



HDP Developer: Apache Pig and Hive

A Hortonworks University
Hadoop Training Course

Title: HDP Developer: Apache Pig and Hive

Version: Revision 3

Date: February 6, 2014

Copyright © 2013-2014 Hortonworks Inc. All rights reserved.

Hadoop and the Hadoop elephant logo are trademarks of the Apache Software Foundation.

The contents of this course and all its related materials, including lab exercises and files, are Copyright © Hortonworks Inc 2014.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Hortonworks Inc. All rights reserved.

Table of Contents

Table of Contents	3
Preface.....	8
Unit 1: Understanding Hadoop	9
Big Data Facts	10
The Three V's of Big Data.....	12
6 Key Hadoop Data Types.....	14
Sentiment Use Case.....	15
Geolocation Use Case	16
What is Hadoop?	17
Relational Databases vs. Hadoop.....	19
What is Hadoop 2.0?.....	20
What's New in Hadoop 2.0?.....	21
The Hadoop Ecosystem	22
Who is Hortonworks?	24
The Hortonworks Data Platform (HDP)	25
The Path to ROI	27
Lab 1.1: Start the HDP 2.0 Virtual Machine	30
Unit 2: The Hadoop Distributed File System (HDFS).....	35
What is HDFS?.....	36
Hadoop vs. RDBMS.....	38
HDFS Components.....	40
Understanding Block Storage	42
Demonstration: Understanding Block Storage	44
The NameNode	48
The DataNodes	50
DataNode Failure.....	52
HDFS Commands	54
Examples of HDFS Commands.....	56
HDFS File Permissions.....	57
Lab 2.1: Using HDFS Commands	60
Unit 3: Inputting Data into HDFS.....	67
Options for Data Input.....	68
The Hadoop Client.....	69
WebHDFS.....	70
Overview of Flume	72
A Flume Example	74
Overview of Sqoop	76
The Sqoop Import Tool.....	78
Importing a Table.....	80
Importing Specific Columns.....	82
Importing from a Query	84
The Sqoop Export Tool	86
Exporting to a Table	88

Lab 3.1: Importing RDBMS Data into HDFS	90
Lab 3.2: Exporting HDFS Data to a RDBMS.....	95
Unit 4: The MapReduce Framework	99
Overview of MapReduce	100
Understanding MapReduce.....	102
The Key/Value Pairs of MapReduce.....	105
WordCount in MapReduce	106
Demonstration: Understanding MapReduce	107
The Map Phase	109
The Reduce Phase	111
Lab 4.1: Running a MapReduce Job	114
Unit 5: Introduction to Pig.....	117
What is Pig?	118
Pig Latin.....	120
The Grunt Shell	121
Demonstration: Understanding Pig	122
Pig Latin Relation Names.....	127
Pig Latin Field Names.....	128
Pig Data Types	130
Pig Complex Types	131
Defining a Schema	133
Lab 5.1: Getting Started with Pig.....	135
The GROUP Operator	139
GROUP ALL.....	141
Relations without a Schema.....	142
The FOREACH GENERATE Operator	144
Specifying Ranges in FOREACH.....	146
Field Names in a FOREACH	148
FOREACH with Groups	150
The FILTER Operator	152
The LIMIT Operator.....	154
Lab 5.2: Exploring Data with Pig	156
Unit 6: Advanced Pig Programming	163
The ORDER BY Operator	164
The CASE Operator.....	166
Parameter Substitution.....	167
The DISTINCT Operator	169
Using PARALLEL	171
The FLATTEN Operator	173
Lab 6.1: Splitting a Dataset.....	175
Nested FOREACH.....	179
Performing an Inner Join.....	181
Performing an Outer Join	183
Replicated Joins	185
The COGROUP Operator	186
Pig User-Defined Functions	188
A UDF Example	189
Invoking a UDF	190

Tips for Optimizing Pig Scripts	191
Lab 6.2: Joining Datasets.....	192
Lab 6.3: Preparing Data for Hive	197
Overview of the DataFu Library	199
Computing Quantiles.....	200
Demonstration: Computing PageRank.....	202
Lab 6.4: Analyzing Clickstream Data	206
Lab 6.5: Analyzing Stock Market Data using Quantiles	212
Unit 7: Hive Programming.....	215
What is Hive?	216
Comparing Hive to SQL.....	218
Hive Architecture	220
Submitting Hive Queries	222
Defining a Hive-Managed Table	224
Defining an External Table	225
Defining a Table LOCATION	225
Loading Data into a Hive Table.....	226
Performing Queries	227
Lab 7.1: Understanding Hive Tables	228
Hive Partitions	234
Hive Buckets	236
Skewed Tables.....	237
Demonstration: Understanding Partitions and Skew.....	238
Sorting Data.....	241
Using Distribute By	243
Storing Results to a File	245
Specifying MapReduce Properties.....	246
Lab 7.2: Analyzing Big Data with Hive	247
Lab 7.3: Understanding MapReduce in Hive	253
Hive Join Strategies.....	257
Shuffle Joins.....	258
Map (Broadcast) Joins	259
Sort-Merge-Bucket (SMB) Joins	260
Invoking a Hive UDF	262
Computing ngrams in Hive	263
Demonstration: Computing ngrams	265
Lab 7.4: Joining Datasets in Hive	269
Lab 7.5: Computing ngrams of Emails in Avro Format	273
Unit 8: Using HCatalog.....	279
What is HCatalog?.....	280
HCatalog in the Ecosystem	282
Defining a New Schema	283
Using HCatLoader with Pig.....	284
Using HCatStorer with Pig	285
Lab 8.1: Using HCatalog with Pig	287
Unit 9: Advanced Hive Programming	291
Performing a Multi Table/File Insert	292
Understanding Views	294

Defining Views.....	296
Using Views	298
Overview of Indexes	299
Defining Indexes.....	301
The OVER Clause.....	303
Using Windows.....	304
Hive Analytics Functions	306
Lab 9.1: Advanced Hive Programming.....	307
Hive File Formats	317
Hive SerDes	319
Hive ORC Files.....	321
Demonstration: ORC Files	323
Computing Table Statistics	325
Vectorization	327
Using HiveServer2.....	328
Understanding Hive on Tez.....	329
Using Tez for Hive Queries.....	330
Hive Optimization Tips.....	331
Hive Query Tunings	332
Lab 9.2: Streaming Data with Hive and Python.....	335
Unit 10: Hadoop 2.0 and YARN.....	339
What is HDFS Federation?	340
Multiple Federated NameNodes.....	341
Multiple Namespaces	342
Overview of HDFS High Availability	343
Quorum Journal Manager	344
Configuring Automatic Failover	345
What is YARN?	346
Open-source YARN Use Cases.....	347
The Components of YARN	348
Lifecycle of a YARN Application.....	350
A Cluster View Example	351
Lab 10.1: Running a YARN Application	353
Appendix A: Defining Workflow with Oozie	356
Overview of Oozie	357
Defining an Oozie Workflow.....	358
Pig Actions	359
Hive Actions	361
MapReduce Actions	362
Submitting a Workflow Job	364
Making Decisions	366
Defining an Oozie Coordinator Job.....	367
Schedule a Job Based on Time.....	368
Schedule a Job Based on Data Availability.....	370
Lab: Defining an Oozie Workflow	373
Appendix B: Hadoop Streaming.....	378
Hadoop Streaming	379
Running a Hadoop Streaming Job.....	380

Appendix C: Unit Review Answers.....381

Preface

- Course Outline
- Introductions
- Class Logistics

Unit 1: Understanding Hadoop

Topics covered:

- Big Data Facts
- The Three V's of Big Data
- 6 Key Hadoop Data Types
- Sentiment Use Case - the Iron Man 3 Movie Release
- Geolocation Use Case - Monitoring Truck Drivers
- What is Hadoop?
- What is Hadoop 2.0?
- Relational Databases vs. Hadoop
- What's New in Hadoop 2.0?
- The Hadoop Ecosystem
- Who is Hortonworks?
- The Hortonworks Data Platform (HDP)
- The Path to ROI
- Lab 1.1: Start the HDP 2.0 Virtual Machine

Big Data Facts

1. In what timeframe do we now create the same amount of information that we created from the dawn of civilization until 2003?
2. 90% of the world's data was created in the last (how many years)?
3. What is 1024 petabytes also known as?
4. What is the anticipated shortage in the U.S. of skilled workers with deep analytical skills by 2018?

Sources:
<http://www.itbusinessedge.com/cm/blogs/lawson/iust-the-stats-big-numbers-about-big-data/?cs=48051>
<http://techcrunch.com/2010/08/04/schmidt-data/>
© Hortonworks Inc. 2013



Big Data Facts

Big Data is a common buzzword in the world of IT nowadays, and it is important to understand what the term means:

Big Data describes the realization of greater business intelligence by storing, processing, and analyzing data that was previously ignored or siloed due to the limitations of traditional data management technologies.

Notice from this definition that there is more to Big Data than just “a lot of data”, and there is more to Big Data than just storing it:

- **Processing:** If you are just storing a lot of data, then you probably do not have a use case for Big Data. Big Data is data that you want to be able to process and use as part of a business application.
- **Analyzing:** In addition to making the data a part of your applications, Big Data is also data that you want to analyze (i.e. mine the data) to find information that was otherwise unknown.

NOTE: A common aspect of Big Data is that it is often data that was otherwise *ignored* in your business because you did not have the capability to store, process and analyze it.

For example, your customers' personal information stored in a RDBMS and used in online transactions is not Big Data. However, the three terabytes of Web log files from millions of visits to your Web site over the last ten years probably is Big Data.

The Three V's of Big Data

Variety

Unstructured and semi-structured data is becoming as strategic as the traditional structured data.

Volume

Data coming in from new sources as well as increased regulation in multiple areas means storing more data for longer periods of time.

Velocity

Machine data as well as data coming from new sources is being ingested at speeds not even imagined a few years ago.

© Hortonworks Inc. 2013



The Three V's of Big Data

The characteristics of Big Data are often defined as the **3 V's**:

- **Variety:** any type of structured or unstructured data
- **Volume:** terabytes and petabytes (and even exabytes) of data
- **Velocity:** data flows in to your organization at increasing rates

NOTE: Many articles have been published that add other “V” words to this list, like veracity, viability and value. These terms (and many others) can certainly be used to describe aspects of Big Data.

Big Data includes all types of data:

- **Structured**: the data has a schema, or a schema can be easily assigned to it.
- **Semi-structured**: has some structure, but typically columns are often missing or rows have their own unique columns.
- **Unstructured**: data that has no structure, like JPGs, PDF files, audio and video files, etc.

Big Data also has two inherent characteristics:

- **Time-based**: a piece of data is something known at a certain moment in time, and that time is an important element. For example, you might live in San Francisco and tweet about a restaurant that you enjoy. If you later move to New York, the fact that you once liked a restaurant in San Francisco does not change.
- **Immutable**: because of its connection to a point in time, the truthfulness of the data does not change. We look at changes in Big Data as “new” entries, not “updates” of existing entries.

6 Key Hadoop DATA TYPES

1. **Sentiment**
How your customers feel
2. **Clickstream**
Website visitors' data
3. **Sensor/Machine**
Data from remote sensors and machines
4. **Geographic**
Location-based data
5. **Server Logs**
6. **Text**
Millions of web pages, emails, and documents



Value

© Hortonworks Inc. 2014



6 Key Hadoop Data Types

The type of Big Data that ends up in Hadoop typically fits into one of the following types:

1. **Sentiment:** Understand how your customers feel about your brand and products - right now!
2. **Clickstream:** Capture and analyze website visitors' data trails and optimize your website.
3. **Sensor/Machine:** Discover patterns in data streaming automatically from remote sensors and machines
4. **Geographic:** Analyze location-based data to manage operations where they occur.
5. **Server Logs:** Research log files to diagnose and process failures and prevent security breaches
6. **Text:** Understand patterns in text across millions of web pages, emails, and documents

Sentiment Use Case



TM Marvel Comics

© Hortonworks Inc. 2014

- Analyze customer sentiment on the days leading up to and following the release of the movie *Iron Man 3*.
- Questions to answer:
 - How did the public feel about the debut?
 - How might the sentiment data have been used to better promote the launch of the movie?



Sentiment Use Case

The goal was to determine how the public felt about the debut of the Iron Man 3 movie using Twitter, and how the movie company might better promote the movie based on the initial feedback. Here are the steps that were performed:

1. Use Flume to get the Twitter feeds into HDFS.
2. Use HCatalog to define a shareable schema for the data.
3. Use Hive to determine sentiment.
4. Use an Excel bar graph to visualize the volume of tweets.
5. Use MS PowerView to view sentiment by country on a map.

REFERENCE: Visit <http://hortonworks.com/use-cases/sentiment-analysis-hadoop-example/> to watch a video that walks through the steps above.

Geolocation Use Case

- A trucking company has over 100 trucks.
- The geolocation data collected from the trucks contains events generated while the truck drivers are driving the trucks.
- The company's goal with Hadoop is to:
 - reduce fuel costs
 - improve driver safety

© Hortonworks Inc. 2014



Geolocation Use Case

A trucking company collects sensor data from its trucks based on GPS coordinates and logs driving events like speed, acceleration, stopping too quickly, driving too close to other vehicles, stopping too quickly, and so on. These events get collected and put into Hadoop for analysis. The goal of the trucking company is to reduce fuel costs and improve driver safety by recognizing high-risk drivers.

1. Flume is used to get the raw sensor data into Hadoop.
2. Sqoop is used to get the data about each vehicle from an RDBMS into Hadoop.
3. HCatalog contains all the schema definitions.
4. Hive is used to analyze the gas mileage of trucks.
5. Pig is used to compute a risk factor for each truck driver based on his/her events.
6. Excel is used for creating bar graphs and maps showing where and how often events are occurring.

REFERENCE: Visit <http://hortonworks.com/use-cases/analyze-geolocation-data-hadoop/> to view a video of the trucking company geolocation use case.

What is Hadoop?

- Framework for solving data-intensive processes.
- Designed to scale massively
- Processes all the contents of a file (instead of attempting to read portions of a file)
- Hadoop is very fast for very large jobs
- Hadoop is not fast for small jobs
- It does not provide caching or indexing (tools like HBase can provide these features if needed)
- Designed for hardware and software failures

© Hortonworks Inc. 2014



What is Hadoop?

Apache Hadoop is one such system. Hadoop ties together a cluster of commodity machines with local storage using free and open source software to store and process vast amounts of data at a fraction of the cost of any other system.

- **Framework for solving data-intensive processes:** Meaning the bottleneck was waiting to read data from the disk. Of the potential bottlenecks in a computing system are CPU, RAM, network, and disk IO. Hadoop was designed to solve the problem of disk IO.
- **Designed to scale massively:** In order to scale massively it is important to keep things as simple as possible and provide redundancy and avoid the need for any sharing of a single system, such as locking files for operations. To meet these goals, the Hadoop file system is write once and files are immutable.
- **Processes all the contents of a file:** Although a file in HDFS can be accessed by opening the file at any byte offset point, the typical assumption is that an application will read all of a file. The application layer will read the whole file or often a directory of files. The reasoning is that once the system has paid the expensive price of seeking to the top of the file, in order to satisfy the goal of massive throughput, the most efficient use of the disk is to have it spend most of its time transferring data, not seeking to find data.

- **Hadoop is very fast for big jobs:** Hadoop does scale. A 20 node cluster with 10 disks per machine running a large MapReduce job will have close to 200 disks reading and processing data ALL AT ONCE. The relative speed of work done in parallel when compared to a non-parallel system will be significant.
- **Hadoop is not fast for small jobs:** If your data set is small, and could fit on a single machine, then you will find that Hadoop is slow compared to another system. In general, if you think 5 minutes is "slow" then perhaps the problem you have is not what is typically considered a Hadoop problem.
- **No caching, no indexes:** Core Hadoop does not provide such features. HBase and apps built on top of Hadoop provide caching and a type of indexing.
- **Designed for hardware and software failures:** which is accomplished by "sharing nothing". Core Hadoop systems are designed to share as little information about state as possible. DataNodes do not know what file a block belongs to; a map task writes to a temporary directory and that data is thrown away at failure; a task is either running to success or it fails completely and subsequent attempts do not acquire state from the failed task; a DataNode does not know what block a file belongs to - it has a more simple job to do, read a block when requested.

All of these features put together create a powerful data processing framework that not only store large amounts of data, but also process large amounts of data in a relatively short amount of time.

Relational Databases vs. Hadoop

Relational	Hadoop
Required on write	schema
Reads are fast	speed
Standards and structured	governance
Limited, no data processing	processing
Structured	data types
Interactive OLAP Analytics Complex ACID Transactions Operational Data Store	best fit use
	Data Discovery Processing unstructured data Massive Storage/Processing

© Hortonworks Inc. 2013



Relational Databases vs. Hadoop

Understanding how schemas work in Hadoop might help you better understand how Hadoop is different from relational databases:

- With a relational database, a schema must exist **before** the data is written to the database, which forces the data to fit into a particular model.
- With Hadoop, data is input into HDFS in its raw format without any schema. When data is **retrieved** from HDFS, a schema can be applied then to fit the specific use case and needs of your application.

IMPORTANT: Hadoop is not meant to replace your relational database. Hadoop is for storing Big Data, which is often the type of data that you would otherwise not store in a database due to size or cost constraints. You will still have your database for relational, transactional data.

What is Hadoop 2.0?

- The Apache Hadoop 2.0 project consists of the following modules:
 - **Hadoop Common**: the utilities that provide support for the other Hadoop modules.
 - **HDFS**: the Hadoop Distributed File System
 - **YARN**: a framework for job scheduling and cluster resource management.
 - **MapReduce**: for processing large data sets in a scalable and parallel fashion.

© Hortonworks Inc. 2013



What is Hadoop 2.0?

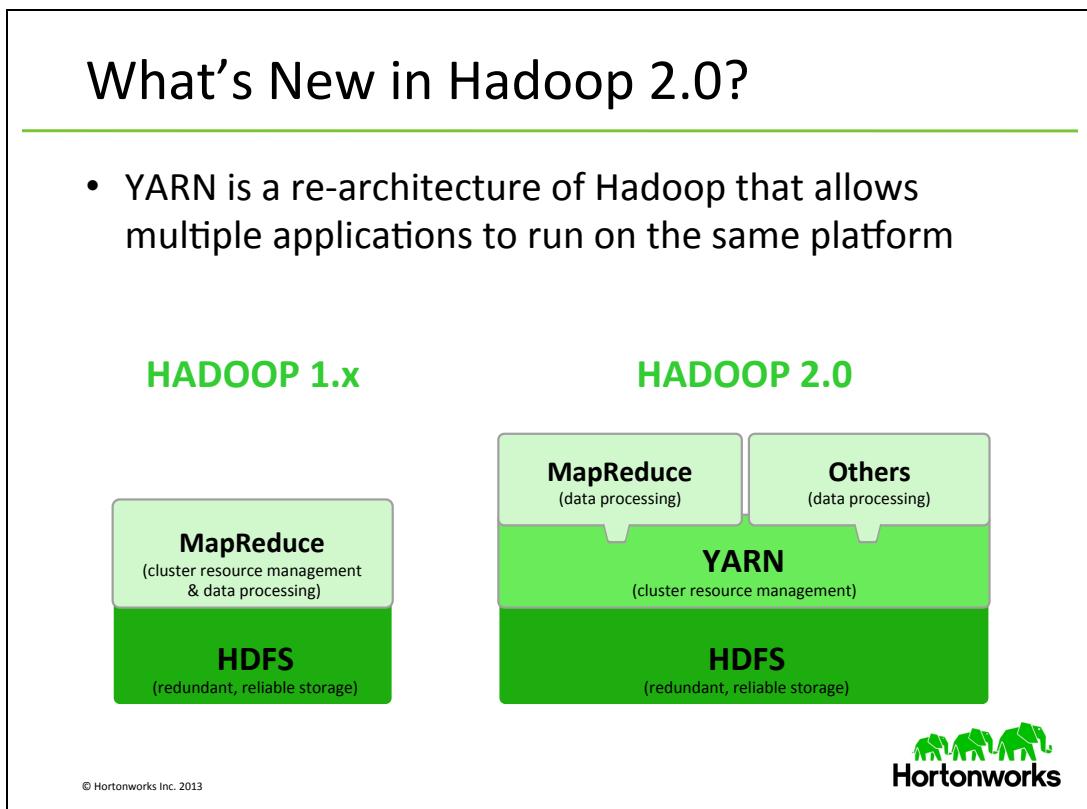
Hadoop 2.0 refers to the next generation of Hadoop. As expected, the Hadoop framework has grown to meet the demands of its own popularity and usage, and 2.0 reflects the natural maturing of the open-source project.

The Apache Hadoop 0.23 project (the open-source version number) consists of the following modules:

- **Hadoop Common**: the utilities that provide support for the other Hadoop modules.
- **HDFS**: the Hadoop Distributed File System
- **YARN**: a framework for job scheduling and cluster resource management.
- **MapReduce**: for processing large data sets in a scalable and parallel fashion.

What's New in Hadoop 2.0?

- YARN is a re-architecture of Hadoop that allows multiple applications to run on the same platform



What's New in Hadoop 2.0?

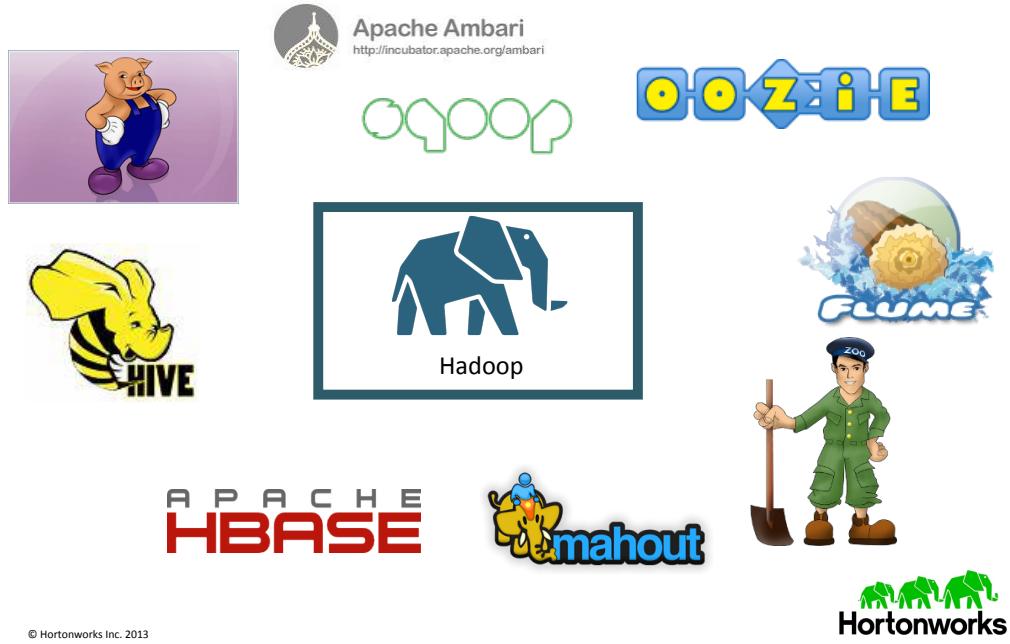
There are two exciting and significant additions to the Hadoop framework:

- **HDFS Federation:** provides a name service that is both scalable and reliable.
- **YARN:** which stands for *Yet Another Resource Negotiator*, it divides the two major functions of the JobTracker (resource management and job lifecycle management) into separate components.

A key issue with Hadoop 1.x was providing a NameNode that was highly-available. Not only does HDFS Federation provide an HA name service, but it also allows for distribution of workload since the NameNodes can now scale horizontally.

YARN provides a logical separation of duties for negotiating and executing jobs across a Hadoop cluster. The end result of YARN is a new, more-generic resource management framework that works with more than just MapReduce jobs.

The Hadoop Ecosystem



The Hadoop Ecosystem

Hadoop is more than HDFS and MapReduce. There is a large group of technologies and frameworks that are associated with Hadoop, including:

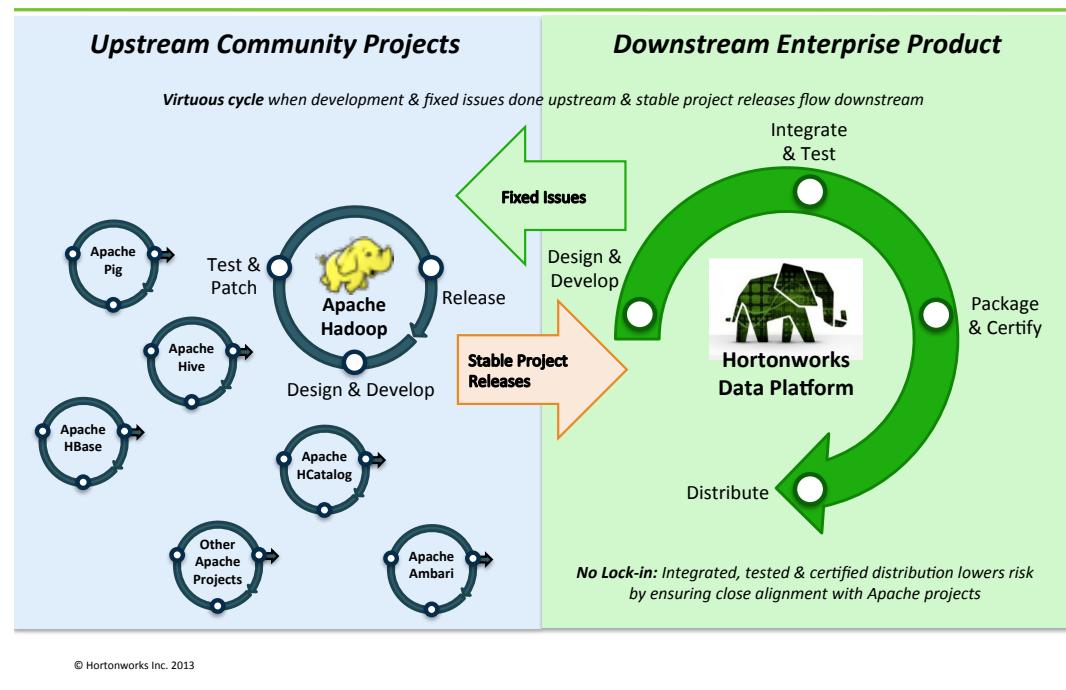
- **Pig**: a scripting language that simplifies the creation of MapReduce jobs and excels at exploring and transforming data
- **Hive**: provides SQL-like access to your Big Data
- **HBase**: a Hadoop database
- **HCatalog**: for defining and sharing schemas
- **Ambari**: for provisioning, managing, and monitoring Apache Hadoop clusters
- **ZooKeeper**: an open-source server which enables highly reliable distributed coordination
- **Sqoop**: for efficiently transferring bulk data between Hadoop and relation databases
- **Oozie**: a workflow scheduler system to manage Apache Hadoop jobs

- **Mahout**: an Apache project whose goal is to build scalable machine learning libraries
- **Flume**: for efficiently collecting, aggregating, and moving large amounts of log data

There are many other products and tools in the Hadoop ecosystem, including:

- **Hadoop as a Service**: includes Microsoft HDInsight and Rackspace Private Cloud
- **Programming Frameworks**: includes Cascading, Hama and Tez
- **Data Integration Tools**: includes Talend Open Studio

Who is Hortonworks?

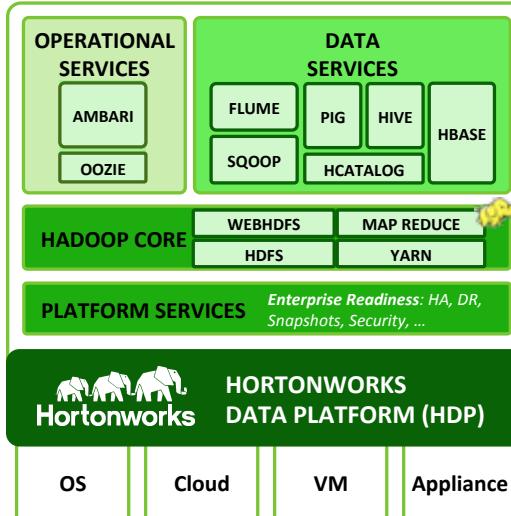


Who is Hortonworks?

Hortonworks **develops, distributes and supports** Enterprise Apache Hadoop:

- **Develop:** Hortonworks was formed by the key architects, builders and operators from the Yahoo! Hadoop software engineering team that led the effort to design and build every major release of Apache Hadoop from 0.1 to the most current stable release, contributing more than 80% of the code along the way.
- **Distribute:** We provide a 100% Open Source Distribution of Apache Hadoop, adding the required Operational, Data and Platform services from the open source community in the Hortonworks Data Platform (HDP).
- **Support:** We provide a range of support options for customers of the Hortonworks Data Platform and are the leading provider of expert Hadoop training available today.

The Hortonworks Data Platform (HDP)



Enterprise Hadoop

- The ONLY 100% open source and complete distribution
- Enterprise grade, proven and tested at scale
- Ecosystem endorsed to ensure interoperability



The Hortonworks Data Platform (HDP)

The **Hortonworks Data Platform**, or **HDP** for short, is the only 100% open source data management platform for Apache Hadoop, and is the most stable and reliable Apache Hadoop distribution. It delivers the cost-effectiveness of Hadoop and the advanced services required for enterprise deployments.

