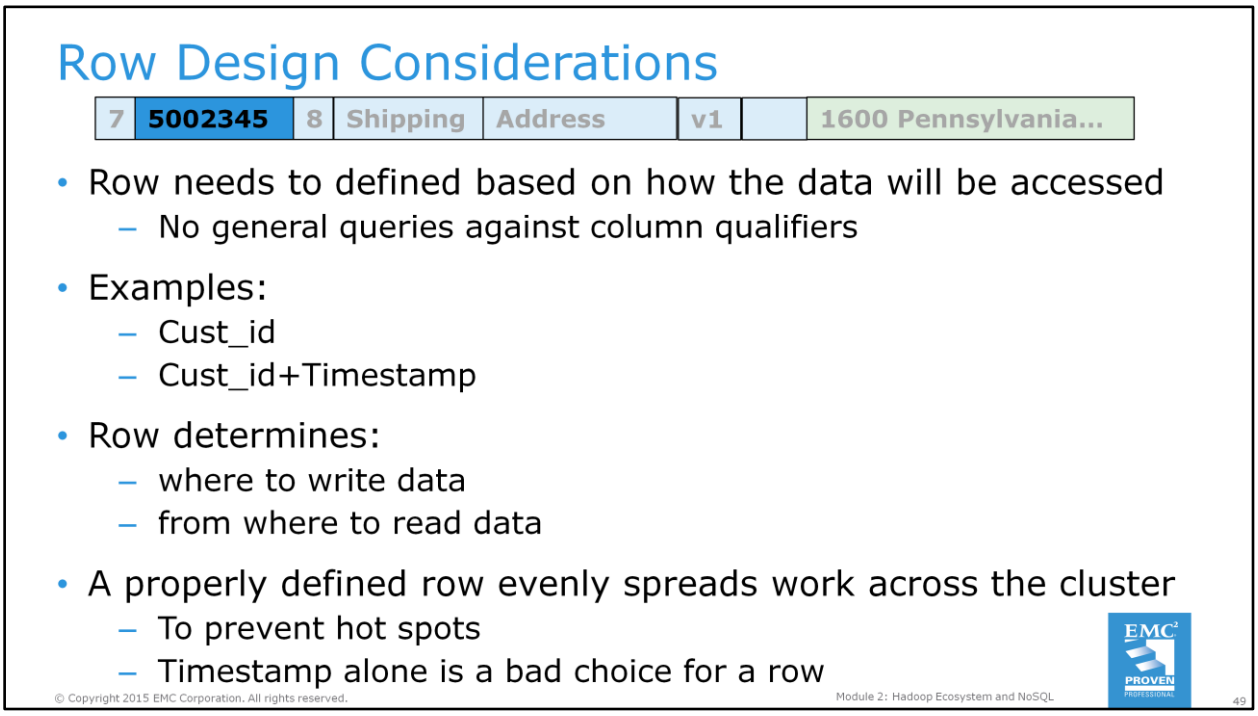
1. Under the column family, Shipping, two column qualifiers are created to store each customer's shipping address and the corresponding order date for that shipping address.

2. The attributes for the most recent order are assigned the largest version designation; v3 in this example. The previous values were assigned version v2 and v1 at the time of the orders.

3. The customer's identification (cust_id) number is used for the row. Thus, as the customer checks out, the recent shipping addresses can be displayed to streamline the process.

The key/values for the most recent order are illustrated on this slide. By default, the type is not entered since it is a simple write to HBase. Writes constitute the majority of the entries in an HBase table. Note: the lengths of the row and column family (cf), 7 and 8 respectively, are included in the key, but not labeled.

Next, the design considerations of the critical elements of the key are examined.



The row needs to be properly designed so that data can be easily written and quickly retrieved. Since the shipping addresses will most certainly be retrieved by the customer identification number (cust_id), it makes sense to set the row to equal the cust_id. However, concatenating a timestamp to the cust_id will ensure that the data be easily retrieved in chronological order.

The row can be as complex as necessary. It is always necessary for the user or the application to precisely understand how the row is designed so that the embedded information can be extracted from the row.

As will be seen later, the data is distributed across the cluster based on the row. Thus, solely using a sequential row designation, such as a timestamp, will create hot spots where one server is performing all of the writes.

Apart from the standard connotation of the term, row, it is important to keep in mind that in HBase, row refers to a portion of a key's contents in the key/value pair. Next, HBase's concept of a column will be presented.