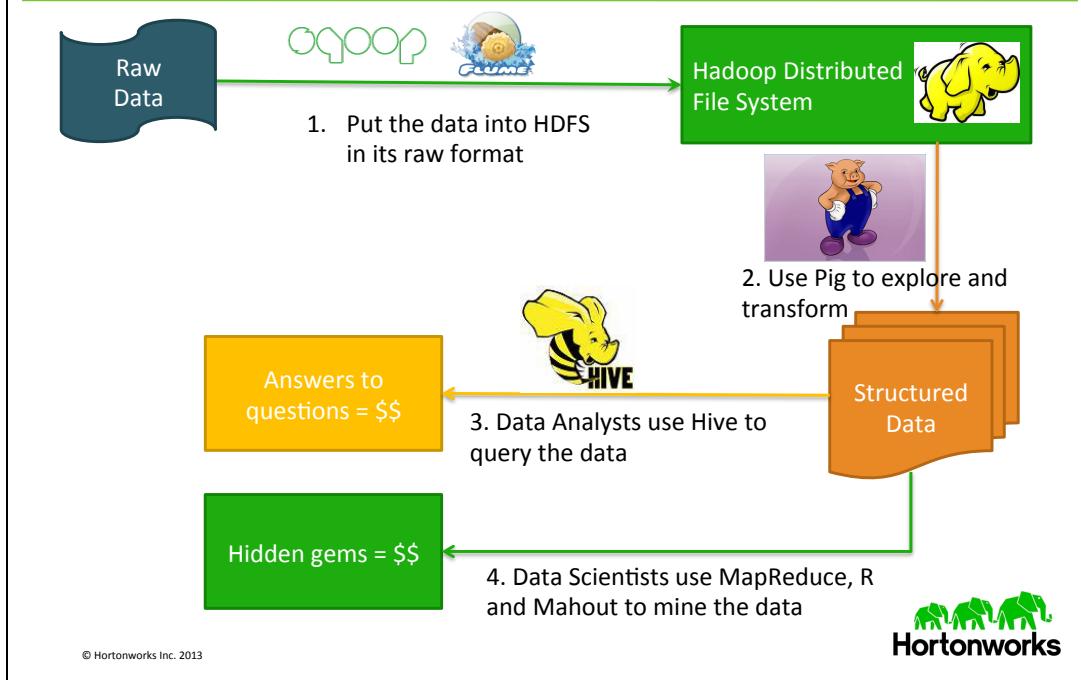
The key features of HDP include:

- **High Availability:** HA is now achievable in HDP 2.0 without the use of an outside technology.
- **Open Source Cluster Management:** HDP includes Apache Ambari, the only open source operations tools that allows you to provision, manage and monitor a Hadoop cluster of any size.
- **Metadata Services & HCatalog:** HCatalog provides metadata services and a REST interface that provides an additional SQL-like interface to Hadoop.
- **Data Integration Services:** including Sqoop, Flume and WebHDFS.

- **ODBC Done Right:** Hive has a free high-performance ODBC driver that includes a SQL engine so you can interact with nearly every BI tool, including all SQL-92 interfaces.

NOTE: Apache Hadoop has become a core component of the enterprise data architecture as a complement to existing data management systems. Accordingly, HDP is architected to easily interoperate so you can extend your existing investments in applications, tools and processes with Hadoop.

The Path to ROI



The Path to ROI

Along with the tools and frameworks in the Hadoop ecosystem, there are also the individuals who must push the data through Hadoop, answer questions, and find hidden gems within the Big Data. The path to ROI in Hadoop involves several steps and roles, including:

- Put the data into HDFS:** Because you do not need to apply a schema to the data, it is best to keep it in its raw format and try not to force a structure on the data that may only fit a few use cases. By keeping all of the original raw data, you leave the door open for answers to future questions that you may not have thought to ask yet.
- Explore and Transform:** Often the raw data needs to be transformed. Pig is an excellent tool for exploring the raw data and transforming it into a structure more suitable for your specific use case.
- Answer questions:** Hive is a great tool for performing queries on structured data. The Hive query language is essentially SQL, so it is familiar and comfortable to use for Data Analysts.

4. **Find hidden gems:** The real ROI comes from mining the data, a task that fits under the moniker of *data science*. The Data Scientist uses a variety of tools and frameworks, including Java MapReduce, R, Mahout, Python and other tools and scripting languages.

NOTE: The diagram above is meant only to show a typical use case of how data might flow through Hadoop and how the various elements of the Hadoop ecosystem are typically used. There are certainly many other scenarios and use cases, along with many other tools available for answering questions and mining Big Data.

Unit 1 Review

1. What are 1024 petabytes known as? _____
2. What are 1024 exabytes known as? _____
3. List the 3 V's of Big Data: _____
4. Sentiment is one of the 6 key types of Big Data. List the other 5:

5. What technology might you use to stream Twitter feeds into Hadoop?

6. What technology might you use to define, store and share the schemas of your Big Data stored in Hadoop?

7. What are the two main new components in Hadoop 2.0?

Lab 1.1: Start the HDP 2.0 Virtual Machine

Objective:	Start and access your HDP 2.0 Sandbox.
Location of Files:	n/a
Successful Outcome:	The HDP 2.0 virtual machine will be running on your local machine.
Before You Begin:	VMWare should be installed on your machine and the classroom VM should be imported.

Step 1: Start the VM

1.1. Start VMWare on your local machine.

1.2. Select the classroom instance by clicking on it.

1.3. Click the **Start** icon on the toolbar. The VM will start, which may take several minutes. Once the VM startup is complete, the console should look like the following:

Hortonworks Sandbox 2.0
http://hortonworks.com

To initiate your Hortonworks Sandbox session,
please open a browser and enter this address
in the browser's address field:
http://192.168.215.139/

You can access SSH by \$ ssh root@192.168.215.139

Log in to this virtual machine: Linux/Windows <Alt+F5>, Mac OS X <Ctrl-Alt-F5>

Step 2: SSH Into Your Sandbox

NOTE: The VM for this class is a modified version of the Hortonworks Sandbox that has the all the necessary lab files copied onto it. The Sandbox has an inconvenient feature in that it does not allow you to scroll. This will become tedious in the labs, so you are going to SSH into your VM using an SSH client.

2.1. On Windows: Start **Putty** (or whatever SSH client application you are using). In Putty, enter the IP address of your VM in the **Host Name**. Click the **Open** button, and then enter the login credentials (**root/hadoop**).

2.2. On a Mac/Linux, enter the following command in a terminal window:

```
$ ssh root@ipaddress
```

2.3. Once logged in successfully, you should see a prompt that looks like the following:

```
[root@sandbox ~] #
```

Step 3: Verify Hadoop is Running

3.1. The Java Development Kit (JDK) comes with a tool called **jps** that shows you all the Java processes running. From the command prompt, enter the following command:

```
[root@sandbox ~] # jps
```

The output should look similar to the following:

```
2615 RunJar
2307 NodeManager
2300 ResourceManager
1739 SecondaryNameNode
2381 RunJar
1742 DataNode
2229 RunJar
2078 Bootstrap
1745 NameNode
2284 JobHistoryServer
3483 Jps
```

```
3278 RunJar  
2030 QuorumPeerMain
```

Step 4: Check the Health of the Cluster

- 4.1.** From the command line enter the following command, which displays the usage of the **hdfs dfsadmin** utility:

```
# hdfs dfsadmin
```

NOTE: The “*dfs*” in **dfsadmin** stands for *distributed file system*, and the **dfsadmin** utility contains administrative commands for communicating with the Hadoop Distributed File System.

- 4.2.** Notice the **dfsadmin** utility has a **-report** option, which outputs the current health of your cluster. Enter the following command to view this report:

```
# hdfs dfsadmin -report
```

4.3. What is the configured capacity of your distributed file system? _____

4.4. What is the present capacity? _____

4.5. How much of your distributed file system is used right now? _____

4.6. What do you think an “**Under replicated block**” is? _____

4.7. How many available DataNodes does your cluster have? _____

Step 5: View the Various Hadoop Web User Interfaces

- 5.1.** Open a Web browser and enter the following URL:

```
http://ipaddress:8088/
```

You have to replace *ipaddress* with the IP address of your VM.

- 5.2.** Notice the URL shows the ResourceManager Web UI:



All Applications

Logged in as: dr.who

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
0	0	0	0	0	0 B	2.20 GB	0 B	1	0	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
No data available in table										

Showing 0 to 0 of 0 entries First Previous Next Last

About Nodes Applications

- NEW
- NEW_SAVING
- SUBMITTED
- ACCEPTED
- RUNNING
- REMOVING
- FINISHING
- FINISHED
- FAILED
- KILLED

Scheduler

Tools

The ResourceManager UI displays information about the applications that have been executed on your Hadoop cluster.

5.3. Point your browser to the NameNode UI:

`http://ipaddress:50070/`

NameNode 'sandbox.hortonworks.com:8020' (active)

Started:	Fri Jan 10 08:08:01 PST 2014
Version:	2.2.0.2.0.6.0-76, 8656b1cfad13b03b29e98cad042626205e7a1c86
Compiled:	2013-10-18T00:19Z by jenkins from bigwheel-GA-2.2.0
Cluster ID:	CID-6534e889-85e8-46e2-8034-8d7c8930848a
Block Pool ID:	BP-1578958328-10.0.2.15-1382306880516

[Browse the filesystem](#)
[NameNode Logs](#)

Cluster Summary

Security is OFF

918 files and directories, 627 blocks = 1545 total.

Heap Memory used 153.03 MB is 15% of Committed Heap Memory 960 MB. Max Heap Memory is 960 MB.

Non Heap Memory used 36.64 MB is 98% of Committed Non Heap Memory 37.25 MB. Max Non Heap Memory is 130 MB.

Configured Capacity	:	44.85 GB
DFS Used	:	574.75 MB
Non DFS Used	:	5.67 GB
DFS Remaining	:	38.62 GB
DFS Used%	:	1.25%
Non DFS Remaining%	:	88.11%

Notice the NameNode UI contains similar information as the **hdfs dfsadmin** report, plus you can browse the filesystem, logs, and also view the DataNodes of the cluster.

5.4. View the JobHistory UI at:

`http://ipaddress:19888/`

As it sounds, the JobHistory UI shows the applications that have executed on your cluster. You will visit this page often in the upcoming labs throughout the course.

RESULT: You now have the Hortonworks Data Platform 2.0 running in a virtual machine on your local machine, as well as SSH access to the VM.

ANSWERS:

4.3: 44.85 GB

4.4: Around 38 GB

4.5: Look for the “DFS Used:” value. It should be around 332MB.

4.6: We will discuss this in the next Unit, but for now know that your Big Data is split up into small blocks that get replicated across the cluster.

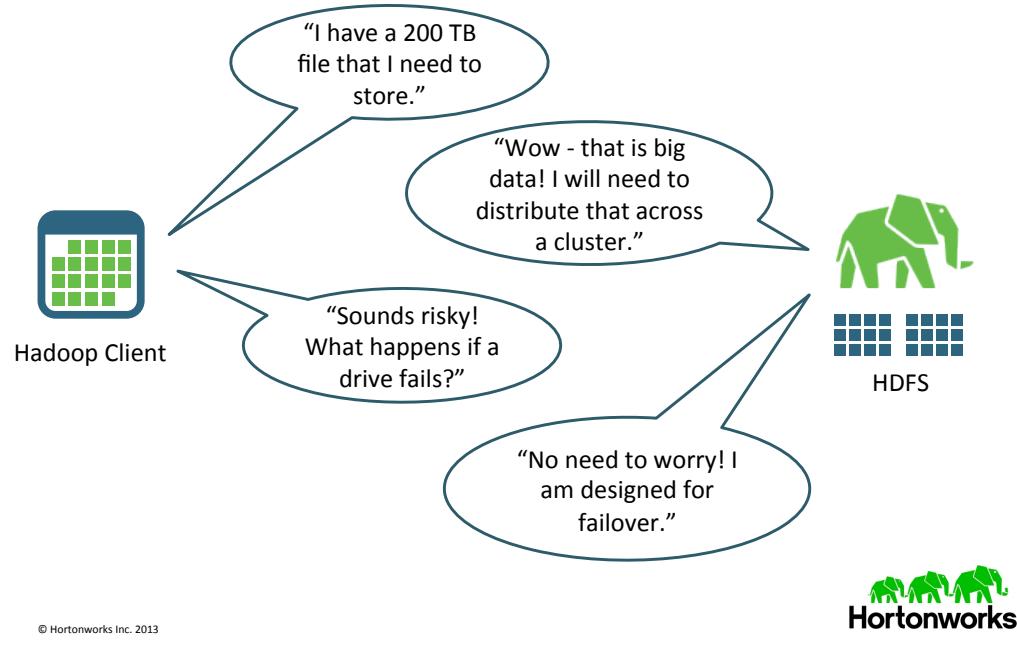
4.7: 1

Unit 2: The Hadoop Distributed File System (HDFS)

Topics covered:

- What is HDFS?
- Hadoop vs. RDBMS
- HDFS Components
- Understanding Block Storage
- The NameNode
- The DataNodes
- DataNode Failure
- HDFS Commands
- Examples of HDFS Commands
- HDFS File Permissions
- Lab 2.1: Using HDFS Commands

What is HDFS?



What is HDFS?

Data in Hadoop is stored on a filesystem referred to as **HDFS** - the **Hadoop Distributed File System**. With HDFS, data is broken down into chunks and distributed across a cluster of machines.

HDFS has the following characteristics:

- Primary storage system for Hadoop, it stores large files as small blocks
- Designed to be deployed on low-cost hardware
- Designed to scale easily and effectively (adding more nodes increases both storage space and computing throughput)
- Reliability - data is replicated so that disk failover is not only acceptable, but expected and handled seamlessly

NOTE: HDFS is the storage mechanism for Hadoop. In Unit 4, *The MapReduce Framework and YARN*, we will discuss how YARN is the computing mechanism for Hadoop.

NOTES

Hadoop

- Assumes a task will require reading a significant amount of data off of a disk
- It does not maintain any data structure
- Simply reads the entire file
- Scales well (increase the cluster size to decrease the read time of each task)

RDBMS

- Uses indexes to avoid reading an entire file (very fast lookups)
- Maintains a data structure in order to provide a fast execution layer
- Works well as long as the index fits in RAM

- 2,000 blocks of size 256MB
- 1.9 seconds of disk read for each block
- On a 40 node cluster with 8 disks on each node, it would take about 14 seconds to read the entire 500 GB

500 GB
data file

61 minutes to read this data off of a disk (assuming a transfer rate of 1,030 Mbps)



© Hortonworks Inc. 2013

Hadoop vs. RDBMS

To help better understand how Hadoop works, let's compare it to something you may be very familiar with: a relational database. From a very high level, the difference between Hadoop and RDMBS is:

- A relational database uses complex in-memory data structures to avoid the expense of disk access.
- Hadoop uses a collection of disks to parallelize that expense of disk access.

A relational database's performance is largely optimized by using indexes to avoid disk access. In order to store a lot of data and access it efficiently, RDBMS use a smaller organized representation of the data (an index) that can be loaded into memory and allow a lightning fast lookup as to whether or not a disk seek and read is needed. This works very well up to the point that your index no longer fits in RAM. Or up to the point that your final result set, or the operations performed while generating this result set, require a lot of disk access.

Hadoop looks at this problem in another way. Hadoop assumes that the operation will require reading a significant amount of data off of disk. To avoid seeks, Hadoop simply reads the entire file.

An Example of Disk Read Performance

Suppose a RDBMS had to process a 500G data file. The time it takes to read this data off of disk would be 61 minutes. This assumes a transfer rate of 1030Mbps. (Source: http://www.calctool.org/CALC/prof/computing/transfer_time). Typically you would look at your queries, add some indexes, and try to optimize the access to avoid this disk seek.

In Hadoop, this file could be stored as 2,000 256Mb chunks. If we processed it in Hadoop doing a single search for records matching a pattern, then Hadoop would perform 2,000 individual file reads. Each of these 2,000 tasks will require 1.9 seconds of disk read. A cluster of 40 DataNodes with 8 disks each (so a total of 320 disks) will get an average 6 or 7 of these file chunk reads, for a total transfer time of 14 seconds. The bottleneck of processing this 500G file has been taken from 60 minutes to $7 * 1.9$ seconds, or roughly 14 seconds.

NOTE: This doesn't mean the overall MapReduce job would take 14 seconds. This example is ignoring the overhead of both MapReduce and RDBMS and is only comparing the amount of time spent reading from disk. Regardless of the overhead, this demonstrates how Hadoop reads large amounts of data in an extremely efficient manner!

HDFS Components

- **NameNode**
 - The “master” node of HDFS
 - Determines and maintains how the chunks of data are distributed across the DataNodes
- **DataNode**
 - Stores the chunks of data, and is responsible for replicating the chunks across other DataNodes

© Hortonworks Inc. 2013



HDFS Components

A Hadoop instance consists of a cluster of HDFS machines, often referred to as the *Hadoop cluster* or *HDFS cluster*. There are two main components of an HDFS cluster:

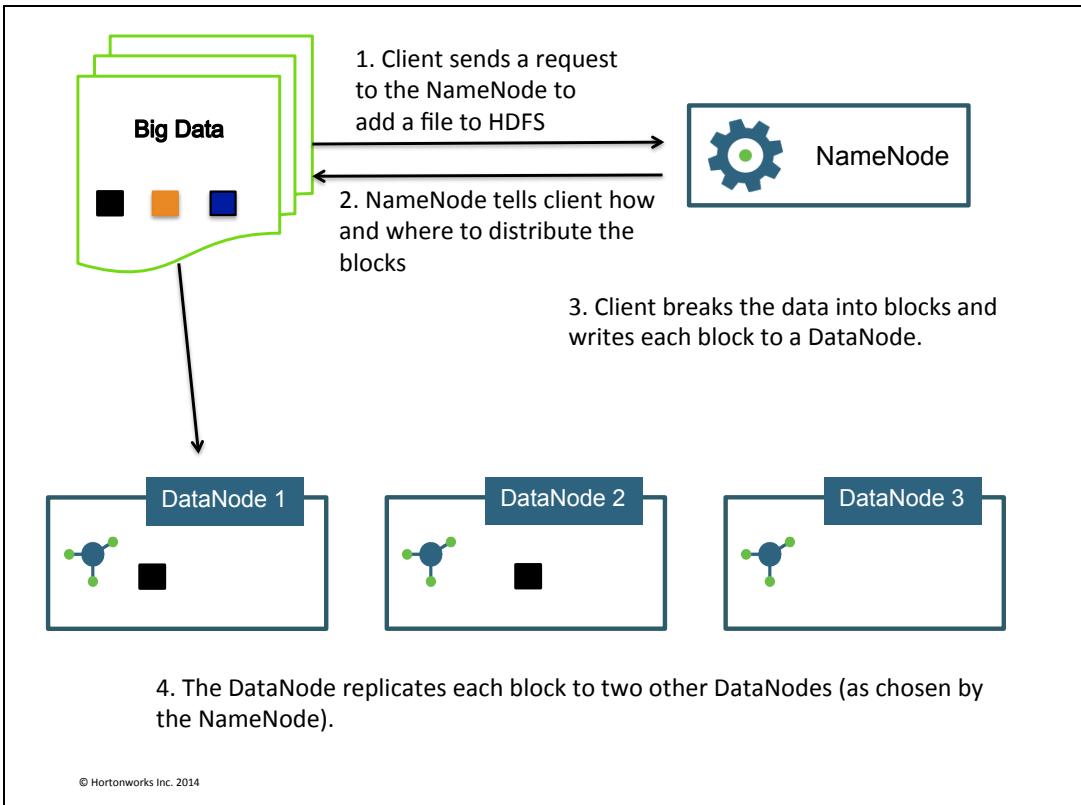
1. **NameNode:** The “master” node of HDFS that manages the data (without actually storing it) by determining and maintaining how the chunks of data are distributed across the DataNodes.
2. **DataNode:** Stores the chunks of data, and is responsible for replicating the chunks across other DataNodes.

The NameNode and DataNode are daemon processes running in the cluster. Some important concepts involving the NameNode and DataNodes:

- A NameNode represents a single namespace. A cluster can have multiple NameNodes if multiple namespaces are desired.
- Data never resides on or passes through the NameNode. Your big data only resides on DataNodes.
- DataNodes are referred to as “slave” daemons to the NameNode and are constantly communicating their state with the NameNode.

- The NameNode keeps track of how the data is broken down into chunks on the DataNodes.
- The default chunk size is 128MB (but is configurable).
- The default replication factor is 3 (and is also configurable), which means each chunk of data is replicated across 3 DataNodes.
- DataNodes communicate with other DataNodes (through commands from the NameNode) to achieve data replication.

NOTE: HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories.



Understanding Block Storage

Putting a file into HDFS involves the following steps:

1. A client application sends a request to the NameNode that specifies where they want to put the file in the file system.
2. The NameNode determines how the data is broken down into blocks and which DataNodes will be used to store those blocks. That information is given to the client application.
3. The client application communicates directly with each DataNode, writing the blocks onto the DataNodes.
4. The DataNodes replicate the newly-created blocks, based on instructions from the NameNode.

You can specify the block size for each file using the **dfs.blocksize** property. If you do not specify a block size at the file level, then the global value of **dfs.blocksize** defined in **hdfs-site.xml** is used.

IMPORTANT: Notice the data never actually passes through the NameNode. The client program that is uploading the data into HDFS performs I/O directly with the DataNodes. The NameNode only stores the metadata of the file system, but is not responsible for storing or transferring the data.

Demonstration: Understanding Block Storage

Objective:	To understand how data is partitioned into blocks and stored in HDFS.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Put the File into HDFS

1.1. Review the contents of the file **stocks.csv** located in **labs/demos**.

1.2. Try putting the file into HDFS with a block size of 30 bytes:

```
# hadoop fs -D dfs.blocksize=30 -put stocks.csv stocks.csv
```

1.3. Notice 30 bytes is not a valid blocksize. The blocksize needs to be at least 1048576 according to the **dfs.namenode.fs-limits.min-block-size** property:

```
put: Specified block size is less than configured minimum value (dfs.namenode.fs-limits.min-block-size): 30 < 1048576
```

1.4. Try the **put** again, but use a block size of 2,000,000:

```
# hadoop fs -D dfs.blocksize=2000000 -put stocks.csv  
stocks.csv
```

1.5. Notice 2,000,000 is not a valid block size because it is not a multiple of 512 (the checksum size).

1.6. Try the **put** again, but this time use 1,048,576 for the block size:

```
# hadoop fs -D dfs.blocksize=1048576 -put stocks.csv  
stocks.csv
```

1.7. This time the **put** command should have worked. Use **ls** to verify the file is in HDFS:

```
# hadoop fs -ls
Found 1 items
-rw-r--r-- 3 root root 3613198 2013-08-28 21:55
stocks.csv
```

Step 2: View the Number of Blocks

2.1. Run the following command to view the number of blocks that were created for **stocks.csv**:

```
# hdfs fsck /user/root/stocks.csv
```

2.2. Notice there are four blocks. Look for the following line in the output:

```
Total blocks (validated) : 4 (avg. block size 903299 B)
```

Step 3: Find the Actual Blocks

3.1. Enter the same **fsck** command as before, but add the **-files** and **-blocks** options:

```
# hdfs fsck /user/root/stocks.csv -files -blocks
```

Notice the output contains the block IDs, which coincidentally are the names of the files on the DataNodes.

3.2. Change directories to the following:

```
# cd /hadoop/hdfs/data/current/BP-xxx/current/finalized/
```

replacing **BP-xxx** with the actual folder name.

3.3. Notice the actual blocks appear in this folder. List the contents of the folder and look for files that are exactly 1048576. These are 3 of the blocks, and notice the 4th block is smaller: 467470 bytes.

```
-rw-r--r-- 1 hdfs hadoop 1048576 Aug 28 21:55
blk_1073741904
-rw-r--r-- 1 hdfs hadoop 8199 Aug 28 21:55
blk_1073741904_1086.meta
```

```
-rw-r--r-- 1 hdfs hadoop 1048576 Aug 28 21:55
blk_1073741905
-rw-r--r-- 1 hdfs hadoop      8199 Aug 28 21:55
blk_1073741905_1087.meta
-rw-r--r-- 1 hdfs hadoop 1048576 Aug 28 21:55
blk_1073741906
-rw-r--r-- 1 hdfs hadoop      8199 Aug 28 21:55
blk_1073741906_1088.meta
-rw-r--r-- 1 hdfs hadoop 467470 Aug 28 21:55
blk_1073741907
-rw-r--r-- 1 hdfs hadoop      3663 Aug 28 21:55
blk_1073741907_1089.meta
```

3.4. You can view the contents of a block (although this is not a typical task in Hadoop!). Here is the tail of the 2nd block:

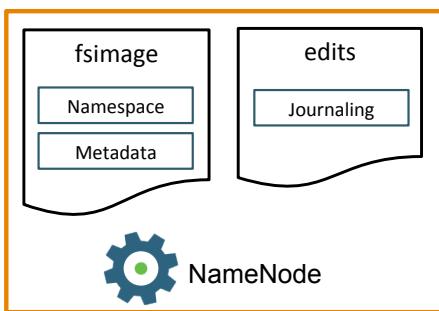
```
# tail blk_1073741905
NYSE,XKK,2007-08-20,9.51,9.64,9.30,9.51,4700,7.17
NYSE,XKK,2007-08-17,9.30,9.99,9.26,9.57,3900,7.21
NYSE,XKK,2007-08-16,9.45,10.00,8.11,9.05,23400,6.82
NYSE,XKK,2007-08-15,9.51,9.51,9.18,9.35,4900,7.04
NYSE,XKK,2007-08-14,9.52,9.52,9.51,9.51,1100,7.17
NYSE,XKK,2007-08-13,9.60,9.60,9.56,9.56,3000,7.20
NYSE,XKK,2007-08-10,9.82,9.82,9.60,9.60,2500,7.23
NYSE,XKK,2007-08-09,9.83,9.87,9.82,9.82,4500,7.40
NYSE,XKK,2007-08-08,9.45,9.90,9.45,9.66,6000,7.28
NYSE,XKK,2007-08-07,9.25,9.50,9.25,9.40
```

Notice the last record in this file is not complete and spills over to the next block - a common occurrence in HDFS.

NOTES

The NameNode

1. When the NameNode starts, it reads the **fsimage_N** and **edits_N** files.
 2. The transactions in **edits_N** are merged with **fsimage_N**.
 3. A newly-created **fsimage_N+1** is written to disk, and a new, empty **edits_N+1** is created.
- The NameNode will be in *safemode*, a read-only mode.
4. Now a client application can create a new file in HDFS.
 5. The NameNode journals that create transaction in the **edits_N+1** file.



© Hortonworks Inc. 2014

The NameNode

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, which is a master server that manages the file system namespace and regulates access to files by clients.

The NameNode has the following characteristics:

- The master of the DataNodes
- Executes file system namespace operations like opening, closing, and renaming files and directories
- Determines the mapping of blocks to DataNodes
- Maintains the file system namespace

The NameNode performs these tasks by maintaining two files:

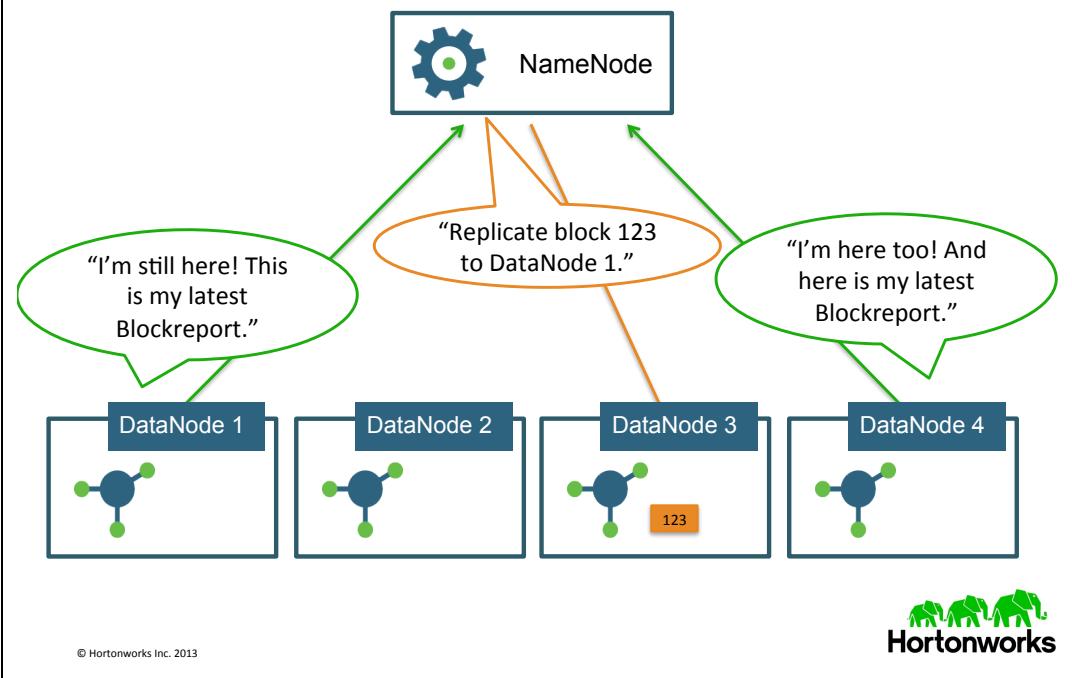
- **fsimage_N**: Contains the entire file system namespace, including the mapping of blocks to files and file system properties.
- **edits_N**: A transaction log that persistently records every change that occurs to file system metadata.

When the NameNode starts up, it enters *safemode* (a read-only mode). It loads the **fsimage_N** and **edits_N** from disk, applies all the transactions from the **edits_N** to the in-memory representation of the **fsimage_N**, and flushes out this new version into a new **fsimage_N+1** on disk.

For example, initially you will have an **fsimage_0** file and an **edits_0** file. When the merging occurs, the transactions in **edits_0** are merged with **fsimage_0**, and a new **fsimage_1** file is created. In addition, a new, empty **edits_1** file is created for all future transactions that occur after the creation of **fsimage_1**.

This process is called a **checkpoint**. Once the NameNode has successfully checkpointed, it will leave safemode, thus enabling writes.

The DataNodes



The DataNodes

HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of **DataNodes**.

The NameNode determines the mapping of blocks to DataNodes. The DataNodes are responsible for:

- Handling read and write requests from application clients.
- Performing block creation, deletion, and replication upon instruction from the NameNode. (The NameNode makes all decisions regarding replication of blocks.)
- Sending heartbeats to the NameNode.
- Sending a **Blockreport** to the NameNode.

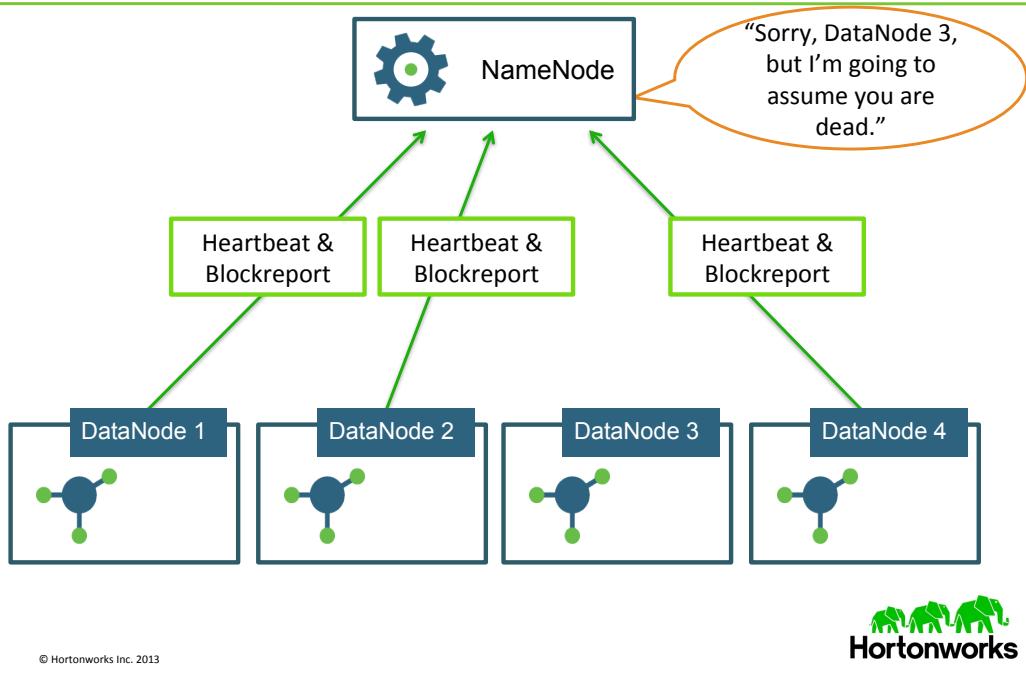
The NameNode periodically receives a **Heartbeat** and a **Blockreport** from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

DataNodes have the following characteristics:

- The DataNode has no knowledge about HDFS files.
- It stores each block of HDFS data in a separate file on its local file system.
- The DataNode does not create all files in the same local directory. Instead, it uses a discovery technique to determine the optimal number of files per directory and creates subdirectories appropriately.
- When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files, and then sends this information to the NameNode (as a Blockreport).

REFERENCE: For tips on configuring a network for a Hadoop cluster, visit
<http://hortonworks.com/kb/best-practices-for-cluster-network-configuration/>.

DataNode Failure



DataNode Failure

The primary objective of HDFS is to store data reliably even in the presence of failures. Hadoop is designed to recover gracefully from a disk failure or network failure of a DataNode:

- If a DataNode fails to send a Heartbeat to the NameNode, that DataNode is labeled as *dead*.
- Any data that was registered to a dead DataNode is not available to HDFS any more.
- The NameNode does not send new I/O requests to a dead DataNode, and its blocks are replicated to live DataNodes.

DataNode death typically causes the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary.

NOTE: It is possible that a block of data fetched from a DataNode arrives corrupted, either from a disk failure or network error. HDFS implements checksum checking on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents, it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block.

HDFS Commands

```
hadoop fs -command [args]
```

Here are a few (of the almost 30) HDFS commands:

- cat**: display file content (uncompressed)
- text**: just like cat but works on compressed files
- chgrp**, **chmod**, **chown**: changes file permissions
- put**, **get**, **copyFromLocal**, **copyToLocal**: copies files from the local file system to the HDFS and vice-versa.
- ls**, **-ls -R**: list files/directories
- mv**, **moveFromLocal**, **moveToLocal**: moves files
- stat**: statistical info for any given file (block size, number of blocks, file type, etc.)

© Hortonworks Inc. 2014



HDFS Commands

The **hadoop** application is a Hadoop client application that allows you to issue commands to HDFS from a command line. The **hadoop** application has the following syntax:

```
hadoop fs -command <args>
```

where *command* is one of the following:

- **ls**: lists the contents of folders
- **du**: shows the disk usage
- **count**: counts the number of directories, files and bytes in a path
- **chgrp**, **chown** and **chmod**: change file and directory permissions
- **stat**: print statistics about a file or directory
- **cat** and **text**: display the contents of files (**text** works on compressed files)
- **tail**: show the last 1KB of a file's contents

- **get** and **copyToLocal**: identical commands that copy a file from HDFS to the local file system
- **put** and **copyFromLocal**: identical commands that copy a file from the local file system into HDFS
- **getmerge**: gets a collection of files and merges them into a single file
- **mv**: move a file in HDFS
- **cp**: copy a file to another location in HDFS
- **mkdir**: make a new directory in HDFS
- **rm**: remove a file (to the **Trash** folder)
- **rm -R**: remove folders and recursively remove any files and subfolders (to the **Trash** folder)
- **test**: checks if a file exists
- **touchz**: writes a timestamp into a new file

Examples of HDFS Commands

The following **mkdir** command makes a new directory named **mydata**:

```
hadoop fs -mkdir mydata
```

This **put** command copies a local file named **numbers.txt** into **mydata** in HDFS:

```
hadoop fs -put numbers.txt mydata/
```

Use the **ls** command to view the contents of the **mydata** folder:

```
# hadoop fs -ls mydata
Found 1 items
-rw-r--r-- 3 root root 2549 2013-08-29 mydata/numbers.txt
```

NOTE: The logs for HDFS are, by default, in the **/var/log/hadoop/hdfs** folder. Hadoop uses **log4j** via the Apache Commons Logging framework for logging.

HDFS File Permissions

- File and directories have owners and groups
- r = read
- w = write
- x = permission to access the contents of a directory

```
drwxr-xr-x  - hue      hue          0 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/shell  
-rwxr-xr-x  3 hue      hue          77 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/shell/hello.py  
drwxr-xr-x  - hue      hue          0 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/sleep  
-rwxr-xr-x  3 hue      hue          0 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/sleep/empty  
drwxr-xr-x  - hue      hue          0 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/sqoop  
-rwxr-xr-x  3 hue      hue          7175 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/sqoop/TT.java  
-rwxr-xr-x  3 hue      hue          420 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/sqoop/db.hsqldb.properties  
-rwxr-xr-x  3 hue      hue          276 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/sqoop/db.hsqldb.script  
drwxr-xr-x  - hue      hue          0 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/ssh  
-rwxr-xr-x  3 hue      hue          0 2013-08-27 23:00 /user/hue/oozie/worksheets/unmanaged/ssh/empty  
drwxr-xr-x  - root     root         0 2013-08-27 03:22 /user/root  
drwxr-xr-x  - root     root         0 2013-08-29 03:23 /user/root/mydata  
-rw-r--r--  3 root     root        2549 2013-08-29 03:23 /user/root/mydata/numbers.txt  
-rw-r--r--  3 root     root       3613198 2013-08-28 21:55 /user/root/stocks.csv  
[root@sandbox demos]#
```

© Hortonworks Inc. 2013



HDFS File Permissions

HDFS implements a permissions model for files and directories that shares much of the POSIX model:

- Each file and directory is associated with an owner and a group.
- The file or directory has separate permissions for the user that is the owner, for other users that are members of the group, and for all other users.
- For files, the **r** permission is required to read the file, and the **w** permission is required to write or append to the file.
- For directories, the **r** permission is required to list the contents of the directory, the **w** permission is required to create or delete files or directories, and the **x** permission is required to access a child of the directory.

The output of the **ls** and **ls -R** commands shows the file permissions:

```
drwxr-xr-x  - root root          0 2013-08-29 03:23
/usr/root/mydata
-rw-r--r--  3 root root      2549 2013-08-29 03:23
/usr/root/mydata/numbers.txt
-rw-r--r--  3 root root  3613198 2013-08-28 21:55
/usr/root/stocks.csv
```

Unit 2 Review

1. Which component of HDFS is responsible for maintaining the namespace of the distributed filesystem? _____
2. What is the default file replication factor in HDFS? _____
3. True or False: To input a file into HDFS, the client application passes the data to the NameNode, which then divides the data into blocks and passes the blocks to the DataNodes. _____
4. Which property is used to specify the block size of a file stored in HDFS?

5. The NameNode maintains the namespace of the filesystem using which two sets of files? _____
6. What does the following command do?

```
hadoop fs -ls -R /user/thomas/
```

7. What does the following command do?

```
hadoop fs -ls /user/thomas/
```

Lab 2.1: Using HDFS Commands

Objective:	To become familiar with how files are added to and removed from HDFS, and how to view files in HDFS.
Location of Files:	/root/labs/Lab2.1
Successful Outcome:	You will have added and deleted several files and folders in HDFS.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: View the **hadoop fs** Command

- 1.1.** From the command line, enter the following command to view the usage of **hadoop fs**:

```
# hadoop fs
```

- 1.2.** Notice the usage contains options for performing file system tasks in HDFS, like copying files from a local folder into HDFS, retrieving a file from HDFS, copying and moving files around, and making and removing directories. In this lab, you will perform these commands and many others, to help you become comfortable with working with HDFS.

Step 2: Create a Directory in HDFS

- 2.1.** Enter the following **-ls** command to view the contents of the user's root directory in HDFS, which is **/user/root**:

```
# hadoop fs -ls
```

You do not have any files in **/user/root** yet, so no output is displayed.

2.2. Run the **-ls** command again, but this time specify the root HDFS folder:

```
# hadoop fs -ls /
```

The output should look like:

```
Found 7 items
drwxrwxrwt  - yarn    hadoop  0 2014-01-08 02:34 /app-logs
drwxr-xr-x  - hdfs    hdfs   0 2013-10-20 15:08 /apps
drwxr-xr-x  - mapred  hdfs   0 2013-10-20 15:10 /mapred
drwxr-xr-x  - hdfs    hdfs   0 2013-10-20 15:10 /mr-history
drwxr-xr-x  - root    hdfs   0 2014-01-10 15:35 /test
drwxrwxrwx  - hdfs    hdfs   0 2014-01-09 23:54 /tmp
drwxr-xr-x  - hdfs    hdfs   0 2014-01-10 15:26 /user
```

IMPORTANT: Notice how adding the **/** in the **-ls** command caused the contents of the root folder to display, but leaving off the **/** showed the contents of **/user/root**, which is the default prefix if you leave off the leading **/** on any of the **hadoop** commands.

2.3. Enter the following command to create a directory named **test** in HDFS:

```
# hadoop fs -mkdir test
```

2.4. Verify the folder was created successfully:

```
# hadoop fs -ls
Found 1 items
drwxr-xr-x  - root    root   0 2013-08-29 03:51 test
```

2.5. Create a couple of subdirectories of **test**:

```
# hadoop fs -mkdir test/test1
# hadoop fs -mkdir test/test2
# hadoop fs -mkdir test/test2/test3
```

2.6. Use the **-ls** command to view the contents of **/user/root**:

```
# hadoop fs -ls
```

Notice you only see the **test** directory. To recursively view the contents of a folder, use **-ls -R**:

```
# hadoop fs -ls -R
```

The output should look like:

```
drwxr-xr-x    - root root  0 2013-08-29 03:54 test
drwxr-xr-x    - root root  0 2013-08-29 03:52 test/test1
drwxr-xr-x    - root root  0 2013-08-29 03:54 test/test2
drwxr-xr-x    - root root  0 2013-08-29 03:54
                  test/test2/test3
```

Step 3: Delete a Directory

3.1. Delete the **test2** folder (and recursively its subcontents) using the **-rm -R** command:

```
# hadoop fs -rm -R test/test2
```

3.2. Now run the **-ls -R** command:

```
# hadoop fs -ls -R
```

The directory structure of the output should look like:

```
.Trash
.Trash/Current
.Trash/Current/user
.Trash/Current/user/root
.Trash/Current/user/root/test
.Trash/Current/user/root/test/test2
.Trash/Current/user/root/test/test2/test3
test
test/test1
```

NOTE: Notice Hadoop created a **.Trash** folder for the **root** user and moved the deleted content there. The **.Trash** folder empties automatically after a configured amount of time.

Step 4: Upload a File to HDFS

4.1. Now let's put a file into the **test** folder. Change directories to **/root/labs/Lab2.1**:

```
# cd ~/labs/Lab2.1/
```

4.2. Notice this folder contains a file named **data.txt**:

```
# tail data.txt
```

4.3. Run the following **-put** command to copy **data.txt** into the **test** folder in HDFS:

```
# hadoop fs -put data.txt test/
```

4.4. Verify the file is in HDFS by listing the contents of **test**:

```
# hadoop fs -ls test
```

The output should look like the following:

```
Found 2 items
-rw-r--r--    3 root root 1529355 2013-08-29  test/data.txt
drwxr-xr-x    - root root      0 2013-08-29  test/test1
```

Step 5: Copy a File in HDFS

5.1. Now copy the **data.txt** file in **test** to another folder in HDFS using the **-cp** command:

```
# hadoop fs -cp test/data.txt test/test1/data2.txt
```

5.2. Verify the file is in both places by using the **-ls -R** command on **test**. The output should look like the following:

```
# hadoop fs -ls -R test
-rw-r--r--    3 root root      1529355 test/data.txt
drwxr-xr-x    - root root      0 test/test1
-rw-r--r--    3 root root      1529355 test/test1/data2.txt
```

5.3. Now delete the **data2.txt** file using the **-rm** command:

```
# hadoop fs -rm test/test1/data2.txt
```

5.4. Verify the **data2.txt** file is in the **.Trash** folder.

Step 6: View the Contents of a File in HDFS

6.1. You can use the **-cat** command to view text files in HDFS. Enter the following command to view the contents of **data.txt**:

```
# hadoop fs -cat test/data.txt
```

6.2. You can also use the **-tail** command to view the end of a file:

```
# hadoop fs -tail test/data.txt
```

Notice the output this time is only the last 20 rows of **data.txt**.

Step 7: Getting a File from HDFS

7.1. See if you can figure out how to use the **get** command to copy **test/data.txt** from HDFS into your local **/tmp** folder.

Step 8: The **getmerge** Command

8.1. Put the file **/root/labs/demos/small_blocks.txt** into the **test** folder in HDFS. You should now have two files in test: **data.txt** and **small_blocks.txt**.

8.2. Run the following **getmerge** command:

```
# hadoop fs -getmerge test /tmp/merged.txt
```

8.3. What did the previous command do? Open the file **merged.txt** to see what happened?

Step 9: Specify the Block Size and Replication Factor

9.1. Put **/root/labs/Lab2.1/data.txt** into **/user/root** in HDFS, giving it a blocksize of 1048576 bytes. **HINT:** The blocksize is defined using the **dfs.blocksize** property on the command line.

9.2. Run the following **fsck** command on **data.txt**:

```
# hdfs fsck /user/root/data.txt
```

9.3. How many blocks are there for this file? _____

RESULT: You should now be comfortable with executing the various HDFS commands, including creating directories, putting files into HDFS, copying files out of HDFS, and deleting files and folders.

ANSWERS:

Step 7.1:

```
hadoop fs -get test/data.txt /tmp/
cd /tmp
ls
```

Step 8.1:

```
hadoop fs -put /root/labs/demos/small_blocks.txt test/
```

Step 8.3: The two files that were in the **test** folder in HDFS were merged into a single file and stored on the local file system.

Step 9.1:

```
hadoop fs -D dfs.blocksize=1048576 -put data.txt data.txt
```

Step 9.3: The file should be broken down into 2 blocks.

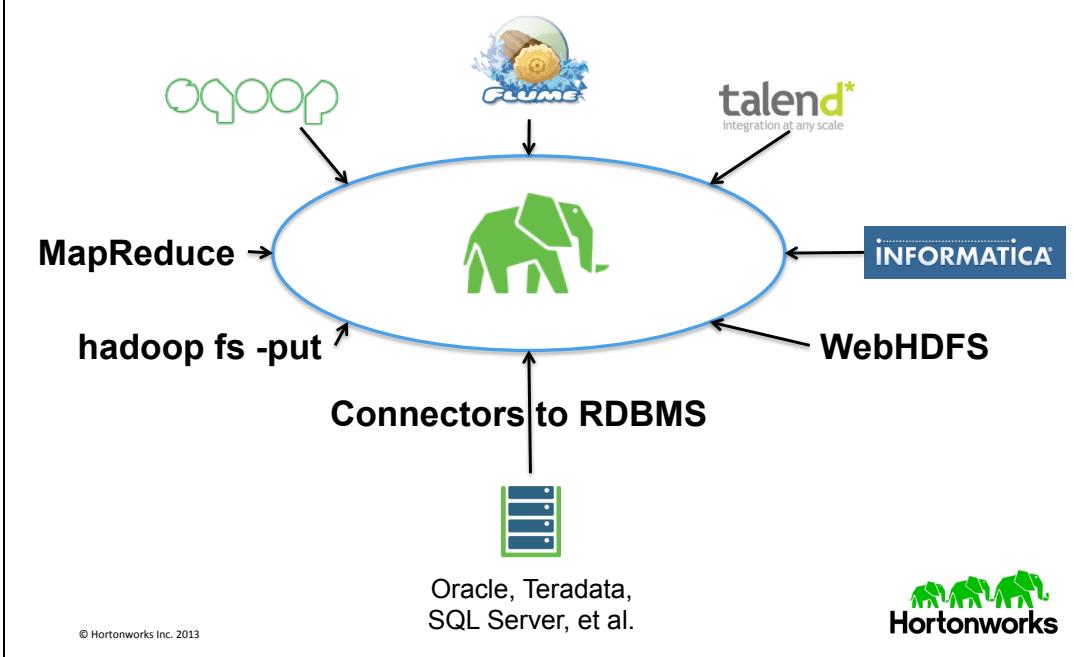
NOTES

Unit 3: Inputting Data into HDFS

Topics covered:

- Options for Data Input
- The Hadoop Client
- WebHDFS
- Overview of Flume
- A Flume Example
- Overview of Sqoop
- The Sqoop Import Tool
- Importing a Table
- Import Specific Columns
- Importing from a Query
- The Sqoop Export Tool
- Exporting to a Table
- Lab 3.1: Importing RDBMS Data into HDFS
- Lab 3.2: Exporting HDFS Data to a RDBMS

Options for Data Input



Options for Data Input

Typically the first task in using a Hadoop cluster is getting your Big Data into HDFS. You have several options to choose from, and typically you may need to use more than one tool depending on the sources of your Big Data.

In this Unit, we will discuss some of the common techniques for inputting data into a Hadoop cluster.

BEST PRACTICE: When putting data into Hadoop, do not forget one of the essentials of Hadoop: *no schema is applied when the data goes in*. In other words, keep your Big Data in its raw format, and worry about applying structure and schema to it later when you transform and analyze the data.

The Hadoop Client

- The **put** Command
 - Same as **copyFromLocal**
- Perfect for inputting local files into HDFS
 - Useful in batch scripts
- Usage:

```
hadoop fs -put <localsrc> ... <dst>
```

© Hortonworks Inc. 2014



The Hadoop Client

As you have already seen, the **hadoop** client works well for inputting files from a local file system into HDFS.

```
Usage: hadoop fs -put <localsrc> ... <dst>
```

Obviously you do not have a 2 Petabyte file sitting around on your local hard drive that you want to store into HDFS, but the **put** command is still an extremely useful tool that you will use on a regular basis when doing development.

NOTE: The **put** command also reads input from **stdin** and writes to a specified file in HDFS. Just use a dash “-” for the localsrc:

```
# hadoop fs -put - myinput.txt
```

WebHDFS

- REST API for accessing all of the HDFS file system interfaces:
 - `http://host:port/webhdfs/v1/test/mydata.txt?op=OPEN`
 - `http://host:port/webhdfs/v1/user/root/data?op=MKDIRS`
 - `http://host:port/webhdfs/v1/test/mydata.txt?op=APPEND`

© Hortonworks Inc. 2013



WebHDFS

WebHDFS is a REST API for accessing all of the HDFS file system interfaces. WebHDFS supports all HDFS user operations including reading files, writing to files, making directories, changing permissions and renaming. With WebHDFS, you can use common tools like `curl`, `wget`, or any Web Services client to access the files in a Hadoop cluster.

Some of the features of WebHDFS include:

- **Secure Authentication:** uses Kerberos (SPNEGO) and Hadoop delegation tokens for authentication.
- **Data Locality:** The file read and file write calls are redirected to the corresponding datanodes. It uses the full bandwidth of the Hadoop cluster for streaming data.
- **Built-in to Hadoop:** runs inside NameNode and DataNodes, so are no additional servers to install.

The syntax for an HTTP request looks like:

```
http://host:port/webhdfs/v1/<PATH>?op=...
```

For example, the following GET request reads a file named **/test/mydata.txt**:

```
http://host:port/webhdfs/v1/test/mydata.txt?op=OPEN
```

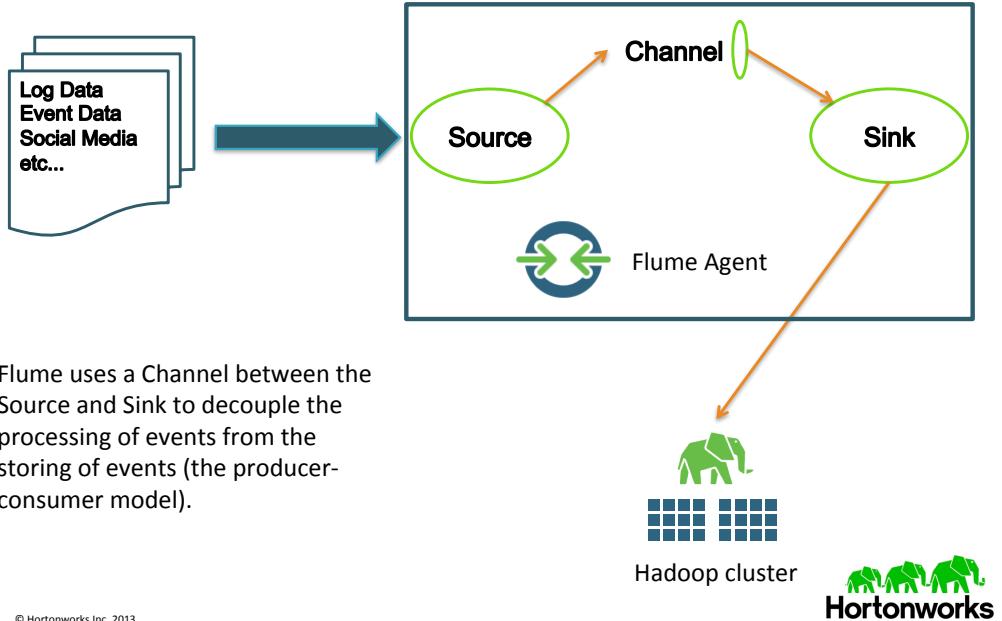
The following PUT request makes a new directory in HDFS named **/user/root/data**:

```
http://host:port/webhdfs/v1/user/root/data?op=MKDIRS
```

The following is a POST request that appends the posted data to the file named **/test/mydata.txt**:

```
http://host:port/webhdfs/v1/test/mydata.txt?op=APPEND
```

Overview of Flume



Overview of Flume

Flume is an open-source Apache project that is a system for efficiently collecting, aggregating and moving large amounts of log data from many different sources into HDFS. You can also customize Flume to work with network traffic data, social-media-generated data, email messages and pretty much any data source possible.

Flume uses a producer-consumer model for handling events where the **Source** is the producer and the **Sink** is the consumer of the events. Examples of a Source include:

- System log files
- Network traffic log files
- Website traffic logs
- Twitter feeds and other social media sources

The events travel through an asynchronous Channel to a Sink. Examples of a Sink include:

- HDFS
- HBase

- Event Serializer: which converts the event data into a custom format and writes to an output stream

A **Channel** drains into a Sink, but because it is asynchronous the Channel is not required to send events to the Sink at the same rate that it receives them from the Source. This allows for a Source to not have to wait for Flume to store the event in its final destination, which can improve performance by decoupling the Sink from the Source.

NOTE: A Flume process can consist of more than one Agent with a single Source and Sink. You can have multiple Agents that aggregate data from multiple Sources, and you can configure multiple Sinks that output events to different destinations.

A Flume Example

```
agent.sources = webserver
agent.channels = memoryChannel
agent.sinks = mycluster

agent.sources.webserver.type = exec
agent.sources.webserver.command = tail -F /var/log/hadoop/hdfs/hdfs-audit.log
agent.sources.webserver.batchSize = 1
agent.sources.webserver.channels = memoryChannel

agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 10000

agent.sinks.mycluster.type = hdfs
agent.sinks.mycluster.channel = memoryChannel
agent.sinks.mycluster.hdfs.path = hdfs://127.0.0.1:8020/hdfsaudit/
```



© Hortonworks Inc. 2013

A Flume Example

To use Flume, you start an **Agent**. An Agent has a configuration file associated with it that defines its Sources and Sinks. The command to start an Agent looks like:

```
flume-ng agent -n my_agent -c conf -f myagent.conf
```

where **myagent.conf** is the configuration file.

The following Agent config file demonstrates streaming a Web server's log file into HDFS as a sequence file:

```
agent.sources = webserver
agent.channels = memoryChannel
agent.sinks = mycluster

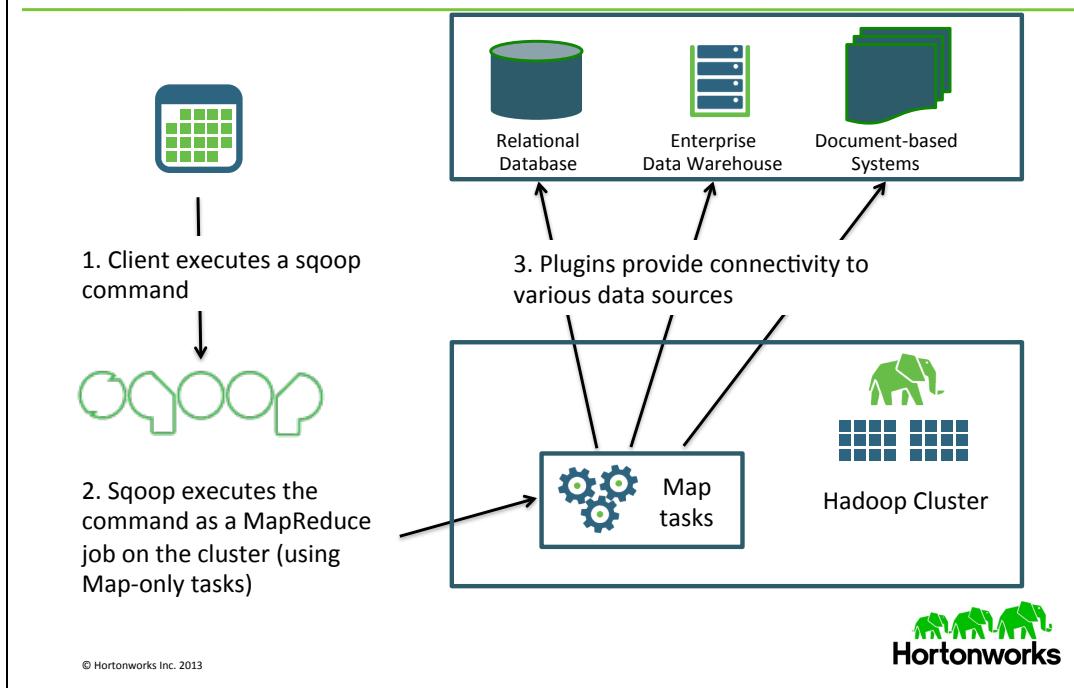
agent.sources.webserver.type = exec
agent.sources.webserver.command = tail -F
    /var/log/hadoop/hdfs/hdfs-audit.log
agent.sources.webserver.batchSize = 1
agent.sources.webserver.channels = memoryChannel

agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 10000
```

```
agent.sinks.mycluster.type = hdfs
agent.sinks.mycluster.channel = memoryChannel
agent.sinks.mycluster.hdfs.path =
hdfs://127.0.0.1:8020/hdfsaudit/
```

- The names of the Sink, Source and Channel are arbitrary.
- This Flume Agent has one Source named **webserver**.
- The **webserver** Source is of type **exec**, which means it executes a given Unix command. In this example, it executes the **tail** command on the httpd access log file.
- The Agent has one Sink named **mycluster**, which sends the events to a sequence file in a specified folder in HDFS.
- The Agent has one Channel named **memoryChannel**.
- The **memoryChannel** is configured with a memory type, which means it stores the events in memory, and notice it is configured with a capacity of 10,000. No more than 10,000 events can fit in this Channel.
- Other options for a Channel include a database, a file, or you can define your own custom Channel.
- Other options for a Sink include a system log (as INFO events), an IRC destination, local files, HBase, and Elastic Search.