Column Design Considerations

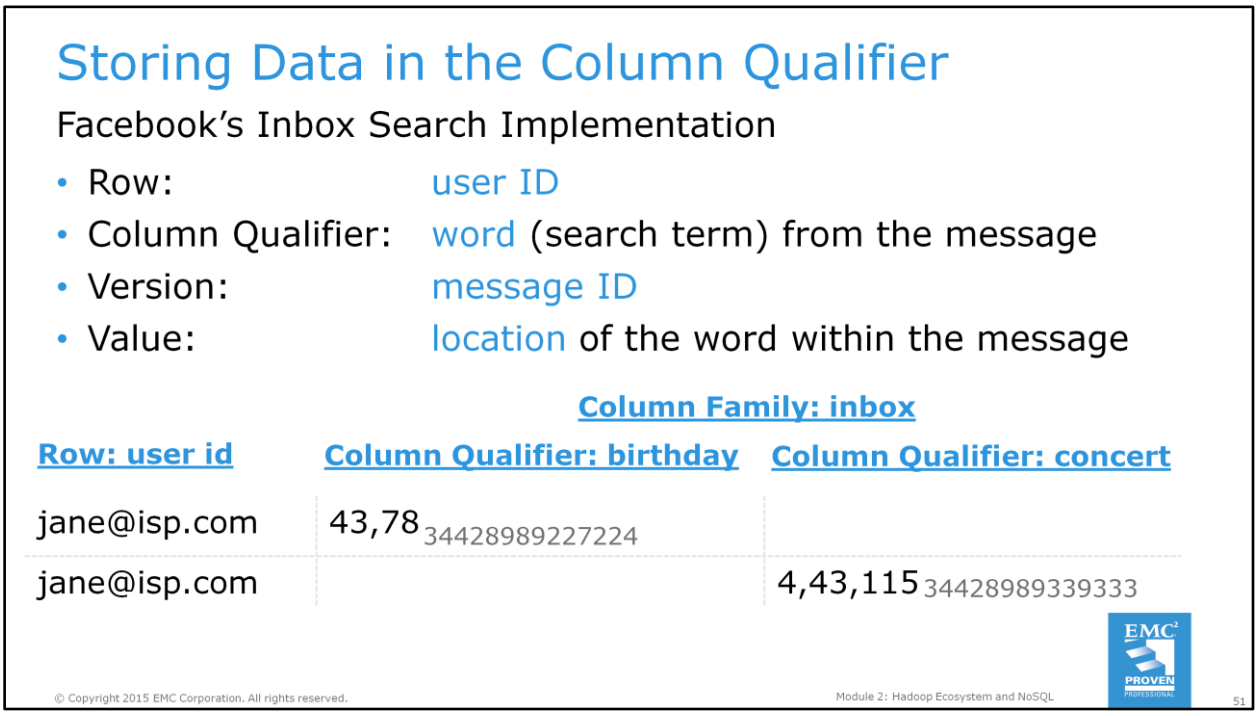
7	5002345	8	Shipping	Address	v1		1600 Pennsylvania…

- A column is a combination of column family and column qualifier
  - Denoted *cf:cq*
  - Column family
    - Groups related column qualifiers together (e.g. Shipping)
      - Reads and writes should involve only one column family at a time
    - Use short column family names
  - Column qualifier
    - May label the contents of value (e.g. Address)
    - Or is actually the data to be stored (e.g. www.emc.com)
      - **!!!! This feature is how millions of columns are possible !!!!**
- Again, don't plan to search directly on column contents
  - e.g. do not plan to access data where cf1:name = 'Cutting'

A column consists of two parts: the column family and the column qualifier. The column family is used to group several column qualifiers together. The column family helps to define the HDFS files where the key/values will be physically stored. Thus, if it is likely that reads or writes may occur across multiple column families, then it is advisable to use just one column family to avoid access to multiple HDFS files.

For customer data, a column qualifier may be a typical label such as name or address; however, the real strength of HBase is that a column qualifier can be used to store data. For example, the column qualifier could be a web address such as www.emc.com. This approach is how it is possible for HBase to accommodate millions of columns as well as to add new columns whenever necessary.

Recall that data will be accessed using the row. So, don't plan to employ SQL like approaches, such as using a WHERE clause, to query all of the data that matches a certain criterion and expect real-time performance.



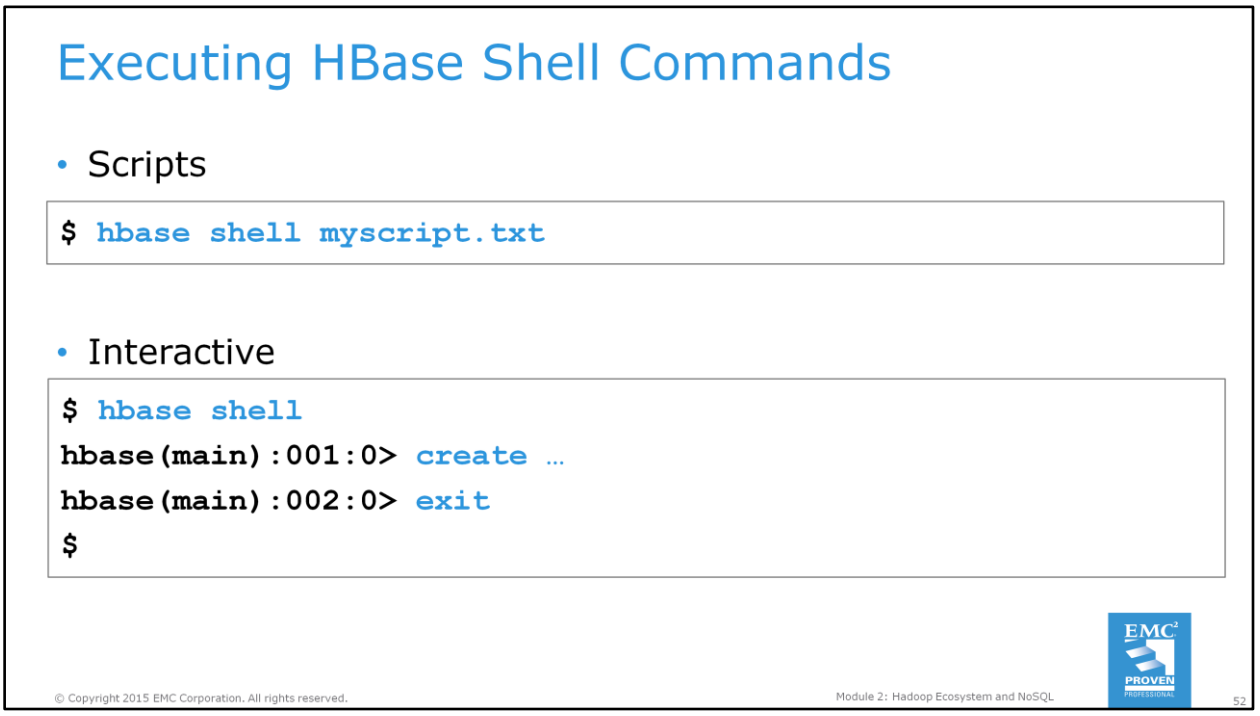
To enable search functionality in a user's inbox, HBase was used at Facebook to build a table to store the words that appear in each user's inbox message. A row denoted a particular user's inbox.

For a column family, inbox, the column qualifiers are words, such as birthday or concert, that appear in the message text. If one of these words appeared in a message, a version would be assigned to the corresponding message ID and the value would be set to the location of the word in the message. If the word appears multiple times in a single message, the locations could be separated by commas. This use of HBase illustrates how schema on write can be implemented; the column qualifiers are data of interest, not just labels.

With this background of HBase and its key/value modeling approach, the next part of the lesson will address how to create a table, read and write to a table, and how to delete data from a table.

Ref: George Lars (2011). *HBase: The Definitive Guide.* O'Reilly Media, pg. 374.

More details on Facebook's requirements of high write throughputs and low latency reads can be found at:
https://www.facebook.com/UsingHbase
http://sites.computer.org/debull/A12june/facebook.pdf
https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919



HBase shell commands can be executed individually from the interactive environment or as a group of commands from a script. In production, these commands are typically embedded into the application that is accepting inputs and providing outputs. This lesson assumes that the shell commands are being entered at the HBase prompt, which has been omitted for readability purposes. As addressed in the next slide, the create command is used to create an HBase table. The exit command is used to return to the system command prompt.

The next few slides cover several common HBase shell commands. An extensive list of the HBase shell commands and their descriptions can be found at: http://wiki.apache.org/hadoop/Hbase/Shell or by typing help at the HBase prompt.

Creating an HBase table (Create)

```
create 'my_table', 'cf1', 'cf2',
                        {SPLITS =>['250000','500000','750000']}
```

- Creates a table with two column families, cf1 and cf2
  - Note: no column qualifiers
- SPLITS defines how data will be distributed across the cluster
  - Based on row portion of the key
  - Four regions are created in this example

Shown here is the command to create an HBase table. Two column families, cf1 and cf2, are defined in the table. The SPLITS option specifies how the table will be divided based on the row portion of the key. In this example, the table is split into four parts, called **regions**. Rows less than 250000 are added to the first region; rows from 250000 to less than 500000 are added to the second region, and likewise for the remaining splits. These splits provide the primary mechanism for achieving the real-time read and write access.

In this example, my_table is initially split into four regions, each on its own worker node in the Hadoop cluster. Thus, as the table size increases or the user load increases, additional worker nodes and region splits can be added to scale the cluster appropriately. The reads and writes are based on the contents of the row. HBase can quickly determine the appropriate region to direct a read or write command.

Each region is managed by a **RegionServer**. The HBase API communicates with the HBase master to determine which region server will read or write the data of interest. The coordination of the region servers is managed by Apache Zookeeper, "a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services." Ref: http://zookeeper.apache.org

Each region consists of a **memstore** which stores recent write operations, the **WAL** (write ahead log) which persists the memstore contents to a local disk outside of HDFS, and **HFiles** which are the part of the table persisted to HDFS files.

Adding Data to a Table (Put)

```
put 'my_table', '000700', 'cf1:cq1', 'data1'
put 'my_table', '000700', 'cf1:cq2', 'data2'
put 'my_table', '000700', 'cf2:cq3', 'data3'
```

- Put command
  - Table: mytable
  - Row: 000700
  - Column: cf1:cq1, cf1:cq2, and cf2:cq3
  - Value: data1, data2, and data3

Three put commands are executed to add three values to three columns for row = ' 000700 '.

A write operation (put) is performed in a series of steps to persist the transaction and to make the data available in real-time for a subsequent get or scan operation. Based on the provided row included in the put statement, HBase determines which region will hold the data and which RegionServer is responsible for that region. The RegionServer will log the transaction to the Write-Ahead Log (WAL) to a local disk. The transaction is then written to a the memstore residing in RAM. If there is a power outage, for example, and the RAM contents are lost, the region can be recovered by writing the contents of the WAL back to RAM.

Retrieving Data from a Table (Get)

```
get 'my_table', '000700', 'cf2:cq3'
COLUMN          CELL
 cf2:cq3        timestamp=1412000306594, value=data3
```

- Only returns data for a single row
- Timestamp
  - Version of the row and column value
  - Number of milliseconds since January 1, 1970 (default)

In this example, the value of column, 'cf2:cq3' for row 000700 is retrieved. Earlier, it was mentioned that the version of each value is stored with each row and column. By default, HBase uses the number of milliseconds since January 1, 1970 for the version.

A single get command can retrieve multiple columns for a single row. For example, this get command returns the two columns from column family, cf1.

get 'my_table' , '000700', {COLUMN => ['cf1:cq1', 'cf1:cq2']}

Although the design of the HBase table should avoid the need to pull all column qualifiers from all of the column families, the following get command will do just that.

get 'my_table' , '000700'

Working with Multiple Versions

- Add a 2nd value for a cell

```
put 'my_table', '000700', 'cf2:cq3', 'data4'
```

- Get command only returns the most recent version

```
get 'my_table', '000700', 'cf2:cq3'
COLUMN          CELL
 cf2:cq3        timestamp=1412000389075, value=data4
```


For the given row and column, a value of 'data3' was already entered into the table. The put statement adds another value for the intersection of this row and column. When multiple versions of a cell exist, by default, the get command only returns the most recent version.