Overview of Sqoop



Overview of Sqoop

Sqoop is a tool designed to transfer data between Hadoop and external structured datastores like RDBMS and data warehouses. Using Sqoop, you can provision the data from an external system into HDFS. Sqoop uses a connector-based architecture that supports plugins that provide connectivity to additional external systems.

As you can see in the slide, Sqoop uses MapReduce to distribute its work across the Hadoop cluster:

1. A Sqoop job gets executed using the **sqoop** command line.
2. Sqoop uses Map tasks (4 by default) to execute the command.
3. Plugins are used to communicate with the outside data source. The schema is provided by the data source, and Sqoop generates and executes SQL statements using JDBC or other connectors.

NOTE: Using MapReduce to perform Sqoop commands provides parallel operation as well as fault tolerance.

HDP provides the following connectors for Sqoop:

- **Teradata**
- **MySQL**
- **Oracle JDBC connector**
- **Netezza**

A Sqoop connector for SQL Server is also available from Microsoft:

- **SQL Server R2 connector**

The Sqoop Import Tool

- The **import** command has the following requirements:
 - Must specify a connect string using the **--connect** argument
 - Credentials can be included in the connect string, so using the **--username** and **--password** arguments
 - Must specify either a table to import using **--table**, or the result of a SQL query using **--query**

© Hortonworks Inc. 2013



The Sqoop Import Tool

With Sqoop, you can import data from a relational database system into HDFS:

- The input to the import process is a database table.
- Sqoop will read the table row-by-row into HDFS. The output of this import process is a set of files containing a copy of the imported table.
- The import process is performed in parallel. For this reason, the output will be in multiple files.
- These files may be delimited text files (for example, with commas or tabs separating each field), or binary Avro or SequenceFiles containing serialized record data.

The **import** command looks like:

```
sqoop import (generic-args) (import-args)
```

The **import** command has the following requirements:

- Must specify a connect string using the **--connect** argument
- Credentials can be included in the connect string, so using the **--username** and **--password** arguments
- Must specify either a table to import using **--table**, or the result of a SQL query using **--query**

Importing a Table

```
sqoop import  
--connect jdbc:mysql://host/nyse  
--table StockPrices  
--target-dir /data/stockprice/  
--as-textfile
```

© Hortonworks Inc. 2013



Importing a Table

The following Sqoop command imports a database table named **StockPrices** into a folder in HDFS named **/data/stockprices**:

```
sqoop import  
--connect jdbc:mysql://host/nyse  
--table StockPrices  
--target-dir /data/stockprice/  
--as-textfile
```

Based on the **import** command above:

- The connect string in this example is for MySQL. The database name is **nyse**.
- The **--table** argument is the name of the table in the NYSE database.
- The **--target-dir** is where in HDFS the data will be imported.
- The default number of map tasks for Sqoop is 4, so the result of this import will be in 4 files.
- The **--as-textfile** argument imports the data as plain text.

NOTE: You can use **--as-avrodatafile** to import the data to Avro files, and use **--as-sequencefile** to import the data to sequence files.

Other useful import arguments include:

- **--columns**: a comma-separated list of the columns in the table to import (as opposed to importing all columns, which is the default behavior).
- **--fields-terminated-by**: specify the delimiter. Sqoop uses a comma by default.
- **--append**: the data is appended to an existing dataset in HDFS.
- **--split-by**: the column used to determine how the data is split between mappers. If you do not specify a split-by column, then the primary key column is used.
- **-m**: the number of map tasks to use.
- **--query**: use instead of **--table**, the imported data is the resulting records from the given SQL query.
- **--compress**: enables compression.
- **--direct**: Sqoop will attempt the direct import channel, which may be higher performance than using JDBC.

NOTE: The **import** command shown here looks like it entered over multiple lines, but you have to enter this entire Sqoop command on a single command line.

REFERENCE: Visit <http://sqoop.apache.org/docs/1.4.2/SqoopUserGuide.html> for a list of all arguments available for the import command.

Importing Specific Columns

```
sqoop import  
--connect jdbc:mysql://host/nyse  
--table StockPrices  
--columns StockSymbol,Volume,  
High,ClosingPrice  
--target-dir /data/dailyhighs/  
--as-textfile  
--split-by StockSymbol  
-m 10
```

© Hortonworks Inc. 2013



Importing Specific Columns

Use the **--columns** argument to specify which columns from the table to import. For example:

```
sqoop import  
--connect jdbc:mysql://host/nyse  
--table StockPrices  
--columns StockSymbol,Volume,High,ClosingPrice  
--target-dir /data/dailyhighs/  
--as-textfile  
--split-by StockSymbol  
-m 10
```

Based on the **import** command above:

- How many columns will be imported? _____
- How many files will be created in **/data/dailyhighs/**? _____
- Which column will Sqoop use to split the data up between the mappers?

NOTES

Importing from a Query

```
sqoop import  
--connect jdbc:mysql://host/nyse  
--query "SELECT * FROM StockPrices s  
WHERE s.Volume >= 1000000  
AND \$CONDITIONS"  
--target-dir /data/highvolume/  
--as-textfile  
--direct  
--split-by StockSymbol
```

© Hortonworks Inc. 2014



Importing from a Query

Use the **--query** argument to specify which rows to select from a table. For example:

```
sqoop import  
--connect jdbc:mysql://host/nyse  
--query "SELECT * FROM StockPrices s  
WHERE s.Volume >= 1000000  
AND \$CONDITIONS"  
--target-dir /data/highvolume/  
--as-textfile  
--split-by StockSymbol
```

Based on the command above:

- Only rows whose **Volume** column is greater than 1,000,000 will be imported.
- The **\\$CONDITIONS** token must appear somewhere in the **WHERE** clause of your SQL query. Sqoop replaces this token with **LIMIT** and **OFFSET** clauses so that the data can be split between mappers.
- If you use **--query**, then you must also specify a **--split-by** column or the Sqoop command will fail to execute.

NOTE: Using **--query** is limited to simple queries where there are no ambiguous projections and no **OR** conditions in the **WHERE** clause. Use of complex queries (such as queries that have sub-queries, or joins leading to ambiguous projections) can lead to unexpected results.

IMPORTANT: You either use **--query** or **--table**, but attempting to define both results in an error.

The Sqoop Export Tool

- The export command transfers data from HDFS to a database:
 - Use **--table** to specify the database table
 - Use **--export-dir** to specify the data to export
- Rows are appended to the table by default
- If you define **--update-key**, then existing rows will be updated with the new data
- Use **--call** to invoke a stored procedure (instead of specifying the **--table** argument)

© Hortonworks Inc. 2013



The Sqoop Export Tool

Sqoop's export process will read a set of delimited text files from HDFS in parallel, parse them into records, and insert them as new rows in a target database table. The syntax for the **export** command is:

```
sqoop export (generic-args) (export-args)
```

The Sqoop **export** tool runs in three modes:

1. **Insert Mode:** the records being exported are inserted into the table using a SQL INSERT statement.
2. **Update Mode:** an UPDATE SQL statement is executed for existing rows, and an INSERT can be used for new rows.
3. **Call Mode:** a stored procedure is invoked for each record.

The mode used is determined by the arguments specified:

- **--table:** the table to populate in the database. This table must already exist in the database. If no **--update-key** is defined, then the command is executed in Insert Mode.

- **--update-key**: the primary key column for supporting updates. If you define this argument, the Update Mode is used and existing rows are updated with the exported data.
- **--call**: invokes a stored procedure for every record, thereby using Call Mode. If you define **--call**, then do not define the **--table** argument or an error will occur.

The following are **sqoop export** arguments:

- **--export-dir**: the directory in HDFS that contains the data to export.
- **--input-fields-terminated-by**: the input field delimiter. A comma is the default.
- **--update-mode**: Specify how updates are performed when new rows are found with non-matching keys in database. Values are **updateonly** (the default) and **allowinsert**.

Exporting to a Table

```
sqoop export  
--connect jdbc:mysql://host/mylogs  
--table LogData  
--export-dir /data/logfiles/  
--input-fields-separated-by "\t"
```

© Hortonworks Inc. 2014



Exporting to a Table

The following Sqoop command exports the data in the **/data/logfiles/** folder in HDFS to a table named **LogData**:

```
sqoop export  
--connect jdbc:mysql://host/mylogs  
--table LogData  
--export-dir /data/logfiles/  
--input-fields-separated-by "\t"
```

Based on the command above:

- The table **LogData** needs to already exist in the **Weblogs** database.
- The column values are determined by the delimiter, which is a tab in this example.
- All files in the **/data/logfiles/** directory will be exported.
- Sqoop will perform this job using 4 mappers, but you can specify the number to use with the **-m** argument.

Unit 3 Review

1. What tool would work best for importing a Twitter feed into HDFS?

2. What tool would work best for importing data from a relational database into HDFS? _____
3. What tool would work best for putting a file on your local filesystem into HDFS?

4. List the three main components of a typical Flume agent: _____

5. What is the default number of map tasks for a Sqoop job? _____
6. How do you specify a different number of mappers in a Sqoop job?

7. What is the purpose of the **\$CONDITIONS** value in the **WHERE** clause of a Sqoop query?

Lab 3.1: Importing RDBMS Data into HDFS

Objective:	Import data from a database into HDFS.
Location of Files:	n/a
Successful Outcome:	You will have imported data from MySQL into folders in HDFS.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Perform the following steps:

Step 1: Create a Table in MySQL

1.1. From the command prompt, change directories to **Lab3.1**:

```
# cd ~/labs/Lab3.1
```

1.2. View the contents of **salaries.txt**:

```
# tail salaries.txt
```

The comma-separated fields represent a gender, age, salary and zipcode.

1.3. Notice there is a **salaries.sql** script that defines a new table in MySQL named **salaries**. For this script to work, you need to copy **salaries.txt** into the publicly-available **/tmp** folder:

```
# cp salaries.txt /tmp
```

1.4. Now run the **salaries.sql** script using the following command:

```
# mysql test < salaries.sql
```

Step 2: View the Table

- 2.1.** To verify the table is populated in MySQL, open the **mysql** prompt:

```
# mysql
```

- 2.2.** Switch to the **test** database, which is where the **salaries** table was created:

```
mysql> use test;
```

- 2.3.** Run the **show tables** command and verify **salaries** is defined:

```
mysql> show tables;
+-----+
| Tables_in_test |
+-----+
| salaries      |
+-----+
1 row in set (0.00 sec)
```

- 2.4.** Select 10 items from the table to verify it is populated:

```
mysql> select * from salaries limit 10;
+-----+-----+-----+-----+-----+
| gender | age   | salary | zipcode | id   |
+-----+-----+-----+-----+-----+
| F     | 66    | 41000 | 95103  | 1    |
| M     | 40    | 76000 | 95102  | 2    |
| F     | 58    | 95000 | 95103  | 3    |
| F     | 68    | 60000 | 95105  | 4    |
| M     | 85    | 14000 | 95102  | 5    |
| M     | 14    | 0      | 95105  | 6    |
| M     | 52    | 2000  | 94040  | 7    |
| M     | 67    | 99000 | 94040  | 8    |
| F     | 43    | 11000 | 94041  | 9    |
| F     | 37    | 65000 | 94040  | 10   |
+-----+-----+-----+-----+-----+
```

- 2.5.** Exit the **mysql** prompt:

```
mysql> exit
```

Step 3: Import the Table into HDFS

- 3.1.** Enter the following Sqoop command (all on a single line), which imports the **salaries** table in the **test** database into HDFS:

```
# sqoop import  
--connect jdbc:mysql://localhost/test  
--table salaries
```

3.2. A MapReduce job should start executing, and it may take a couple minutes for the job to complete.

Step 4: Verify the Import

4.1. View the contents of your HDFS folder:

```
# hadoop fs -ls
```

4.2. You should see a new folder named **salaries**. View its contents:

```
# hadoop fs -ls salaries  
Found 4 items  
-rw-r--r-- 1 root hdfs 272 salaries/part-m-00000  
-rw-r--r-- 1 root hdfs 241 salaries/part-m-00001  
-rw-r--r-- 1 root hdfs 238 salaries/part-m-00002  
-rw-r--r-- 1 root hdfs 272 salaries/part-m-00003
```

4.3. Notice there are four new files in the **salaries** folder named **part-m-0000x**. Why are there four of these files?

4.4. Use the **cat** command to view the contents of the files. For example:

```
# hadoop fs -cat salaries/part-m-00000
```

Notice the contents of these files are the rows from the **salaries** table in MySQL. You have now successfully imported data from a MySQL database into HDFS.

Notice you imported the entire table with all of its columns. In the next step, you will import only specific columns of a table.

Step 5: Specify Columns to Import

5.1. Using the **--columns** argument, write a Sqoop command that imports the **salary** and **age** columns (in that order) of the **salaries** table into a directory in HDFS named **salaries2**. In addition, set the **-m** argument to 1 so that the result is a single file.

5.2. After the import, verify you only have one **part-m** file in **salaries2**:

```
# hadoop fs -ls salaries2
Found 1 items
-rw-r--r-- 1 root hdfs 482 salaries2/part-m-00000
```

5.3. Verify the contents of **part-m-00000** are only the 2 columns you specified:

```
# hadoop fs -cat salaries2/part-m-00000
```

The last few lines should look like the following:

```
69000.0,97
91000.0,48
0.0,1
48000.0,45
3000.0,39
14000.0,84
```

Step 6: Importing from a Query

6.1. Write a Sqoop import command that imports the rows from **salaries** in MySQL whose **salary** column is greater than 90,000.00. Use **gender** as the **--split-by** value, specify only 2 mappers, and import the data into the **salaries3** folder in HDFS.

TIP: The Sqoop command will look similar to the ones you have been using throughout this lab, except you will use **--query** instead of **--table**. Recall that when you use a **--query** command you must also define a **--split-by** column, or define **-m** to be 1.

Also, do not forget to add **\$CONDITIONS** to the **WHERE** clause of your query, as demonstrated earlier in this Unit.

6.2. To verify the result, view the contents of the files in **salaries3**. You should have only two output files.

6.3. View the contents of **part-m-00000** and **part-m-00001**. Notice one file contains females, and the other file contains males. Why? _____

6.4. Verify the output files contain only records whose **salary** is greater than 90,000.00.

RESULT: You have imported the data from MySQL to HDFS using the entire table, specific columns, and also using the result of a query.

SOLUTIONS:

Step 5.1 is the following command (entered on a single line):

```
# sqoop import --connect jdbc:mysql://localhost/test  
--table salaries  
--columns salary,age  
-m 1  
--target-dir salaries2
```

Step 6.1:

```
sqoop import --connect jdbc:mysql://localhost/test  
--query "select * from salaries s where s.salary > 90000.00  
and \$CONDITIONS"  
--split-by gender  
-m 2  
--target-dir salaries3
```

ANSWERS:

Step 4.3: The MapReduce job that executed the Sqoop command used four reducers, so there are four output files (one from each reducer).

Step 6.3: You used **gender** as the **split-by** column, so all records with the same gender are sent to the same mapper.

Lab 3.2: Exporting HDFS Data to a RDBMS

Objective:	Export data from HDFS into a MySQL table using Sqoop.
Location of Files:	/root/labs/Lab3.2
Successful Outcome:	The data in salarydata.txt in HDFS will appear in a table in MySQL named salary2 .
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Perform the following steps:

Step 1: Put the Data into HDFS

1.1. Change directories to /root/labs/Lab3.2:

```
# cd ~/labs/Lab3.2
```

1.2. View the contents of **salarydata.txt**:

```
# tail salarydata.txt
M,49,29000,95103
M,44,34000,95102
M,99,25000,94041
F,93,96000,95105
F,75,9000,94040
F,14,0,95102
M,68,1000,94040
F,45,78000,94041
M,40,6000,95103
F,82,5000,95050
```

Notice the records in this file contain 4 values separated by commas, and the values represent a **gender**, **age**, **salary** and **zip code**, respectively.

1.3. Create a new directory in HDFS named **salarydata**.

1.4. Put **salarydata.txt** into the **salarydata** directory in HDFS.

Step 2: Create a Table in the Database

2.1. There is a script in the Lab3.2 folder that creates a table in MySQL that matches the records in salarydata.txt. View the SQL script:

```
# more salaries2.sql
```

2.2. Run this script using the following command:

```
# mysql test < salaries2.sql
```

2.3. Verify the table was created successfully in MySQL:

```
# mysql
mysql> use test;
mysql> describe salaries2;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| gender | varchar(1) | YES  |     | NULL    |       |
| age    | int(11)    | YES  |     | NULL    |       |
| salary | double     | YES  |     | NULL    |       |
| zipcode | int(11)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

2.4. Exit the mysql prompt:

```
mysql> exit
```

Step 3: Export the Data

3.1. Run a Sqoop command that exports the **salarydata** folder in HDFS into the **salaries2** table in MySQL. At the end of the MapReduce output, you should see a log event stating that 10,000 records were exported.

3.2. Verify it worked by viewing the table's contents from the **mysql** prompt. The output should look like the following:

```
mysql> use test;
mysql> select * from salaries2 limit 10;
+-----+-----+-----+-----+
```

gender	age	salary	zipcode
M	57	39000	95050
F	63	41000	95102
M	55	99000	94040
M	51	58000	95102
M	75	43000	95101
M	94	11000	95051
M	28	6000	94041
M	14	0	95102
M	3	0	95101
M	25	26000	94040

RESULT: You now have used Sqoop to export data from HDFS into a database table in MySQL.

SOLUTION: The Sqoop export command in this lab looks like:

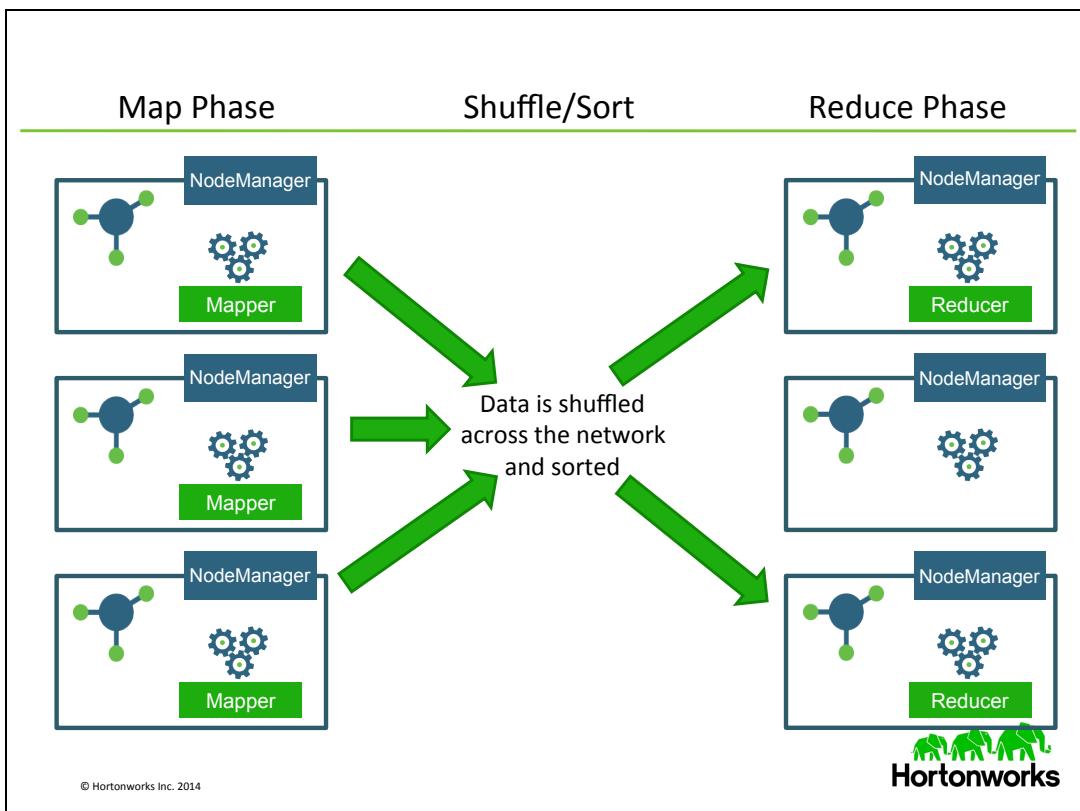
```
sqoop export  
--connect jdbc:mysql://localhost/test  
--table salaries2  
--export-dir salarydata  
--input-fields-separated-by ","
```

NOTES

Unit 4: The MapReduce Framework

Topics covered:

- Overview of MapReduce
- Understanding MapReduce
- The Key/Value Pairs of MapReduce
- WordCount in MapReduce
- Demonstration: Understanding MapReduce
- The Map Phase
- The Reduce Phase
- Lab 4.1: Running a MapReduce Job



Overview of MapReduce

MapReduce is a software framework for developing applications that process large amounts of data in parallel across a distributed environment. As its name implies, a MapReduce program consists of two main phases: a Map phase and a Reduce phase:

1. **Map phase:** data is input into the Mapper, where it is transformed and prepared for the Reducer.
2. **Reduce phase:** retrieves the data from the Mapper and performs the desired computations or analyses.

To write a MapReduce program, you define a **Mapper** class to handle the map phase and a **Reducer** class to handle the Reduce phase.

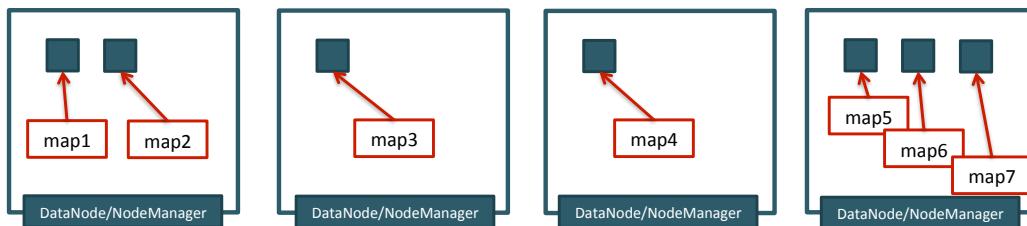
NOTE: The shuffle/sort phase of MapReduce is a part of the framework, so it does not require any programming on your part.

Some important concepts to understand about MapReduce:

- The map and reduce tasks run in their own JVM on the DataNodes.
- The Mapper inputs key/value pairs from HDFS files and outputs intermediate key/value pairs. The data types of the input and output pairs can be different.
- After all of the Mappers finish executing, the intermediate key/value pairs go through a shuffle and sort phase where all the values that share a key are combined and sent to the same Reducer.
- The Reducer inputs the intermediate <key, value> pairs and outputs its own <key, value> pairs, which are typically written to HDFS.
- The number of Mappers is determined by the Input Format.
- The number of Reducers is determined by the MapReduce job configuration.
- A **Partitioner** is used to determine which <key, value> pairs are sent to which Reducer.
- A **Combiner** can be optionally configured to combine the output of the Mapper, which can increase performance by decreasing the network traffic of the shuffle and sort phase.

Understanding MapReduce

1. Suppose a file is the input to a MapReduce job. That file is broken down into blocks stored on DataNodes across the Hadoop cluster.



2. During the Map phase, map tasks process the input of the MapReduce job, with a map task assigned to each Input Split. The map tasks are Java processes that ideally run on the DataNodes where the blocks are stored.

© Hortonworks Inc. 2014



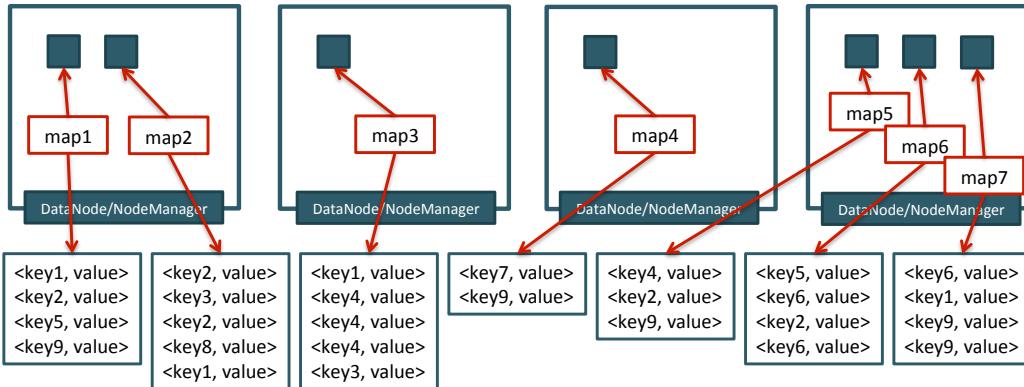
Understanding MapReduce

The Map Phase involves running map tasks on NodeManagers. The main purpose of the map phase is to read all the input data. The goal (in order to gain the best performance) is to achieve data locality, where a map task runs on a DataNode where its Input Split (or at least most of the split) is stored.

- A block of data rarely maps exactly to an Input Split, but it is often close, especially when processing text data. Records that spill over to a subsequent block have to be pulled over the network so the map task can process the entire record, but this is normally an acceptable overhead.
- The number of map tasks in a MapReduce job is based on the number of Input Splits.
- If no NodeManager is available where a specific block resides, then you lose data locality and the block has to be pulled across the network.

Understanding MapReduce - cont.

3. Each map tasks processes its Input Split and outputs records of <key, value> pairs.



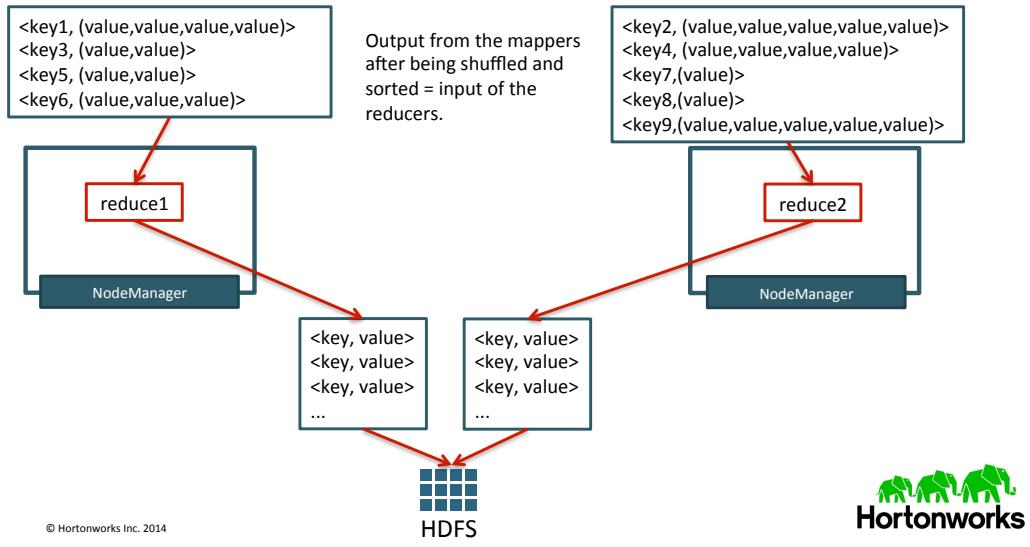
4. The <key,value> pairs go through a shuffle/sort phase, where records with the same key end up at the same reducer. The specific pairs sent to a reducer are sorted by key, and the values are aggregated into a collection.



- Map tasks output <key, value> pairs, which are written to a temporary file on the local filesystem.
- When a map task finishes, its output becomes immediately available to the reduce tasks. Each Reducer asks each mapper for the <key, value> pairs designated for that Reducer. This designating of records is called **partitioning**.
- As a Reducer reads in its <key, value> pairs, the values are aggregated into a collection, and the entire input to the Reducer is sorted by keys. This is referred to as the **shuffle/sort phase**.

Understanding MapReduce - cont.

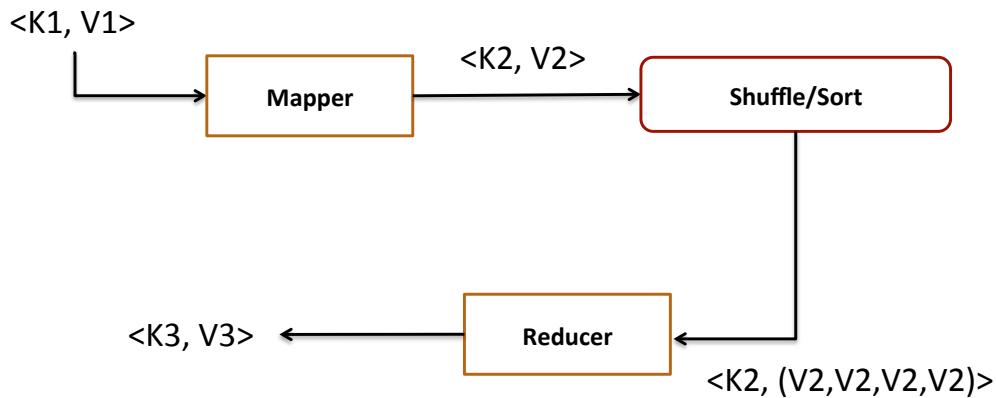
5. Reduce tasks run on a NodeManager as a Java process.
Each Reducer processes its input and outputs `<key,value>` pairs that are typically written to a file in HDFS.



The main purpose of the reduce phase is typically business logic - going through the data output by the mappers and answering a question or solving a problem. The `<key, value>` pairs coming in to the reducer are combined by key, meaning each key is presented once to the reducer, along with all of the values that belong to that key.

- Reducers also output `<key, value>` pairs.
- The output of a Reducer is typically a file in HDFS. For example, if you have 5 Reducers, the output will be 5 different files.
- The number of reduce tasks in a MapReduce job is a setting that you get to choose.

The Key/Value Pairs of MapReduce



© Hortonworks Inc. 2013



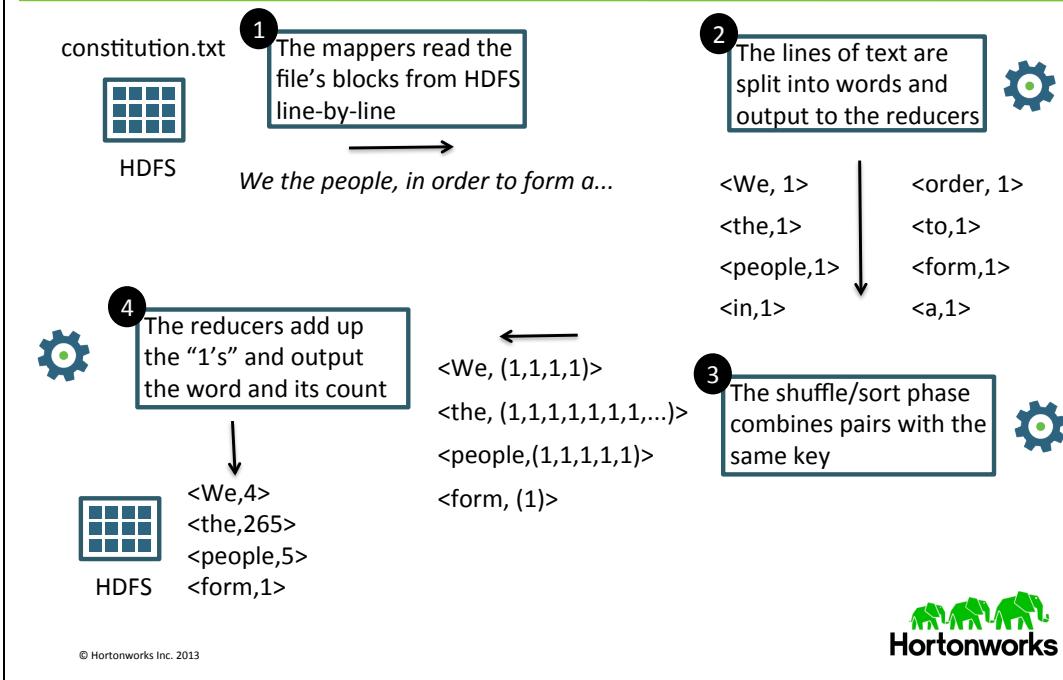
The Key/Value Pairs of MapReduce

The data types of the $\langle \text{key}, \text{value} \rangle$ pairs in a MapReduce job look like:

1. $\langle K1, V1 \rangle$: input to the Mapper
2. $\langle K2, V2 \rangle$: output from the Mapper
3. $\langle K2, \text{Iterable}<V2> \rangle$: input to the Reducer
4. $\langle K3, V3 \rangle$: output from the Reducer

NOTE: Keys are constantly being compared and sorted in MapReduce, and both keys and values get serialized and deserialized between the map and reduce phases.

WordCount in MapReduce



WordCount in MapReduce

The “Hello, World” of Hadoop programming is the word count application, which reads in a text file and counts the number of occurrences of each distinct word.

The diagram above shows how the `<key,value>` pairs of the word count application are passed through the MapReduce job.

Demonstration: Understanding MapReduce

Objective:	To understand how MapReduce works.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Put the File into HDFS

- 1.1. Change directories to the **demos** folder:

```
# cd /root/labs/demos
```

- 1.2. Notice a file named **constitution.txt**:

```
# more constitution.txt
```

- 1.3. Put the file into HDFS:

```
# hadoop fs -put constitution.txt constitution.txt
```

Step 2: Run the WordCount Job

- 2.1. The following command runs a wordcount job on the **constitution.txt** and writes the output to **wordcount_output**:

```
# hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.2.0.2.0.6.0-76.jar wordcount constitution.txt wordcount_output
```

- 2.2. Notice a MapReduce job gets submitted to the cluster. Wait for the job to complete.

Step 3: View the Results

3.1. View the contents of the **wordcount_output** folder:

```
# hadoop fs -ls wordcount_output
```

You should see a single file named **part-r-00000**:

```
Found 1 items  
-rw-r--r-- 1 root hdfs 17054 2013-08-29 21:57  
wordcount_output/part-r-00000
```

3.2. Why is there one file in this directory? _____

3.3. What does the “r” in the filename stand for? _____

3.4. View the contents of **part-r-00000**:

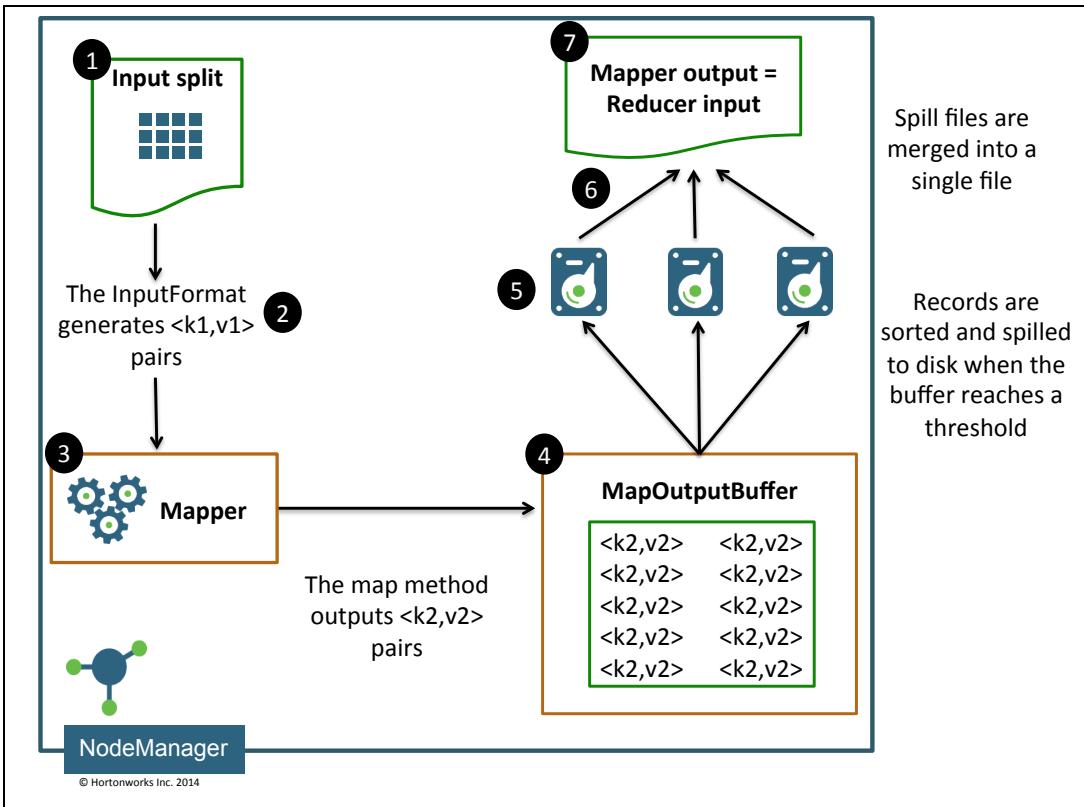
```
# hadoop fs -cat wordcount_output/part-r-00000
```

3.5. Why are the words sorted alphabetically? _____

3.6. What was the key output by the WordCount reducer? _____

3.7. What was the value output by the WordCount reducer? _____

3.8. Based on the output of the reducer, what do you think the mapper output as key/value pairs? _____



The Map Phase

The data is passed into the Mapper as a **<key, value>** pair generated by an *InputFormat* instance. The key and value are determined by the specific *InputFormat* that you configure.

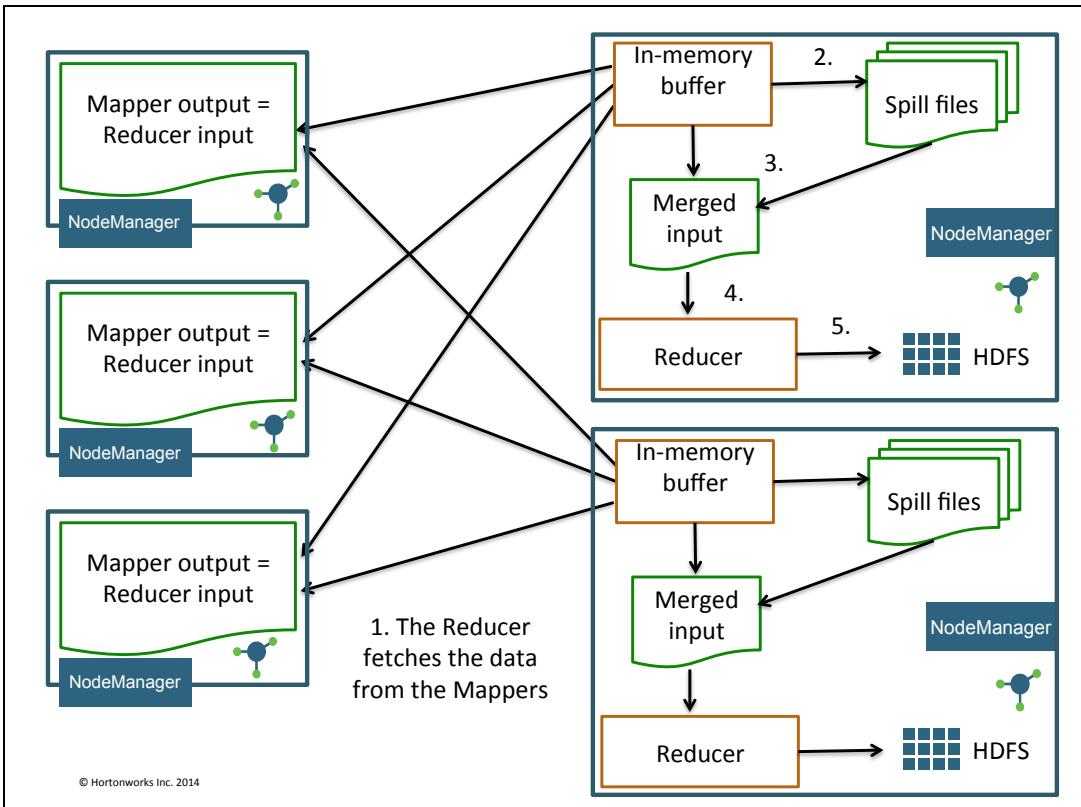
Here is how data flows through the map phase:

1. The *InputFormat* determines where the input data needs to be *split* between the Mappers, and then it generates an *InputSplit* instance for each split.
2. MapReduce spawns a map task for each *InputSplit* generated by the *InputFormat*.
3. Each **<key, value>** pair generated by the *InputFormat* is passed to the *map* method of the *Mapper* class.
4. The *map* method outputs a **<key, value>** pair that is serialized into an unsorted buffer in memory.
5. When the buffer fills up, or when the map task is complete, the **<key, value>** pairs in the buffer are sorted, then *spilled* to the disk.

6. If more than one spill file was created, these files are merged into a single file of sorted <key, value> pairs.
7. The sorted records in the spill file wait to be retrieved by a Reducer.

NOTE: The size of the Mapper's output memory buffer is configurable with the **io.sort.mb** property. A spill occurs when the buffer reaches a certain capacity configured by the **io.sort.spill.percent** property.

IMPORTANT: Spilling to disk cannot be entirely avoided because there is always one spill to disk when the Mapper is complete. However, the ideal scenario is to avoid any intermediate spills. If an intermediate spill occurs, those <key, value> pairs need to be written to disk, then read and re-written one more time, which results in 3-times the disk I/O for those spilled records!



The Reduce Phase

The Reducer fetches the records from the Mapper and uses them to generate and output another set of `<key, value>` pairs that are output to HDFS (or some other configurable location).

The reduce phase can actually be broken down in three phases:

1. **Shuffle:** also referred to as the *fetch* phase, this is when Reducers retrieve the output of the Mappers. All records with the same key are combined and sent to the same Reducer.
2. **Sort:** this phase happens simultaneously with the shuffle phase. As the records are fetched and merged, they are sorted by key.
3. **Reduce:** The reduce method is invoked for each key, with the records combined into an iterable collection.

Here is how data flows through the reduce phase:

1. As Mappers finish their tasks, the Reducers start fetching the records and storing them into a buffer in their JVM's memory.

2. If the buffer fills, it is spilled to disk.
3. Once all Mappers complete and the Reducer has fetched all its relevant input, all spill records are merged and sorted (along with any records still in the buffer).
4. The reduce method is invoked on the Reducer for each key.
5. The output of the reducer is written to HDFS (or wherever the output was configured to be sent).

Some comments about the reduce phase:

- All records that share the same key are sent to the same Reducer.
- During the shuffling, the records are sorted by key and the values are combined into a collection.
- The values in the collection are not sorted by default.
- The number of Reducers is determined by the **mapreduce.job.reduces** property.
- A MapReduce job does not require a Reducer. Setting the number of Reducers to 0 results in the Mapper sending its output directly to HDFS.
- A Reducer can actually start fetching the output of Mappers after the first Mappers finish (but others are still working). This is done using threads, and the number of threads is configurable with the **mapreduce.reduce.shuffle.parallelcopies** property.

Unit 4 Review

1. What are the three main phases of a MapReduce job? _____

2. Suppose the Mappers of a MapReduce job output `<key,value>` pairs that are of type `<integer,string>`. What will the pairs look like that are processed by the corresponding Reducers? _____
3. What happens if all the `<key,value>` pairs output by a Mapper do not fit into the memory of the Mapper? _____
4. What determines the number of Mappers of a MapReduce job? _____

5. What determines the number of Reducers of a MapReduce job? _____

6. True or False: The shuffle/sort phase sorts the keys and values as they are passed to the Reducer. _____

Lab 4.1: Running a MapReduce Job

Objective:	Run a Java MapReduce job.
Location of Files:	/root/labs/Lab4.1
Successful Outcome:	You will see the results of the Inverted Index job in the inverted/output folder in HDFS.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: Put the Data in HDFS

- 1.1.** The MapReduce job you are going to execute is an Inverted Index application, one of the very first use cases for MapReduce. Open a command prompt and change directories to **/root/labs/Lab4.1**:

```
# cd ~/labs/Lab4.1
```

- 1.2.** View the contents of the file **hortonworks.txt**. Each line looks like:

```
http://hortonworks.com/,hadoop,webinars,articles,download,e  
nterprise,team,reliability
```

Each line of text consists of a Web page URL, followed by a comma-separated list of keywords found on that page.

- 1.3.** Make a new folder in HDFS named **inverted/input**:

```
# hadoop fs -mkdir inverted  
# hadoop fs -mkdir inverted/input
```

1.4. Put **hortonworks.txt** into HDFS in the **inverted/input** folder. This file will be the input to the MapReduce job.

Step 2: Run the Inverted Index Job

2.1. From the **/root/labs/Lab4.1** folder, enter the following command (all on a single line):

```
# hadoop jar invertedindex.jar inverted.IndexInverterJob  
inverted/input inverted/output
```

2.2. Wait for the MapReduce job to execute. The final output should look like:

```
File Input Format Counters  
    Bytes Read=1126  
File Output Format Counters  
    Bytes Written=2997
```

Step 3: View the Results

3.1. List the contents of the **inverted/output** folder. How many reducers did this job use? _____ How can you determine this from the contents of **inverted/output**?

3.2. Use the **cat** command to view the contents of **inverted/output/part-r-00000**. The file should look like:

```
# # hadoop fs -cat inverted/output/part-r-00000  
about http://hortonworks.com/about-us/,  
apache  
http://hortonworks.com/products/hortonworksdataplatform/, h  
ttp://hortonworks.com/about-us/,  
articles  
http://hortonworks.com/community/, http://hortonworks.com/,  
...
```

Step 4: Specify the Number of Reducers

4.1. Try running the job again, but this time specify the number of Reducers to be three:

```
# hadoop jar invertedindex.jar inverted.IndexInverterJob  
-D mapreduce.job.reduces=3 inverted/input inverted/output
```

4.2. View the contents of **inverted/output**. Notice there are three **part-r** files:

```
# # hadoop fs -ls inverted/output
Found 3 items
1 root hdfs      1221 inverted/output/part-r-00000
1 root hdfs      977 inverted/output/part-r-00001
1 root hdfs      799 inverted/output/part-r-00002
```

4.3. View the contents of the three files. How did the MapReduce framework determine which <key,value> pair to send to which reducer?

RESULT: You have now executed a Java MapReduce job from the command line that takes an input text file and outputs the inverted indexes of the lines of text. This common task is what Web search engines like Google and Yahoo! use to determine the pages associated with search terms.

ANSWERS:

3.1: The job used one reducer, which you can determine by the existence of only one **part-r-n** file in the output directory.

4.3: <key,value> pairs are sent to the reducer based on the hashing of the key and using the remainder of dividing by the number of reducers.

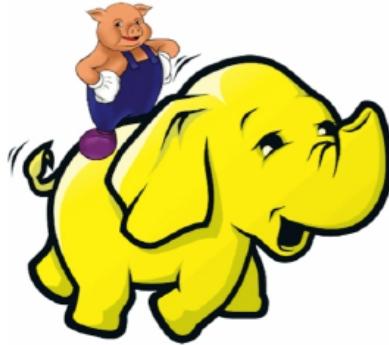
Unit 5: Introduction to Pig

Topics covered:

- What is Pig?
- Pig Latin
- The Grunt Shell
- Demonstration: Understanding Pig
- Pig Latin Relation Names
- Pig Latin Field Names
- Pig Data Types
- Pig Complex Types
- Defining a Schema
- Lab 5.1: Getting Started with Pig
- The GROUP Operator
- GROUP ALL
- Relations without a Schema
- The FOREACH GENERATE Operator
- Specifying Ranges in FOREACH
- Field Names in FOREACH
- FOREACH with Groups
- The FILTER Operator
- The LIMIT Operator
- Lab 5.2: Exploring Data with Pig

What is Pig?

- An engine for executing programs on top of Hadoop
- It provides a language, Pig Latin, to specify these programs



© Hortonworks Inc. 2013

What is Pig?

Apache Pig is a Hadoop platform for creating MapReduce jobs. Pig uses a high-level, SQL-like programming language named **Pig Latin**. The benefits of Pig include:

- Run a MapReduce job with a few simple lines of code.
- Process structured data with a schema, or Pig can process unstructured data without a schema. (Pigs eat anything!)
- Pig Latin uses a familiar SQL-like syntax.
- Pig scripts read and write data from HDFS.
- Pig Latin is a data flow language, a logical solution for many MapReduce algorithms.

NOTE: Pig was created at Yahoo! to make it easier to analyze the data in your HDFS without the complexities of writing a traditional MapReduce program.

The developers of Pig published their philosophy to summarize the goals of Pig, using comparisons to actual pigs:

- **Pigs eat anything:** Pig can process any data, structured or unstructured.
- **Pigs live anywhere:** Pig can run on any parallel data processing framework, so Pig scripts do not have to run just on Hadoop.
- **Pigs are domestic animals:** Pig is designed to be easily controlled and modified by its users.
- **Pigs fly:** Pig is designed to process data quickly!

Pig Latin

- High-level data flow scripting language
- Pig executes in a unique fashion:
 1. During execution each statement is processed by the Pig interpreter
 2. If a statement is valid, it gets added to a **logical plan** built by the interpreter
 3. The steps in the logical plan do not actually execute until a DUMP or STORE command

© Hortonworks Inc. 2012

Pig Latin

Pig Latin is a high-level data flow scripting language. Pig Latin scripts can be executed one of three ways:

- **Pig script:** write a Pig Latin program in a text file and execute it using the **pig** executable.
- **Grunt shell:** enter Pig statements manually one-at-a-time from a CLI tool known as the Grunt interactive shell.
- **Embedded in Java:** use the `PigServer` class to execute a Pig query from within Java code.

Pig executes in a unique fashion - some commands build on previous commands, while certain commands trigger a MapReduce job:

- During execution each statement is processed by the Pig interpreter.
- If a statement is valid, it gets added to a **logical plan** built by the interpreter.
- The steps in the logical plan do not actually execute until a DUMP or STORE command.

The Grunt Shell

- An interactive shell for entering Pig Latin statements
- Start it by running the **pig** executable



Grunt shell

```
rich — root@sandbox:~ — ssh — 59x5
grunt> employees = LOAD 'pigdemo.txt' AS (state, name);
grunt> describe employees;
employees: {state: bytearray, name: bytearray}
grunt> employees_grp = group employees by state;
grunt> dump employees;
```

© Hortonworks Inc. 2013



The Grunt Shell

Grunt is an interactive shell that enables users to enter Pig Latin statements and also interact with HDFS. To enter the Grunt shell, run the **pig** executable in the **PIG_HOME\bin** folder:

```
# pig
grunt>
```

The Grunt shell provides tab completion for commands (unfortunately there is no tab completion for files or folders), as well as command-line history and editing.

NOTE: You can run HDFS commands directly from the Grunt shell, and it also has the concept of a “present working directory” with the ability to change directories using the **cd** command.

Demonstration: Understanding Pig

Objective:	To understand Pig scripts and relations.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Start the Grunt Shell

1.1. Review the contents of the file **pigdemo.txt** located in **/root/labs/demos**.

1.2. Start the Grunt shell:

```
# pig
```

1.3. Notice the output includes where the logging for your Pig session will go, as well as a statement about connecting to your Hadoop file system:

```
[main] INFO org.apache.pig.Main - Logging error messages to: /root/labs/demos/pig_1377892197767.log
[main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at:
hdfs://sandbox:8020
```

Step 2: Make a New Directory

2.1. Notice you can run HDFS commands easily from the Grunt shell. For example, run the **ls** command:

```
grunt> ls
```

2.2. Make a new directory named **demos**:

```
grunt> mkdir demos
```

2.3. Use **copyFromLocal** to copy the **pigdemo.txt** file into the **demos** folder:

```
grunt> copyFromLocal /root/labs/demos/pigdemo.txt demos/
```

2.4. Verify the file was uploaded successfully:

```
grunt> ls demos
hdfs://sandbox:8020/user/root/demos/pigdemo.txt<r 1> 87
```

2.5. Change the present working directory to **demos**:

```
grunt> cd demos
grunt> pwd
hdfs://sandbox:8020/user/root/demos
```

2.6. View the contents using the **cat** command:

```
grunt> cat pigdemo.txt
SD      Rich
NV      Barry
CO      George
CA      Ulf
IL      Danielle
OH      Tom
CA      manish
CA      Brian
CO      Mark
```

Step 3: Define a Relation

3.1. Define the **employees** relation, using a schema:

```
grunt> employees = LOAD 'pigdemo.txt' AS (state, name);
```

3.2. Demonstrate the **describe** command, which describes what a relation looks like:

```
grunt> describe employees;
employees: {state: bytearray, name: bytearray}
```

NOTE: Fields have a data type, and we will discuss data types later in this Unit. Notice the default data type of a field (if you do not specify one) is **bytearray**.

3.3. Let's view the records in the **employees** relation:

```
grunt> DUMP employees;
```

Notice this requires a MapReduce job to execute, and the result is a collection of **tuples**:

```
(SD,Rich)
(NV,Barry)
(CO,George)
(CA,Ulf)
(IL,Danielle)
(OH,Tom)
(CA,manish)
(CA,Brian)
(CO,Mark)
```

Step 4: Filter the Relation by a Field

4.1. Let's filter the **employees** whose **state** field equals CA:

```
grunt> ca_only = FILTER employees BY (state=='CA');
grunt> DUMP ca_only;
```

4.2. The output is still tuples, but only the records that match the filter:

```
(CA,Ulf)
(CA,manish)
(CA,Brian)
```

Step 5: Create a Group

5.1. Define a relation that groups the **employees** by the **state** field:

```
grunt> emp_group = GROUP employees BY state;
```

5.2. Groups in Pig are represented by **bags**. A bag is an unordered collection of tuples:

```
grunt> describe emp_group;
emp_group: {group: chararray, employees: {(state: bytearray, name: bytearray)}}
```

5.3. All records with the same state will be grouped together, as shown by the output of the **emp_group** relation:

```
grunt> DUMP emp_group;
```

The output is:

```
(CA, { (CA,Ulf), (CA,manish), (CA,Brian) })
(CO, { (CO,George), (CO,Mark) })
(IL, { (IL,Danielle) })
(NV, { (NV,Barry) })
(OH, { (OH,Tom) })
(SD, { (SD,Rich) })
```

NOTE: *Tuples* are displayed in parentheses. *Bags* are represented by curly braces.

Step 6: The STORE Command

6.1. The **DUMP** command dumps the contents of a relation to the console. The **STORE** command sends the output to a folder in HDFS. For example:

```
grunt> STORE emp_group INTO 'emp_group';
```

Notice at the end of the MapReduce job that no records are output to the console.

6.2. Verify a new folder is created:

```
grunt> ls
hdfs://sandbox:8020/user/root/demos/emp_group      <dir>
hdfs://sandbox:8020/user/root/demos/pigdemo.txt<r 1>  87
```

6.3. View the contents of the output file:

```
grunt> cat emp_group/part-r-00000
CA      { (CA,Ulf), (CA,manish), (CA,Brian) }
CO      { (CO,George), (CO,Mark) }
```

```
IL      { (IL,Danielle) }
NV      { (NV,Barry) }
OH      { (OH,Tom) }
SD      { (SD,Rich) }
```

- a. Notice the fields of the records (which in this case is the **state** field followed by a **bag**) are separated by a tab character, which is the default delimiter in Pig. Use the **PigStorage** object to specify a different delimiter:

```
grunt> STORE emp_group INTO 'emp_group_csv' USING
PigStorage(',');
```

Step 7: Show All Aliases

- 7.1.** The **aliases** command shows a list of currently-defined aliases:

```
grunt> aliases;
aliases: [ca_only, emp_group, employees]
```

Step 8: Monitor the Pig Jobs

- 8.1.** Point your browser to the JobHistory UI at <http://host:19888/>.

- 8.2.** View the list of jobs, which should contain the MapReduce jobs that were executed from your Pig Latin code in the Grunt shell.

- 8.3.** Notice you can view the log files of the ApplicationMaster, and also each Map and Reduce task.

NOTE: Three commands trigger a logical plan to be converted to a physical plan and execute as a MapReduce job: STORE, DUMP and ILLUSTRATE.

Pig Latin Relation Names

- A **relation** is the result of a processing step
- The name given to a relation is called an **alias**
- For example, **stocks** is an alias:

```
stocks = LOAD 'mydata.txt'  
          USING TextLoader();
```

© Hortonworks Inc. 2014



Pig Latin Relation Names

Each processing step of a Pig Latin script results in a new data set, referred to as a **relation**. You assign names to relations, and the name of a relation is referred to as its **alias**. For example, consider the following Pig Latin statement:

```
stocks = LOAD 'mydata.txt' using TextLoader();
```

The alias **stocks** is assigned to the relation created by the **LOAD** statement, which in this statement is a line of text from the **mydata.txt** file. The **stocks** alias now represents the collection of records in **mydata.txt**.

NOTE: **TextLoader** is a simple way of loading each line of text in a file into a record, no matter what the format of the data is.

Relation names (aliases) are not variables, even though they look like variables. You can reassign an alias to a different relation, but that is not recommended.

Pig Latin Field Names

- Relations can define and use field names, which are associated with an alias
- For example:

```
salaries = LOAD 'salary.data'  
    USING PigStorage(',')  
    AS (gender, age, income, zip);  
highsalaries = FILTER salaries BY income >  
1000000;
```

© Hortonworks Inc. 2014



Pig Latin Field Names

You can also define ***field names*** when using the **LOAD** command to define a relation. Use the **AS** keyword to define field names:

```
salaries = LOAD 'salary.data' USING PigStorage(',') AS  
(gender, age, income, zip);
```

The alias for this relation is **salaries**, and **salaries** has four field names: **gender**, **age**, **income** and **zip**.

Field names can be used in subsequent processing commands. For example, when filtering a relation, you can refer to its fields in the **BY** clause, as shown in the following statement:

```
highsalaries = FILTER salaries BY income > 1000000;
```

Field names contain the values of the current record as the data passes through the pipeline of the Pig application. The **highsalaries** relation will contain all records whose **income** field is greater than 1,000,000.

Both field names and relation names must satisfy the following naming criteria:

- Start with an alphabetic character
- Can contain alphabetic and numeric characters, as well as the underscore (_) character
- All characters must be ASCII

IMPORTANT: Field names and relation names are case sensitive in your Pig Latin scripts. User Defined Functions (UDFs) are also case sensitive. However, Pig Latin keywords (like LOAD and AS) are not case sensitive.

Pig Data Types

- **int**
- **long**
- **float**
- **double**
- **chararray**
- **bytearray**
- **boolean**
- **datetime**
- **bigdecimal**
- **biginteger**

© Hortonworks Inc. 2014



Pig Data Types

Pig has six built-in scalar data types:

- **int**: a 32-bit signed integer
- **long**: a 64-bit signed integer
- **float**: 32-bit floating-point number
- **double**: 64-bit floating-point number
- **chararray**: strings of Unicode characters (represented as **java.lang.String** objects)
- **bytearray**: a blob or array of bytes
- **boolean**: can be either **true** or **false** (case-sensitive)
- **datetime**: stores a date and time in the format 1970-01-01T00:00:00.000+00:00.
- **bigdecimal** and **biginteger**: for performing precision arithmetic.

Pig Complex Types

- **Tuple:** ordered set of values
(OH,Mark,Twain,31225)
- **Bag:** unordered collection of tuples
 - {
 (OH,Mark,Twain,31225),
 (UK,Charles,Dickens,42207),
 (ME,Robert,Frost,11496)
}
- **Map:** collection of key value pairs
[state#OH,name#Mark Twain,zip#31225]

© Hortonworks Inc. 2013



Pig Complex Types

Pig has three complex types:

- **Tuple:** ordered set of fields. A tuple is analogous to a row in a SQL table, with the fields being SQL columns.
- **Bag:** unordered collection of tuples.
- **Map:** collection of key value pairs.

Tuples are indicated by parentheses. For example, the following tuple has four fields:

```
(OH,Mark,Twain,31225)
```

Bags are constructed using curly braces, and the tuples within the bag are separated by commas. For example, the following bag has three tuples in it:

```
{(OH,Mark,Twain,31225),(UK,Charles,Dickens,42207),  
(ME,Robert,Frost,11496)}
```

Maps are key/value pairs, where the key must be a unique **chararray** type, and the value can be any data. Maps are formed using square brackets, with a hashtag between the key and value. For example, the following map has three key#value pairs:

```
[state#OH, name#Mark Twain, zip#31225]
```

As you saw in the Demonstration, the complex types can be nested. For example, a bag can be an element of a tuple, which is the result of the **GROUP BY** operator:

```
(CA, { (CA,Ulf), (CA,manish), (CA,Brian) })
```

Defining a Schema

```
customers = LOAD 'customer_data' AS (
    firstname: chararray,
    lastname:chararray,
    house_number:int,
    street:chararray,
    phone:long,
    payment:double);
```

```
salaries = LOAD 'salaries.txt' AS
(gender:chararray,
 details:bag{
    b (age:int,salary:double,zip:long)
 }) ;
```

© Hortonworks Inc. 2013



Defining a Schema

Pig will eat any kind of data, but if your data has a known structure to it, then you can define a schema for it. The schema is typically defined when you load the data using the **AS** keyword.

For example:

```
customers = LOAD 'customer_data' AS (firstname:
chararray,lastname:chararray,house_number:int,
street:chararray,phone:long,payment:double);
```

The **customers** relation has six fields, and each field is a specific data type.

NOTE: If you load a customer record that has more than six fields, then the extra fields will be truncated. If you load a customer record that has fewer than six fields, then it will pad the end of the record with nulls.

The schema can also specify complex types. For example, suppose we have the following dataset in a file named ‘**bag_demo.txt**’:

```
F, 66, { (41000, 95103), (33000, 57701) }
M, 40, { (76000, 95102) }
F, 58, { (95000, 95103), (60000, 95105) }
M, 85, { (14000, 95102), (0, 95105), (2000, 94040) }
```

The corresponding relation might look like:

```
salaries = LOAD 'bag_demo.txt' AS (gender:chararray,
age:int, details:bag{b:(salary:double,zip:long)});
```

The **salaries** relation is a tuple of three fields: the first field is a **chararray** named **gender**, the second field is an **int** named **age**, and the third field is a **bag** named **details**.

NOTE: Bags are odd in that the tuple inside the bag must have a name, which is **b** in the **salaries** relation. But you will never actually use the **b** name in any future Pig commands.

Pig is very lenient when it comes to schemas:

- If you define a schema, then Pig will perform error-checking with it.
- If you do not define a schema, Pig will make a “best guess” as to how the data should be treated.

Lab 5.1: Getting Started with Pig

Objective:	Use Pig to navigate through HDFS and explore a dataset.
Location of Files:	/root/labs/Lab5.1
Successful Outcome:	You will have a couple of Pig programs that load the White House visitors' data, with and without a schema, and store the output of a relation into a folder in HDFS.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: View the Raw Data

- 1.1. Change directories to the Lab5.1 folder:

```
# cd ~/labs/Lab5.1
```

- 1.2. Unzip the archive in the **Lab5.1** folder, which contains a file named **whitehouse_visits.txt** that is quite large:

```
# unzip whitehouse_visits.zip
```

- 1.3. View the contents of this file:

```
# tail whitehouse_visits.txt
```

This publicly available data contains records of visitors to the White House in Washington, D.C.

Step 2: Load the Data into HDFS

- 2.1. Start the Grunt shell:

```
# pig
```

2.2. From the Grunt shell, make a new directory in HDFS named **whitehouse**:

```
grunt> mkdir whitehouse
```

2.3. Use the **copyFromLocal** command in the Grunt shell to copy the **whitehouse_visits.txt** file to the **whitehouse** folder in HDFS, renaming the file **visits.txt**. (Be sure to enter this command on a single line):

```
grunt> copyFromLocal  
/root/labs/Lab5.1/whitehouse_visits.txt  
whitehouse/visits.txt
```

2.4. Use the **ls** command to verify the file was uploaded successfully:

```
grunt> ls whitehouse  
hdfs://sandbox:8020/user/root/whitehouse/visits.txt<r 1>  
175153242
```

Step 3: Define a Relation

3.1. You will use the **TextLoader** to load the **visits.txt** file.

NOTE: **TextLoader** simply creates a tuple for each line of text , and it uses a single **chararray** field that contains the entire line. It allows you to load lines of text and not worry about the format or schema yet.

Define the following **LOAD** relation:

```
grunt> A = LOAD '/user/root/whitehouse/' USING  
TextLoader();
```

3.2. Use **DESCRIBE** to notice that **A** does not have a schema:

```
grunt> DESCRIBE A;  
Schema for A unknown.
```

3.3. We want to get a sense of what this data looks like. Use the **LIMIT** operator to define a new relation named **A_limit** that is limited to 10 records of **A**.

3.4. Use the **DUMP** operator to view the **A_limit** relation. Each row in the output will look similar to the following and should be 10 arbitrary rows from **visits.txt**:

Step 4: Define a Schema

- 4.1.** Load the White House data again, but this time use the **PigStorage** loader and also define a partial schema:

```
grunt> B = LOAD '/user/root/whitehouse/visits.txt' USING
PigStorage(',') AS (
    lname:chararray,
    fname:chararray,
    mname:chararray,
    id:chararray,
    status:chararray,
    state:chararray,
    arrival:chararray
);
```

- #### 4.2. Use the **DESCRIBE** command to view the schema:

```
grunt> describe B;
B: {lname: chararray, fname: chararray, mname: chararray, id: chararray, status: chararray, state: chararray, arrival: chararray}
```

Step 5: The STORE Command

- 5.1.** Enter the following **STORE** command, which stores the **B** relation into a folder named **whose tab** and separates the fields of each record with tabs:

```
grunt> store B into 'whouse tab' using PigStorage('\t');
```

- ## 5.2. Verify the whose tab folder was created:

```
grunt> ls whose tab
```

You should see two map output files.

5.3. View one of the output files to verify they contain the B relation in a tab-delimited format:

```
grunt> cat whouse_tab/part-m-00000
```

5.4. Each record should contain 7 fields. What happened to the rest of the fields from the raw data that was loaded from **whitehouse/visits.txt**?

Step 6: Use a Different Storer

6.1. In the previous step, you stored a relation using **PigStorage** with a tab delimiter. Enter the following command, which stores the same relation but in a JSON format:

```
grunt> store B into 'whouse_json' using JsonStorage();
```

6.2. Verify the **whouse_json** folder was created:

```
grunt> ls whouse_json
```

6.3. View one of the output files:

```
grunt> cat whouse_json/part-m-00000
```

Notice that the schema you defined for the B relation was used to create the format of each JSON entry:

```
{"lname": "MATTHEWMAN", "fname": "ROBIN", "mname": "H", "id": "U81961", "status": "73574", "state": "VA", "arrival": "2/10/2011 11:14"}  
{"lname": "MCALPINEDILEM", "fname": "JENNIFER", "mname": "J", "id": "U81961", "status": "78586", "state": "VA", "arrival": "2/10/2011 10:49"}
```

RESULT: You have now seen how to execute some basic Pig commands, load data into a relation, and store a relation into a folder in HDFS using different formats.

The GROUP Operator

salaries				salariesbyage	
gender	age	salary	zip	group	salaries
F	17	41000.00	95103	17	<code>{(F,17,41000.0,95103), (F17,35000.0,951034)}</code>
M	19	76000.00	95102	19	<code>{(M,19,76000.0,95102), (F,19,60000.0,95105), (M,19,14000.0,95102)}</code>
F	22	95000.00	95103	22	<code>{(F,22,95000.0,95103)}</code>
F	19	60000.00	95105		
M	19	14000.00	95102		
M	17	35000.00	95103		

`salariesbyage = GROUP salaries BY age;`

```
grunt> DESCRIBE salariesbyage;
salariesbyage: {group:int,
    salaries:{(gender: chararray, age: int,salary: double,zip: int)}}
```

© Hortonworks Inc. 2013



The GROUP Operator

One of the most common operators in Pig is **GROUP**, which collects all records with the same value for a provided key and puts them together into a bag. The result of a **GROUP** operation is a relation that includes one tuple per group. This tuple contains two fields:

- The first field is named "**group**" and is the same type as the group key
- The second field takes the name of the original relation and is type **bag**

For example, suppose we have the following data set:

```
F, 66, 41000, 95103
M, 40, 76000, 95102
F, 58, 95000, 95103
F, 68, 60000, 95105
M, 85, 14000, 95102
...
```

Let's group the records together by **age**:

```
salaries = LOAD 'salaries.txt' USING PigStorage(',') AS
(gender:chararray, age:int,salary:double,zip:int);
salariesbyage = GROUP salaries BY age;
```

The **salariesbyage** relation has two fields. The first is **group**, which will be an **int** because **age** is an **int**, followed by the **salaries** field as a tuple:

```
> DESCRIBE salariesbyage;
salariesbyage: {group:int, salaries:{(gender: chararray,
    age: int,salary: double,zip: int)}}}
```

The records will look like:

```
> DUMP salariesbyage;
(17, { (F,17,0.0,95050), (M,17,0.0,95103), (M,17,0.0,95102) })
(19, { (M,19,0.0,95050) })
(22, { (F,22,90000.0,95102) })
(23, { (M,23,89000.0,95105), (M,23,64000.0,94041) })
```

You can also group a relation by multiple keys. The keys must be listed in parentheses. For example:

```
> mygroup = GROUP salaries BY (gender,age);
> describe mygroup;
mygroup: {group: (gender: chararray,age: int),salaries:
{ (gender: chararray,age: int,salary: double,zip: int)}}}
```

Notice the **group** field is a tuple containing both **gender** and **age**. The resulting records in the **mygroup** relation look like:

```
((M,17), { (M,17,0.0,95103), (M,17,0.0,95102) })
((M,19), { (M,19,0.0,95050) })
((M,23), { (M,23,64000.0,94041), (M,23,89000.0,95105) })
```

GROUP ALL

salaries				salariesbyage	
gender	age	salary	zip	group	salaries
F	17	41000.00	95103		
M	19	76000.00	95102		
F	22	95000.00	95103		
F	19	60000.00	95105		
M	19	14000.00	95102		
M	17	35000.00	95103		

→

salariesbyage = GROUP salaries ALL;

```
grunt> DESCRIBE salaries_group;
salaries_group: {
  group: chararray,
  salaries: {(gender: chararray,age: int,salary: double,zip: int)}}
```

© Hortonworks Inc. 2013



GROUP ALL

You can group all of the records of a relation into a single tuple using the **ALL** option. For example:

```
> salaries_group = GROUP salaries ALL;
> describe salaries_group;
salaries_group: {group: chararray, salaries: {(gender: chararray, age: int, salary: double, zip: int)}}}
```

In this case, the value of **group** will be the chararray “all”, followed by a **bag** of all records:

```
(all, {(F, 66, 41000.0, 95103), (M, 40, 76000.0, 95102), (F, 58, 95000.0, 95103), (F, 68, 60000.0, 95105), (M, 85, 14000.0, 95102), (M, 14, 0.0, 95105), (M, 52, 2000.0, 94040), (M, 67, 99000.0, 94040), (F, 43, 11000.0, 94041), (F, 37, 65000.0, 94040), (M, 72, 83000.0, 94041), (M, 68, 15000.0, 95103), (F, 74, 37000.0, 95105), (F, 15, 0.0, 95050), (F, 83, 0.0, 94040), (F, 30, 10000.0, 95101), (M, 19, 0.0, 95050), (M, 23, 89000.0, 95105), (M, 1, 0.0, 95050), (F, 4, 0.0, 95103)})
```

Relations without a Schema

salaries				salariesgroup	
\$0	\$1	\$2	\$3	group	salaries
F	17	41000.00	95103	95103	{(F,17,41000.0,95103), (F,22,95000.0,95103) (F17,35000.0,95103)}
M	19	76000.00	95102	95102	{(M,19,76000.0,95102), (M,19,14000.0,95102)}
F	22	95000.00	95103		
F	19	60000.00	95105		
M	19	14000.00	95102		
M	17	35000.00	95103	95105	{(F,19,60000.0,95105)}

salariesgroup = GROUP salaries BY \$3;

```
grunt> DESCRIBE salariesbyage;
salariesbyage: {group:bytearray,
    salaries:{()}}
```

© Hortonworks Inc. 2014



Relations without a Schema

If you do not define a schema, then the fields of a relation are specified using an index that starts at **\$0**. This works well for datasets that have a lot of columns or for data that is not structured.

The following relation has four columns but does not define a schema:

```
salaries = LOAD 'salaries.txt' USING PigStorage(',') ;
```

Notice what the output is when you try to describe this relation:

```
> DESCRIBE salaries;
Schema for salaries unknown.
```

The following relation groups **salaries** by its fourth field:

```
salariesgroup = GROUP salaries BY $3;
```

Notice the **salariesgroup** relation does not have a schema for its **salaries** field:

```
> describe salariesgroup  
salariesgroup: {group: bytearray, salaries: {} ()}
```

Why is the datatype of **group** a **bytearray**? _____

The FOREACH GENERATE Operator

salaries				A	
gender	age	salary	zip	age	salary
M	66	41000.00	95103	66	41000.00
M	58	76000.00	57701	58	76000.00
F	40	95000.00	95102	40	95000.00
M	45	60000.00	95105	45	60000.00
F	28	55000.00	95103	28	55000.00

`A = FOREACH salaries GENERATE age, salary;`

```
grunt> DESCRIBE A;
A: {age: int, salary: double}
```

© Hortonworks Inc. 2013



The FOREACH GENERATE Operator

The **FOREACH..GENERATE** operator transforms records based on a set of expressions you define. The operator works on each record in the data set (as in, “for each record”). The result of a **FOREACH** is a new tuple, typically with a different schema.

The **FOREACH** operator is a great tool for transforming your data into different data sets. The expression in a **FOREACH** can contain fields, constants, mathematical expressions, the result of invoking a Pig function, and many other variations and nestings.

Let’s look at an example. The following command takes in the **salaries** relation and generates a new relation that only contains two of the columns in **salaries**:

```
> A = FOREACH salaries GENERATE age, salary;
> DESCRIBE A;
A: {age: int, salary: double}
```

The records in the **A** relation look like:

```
(66,84000.0)
(39,3000.0)
(84,14000.0)
```

You can perform mathematical computations in the **GENERATE** clause:

```
B = FOREACH salaries GENERATE salary, salary * 0.07;
```

The resulting relation contains each salary along with the salary multiplied by 7%:

```
(69000.0,4830.000000000001)
(91000.0,6370.000000000001)
(0.0,0.0)
(48000.0,3360.000000000005)
(3000.0,210.000000000003)
```

Specifying Ranges in FOREACH

```
salaries = LOAD 'salaries.txt' USING  
PigStorage(',') AS (gender:chararray,  
age:int,salary:double,zip:int);  
C = FOREACH salaries GENERATE age..zip;  
D = FOREACH salaries GENERATE age..;  
E = FOREACH salaries GENERATE ..salary;
```

```
customer = LOAD 'data/customers';  
F = FOREACH customer GENERATE $12..$23;
```

© Hortonworks Inc. 2013



Specifying Ranges in FOREACH

In the **GENERATE** clause, you can specify a range of values, which is useful when working with datasets that have a large number of fields. For example:

```
salaries = LOAD 'salaries.txt' USING PigStorage(',') AS  
(gender:chararray, age:int,salary:double,zip:int);  
C = FOREACH salaries GENERATE age..zip;
```

The **C** relation will contain three fields: **age**, **salary** and **zip**:

```
> describe C;  
C: {age: int,salary: double,zip: int}
```

You can also specify an open-ended range. For example:

```
D = FOREACH salaries GENERATE age..;  
E = FOREACH salaries GENERATE ..salary;
```

D will contain **age**, **salary** and **zip**. **E** will contain **gender**, **age**, and **salary**.

This syntax also works well with large relations that do not have a schema:

```
customer = LOAD 'data/customers';
F = FOREACH customer GENERATE $12..$23;
```

Field Names in a FOREACH

```
salaries = LOAD 'salaries.txt' USING  
PigStorage(',') AS (gender:chararray,  
age:int,salary:double,zip:int);  
C = FOREACH salaries GENERATE zip, salary;  
C: {zip: int,salary: double}
```

```
D = FOREACH salaries GENERATE zip,  
    salary * 0.10;  
D: {zip: int,double}
```

```
E = FOREACH salaries GENERATE zip,  
    salary * 0.10 AS bonus;  
E: {zip: int,bonus: double}
```

© Hortonworks Inc. 2013



Field Names in a FOREACH

A relation created from a **FOREACH** statement is a new tuple. The data types of the fields in the new tuple are inferred by Pig, but sometimes the names of the fields are not inferred. For example, in the following simple projection, Pig will use the same field name as the original relation:

```
> salaries = LOAD 'salaries.txt' USING PigStorage(',') AS  
(gender:chararray, age:int,salary:double,zip:int);  
> C = FOREACH salaries GENERATE zip, salary;  
> DESCRIBE C;  
C: {zip: int,salary: double}
```

However, in the following projection, Pig cannot determine a field name for the second value in the new tuple:

```
> D = FOREACH salaries GENERATE zip, salary * 0.10;  
> DESCRIBE D;  
D: {zip: int,double}
```

Notice the second field in **D** only has a datatype, but no name. You would have to use the **\$1** to refer to this field in **D**.

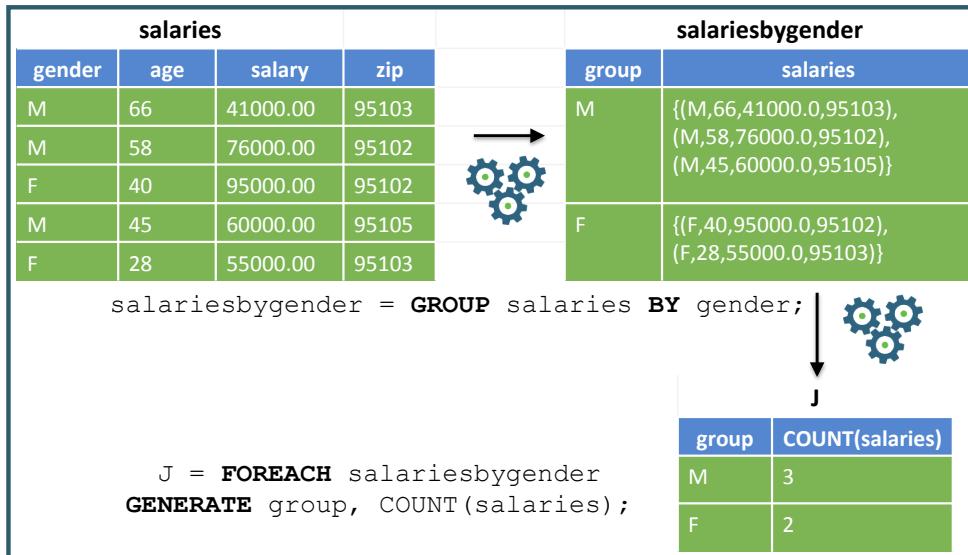
You can use the **AS** keyword to assign a name to the fields in the new tuple. For example:

```
> E = FOREACH salaries GENERATE zip, salary * 0.10 AS  
  bonus;  
> DESCRIBE E;  
E: {zip: int,bonus: double}
```

Notice the second field in **E** has the name **bonus**.

NOTE: You can use the **AS** keyword for any of the fields in the **GENERATE** clause, even if the field name can be inferred by Pig.

FOREACH with Groups



© Hortonworks Inc. 2013



FOREACH with Groups

Let's look at an example of a Pig script that performs a **FOREACH** operation on a group. The following

```
salaries = LOAD 'salaries.txt' USING PigStorage(',') AS  
(gender:chararray, age:int,salary:double,zip:int);  
salariesbygender = GROUP salaries BY gender;
```

The **salariesbygender** relation has two fields - **group** and a bag named **salaries**:

```
salariesbygender: {group: chararray,salaries: {(gender:  
chararray,age: int,salary: double,zip: int)}}
```

Since there are only two possible values of **group** (M or F), then there will be at most two rows. The following **FOREACH** counts the number of tuples in each **salaries** bag:

```
J = FOREACH salariesbygender GENERATE group,  
COUNT(salaries);
```

The **J** relation looks like:

```
J: {group: chararray, long}
```

and output of **J** is:

```
(F, 24)  
(M, 26)
```

This means our **salaries.txt** file contains 24 female records, and 26 male records.

If you need to specifically refer to a field inside the bag of a group relation, you use the *dot operator*. For example, suppose we only want to refer to the **salary** field in the **salaries** bag of the **salariesbygender** relation:

```
K = FOREACH salariesbygender GENERATE group,  
MAX(salaries.salary);
```

The **K** relation will contain **group** (so either M or F) and the maximum **salary** field in that particular **salaries** bag. The output of running this code is:

```
(F, 95000.0)  
(M, 99000.0)
```

The FILTER Operator

salaries				G			
gender	age	salary	zip	gender	age	salary	zip
F	17	41000.00	95103				
M	19	76000.00	95102	M	19	76000.0	95102
F	22	95000.00	95103	F	22	95000.0	95103
F	19	60000.00	95105	F	19	60000.0	95105
M	19	14000.00	95102				
M	17	35000.00	95103				

`G = FILTER salaries BY salary >= 50000.0;`



© Hortonworks Inc. 2013



The FILTER Operator

The **FILTER** operator selects tuples from a relation based on specified Boolean expressions. Use **FILTER** to select the rows you want (or filter out the rows you do not want).

The **FILTER** operator looks like:

```
FILTER alias BY expression;
```

For example, the following command filters the **salaries** relation to contain only those tuples whose **salary** field is greater than 10,000:

```
G = FILTER salaries BY salary >= 10000.0;
```

Conditions can be combined using **AND** or **OR**:

```
H = FILTER salaries BY gender == 'F' AND age >= 50;
```

Use the **NOT** operator to reverse a condition. Suppose we have the following dataset:

```
SD      Rich
NV      Barry
CO      George
CA      Ulf
IL      Danielle
OH      Tom
CA      Manish
CA      Brian
CO      Mark
```

The following **NOT** operator filters out all rows that match a regular expression:

```
> employees = LOAD 'pigdemo.txt' AS (state:chararray,
name:chararray);
> no_b = FILTER employees BY NOT name MATCHES '.*b|B.*';
```

The **no_b** relation will contain all records that do not contain the letter '**b**' or '**B**':

```
(SD,Rich)
(CO,George)
(CA,Ulf)
(IL,Danielle)
(OH,Tom)
(CA,Manish)
(CO,Mark)
```

NOTE: The **FILTER** command does not change the schema of a relation or the structure. It only narrows down the number of records belonging to that relation.

The LIMIT Operator

```
employees = LOAD 'pigdemo.txt' AS  
(state:chararray, name:chararray);  
  
emp_group = GROUP employees BY state;  
  
L = LIMIT emp_group 3;
```

© Hortonworks Inc. 2013



The LIMIT Operator

The **LIMIT** command limits the number of output tuples for a relation:

```
employees = LOAD 'pigdemo.txt' AS (state:chararray,  
name:chararray);  
emp_group = GROUP employees BY state;  
L = LIMIT emp_group 3;
```

Note there is no guarantee which 3 tuples will be returned, and the tuples that are returned can change from one run to the next. Using the data shown earlier, the output of one of the executions was:

```
(CA, { (CA,Ulf), (CA,manish), (CA,Brian) })  
(CO, { (CO,George), (CO,Mark) })  
(IL, { (IL,Danielle) })
```

NOTE: If you define an **ORDER BY** (discussed in the next Unit) immediately before the **LIMIT**, then you will be guaranteed to get the same results each time.

Unit 5 Review

1. List two Pig commands that cause a logical plan to execute: _____
2. Which Pig command stores the output of a relation into a folder in HDFS? _____

Suppose the **prices.csv** file looks like:

```
XFR,2004-05-13,22.90,400  
XFR,2004-05-12,22.60,400000  
XFR,2004-05-11,22.80,2600  
XFR,2004-05-10,23.00,3800  
XFR,2004-05-07,23.55,2900  
XFR,2004-05-06,24.00,2200
```

And assume we have the following relation defined:

```
prices = load 'prices.csv' using PigStorage(',')  
as (symbol:chararray, date:chararray, price:double,  
volume:int);
```

Explain what each of the following Pig commands or relations do:

3. `describe prices;` _____
4. `A = group prices by symbol;` _____
5. `B = foreach prices generate symbol as x, volume as y;`

6. `C = foreach A generate group, SUM(prices.volume);`

7. `D = foreach prices generate symbol..price;`

8. Write a Pig relation that only contains **prices** with **volume** greater than 3,000:

Lab 5.2: Exploring Data with Pig

Objective:	Use Pig to navigate through HDFS and explore a dataset.
Location of Files:	whitehouse/vistis.txt in HDFS
Successful Outcome:	You will have written several Pig scripts that analyze and query the White House visitors' data, including a list of people who visited the President.
Before You Begin:	At a minimum, complete steps 1 and 2 of Lab 5.1.

Step 1: Load the White House Visitor Data

- 1.1.** You will use the **TextLoader** to load the **visits.txt** file. Define the following **LOAD** relation:

```
grunt> A = LOAD '/user/root/whitehouse/' USING  
TextLoader();
```

Step 2: Count the Number of Lines

- 2.1.** Define a new relation named **B** that is a group of all the records in **A**:

```
grunt> B = GROUP A ALL;
```

- 2.2.** Use **DESCRIBE** to view the schema of **B**. What is the datatype of the **group** field? _____ Where did this datatype come from? _____

- 2.3.** Why does the **A** field of **B** contain no schema? _____

- 2.4.** How many groups are in the relation **B**? _____

2.5. The **A** field of the **B** tuple is a bag of all of the records in **visits.txt**. Use the **COUNT** function on this bag to determine how many lines of text are in **visits.txt**:

```
grunt> A_count = FOREACH B GENERATE 'rowcount', COUNT(A);
```

NOTE: The '**rowcount**' string in the **FOREACH** statement is simply to demonstrate that you can have constant values in a **GENERATE** clause. It is certainly not necessary - just makes the output nicer to read.

2.6. Use **DUMP** on **A_count** to view the results. The output should look like:

```
(rowcount, 447598)
```

We can now conclude that there are 447,598 rows of text in **visits.txt**.

Step 3: Analyze the Data's Contents

3.1. We now know how many records are in the data, but we still do not have a clear picture of what the records look like. Let's start by looking at the fields of each record. Load the data using **PigStorage(',')** instead of **TextLoader()**:

```
grunt> visits = LOAD '/user/root/whitehouse/' USING  
PigStorage(' ,');
```

This will split up the fields by comma.

3.2. Use a **FOREACH..GENERATE** command to define a relation that is a projection of the first 10 fields of the **visits** relation.

3.3. Use **LIMIT** to display only 50 records, then **DUMP** the result. The output should be 50 tuples that represent the first 10 fields of **visits**:

```
(PARK,ANNE,C,U51510,0,VA,10/24/2010 14:53,B0402,,)  
(PARK,RYAN,C,U51510,0,VA,10/24/2010 14:53,B0402,,)  
(PARK,MAGGIE,E,U51510,0,VA,10/24/2010 14:53,B0402,,)  
(PARK,SIDNEY,R,U51510,0,VA,10/24/2010 14:53,B0402,,)  
(RYAN,MARGUERITE,,U82926,0,VA,2/13/2011 17:14,B0402,,)  
(WILE,DAVID,J,U44328,,VA,,,)  
(YANG,EILENE,D,U82921,,VA,,,)  
(ADAMS,SCHUYLER,N,U51772,,VA,,,)  
(ADAMS,CHRISTINE,M,U51772,,VA,,,)  
(BERRY,STACEY,,U49494,79029,VA,10/15/2010  
12:24,D0101,10/15/2010 14:06,D1S)
```

NOTE: Because **LIMIT** uses an arbitrary sample of the data, your output will be different names, but the format should look similar.

Notice from the output that the first three fields are the person's name. The next 7 fields are a unique ID, badge number, access type, time of arrival, post of arrival, time of departure and post of departure.

Step 4: Locate the POTUS (President of the United States of America)

4.1. There are 26 fields in each record, and one of them represents the *visitee* (the person being visited in the White House). Your goal now is to locate this column and determine who has visited the President of the United States. Define a relation that is a projection of the last 7 fields (**\$19** to **\$25**) of **visits**. Use **LIMIT** to only output 500 records. The output should look like:

```
(OFFICE, VISITORS, WH, RESIDENCE, OFFICE, VISITORS, HOLIDAY OPEN  
HOUSE/)  
(OFFICE, VISITORS, WH, RESIDENCE, OFFICE, VISITORS, HOLIDAY OPEN  
HOUSES/)  
(OFFICE, VISITORS, WH, RESIDENCE, OFFICE, VISITORS, HOLIDAY OPEN  
HOUSE/)  
(CARNEY, FRANCIS, WH, WW, ALAM, SYED, WW TOUR)  
(CARNEY, FRANCIS, WH, WW, ALAM, SYED, WW TOUR)  
(CARNEY, FRANCIS, WH, WW, ALAM, SYED, WW TOUR)  
(CHANDLER, DANIEL, NEOB, 6104, AGCAOILI, KARL, )
```

It is not necessarily obvious from the output, but field **\$19** in the **visits** relation represents the *visitee*. Even though you selected 500 records in the previous step, you may or may not see POTUS in the output above. (The White House has thousands of visitors each day, but only a few meet the President!)

4.2. Use **FILTER** to define a relation that only contains records of **visits** where field **\$19** matches '**POTUS**'. Limit the output to 500 records. The output should include only visitors who met with the President. For example:

```
(ARGOW, KEITH, A, U83268,, VA, ,,,, 2/14/2011 18:42, 2/16/2011  
16:00, 2/16/2011 23:59,, 154, LC, WIN, 2/14/2011  
18:42, LC, POTUS,, WH, EAST ROOM, THOMPSON, MARGRETTE,, AMERICA'S  
GREAT OUTDOORS ROLLOUT EVENT  
, 5/27/2011, , , , , , , , , , , , , , , , , , , , , , , , , , , ,  
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /  
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /  
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /  
, , )
```

```
(AYERS, JOHNATHAN, T, U84307,, VA, , , , 2/18/2011 19:11, 2/25/2011  
17:00, 2/25/2011 23:59,, 619, SL, WIN, 2/18/2011  
19:11, SL, POTUS,, WH, STATE FLOO, GALLAGHER, CLARE,, RECEPTION  
, 5/27/2011, , , , , , , ,  
, )
```

Step 5: Count the POTUS Visitors

5.1. Let's discover how many people have visited the President. To do this, we need to count the number of records in **visits** where field **\$19** matches '**POTUS**'. See if you can write a Pig script to accomplish this. Use the **potus** relation from the previous step as a starting point. You will need to use **GROUP ALL**, and then a **FOREACH** projection that uses the **COUNT** function.

5.2. If successful, you should get 21,819 as the number of visitors to the White House who visited the President.

Step 6: Finding People Who Visited the President

6.1. So far you have used **DUMP** to view the results of your Pig scripts. In this step, you will save the output to a file using the **STORE** command. Start by loading the data using **PigStorage(',')**, which you may already have defined:

```
grunt> visits = LOAD '/user/root/whitehouse/' USING  
PigStorage('');
```

6.2. Now **FILTER** the relation by visitors who met with the President:

```
potus = FILTER visits BY $19 MATCHES 'POTUS';
```

6.3. Define a projection of the **potus** relationship that contains the name and time of arrival of the visitor:

```
grunt> potus_details = FOREACH potus GENERATE  
(chararray) $0 AS lname:chararray,  
(chararray) $1 AS fname:chararray,  
(chararray) $6 AS arrival_time:chararray,  
(chararray) $19 AS visitee:chararray;
```

6.4. Order the **potus_details** projection by last name:

```
grunt> potus_details_ordered = ORDER potus_details BY lname  
ASC;
```

6.5. Store the records of **potus_details_ordered** into a folder named '**potus**' and using a comma delimiter:

```
grunt> STORE potus_details_ordered INTO 'potus' USING  
PigStorage(',',');
```

6.6. View the contents of the **potus** folder:

```
grunt> ls potus  
hdfs://sandbox.hortonworks.com:8020/user/root/potus/_SUCCESS  
S<r 3> 0  
hdfs://sandbox.hortonworks.com:8020/user/root/potus/part-r-  
00000<r 3> 501378
```

6.7. Notice there is a single output file, so the Pig job was executed with one reducer. View the contents of the output file using **cat**:

```
grunt> cat potus/part-r-00000
```

The output should be in a comma-delimited format and contain the last name, first name, time of arrival (if available), and the string 'POTUS':

```
CLINTON,WILLIAM,,POTUS  
CLINTON,HILLARY,,POTUS  
CLINTON,HILLARY,,POTUS  
CLINTON,HILLARY,,POTUS  
CLONAN,JEANETTE,,POTUS  
CLOOBECK,STEPHEN,,POTUS  
CLOOBECK,CHANTAL,,POTUS  
CLOOBECK,STEPHEN,,POTUS  
CLOONEY,GEORGE,10/12/2010 14:47,POTUS
```

Step 7: View the Pig Log Files

7.1. Each time you executed a **DUMP** or **STORE** command, a MapReduce job executed on your cluster. You can view the log files of these jobs in the JobHistory UI. Point your browser to <http://ipaddress:19888>:



JobHistory

Logged in as: dr.who

Retired Jobs										
Start Time		Finish Time		Job ID	Name	User	Queue	State	Maps Total	Maps Completed
2014.01.08 23:52:01 PST	2014.01.08 23:52:23 PST	job_1389214366903_0007	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	1	1
2014.01.08 23:36:43 PST	2014.01.08 23:36:57 PST	job_1389214366903_0006	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0
2014.01.08 23:27:41 PST	2014.01.08 23:27:55 PST	job_1389214366903_0005	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0
2014.01.08 23:25:52 PST	2014.01.08 23:26:14 PST	job_1389214366903_0004	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0
2014.01.08 13:19:21 PST	2014.01.08 13:19:32 PST	job_1389214366903_0003	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0

7.2. Click on the job's id to view the details of the job and its log files.

RESULT: You have written several Pig scripts to analyze and query the data in the White House visitors' log. You should now be comfortable with writing Pig scripts with the Grunt shell and using common Pig commands like **LOAD**, **GROUP**, **FOREACH**, **FILTER**, **LIMIT**, **DUMP** and **STORE**.

ANSWERS:

Step 2: The **group** field is a **chararray** because it is just the string “all” and is a result of performing a **GROUP ALL**. The **A** field of **B** contains no schema because the **A** relation has no schema. The **B** relation can only contain 1 group because it a grouping of every single record. Note that the **A** field is a **bag**, and **A** will contain any number of tuples.

SOLUTIONS:

Step 3:

```
visits = LOAD '/user/root/whitehouse/' USING
PigStorage(',');
firstten = FOREACH visits GENERATE $0..$9;
firstten_limit = LIMIT firstten 50;
DUMP firstten_limit;
```

Step 4:

```
lastfields = FOREACH visits GENERATE $19..$25;
lastfields_limit = LIMIT lastfields 500;
DUMP lastfields_limit;

--find the POTUS
potus = FILTER visits BY $19 MATCHES 'POTUS';
potus_limit = LIMIT potus 500;
DUMP potus_limit;
```

Step 5:

```
potus = FILTER visits BY $19 MATCHES 'POTUS';
potus_group = GROUP potus ALL;
potus_count = FOREACH potus_group GENERATE COUNT(potus);
DUMP potus_count;
```

Unit 6: Advanced Pig Programming

Topics covered:

- The ORDER BY Operator
- The CASE Operator
- Parameter Substitution
- The DISTINCT Operator
- Using PARALLEL
- The FLATTEN Operator
- Lab 6.1: Splitting a Dataset
- Nested FOREACH
- Performing an Inner Join
- Performing an Outer Join
- Replicated Joins
- The COGROUP Operator
- Pig User-Defined Functions
- A UDF Example
- Tips for Optimizing Pig Scripts
- Lab 6.2: Joining Datasets
- Lab 6.3: Preparing Data for Hive
- Overview of the DataFu Library
- Demonstration: Computing PageRank
- Lab 6.4: Analyzing Clickstream Data
- Lab 6.5: Analyzing Stock Market Data using Quantiles

The ORDER BY Operator

salaries				byage			
gender	age	salary	zip	gender	age	salary	zip
M	66	41000.00	95103				
M	58	76000.00	95102				
F	40	95000.00	95102				
M	45	60000.00	95105				
F	28	55000.00	95103				

byage = **ORDER salaries BY age ASC;**



© Hortonworks Inc. 2013



The ORDER BY Operator

The **ORDER BY** command allows you to sort the data in a relation:

```
salaries = LOAD 'salaries.txt' USING PigStorage(',') AS  
    (gender:chararray,age:int,salary:double,zip:chararray);  
byage = ORDER salaries BY age ASC;
```

The records in the **byage** relation will be sorted by age:

```
(M,19,0.0,95050)  
(F,22,90000.0,95102)  
(M,23,89000.0,95105)  
(M,23,64000.0,94041)  
(F,30,10000.0,95101)  
(M,31,95000.0,94041)
```

You can use **DESC** or **ASC** in the **BY** clause. You can also order by multiple fields. For example:

```
agesalary = ORDER salaries BY age ASC, salary ASC;
```

The output is similar to **byage**, except the **salary** field is sorted ascending. Compare the two outputs of the records with age = 23:

```
(M,19,0.0,95050)
(F,22,90000.0,95102)
(M,23,64000.0,94041)
(M,23,89000.0,95105)
(F,30,10000.0,95101)
(M,31,95000.0,94041)
```

NOTE: The resulting output of an **ORDER BY** relation is a *total ordering*, which means the data will be sorted across all output files. In other words, **part-r-00000** will contain the first set of ordered tuples, then **part-r-00001** will continue where the first records left off, and so on.

IMPORTANT: If you define a relation with an ordering, then process that relation in another expression, then the ordering is no longer guaranteed. For example:

```
A = ORDER visitors BY lastname DESC;
B = FILTER A BY age >= 21;
```

The records in **B** are no longer guaranteed to be ordered by **lastname** descending.

The CASE Operator

salaries				bonuses	
gender	age	salary	zip	salary	bonus
M	66	41000.0	95103		
M	58	76000.0	95102	41000.0	2050.0
F	40	95000.0	95102	76000.0	7600.0
M	45	20000.0	95105	95000.0	9500.0
F	28	55000.0	95103	20000.0	0.0
				55000.00	2750.0

```
bonuses = FOREACH salaries GENERATE salary, (
  CASE
    WHEN salary >= 70000.00 THEN salary * 0.10
    WHEN salary < 70000.00
      AND salary >= 30000.0 THEN salary * 0.05
    WHEN salary < 30000.0 THEN 0.0
  END) AS bonus;
```

© Hortonworks Inc. 2014



The CASE Operator

Pig has a **CASE** operator that allows you to make decisions within a **FOREACH GENERATE** statement. A **CASE** clause contains an arbitrary number of **WHEN..THEN** clauses and contains an **END** statement to denote the end of the **CASE**.

For example:

```
bonuses = FOREACH salaries GENERATE salary, (
  CASE
    WHEN salary >= 70000.00 THEN salary * 0.10
    WHEN salary < 70000.00 AND salary >= 30000.0
      THEN salary * 0.05
    WHEN salary <= 30000.0 THEN 0.0
  END) AS bonus;
```

Parameter Substitution

```
stocks = load '$INPUTFILE' USING  
PigStorage(',');
```

```
pig -p INPUTFILE=NYSE_daily_prices_A.csv  
myscript.pig
```

```
pig -param_file stock.params  
myscript.pig
```

© Hortonworks Inc. 2013



Parameter Substitution

Pig provides a **parameter substitution** feature that allows your Pig scripts to refer to values that can be defined at runtime, either from the command line or in a properties file. A parameter is a value that starts with a dollar sign \$.

For example, **\$INPUTFILE** is a parameter in the following **LOAD** statement:

```
stocks = load '$INPUTFILE' USING PigStorage(',');
```

When you execute the script, specify a value for **\$INPUTFILE** using the **-p** switch:

```
> pig -p INPUTFILE=NYSE_daily_prices_A.csv myscript.pig
```

Use the **-param_file** switch if your properties are stored in a text file:

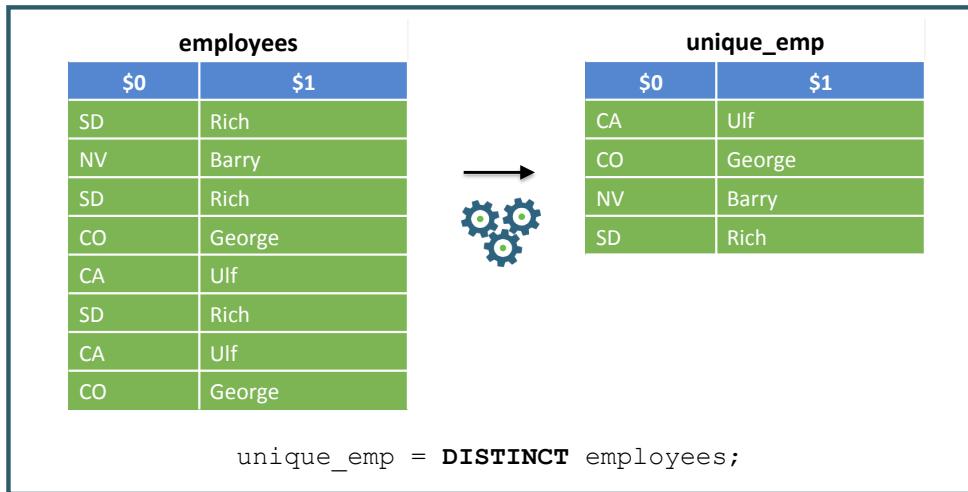
```
> pig -param_file stock.params myscript.pig
```

where the text file **stock.params** looks like:

```
INPUTFILE=NYSE_daily_prices_A.csv
```

NOTES

The DISTINCT Operator



© Hortonworks Inc. 2013



The DISTINCT Operator

The **DISTINCT** operator removes duplicate tuples in a relation. The syntax is:

```
DISTINCT alias;
```

For example, suppose we have the following data:

```
SD      Rich
NV      Barry
SD      Rich
CO      George
CA      Ulf
SD      Rich
CA      Ulf
CO      George
```

Applying **DISTINCT** removes the duplicates:

```
employees = LOAD 'employees.txt';
unique_emp = DISTINCT employees;
```

The tuples in **unique_emp** are:

```
(CA,Ulf)
(CO,George)
(NV,Barry)
(SD,Rich)
```

Using PARALLEL

- PARALLEL determines the number of reducers to use in a particular operation

```
A = LOAD 'data1';  
B = LOAD 'data2';  
C = JOIN A by $1, B by $3 PARALLEL 20;  
D = ORDER C BY $0 PARALLEL 5;
```

© Hortonworks Inc. 2013



Using PARALLEL

The **PARALLEL** operator is a clause used to determine the number of reducers in the subsequent MapReduce job for that particular operation.

The syntax for the **PARALLEL** clause is:

```
PARALLEL n;
```

where *n* is the number of reducers.

For example:

```
A = LOAD 'data1';  
B = LOAD 'data2';  
C = JOIN A by $1, B by $3 PARALLEL 20;  
D = ORDER C BY $0 PARALLEL 5;
```

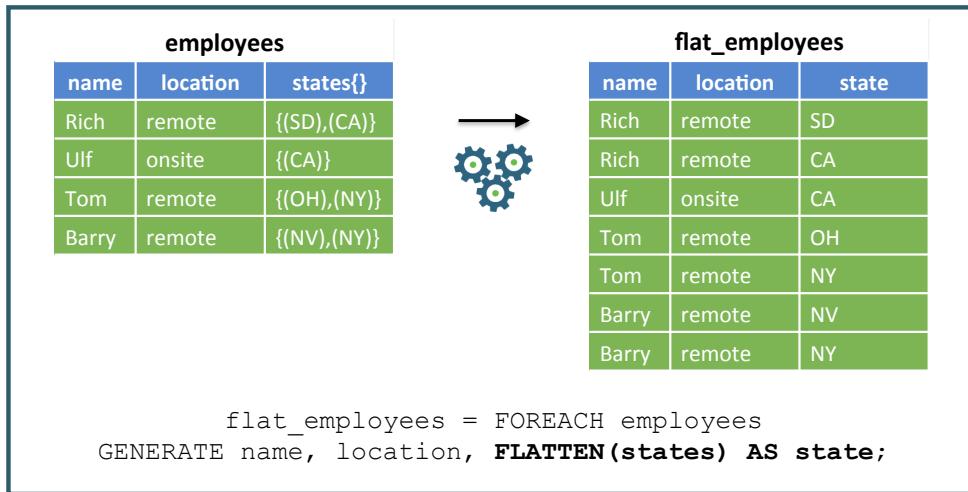
The **JOIN** operation will use 20 reducers, and the **ORDER** operation will use 5 reducers.

You can use the **default_parallel** property to set the number of reducers at the script level. For example, there will be 8 reducers for each reduce task in the following Pig script:

```
SET default_parallel 8;
A = LOAD 'data1';
B = LOAD 'data2';
C = JOIN A by $1, B by $3;
D = ORDER C BY $0;
```

NOTE: Some operators have a reduce phase, like **GROUP**, **ORDER BY**, **DISTINCT**, **JOIN**, **LIMIT** and **COGROUP**. But some Pig operators do not require a reduce phase, like **LOAD**, **FOREACH**, **FILTER** and **SAMPLE**. For those types of operators, it does not make sense to specify a **PARALLEL** value.

The FLATTEN Operator



© Hortonworks Inc. 2013



The FLATTEN Operator

The **FLATTEN** operator removes the nesting of nested tuples and bags. You invoke **FLATTEN** like a function, passing in the tuple or bag that you want to flatten:

```
FLATTEN(relation)
```

The **FLATTEN** operator is best understood by an example. Suppose we have the following data set:

```
Rich    remote      { (SD) , (CA) }
Ulf    onsite       { (CA) }
Tom    remote      { (OH) , (NY) }
Barry  remote      { (NV) , (NY) }
```

The following Pig Latin statements load the data using a schema. Notice the states are in a bag:

```
> employees = LOAD 'locations.txt' AS (
  name:chararray,
  location:chararray,
  states:bag{t:tuple(state:chararray)} );
> describe employees;
```

```
employees: {name: chararray, location: chararray, states: {t:  
  (state: chararray)}}
```

The output of the **employees** relation is the following:

```
(Rich,remote,{(SD),(CA)})  
(Ulf,onsite,{(CA)})  
(Tom,remote,{(OH),(NY)})  
(Barry,remote,{(NV),(NY)})
```

Notice each record has a bag containing one or more states. If you flatten the **states** field in the **employees** relation, then each entry in the bag becomes its own full record. For example:

```
flat_employees = FOREACH employees GENERATE name,  
                  location, FLATTEN(states) AS state;
```

The **FLATTEN** operator produces a cross product of every record in the bag with all of the other expressions in the **GENERATE** clause. The output of **flat_employees** is:

```
(Rich,remote,SD)  
(Rich,remote,CA)  
(Ulf,onsite,CA)  
(Tom,remote,OH)  
(Tom,remote,NY)  
(Barry,remote,NV)  
(Barry,remote,NY)
```

NOTE: The example here flattened a bag, but you can also flatten a nested tuple, which simply removes the nesting so that each field in the tuple is at the top-level. For example, suppose a tuple looks like:

```
(1, (2, 3))
```

After this tuple was flattened, it would look like:

```
(1, 2, 3)
```

Lab 6.1: Splitting a Dataset

Objective:	Research the White House visitor data and look for members of Congress.
Location of Files:	n/a
Successful Outcome:	Two folders in HDFS, congress and not_congress , containing a split of the White House visitor data.
Before You Begin:	You should have the White House visitor data in HDFS in /user/root/whitehouse/visits.txt .

Perform the following steps:

Step 1: Explore the Comments Field

1.1. In this step, you will explore the comments field of the White House visitor data. Start by loading **visits.txt**:

```
cd whitehouse  
visits = LOAD 'visits.txt' USING PigStorage(',') ;
```

1.2. Field **\$25** is the comments. Filter out all records where field **\$25** is null:

```
not_null_25 = FILTER visits BY ($25 IS NOT NULL) ;
```

1.3. Now define a new relation that is a projection of only column **\$25**:

```
comments = FOREACH not_null_25 GENERATE $25 AS comment;
```

1.4. View the schema of **comments** and make sure you understand how this relation ended up as a tuple with one field:

```
grunt> describe comments;
comments: {comment: bytearray}
```

Step 2: Test the Relation

2.1. A common Pig task is to test a relation to make sure it is consistent with what you are intending it to be. But using **DUMP** on a big data relation might take too long or not be practical, so define a **SAMPLE** of **comments**:

```
comments_sample = SAMPLE comments 0.001;
```

2.2. Now **DUMP** the **comments_sample** relation. The output should be non-null comments about visitors to the White House, similar to:

```
(ATTENDEES VISITING FOR A MEETING)
(FORUM ON IT MANAGEMENT REFORM/)
(FORUM ON IT MANAGEMENT REFORM/)
(HEALTH REFORM MEETING)
(DRIVER TO REMAIN WITH VEHICLE)
```

Step 3: Count the Number of Comments

3.1. The **comments** relation represents all non-null comments from **visits.txt**. Write Pig statements that output the number of records in the **comments** relation. The correct result is 222,839 records.

Step 4: Split the Dataset

NOTE: Our end goal is find visitors to the White House who are also members of Congress. We could run our MapReduce job on the entire **visits.txt** dataset, but it is common in Hadoop to split data into smaller input files for specific tasks, which can greatly improve the performance of your MapReduce applications. In this step, you will split **visits.txt** into two separate datasets.

4.1. In this step, you will split **visits.txt** into two datasets: those that contain “CONGRESS” in the comments field, and those that do not. Start by loading the data:

```
visits = LOAD 'visits.txt' USING PigStorage(',') ;
```

4.2. Use the **SPLIT** command to split the **visits** relation into two new relations named **congress** and **not_congress**:

```
SPLIT visits INTO congress IF ($25 MATCHES  
'.* CONGRESS .*'), not_congress IF (NOT ($25 MATCHES  
'.* CONGRESS .*'));
```

4.3. Store the **congress** relation into a folder named '**congress**' using a JSON format:

```
STORE congress INTO 'congress';
```

4.4. Similarly, **STORE** the **not_congress** relation in a folder named '**not_congress**'.

4.5. View the output folders using **ls**. The file sizes should be equivalent to the following:

```
grunt> ls congress  
whitehouse/congress/part-m-00000<r 1> 45618  
whitehouse/congress/part-m-00001<r 1> 0  
grunt> ls not_congress  
whitehouse/not_congress/part-m-00000<r 1> 90741587  
whitehouse/not_congress/part-m-00001<r 1> 272381
```

4.6. View one of the output files in **congress**: and make sure the string "CONGRESS" appears in the comment field:

```
cat congress/part-m-00000
```

Step 5: Count the Results

5.1. Write Pig statements that output the number of records in the **congress** relation. This will tell us how many visitors to the White House have "CONGRESS" in the comments of their visit log. The correct result is 102.

NOTE: You now have two datasets: one in '**congress**' with 102 records, and the remaining records in the '**not_congress**' folder. These records are still in their original, raw format.

RESULT: You have just split ‘**visits.txt**’ into two datasets, and you have also discovered that 102 visitors to the White House had the word “CONGRESS” in their comments field. We will further explore these visitors in the next lab as we perform a join with a dataset containing the names of members of Congress.

SOLUTIONS: Here is a solution to Step 3:

```
comments_all = GROUP comments ALL;
comments_count = FOREACH comments_all GENERATE
    COUNT(comments);
DUMP comments_count;
```

Here is a solution to Step 5:

```
congress_grp = GROUP congress ALL;
congress_count = FOREACH congress_grp GENERATE
    COUNT(congress);
DUMP congress_count;
```

Nested FOREACH

```
dailyA = LOAD 'NYSE_daily_prices_A.csv'  
USING PigStorage(',') AS (exchange,  
symbol);  
  
dailyA_grp = GROUP dailyA BY exchange;  
  
unique_symbols = FOREACH dailyA_grp {  
    symbols = dailyA.symbol;  
    unique_symbol = DISTINCT symbols;  
    GENERATE group, COUNT(unique_symbol);  
};
```

© Hortonworks Inc. 2013



Nested FOREACH

A **nested FOREACH** (also known as an *inner foreach*) is a **FOREACH** statement that contains a nested block of code. The nested block of code has the following criteria:

- Can contain **CROSS**, **DISTINCT**, **FILTER**, **FOREACH**, **LIMIT**, and **ORDER BY** operations.
- Must end with a **GENERATE** statement.

The syntax looks like:

```
FOREACH nested_alias {  
    alias = nested_operation;  
    alias = nested_operation;  
    GENERATE expression;  
};
```

The following example demonstrates how to count the number of unique entries in a group using a nested **FOREACH**. The data is daily stock prices from the New York Stock Exchange (NYSE) and each row looks like:

```
NYSE,AEA,2010-02-08,4.42,4.42,4.21,4.24,205500,4.24
```

The first field is the exchange name and the second field is the stock symbol. These are the only two fields we need for our problem:

```
dailyA = LOAD 'NYSE_daily_prices_A.csv' USING
              PigStorage(',') AS (exchange,symbol);
dailyA_grp = GROUP dailyA BY exchange;
unique_symbols = FOREACH dailyA_grp {
    symbols = dailyA.symbol;
    unique_symbol = DISTINCT symbols;
    GENERATE group, COUNT(unique_symbol);
};
```

- The **dailyA_grp** contains all of the stock symbols grouped by exchange.
- Within the **FOREACH**, the **symbols** relation takes the bag **dailyA.symbol** and produces a new relation that is a bag with tuples that have only the field **symbol**.
- The **unique_symbol** relation is also a list of symbols, but with all of the duplicates removed.
- The **GENERATE** statement projects **group** (which is “NYSE” in this example) and the number of values in **unique_symbol**.

The output is:

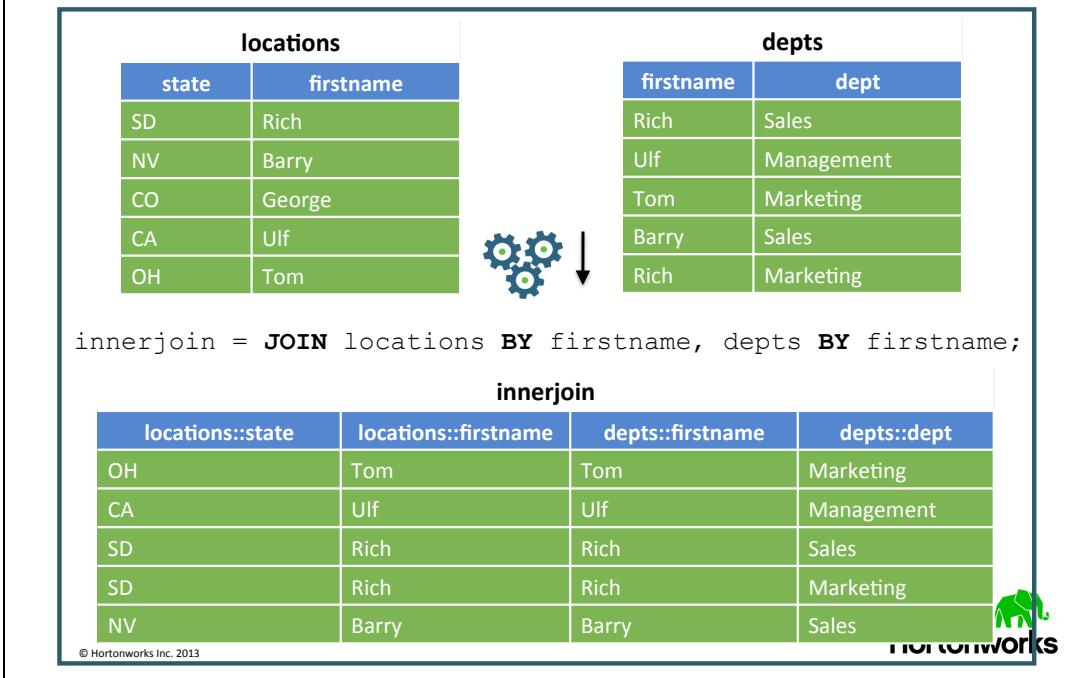
```
(NYSE, 203)
```

which means there are 203 unique stock symbols in the **NYSE_daily_prices_A.csv** file.

NOTE: Another common task inside a nested **FOREACH** is **ORDER BY**. For example:

```
dailyA = LOAD 'NYSE_daily_prices_A.csv' USING
              PigStorage(',') AS (exchange,symbol,date);
dailyA_grp = GROUP dailyA BY symbol;
result = FOREACH dailyA_grp {
    sorted = ORDER dailyA BY date ASC;
    GENERATE group, CONCAT(symbol,date);
};
```

Performing an Inner Join



Performing an Inner Join

Joins are a common occurrence in data processing. The **JOIN** operation in Pig performs both inner and outer joins of two data sets using keys indicated for each input. If the keys are equal, then the two rows are joined.

An inner join in Pig looks like the following:

```
alias = JOIN alias1 BY key1, alias2 BY key2;
```

Let's look at an example. Suppose we have the following file containing states and first names:

```
SD      Rich
NV      Barry
CO      George
CA      Ulf
OH      Tom
```

The second data set contains first names and departments:

```
Rich      Sales
Ulf      Management
Tom      Marketing
Barry    Sales
Rich      Marketing
```

The following Pig Latin commands perform an inner join on these two data sets using the first name in both data sets as the key:

```
locations = LOAD 'pigdemo.txt' AS
            (state:chararray,firstname:chararray);
departments = LOAD 'joindemo.txt' AS
            (firstname:chararray,dept:chararray);
innerjoin = JOIN locations BY firstname, departments BY
firstname;