Viewing Multiple Versions (Scan)

- Retrieve last two versions for a row

```
scan 'my_table', {RAW=>true, VERSIONS=>2, STARTROW=>'000700',
                                          ENDROW=>'000701'}
ROW          COLUMN+CELL
 000700      column=cf1:cq1, timestamp=1412000282204, value=data1
 000700      column=cf1:cq2, timestamp=1412000293748, value=data2
 000700      column=cf2:cq3, timestamp=1412000389075, value=data4
 000700      column=cf2:cq3, timestamp=1412000306594, value=data3
```


The scan command returns a block of rows between the specified STARTROW and ENDROW.

The last two lines of the displayed output correspond to the same column, cf2:cq3 and reflect the different values of 'data3' and 'data4' added with the two earlier put commands.

When the client uses the get or scan command to access the contents of an HBase table, the task is assigned to an appropriate RegionServer based on the specified rows. As required, the contents of the appropriate region's memstore and HFiles are processed to return the requested data.

Removing Cells (Delete)

```
delete 'my_table', '000700', 'cf2:cq3', 1412000306594

scan 'my_table', {RAW=>true, VERSIONS=>2, STARTROW=>'000700',
                                          ENDROW=>'000701'}
ROW          COLUMN+CELL
 000700      column=cf1:cq1, timestamp=1412000282204, value=data1
 000700      column=cf1:cq2, timestamp=1412000293748, value=data2
 000700      column=cf2:cq3, timestamp=1412000389075, value=data4
 000700      column=cf2:cq3, timestamp=1412000306594, type=DeleteColumn
 000700      column=cf2:cq3, timestamp=1412000306594, value=data3
```


Key type is used to identify whether a particular key corresponds to a write operation to the HBase table or a delete operation from the table. Technically, a delete from an HBase table is accomplished with a write to the table. The key type indicates the purpose of the write. For deletes, a special type of entry known as a **tombstone marker** is written to the table to indicate that all cell contents with versions equal to or older than the specified timestamp should be deleted for the corresponding row and column family:column qualifier.

In this example, the delete command identifies, for removal, the contents of the cell for row '000700' and column 'cf2:cq3' at timestamp 1412000306594. Running a scan reveals that the deleted entry still remains, but a tombstone marker is added to the HBase table. This tombstone marker, as well as the deleted entry, will remain in the HBase table until some system maintenance activity is performed such as a **major compaction**. This type of activity is addressed in the upcoming material on region maintenance.

Region Maintenance

- Minor compaction
  - Memstore is dumped to an HFile in HDFS
  - WAL is emptied
- Major compaction
  - Merges and sorts the smaller HFiles into one HFile
  - Physically removes data marked for deletion
  - Removes unwanted older versions
- Balancing
  - Regions can be split among several region servers
  - Manually run when additional nodes are added to the cluster

As the size of the memstore approaches the HDFS block size, a minor compaction is performed to persist the data to HDFS as an HFile. This operation also allows for the WAL entries to be deleted. Over time, more and more of these HFiles are created for a region. A major compaction merges and sorts the small HFiles into one large HFile, as well as removes any deleted entries and the tombstone markers. Also, conditions such as maximum version settings are enforced in order to removed older, unwanted versions.

Region balancing can be accomplished whenever new nodes are added to the cluster or when some regions have relatively higher transaction volumes.

These maintenance activities can be initiated automatically or manually. To prevent overtaxing the cluster, it is best to stagger these operations whenever possible.

Comparing Pig, Hive, and HBase

Aspect	Pig	Hive	HBase
Access	Batch	Batch	Real-time
Storage	HDFS	HDFS	HDFS and RAM
Programming Style	Script	SQL	CRUD
Key "Item"	Relations	Tables	A table

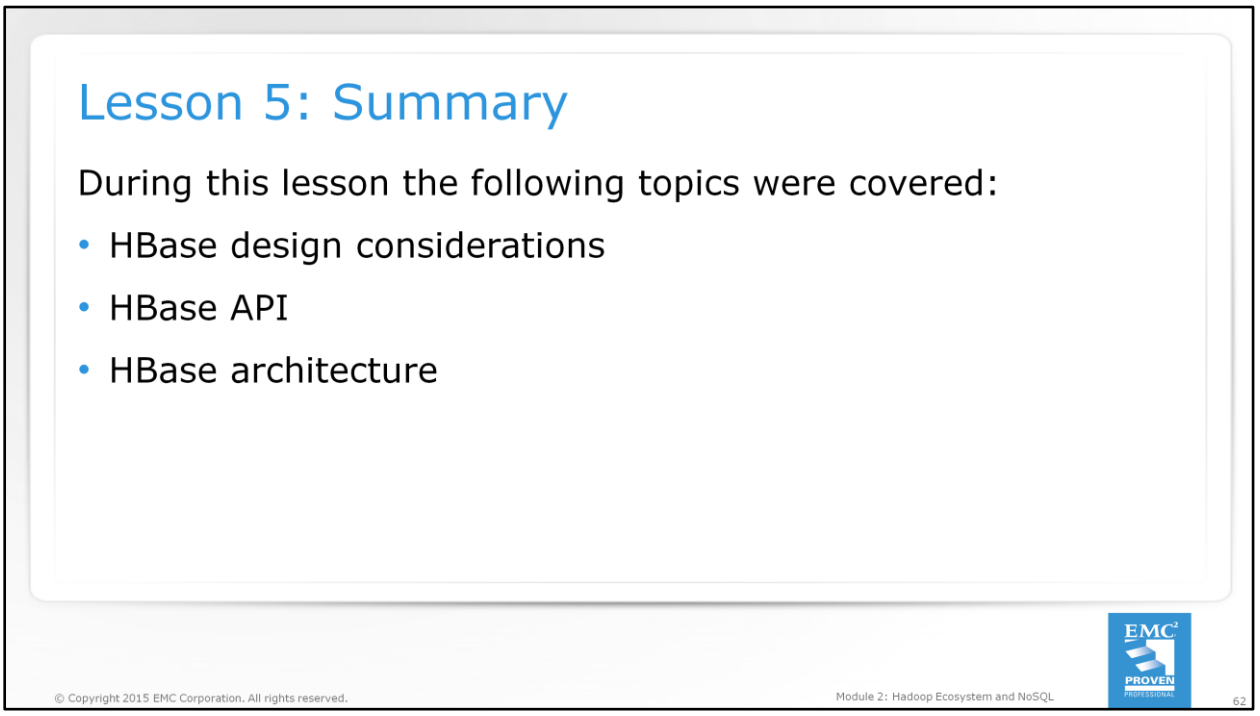
Pig and Hive perform batch processing of HDFS files. Although HBase does store data in HDFS, the ability to add new data into RAM allows HBase to achieve real-time access to data. Pig and Hive are excellent general purpose tools for processing HDFS files. While Hive expects its data to conform nicely to a typical relational database table, Pig is more flexible in dealing with unstructured data. Pig Latin is similar to many scripting language styles. Hive will be the choice for those developers comfortable with SQL.

HBase is typically concerned with four types of transactions: - create, read, update, and delete (CRUD). As seen earlier, the updates are tracked with the use of versions or timestamps.

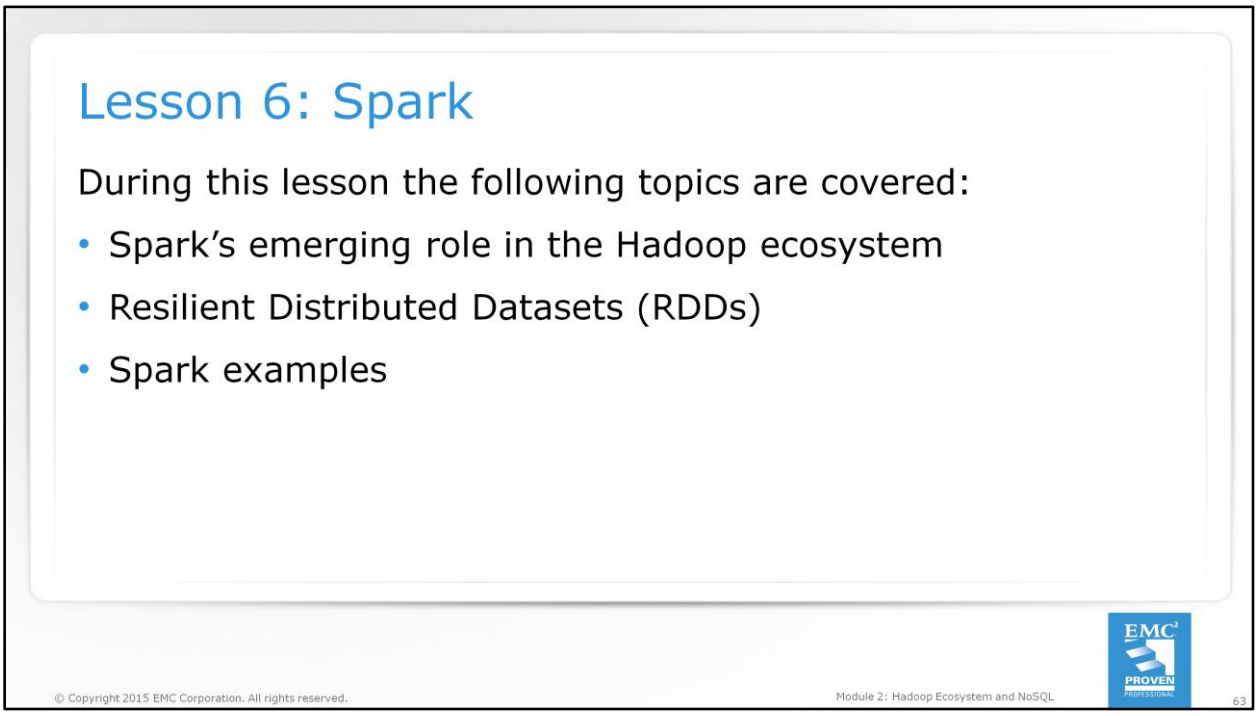
HBase Lab

- Create and populate an HBase table
- Add records with user provided timestamp
- Add columns to a column family
- Delete table cells
- Write an extract to a local file

An overview of the HBase lab is provided. The lab covers the use of the HBase shell commands.

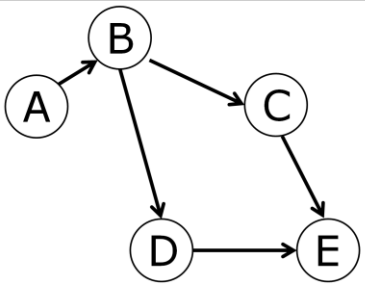


This lesson covered the design considerations for using HBase, in particular, the importance of a properly structured row and the technique of storing data as a column qualifier. The basic HBase shell commands illustrated the use of the HBase API to create an HBase table and to write, read, and delete data from the table. Also, this lesson described how the HBase architecture enables the ability to perform real-time reads and writes.



This lesson focuses on the emerging role of Spark in the Hadoop Ecosystem to provide a more flexible and efficient framework to MapReduce.

Overview of Apache Spark



- A Directed Acyclic Graph (DAG) engine
  - Runs on top of HDFS
  - Utilizes in-memory datasets
    - Called Resilient Distributed Dataset (RDD)
- Java, Scala, or Python can be used to write a Spark application
- Includes several libraries for
  - Graph analysis
  - Machine Learning
  - SQL
  - Streaming data
- Garnered interest in 2014

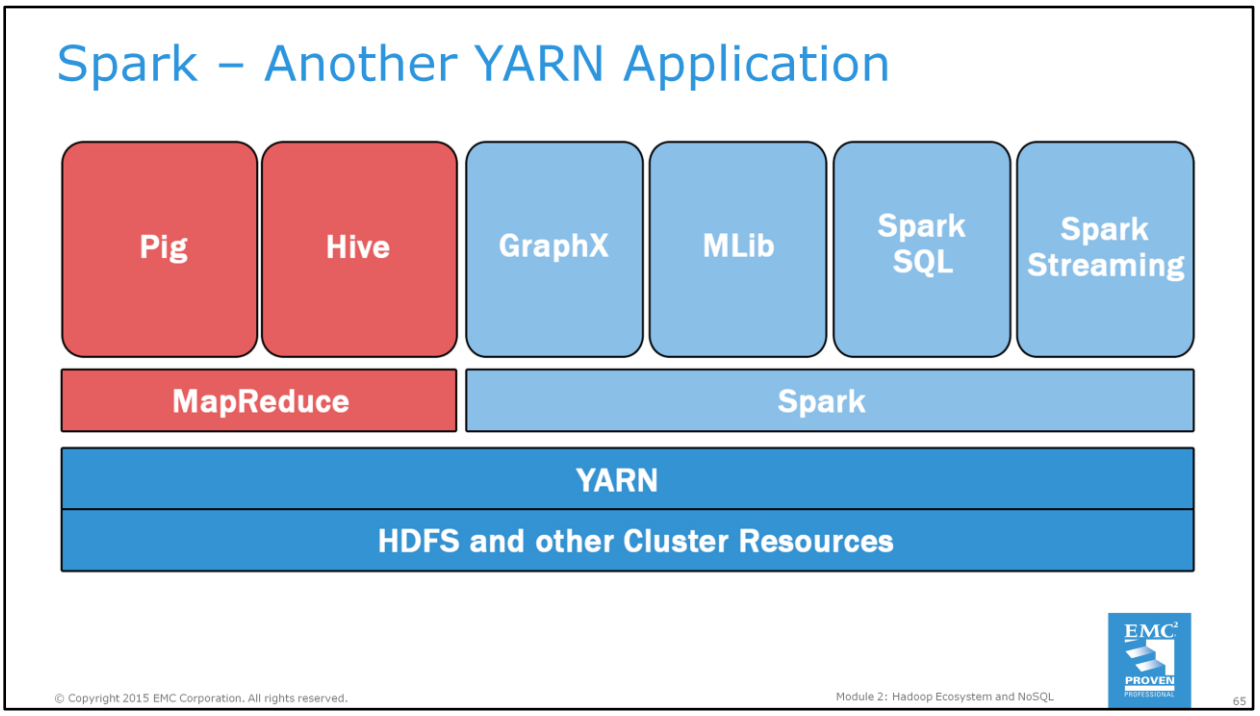
Apache Spark provides a Directed Acyclic Graph (DAG) engine that can efficiently process fairly complex execution plans across a distributed cluster. Directed means that there is a sequence in the performed operations and acyclic means that there are no loops, or cycles between the executed tasks. In Spark's case, a graph represents the order and conditions that the various tasks are executed. A simple DAG is the MapReduce framework, Map ➔ Reduce.

Written in Java, Scala, or Python, a Spark program gains its efficiency by utilizing in-memory datasets called Resilient Distributed Datasets (RDDs), whenever possible. In a series of MapReduce jobs, the reduce output is written to disk in order to be processed by a subsequent Map task. The disk I/O and spinning up the various JVMs can lead to a performance bottleneck.

The Spark distribution also includes several libraries that simplify the writing of Spark programs. These libraries cover graph analysis, machine learning, SQL and processing streaming data.

While MapReduce is ideal for bulk processing large volumes of data, it is not the ideal framework for all applications. This statement is not a ground breaking revelation. Earlier, it was seen how difficult it is perform a JOIN operation with MapReduce. Also, the development of YARN began years ago to enable other frameworks, such as DAGs, to run on a Hadoop cluster.

Of course, Spark has its own set of issues and difficulties with which to contend. For example, how to recover from a lost or corrupted RDD. Projects such as Tachyon (http://tachyon-project.org/), promises reliable in-memory file sharing across frameworks.



As discussed in Module 1, MapReduce is an application that runs on YARN. Similarly, SPARK runs on YARN.

Spark is ideal for those jobs that may require repeated processing of a given dataset or the intermediate output of a previous task. A series of MapReduce jobs can be strung together to perform a SQL-like join. However, by keeping intermediate output in memory, Spark can enable much faster processing times.

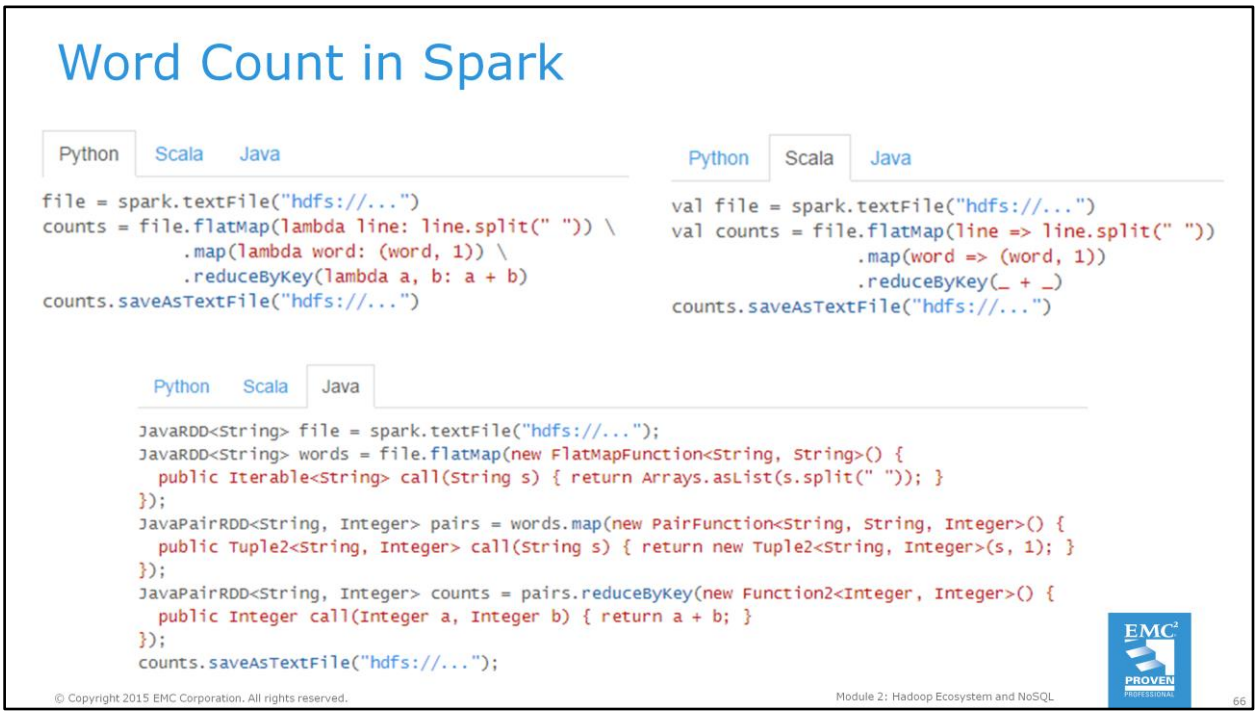
Four projects being developed with Spark and included in the Spark download and installation are:

GraphX: provides tools and algorithms to perform graph analyses

MLib: machine learning library which includes k-means, logistic regression, and naïve Bayes

Spark SQL: apply SQL queries to structured data; accepts unmodified Hive queries

Spark Streaming: enables streaming applications such as processing Twitter feeds



On this slide, the dark blue text indicates Spark operations for the red text that are literals passed to the cluster. Note: the map and reduceByKey functionality are provided by Spark.

Code examples are from "Spark Examples," *spark.apache.org,* accessed on November 5, 2014, http://spark.apache.org/examples.html.

Using Spark to Overcome Small File Issue

- MapReduce does not handle small files well
  - A JVM needs to be opened for each file
  - Several minutes of overhead with each JVM
  - Hundreds of thousands of files (e.g. e-mails) are a problem
- Spark Solution
  - Files can be pre-processed using Spark
    - Parse out the relevant items from each file
    - Build one large file better suited for MapReduce
  - Output can be further processed by Spark or MapReduce jobs

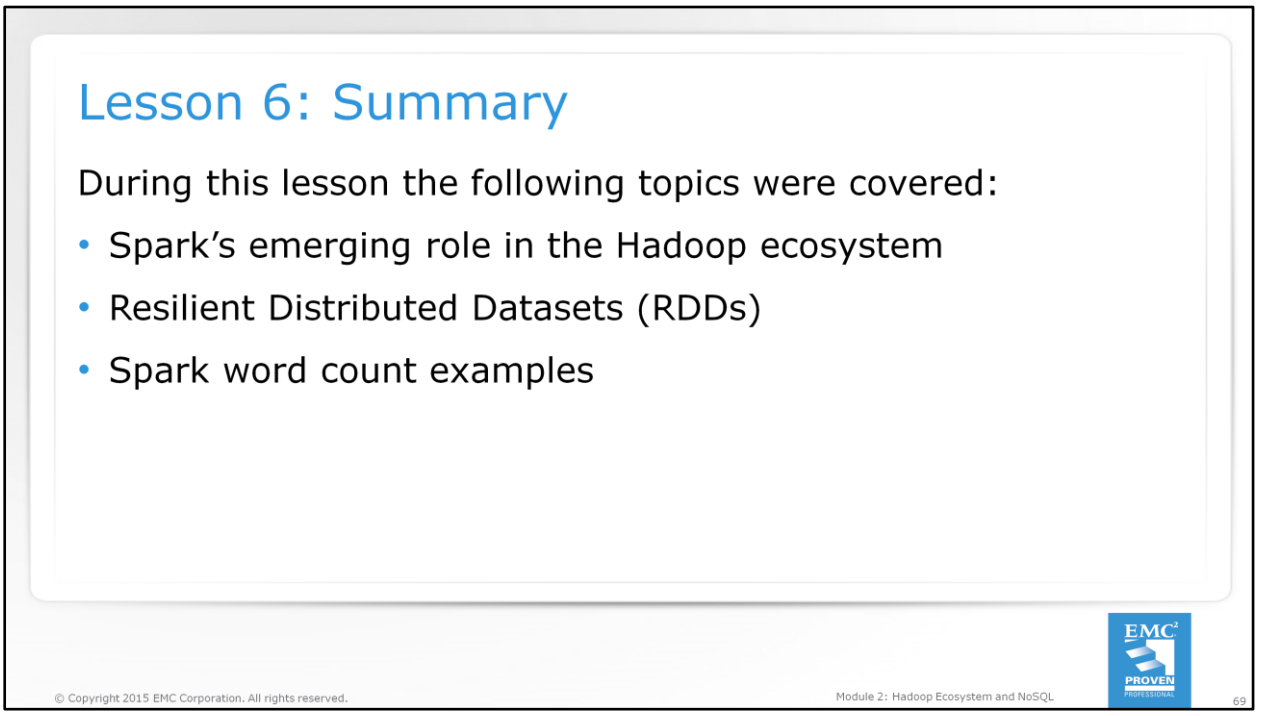
By default, HDFS files are stored in 128 MB blocks. The relatively large block size is chosen so that each map task, in a MapReduce job, will have a sufficient amount of data to process to justify the creation of a JVM to run the task. However, if there are a lot of small files, the overhead of the numerous JVM creations leads to what is known as the small file issue in Hadoop.

In such cases, Spark can be used to construct larger files from the smaller files in preparation for future MapReduce processing. Alternatively, if the specific requirements are known in advance, Spark can be used to process the small files and extract the required data elements for future processing. The latter approach will be seen in the Spark lab which will prep several small e-mail files for analysis during the final lab.

Spark Lab

- Read the final lab problem scenario
- Examine the provided Spark code
- Prepare the Spark code for execution
- Run the Spark job to prep the data for text analysis

An overview of the Spark lab is provided. This lab prepares the data set for use in the final lab. Throughout the remaining modules in this course, keep the final lab scenario in mind as the content is presented. It is important to identify possible analytical approaches to address the problems posed in the final lab scenario.

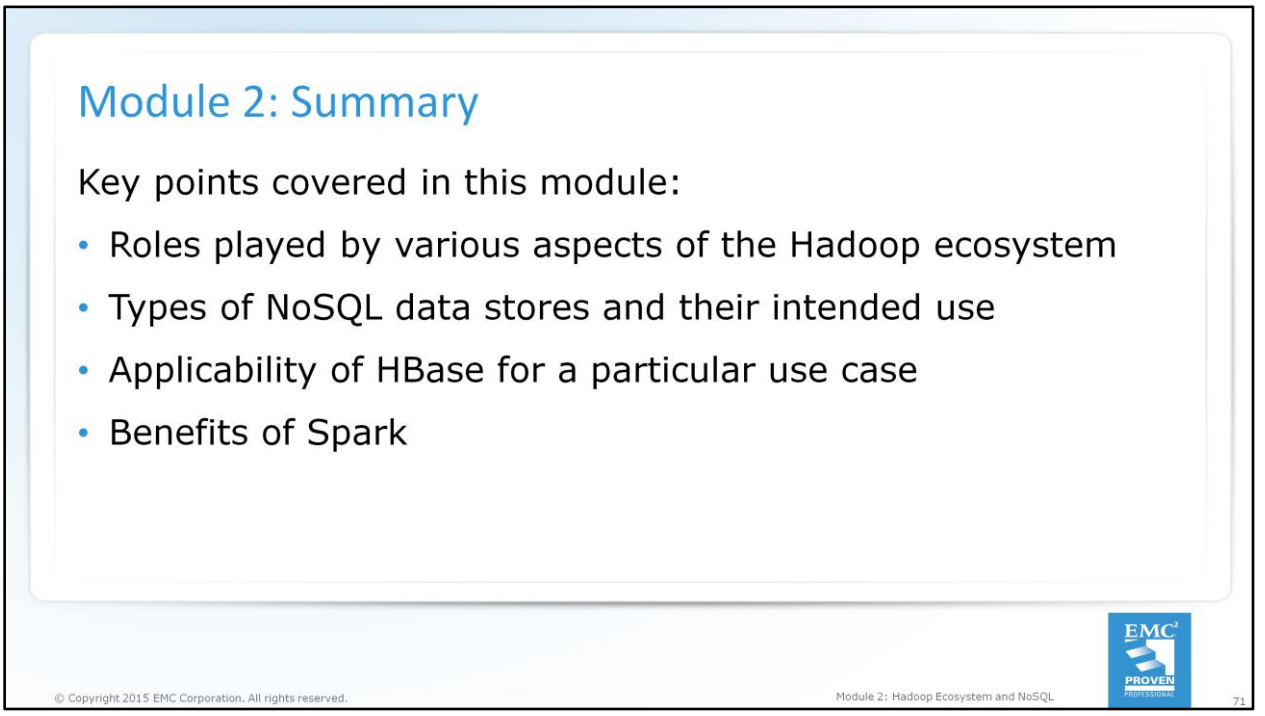


This lesson covered Apache Spark which provides a DAG execution engine. DAG is one of the new frameworks possible on a Hadoop cluster with the release of YARN. Spark gains much of its performance from the use of resilient distributed datasets (RDDs) which keep data in memory instead of writing the data to disk. Also, word count Spark examples using Java, Python, and Scala were presented.

Check Your Knowledge

1. List members of the Hadoop ecosystem and their functions.
2. Which tools are useful for batch processing and why?
3. Which tools are useful for real-time processing and why?
4. What are four types on NoSQL data stores?
5. Before designing an application utilizing HBase, what must be clearly known?
6. Describe a Directed Acyclic Graph (DAG).

Write the answers here.



This module covered several aspects of the Hadoop ecosystem. Apache Pig and Hive were presented as the means to run MapReduce jobs without the need to explicitly code a MapReduce job. After a general discussion of NoSQL data stores, the module focused on the use of Apache HBase to provide real-time reads and writes in a Hadoop cluster. The module ended with a lesson on Apache Spark, a project which began to garner considerable interest in 2014.

This slide intentionally left blank.