> describe innerjoin;
innerjoin:{
```

locations::state: chararray,
locations::firstname: chararray,
departments::firstname: chararray,
departments::dept: chararray

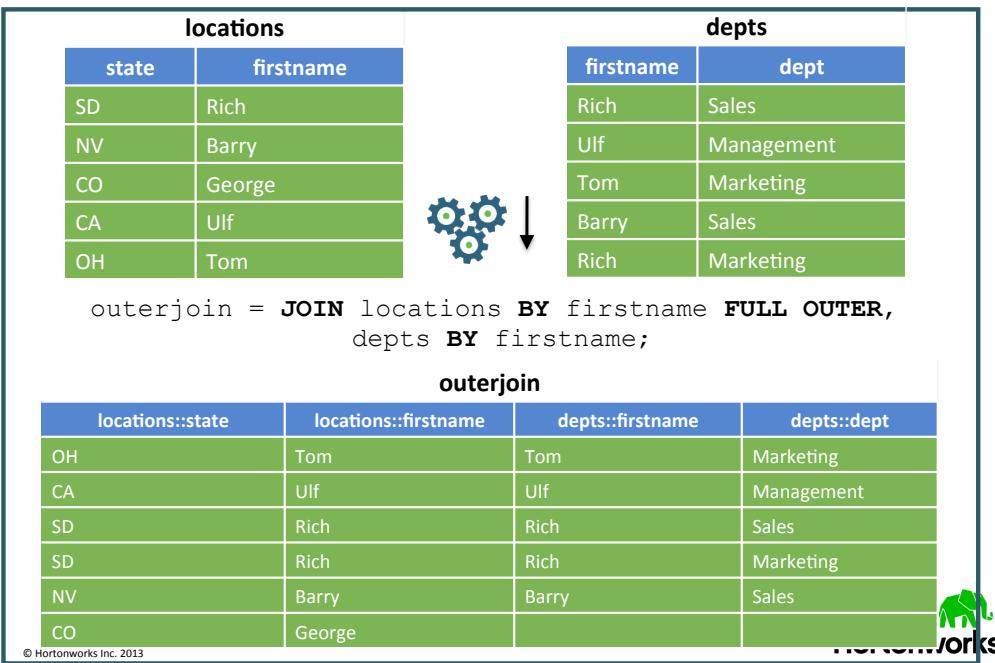
```
}
```

Notice the **innerjoin** relation contains all fields from both data sets in the join. The **::** operator is needed to avoid ambiguity when the two data sets share the same field names (like **firstname** in this example).

The output of **innerjoin** is:

```
(OH,Tom,Tom,Marketing)
(CA,Ulf,Ulf,Management)
(SD,Rich,Rich,Sales)
(SD,Rich,Rich,Marketing)
(NV,Barry,Barry,Sales)
```

Performing an Outer Join



Performing an Outer Join

An outer join in Pig uses the **OUTER** keyword, along with either **LEFT**, **RIGHT** or **FULL**. The syntax looks like:

```
alias = JOIN alias1 BY key1 [LEFT|RIGHT|FULL] OUTER, alias2
      BY key2;
```

NOTE: The main difference between an inner join and an outer join is that records that do not have a match on the other side are included. Pig inserts **null** values into the missing fields.

Let's look at an example, using the same data from the previous example:

```
outerjoin = JOIN locations BY firstname FULL OUTER,
           departments BY firstname;
```

In this case, no records on either side will be omitted. The output looks like:

```
(OH, Tom, Tom, Marketing)
```

```
(CA,Ulf,Ulf,Management)
(SD,Rich,Rich,Sales)
(SD,Rich,Rich,Marketing)
(NV,Barry,Barry,Sales)
(CO,George,,,)
```

If you perform a **LEFT** join, you get all records from the left data set, but non-matching records in the right data set are omitted:

```
leftjoin = JOIN locations BY firstname LEFT OUTER,
           departments BY firstname;
```

In our simple example, the result of **leftjoin** is is the same as **FULL OUTER** because our data on the right does not contain any records that are non-matching:

```
(OH, Tom, Tom, Marketing)
(CA,Ulf,Ulf,Management)
(SD,Rich,Rich,Sales)
(SD,Rich,Rich,Marketing)
(NV,Barry,Barry,Sales)
(CO,George,,,)
```

Replicated Joins

- Uses a LocalResource to perform a map-side join (by storing one dataset in memory)
- Specify '**replicated**', which is applied to the second data set listed

```
replicatedjoin = JOIN locations BY
firstname, departments BY firstname
USING 'replicated';
```

© Hortonworks Inc. 2014



Replicated Joins

A **replicated join** is useful when one of the data sets in the join is small enough to fit into memory. This results in a map-side join, saving an enormous amount of network traffic during the shuffle/sort phase of the resulting MapReduce job.

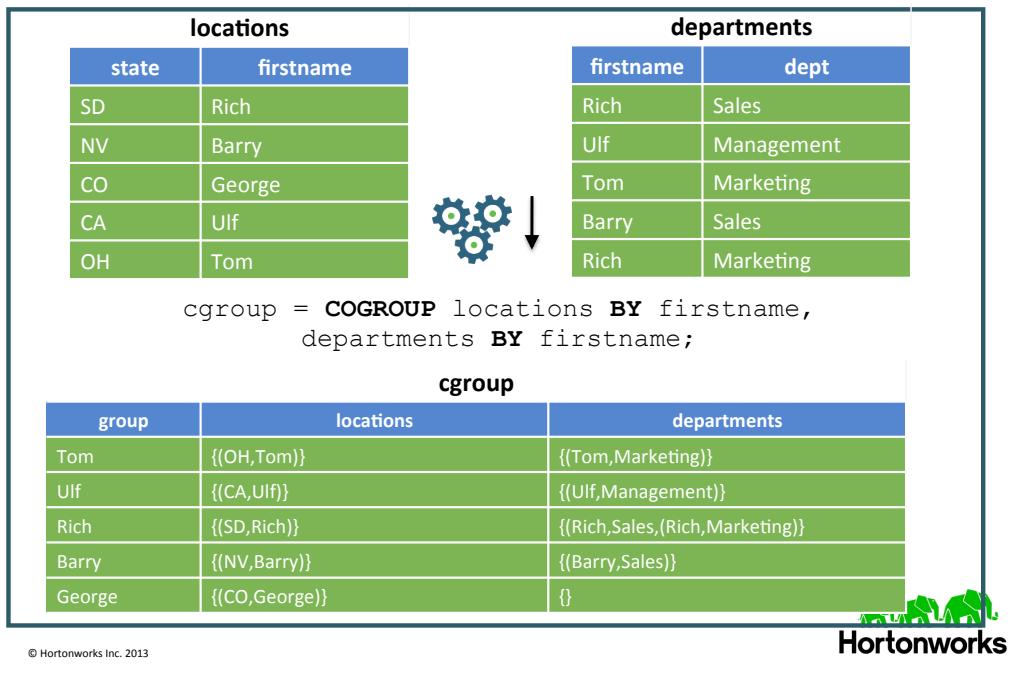
To take advantage of a replicated join, list the smaller data set last in the **BY** clause and follow it with a **USING 'replicated'** statement. For example:

```
replicatedjoin = JOIN locations BY firstname,
departments BY firstname USING 'replicated';
```

The **departments** data set will be distributed across all map tasks (using a feature of MapReduce called a LocalResource), and the join will occur in the map side instead of on the reduce side.

BEST PRACTICE: Use replicated joins whenever you can! The increase in performance is noticeable. Just be careful - if the data set does not fit in memory, the underlying MapReduce will generate an error and fail.

The COGROUP Operator



The COGROUP Operator

The **COGROUP** operator is actually identical to the **GROUP** operator, except we use **COGROUP** when grouping together more than one relation. For each input, the result of a **COGROUP** is a record with a key and one bag. You can think of a **COGROUP** as the first half of a **JOIN**: the keys are collected, but the cross product is not performed.

Let's look at an example using the **locations** and **departments** data:

```
> cgroup = COGROUP locations BY firstname,  
                      departments BY firstname;  
> DESCRIBE cgroup;  
cgroup: {group: chararray,  
locations: {  
    (state: chararray,  
     firstname: chararray)  
},  
departments: {  
    (firstname: chararray,  
     dept: chararray)}  
}
```

Notice the schema of the **cgroup** relation consists of a key followed by a bag for each data set. The output of **cgroup** is:

```
(Tom, { (OH, Tom) }, { (Tom, Marketing) })
(Ulf, { (CA, Ulf) }, { (Ulf, Management) })
(Rich, { (SD, Rich) }, { (Rich, Sales), (Rich, Marketing) })
(Barry, { (NV, Barry) }, { (Barry, Sales) })
(George, { (CO, George) }, { (George, Management) })
```

You could use the **cgroup** relation to count the number of records that would occur in the result of a join. For example:

```
counters = FOREACH cgroup GENERATE group, COUNT(locations),
            COUNT(departments);
```

The first number is the inner join count, and the second number the outer join count:

```
(Tom,1,1)
(Ulf,1,1)
(Mark,1,0)
(Rich,1,2)
(Barry,1,1)
(Brian,1,0)
(George,1,1)
(Manish,0,2)
(manish,1,0)
(Danielle,1,2)
```

NOTE: The only difference between **GROUP** and **COGROUP** is the readability. If you see **GROUP**, that implies the grouping of a single relation. If you see **COGROUP**, that implies the grouping of two or more relations.

Pig User-Defined Functions

You write a UDF in Java following these steps:

1. Write a Java class that extends `EvalFunc`.
2. Deploy the class in a JAR file.
3. Register the JAR file in the Pig script using the `REGISTER` command.
4. Optionally define an alias for the UDF using the `DEFINE` command

© Hortonworks Inc. 2012

Pig User-Defined Functions

The Pig API has a large collection of built-in functions for performing common tasks and computations. However, some Pig scripts may require **User-Defined Functions** (UDFs) to complete their task.

You write a UDF in Java following these steps:

- Write a Java class that extends `EvalFunc`.
- Deploy the class in a JAR file.
- Register the JAR file in the Pig script using the `REGISTER` command.
- Optionally define an alias for the UDF using the `DEFINE` command.

REFERENCE: The Pig API Javadocs are at <http://pig.apache.org/docs/r0.10.0/api/>.

A UDF Example

Let's take a look at an example. The following UDF adds a comma between two input strings:

```
package com.hortonworks.udfs;

public class CONCAT_COMMA extends EvalFunc<String> {

    @Override
    public String exec(Tuple input) throws IOException {
        String first = input.get(0).toString().trim();
        String second = input.get(1).toString().trim();

        return first + ", " + second;
    }
}
```

- Notice the `CONCAT_COMMA` class extends `EvalFunc`.
- The generic of `EvalFunc` represents the data type of the return value. Notice the `exec` method returns a `String`.
- The `exec` method is called when the UDF is invoked from the Pig script.
- The input parameter is a `Tuple` instance - which allows for an arbitrary number of arguments.
- Use the `get` method of `Tuple` to retrieve the arguments passed in.

Invoking a UDF

Before you can invoke a UDF, the function needs to be registered by your Pig script so that the Pig compiler knows where to find the definition of the UDF. Use the REGISTER command to register a JAR:

```
register my.jar;
```

You can specify a relative path or a full path to the JAR file. Once the JAR is registered, you call the UDF using its fully-qualified class name:

```
x = FOREACH logevents  
    GENERATE com.hortonworks.udfs.CONCAT_COMMA(level, code);
```

You can optionally use the DEFINE command to define an alias to simplify the syntax for invoking the UDF:

```
DEFINE CONCAT_COMMA com.hortonworks.udfs.CONCAT_COMMA();
```

Now you can invoke the UDF using the alias:

```
x = FOREACH logevents GENERATE CONCAT_COMMA(level, code);
```

Tips for Optimizing Pig Scripts

Here are a few best practices that can make a big difference in the performance of your Pig scripts:

- **Filter early and often:** getting rid of data as quickly as possible will improve the performance by reducing the amount of data that gets shuffled and sorted across the network.
- **Project early and often:** use a **FOREACH** to remove unwanted or unused fields in your records as soon as possible.
- **Drop nulls before a join:** filter out null records before the **JOIN**. The gain can be significant even if you have a small percentage of null values.
- **Use replicated joins whenever possible:** a map-side join is always much more efficient than a reduce-side join.
- **Use PARALLEL properly:** know your cluster. Setting this value too high can actually slow down the job, and setting it too low is not a good use of your cluster's resources.
- **Use compression:** enable the compression of the temporary data files used between map/reduce tasks and jobs by setting **mapreduce.map.output.compress** to true and specifying a compression codec with **mapreduce.map.output.compress.codec**. Enable compression of the output files between MapReduce jobs within a Pig processing pipeline by setting the **pig.tmpfilecompression** and **pig.tmpfilecompression.codec** properties.
- **Choose the right data types:** if you are treating a field as a specific data type, then define the type in the **LOAD** statement with a schema. This will avoid unnecessary data type conversions later.

TIP: When you start Pig, a special file named **.pigbootup** is searched for in the user's home folder and executed. The **.pigbootup** file is a great place to configure properties, register JAR files, define UDFs, and any other task that can be applied globally to all of your Pig scripts.

Lab 6.2: Joining Datasets

Objective:	Join two datasets in Pig.
Location of Files:	/root/labs/Lab6.2
Successful Outcome:	A file of members of Congress who have visited the White House.
Before You Begin:	If you are in the Grunt shell, exit it using the quit command. In this lab, you will write a Pig script in a text file.

Perform the following steps:

Step 1: Upload the Congress Data

- 1.1.** Put the file /root/labs/Lab6.2/congress.txt into the **whitehouse** directory in HDFS.
- 1.2.** Use the **hadoop fs -ls** command to verify the **congress.txt** file is in **whitehouse**, and use **hadoop fs -cat** to view its contents. The file contains the names and other information about the members of the U.S. Congress.

Step 2: Create a Pig Script File

- 2.1.** In this lab, you will not use the Grunt shell to enter commands. Instead, you will enter your script in a text file. Using a text editor, create a new file named **join.pig** in the **Lab6.2** folder.

- 2.2.** At the top of the file, add a comment:

```
--join.pig: joins congress.txt and visits.txt
```

Step 3: Load the White House Visitors

- 3.1.** Define the following **visitors** relations, which will contain the first and last names of all White House visitors:

```
visitors = LOAD 'whitehouse/visits.txt' USING  
PigStorage(',') AS (lname:chararray, fname:chararray);
```

That is the only data we are going to use from **visits.txt**.

Step 4: Define a Projection of the Congress Data

- 4.1.** Add the following load command that loads the '**congress.txt**' file into a relation named **congress**. The data is tab-delimited, so no special Pig loader is needed:

```
congress = LOAD 'whitehouse/congress.txt' AS (  
    full_title:chararray,  
    district:chararray,  
    title:chararray,  
    fname:chararray,  
    lname:chararray,  
    party:chararray  
) ;
```

- 4.2.** The names in **visits.txt** are all uppercase, but the names in **congress.txt** are not. Define a projection of the **congress** relation that consists of the following fields:

```
congress_data = FOREACH congress GENERATE  
    district,  
    UPPER(lname) AS lname,  
    UPPER(fname) AS fname,  
    party;
```

Step 5: Join the Two Datasets

- 5.1.** Define a new relation named **join_contact_congress** that is a **JOIN** of **visitors** and **congress_data**. Perform the join on both the first and last names.

- 5.2.** Use the **STORE** command to store the result of **join_contact_congress** into a directory named '**joinresult**'.

Step 6: Run the Pig Script

- 6.1.** Save your changes to **join.pig**.

6.2. Run the script using the following command:

```
# pig join.pig
```

6.3. Wait for the MapReduce job to execute. When it is finished, write down the number of seconds it took for the job to complete (by subtracting the **StartedAt** time from the **FinishedAt** time) and write down the result: _____

6.4. The type of join used is also output in the job statistics. Notice the statistics output has “**HASH_JOIN**” underneath the “**Features**” column, which means a hash join was used to join the two datasets.

Step 7: View the Results

7.1. The output will be in the **joinresult** folder in HDFS. Verify the folder was created:

```
# hadoop fs -ls -R joinresult
-rw-r--r-- 1 root hdfs 40892 joinresult/part-r-00000
```

7.2. View the resulting file:

```
# hadoop fs -cat joinresult/part-r-00000
```

The output should look like the following:

```
DUFFY SEAN WI07 DUFFY SEAN Republican
JONES WALTER NC03 JONES WALTER Republican
SMITH ADAM WA09 SMITH ADAM Democrat
CAMPBELL JOHN CA45 CAMPBELL JOHN Republican
CAMPBELL JOHN CA45 CAMPBELL JOHN Republican
SMITH ADAM WA09 SMITH ADAM Democrat
```

Step 8: Try Using Replicated on the Join

8.1. Delete the **joinresult** directory in HDFS:

```
# hadoop fs -rm -R joinresult
```

8.2. Modify your **JOIN** statement in **join.pig** so that it uses replication.

8.3. Save your changes **to join.pig** and run the script again.

8.4. Notice this time that the statistics output shows Pig used a “**REPLICATED_JOIN**” instead of a “**HASH_JOIN**”.

8.5. Compare the execution time of the **REPLICATED_JOIN** vs. the **HASH_JOIN**. Did you have any improvement or decrease in performance?

NOTE: Using replicated does not necessarily increase the join time. There are way too many factors involved, and this example is using small datasets. The point is that you should try both techniques (if one dataset is small enough to fit in memory) and determine which join algorithm is faster for your particular dataset and use case.

Step 9: Count the Results

9.1. In **join.pig**, comment out the **STORE** command:

```
--STORE join_contact_congress INTO 'joinresult';
```

You have already saved the output of the **JOIN**, so there is no need to perform the **STORE** command again.

9.2. Notice in the output of your **join.pig** script that we know which party the visitor belongs to: Democrat, Republican or Independent. Using the **join_contact_congress** relation as a starting point, see if you can figure out how to output the number of Democrat, Republican and Independent members of Congress that visited the White House. Name the relation **counters** and use the **DUMP** command to output the results:

```
DUMP counters;
```

HINT: When you group the **join_contact_congress** relation, group it by the **party** field of **congress_data**. You will need to use the **::** operator in the **BY** clause. It will look like:
congress_data::party

9.3. The correct results are shown here:

```
(Democrat, 637)  
(Republican, 351)  
(Independent, 2)
```

Step 10: Use the EXPLAIN Command

10.1. At the end of **join.pig**, add the following statement:

```
EXPLAIN counters;
```

If you do not have a **counters** relation, then use **join_contact_congress** instead.

10.2. Run the script again. The Logical, Physical and MapReduce plans should display at the end of the output.

10.3. How many MapReduce jobs did it take to run this job? _____

RESULT: You should have a folder in HDFS named **joinresult** that contains a list of members of Congress who have visited the White House (within the timeframe of the historical data in **visits.txt**).

SOLUTIONS: The **JOIN** and **STORE** commands in Step 5 look like:

```
join_contact_congress = JOIN visitors BY (lname, fname),
                           congress_data BY (lname, fname);
STORE join_contact_congress INTO 'joinresult';
```

A solution for Step 9 is:

```
join_group = GROUP join_contact_congress
               BY congress_data::party;
counters = FOREACH join_group GENERATE group,
               COUNT(join_contact_congress);
DUMP counters;
```

Lab 6.3: Preparing Data for Hive

Objective:	Transform and export a dataset for use with Hive.
Location of Files:	/root/labs/Lab6.3
Successful Outcome:	The resulting Pig script stores a projection of visits.txt in a folder in the Hive warehouse named wh_visits .
Before You Begin:	You should have visits.txt in a folder named whitehouse in HDFS.

Perform the following steps:

Step 1: Review the Pig Script

1.1. From a command prompt, change directories to **Lab6.3**:

```
# cd ~/labs/Lab6.3
```

1.2. View the contents of **wh_visits.pig**:

```
# more wh_visits.pig
```

1.3. Notice the **potus** relation is all White House visitors who met with the President.

1.4. Notice the **project_potus** relation is a projection of the last name, first name, time of arrival, location and comments from the visit.

Step 2: Store the Projection in the Hive Warehouse

2.1. Open **wh_visits.pig** with a text editor.

2.2. Add the following command at the bottom of the file, which stores the **project_potus** relation into a very specific folder in the Hive warehouse:

```
STORE project_potus INTO '/apps/hive/warehouse/wh_visits/';
```

Step 3: Run the Pig Script

3.1. Save your changes to **wh_visits.pig**.

3.2. Run the script from the command line:

```
# pig wh_visits.pig
```

Step 4: View the Results

4.1. The **wh_visits.pig** script creates a directory in the Hive warehouse named **wh_visits**. Use **ls** to view its contents:

```
# hadoop fs -ls /apps/hive/warehouse/wh_visits/
-rw-r--r-- 3 root hdfs 971339
/apps/hive/warehouse/wh_visits/part-m-00000
-rw-r--r-- 3 root hdfs 142850
/apps/hive/warehouse/wh_visits/part-m-00001
```

4.2. View the contents of one of the result files. It should look like the following:

```
hadoop fs -cat /apps/hive/warehouse/wh_visits/part-m-00000
...
FRIEDMAN THOMAS 10/12/2010 12:08 WH PRIVATE LUNCH
BASS EDWIN 10/18/2010 15:01 WH
BLAKE CHARLES 10/18/2010 15:00 WH
OGLETREE CHARLES 10/18/2010 15:01 WH
RIVERS EUGENE 10/18/2010 15:01 WH
```

RESULT: You now have a folder in the Hive warehouse named **wh_visits** that contains a projection of the data in **visits.txt**. We will use this file in an upcoming Hive lab.

Overview of the DataFu Library

- DataFu is a collection of Pig UDFs for data analysis on Hadoop
- Started by LinkedIn and open sourced under the Apache 2.0 license
- Includes functions for:
 - Bag and set operations
 - PageRank
 - Quantiles
 - Variance
 - Sessionization

© Hortonworks Inc. 2013



Overview of the DataFu Library

The DataFu library is an open source library of Pig UDFs for performing data analysis on Hadoop. DataFu contains UDFs for:

- Bag operations like append and concatenate
- Set operations like union and intersect
- Running PageRank on a collection of graphs
- Statistical computations like quantiles and variance
- Sessionization functions for working with page views

To use the functions in the DataFu library, you need to register the DataFu JAR file, just like you would with any other Pig UDF library:

```
register datafu-0.0.10.jar;
```

Computing Quantiles

```
define
Quintile datafu.pig.stats.Quantile
('0.0','0.20','0.40','0.60',
 '0.80','1.0');
```

```
temps_filter = FILTER temperatures
                  BY hightemp is not null;
temps_group = GROUP temps_filter
                  BY location;
quintiles = FOREACH temps_group {
    sorted = ORDER temps_filter BY hightemp;
    GENERATE group AS location,
        Quintile(sorted.hightemp) AS quant;
}
```

© Hortonworks Inc. 2013



Computing Quantiles

A **quantile** is a set of points from the cumulative distribution function of a random variable, taken at regular intervals. The number of points, **n**, is the name of the quantile. For example, if **n** = 4, you have a **4-quantile** (commonly called a *quartile*). If **n** = 5, you have a **5-quantile**, and so on.

The datafu library has a quantile UDF named datafu.pig.stats.Quantile that computes a quantile based on provided intervals and passed in to the UDF's constructor. For example, an evenly-distributed 5-quantile function would be defined as:

```
define Quintile datafu.pig.stats.Quantile('0.0','0.20',
                                             '0.40','0.60','0.80','1.0');
```

5-quantiles are known as *quintiles*, but we could have used any alias. Invoking this UDF requires passing in a sorted bag. This is typically accomplished using a nested FOREACH.

Here is what the entire Pig script might look like for computing the quintiles of a collection of high temperatures gathered at various weather stations:

```
register datafu-0.0.10.jar;

define Quintile datafu.pig.stats.Quantile('0.0','0.20',
                                             '0.40','0.60','0.80','1.0');

temperatures = LOAD 'data.txt' AS (
    location:chararray,
    hightemp:double,
    lowtemp:double
);

temps_filter = FILTER temperatures BY hightemp is not null;
temps_group = GROUP temps_filter BY location;

quintiles = FOREACH temps_group {
    sorted = ORDER temps_filter BY hightemp;
    GENERATE group AS location,
                Quintile(sorted.hightemp) AS quant;
}

dump quintiles;
```

The output for each location is going to be six values, which define five equally numerous subsets of the high temperatures:

```
(Toronto, (3.22,7.48,13.6,16.05,21.49,41.5))
(Moscow, (9.0,22.04,28.5,36.975,43.205,69.98))
(NorthPole, (-20.5,-14.6,-8.76,-2.57,1.475,2.445,3.61))
(Houston, (21.9,30.69,35.41,40.75,50.55,73.87))
(IntlFalls, (-14.41,-4.25,-1.15,12.15,24.6,36.73))
```

For example, in Toronto you have an equal number of days where the high temp was between 3.22 and 7.48 degrees Celcius, between 7.48 and 13.6 degrees, between 13.6 and 16.05 degrees, and so on.

Demonstration: Computing PageRank

Objective:	To understand how to use the PageRank UDF in DataFu.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: View the Data

1.1. Review the `edges.txt` file in the `/root/labs/demos` folder:

```
# cd ~/labs/demos/
# more edges.txt
0      2      3      1.0
0      3      2      1.0
0      4      1      1.0
0      4      2      1.0
0      5      4      1.0
0      5      2      1.0
0      5      6      1.0
0      6      5      1.0
0      6      2      1.0
0     100     2      1.0
0     100     5      1.0
0     101     2      1.0
0     101     5      1.0
0     102     2      1.0
0     102     5      1.0
0     103     5      1.0
0     104     5      1.0
```

1.2. The first column is the topic, but since we only have a single graph, the topic is 0 for all the edges.

1.3. The second and third columns are the source and destination of each edge. For example, there is an edge from 2 to 3 based on the first row.

1.4. The fourth column is the weight of the edge. Our graph is all evenly weighted.

1.5. Based on the data above, which pages should be ranked toward the top?

Step 2: Put the Data in HDFS

2.1. Put **edges.txt** into HDFS:

```
# hadoop fs -put edges.txt edges.txt
```

Step 3: Define the PageRank UDF

3.1. View the contents of **demos/pagerank.pig**. The first two lines register the DataFu library and define the **PageRank** function:

```
register /root/labs/Lab10.1/datafu-0.0.10.jar;
define PageRank datafu.pig.linkanalysis.PageRank();
```

3.2. The edges are loaded and grouped by **topic** and **source**:

```
topic_edges = LOAD '/user/root/edges.txt' as
    (topic:INT,source:INT,dest:INT,weight:DOUBLE);
topic_edges_grouped = GROUP topic_edges by (topic, source);
```

3.3. The data is then prepared for the **PageRank** function, which is expecting a topic, a source, and its edges:

```
topic_edges_data = FOREACH topic_edges_grouped GENERATE
    group.topic as topic,
    group.source as source,
    topic_edges.(dest,weight) as edges;
```

3.4. We only have one topic, but the edges still need to be grouped by topic:

```
topic_edges_data_by_topic = GROUP topic_edges_data
    BY topic;
```

3.5. We can now invoke the **PageRank** function:

```
topic_ranks = FOREACH topic_edges_data_by_topic GENERATE
    group as topic,
    FLATTEN(PageRank(topic_edges_data.(source,edges)));
```

3.6. The results are stored in HDFS:

```
store topic_ranks into 'topicranks';
```

Step 4: Run the Script

- 4.1.** From the command prompt, enter:

```
# pig pagerank.pig
```

The job will take a couple minutes to run.

Step 5: View the Results

- 5.1.** View the contents of the **topicranks** folder in HDFS:

```
# hadoop fs -ls topicranks
Found 1 items
root hdfs      181    topicranks/part-r-00000
```

- 5.2.** View the contents of the output file:

```
# hadoop fs -cat topicranks/part-r-00000
0      104  0.013636362
0      1    0.02764593
0      103  0.013636362
0      5    0.06821412
0      100  0.013636362
0      102  0.013636362
0      6    0.032963693
0      3    0.2891899
0      2    0.32418048
0      4    0.032963693
0      101  0.013636362
```

Step 6: Analyze the Results

- 6.1.** Which page should be ranked the highest? _____

- 6.2.** Which page should be ranked the lowest? _____

- 6.3.** Compare the actual results with your guess from Step 1.

Unit 6 Review

1. If a relation is sorted using **ORDER BY** and the resulting MapReduce job runs with 3 reducers, how is the output actually sorted? _____

Suppose the **prices.csv** file looks like:

```
XFR,2004-05-13,22.90,400
XFR,2004-05-12,22.60,400000
XFR,2004-05-11,22.80,2600
XFR,2004-05-10,23.00,3800
XFR,2004-05-07,23.55,2900
XFR,2004-05-06,24.00,2200
```

And assume we have the following relation defined:

```
prices = load 'prices.csv' using PigStorage(',')  
as (symbol:chararray, date:chararray, price:double,  
volume:int);
```

Explain what each of the following Pig commands or relations do:

2. F = foreach prices generate
(volume > 3000 ? volume : -1);

3. G = distinct prices; _____
4. H = GROUP prices BY symbol;
I = foreach H {
 J = filter prices by volume > 3000;
 GENERATE group, SUM(J.price);
};

5. What is the benefit of the **using 'replicated'** clause in a Pig join? _____

6. Why is filtering and projecting a relation early a performance benefit in Pig?

Lab 6.4: Analyzing Clickstream Data

Objective:	Become familiar with using the DataFu library to sessionize clickstream data.
Location of Files:	/root/labs/Lab6.4
Successful Outcome:	You will have computed the length of each session, along with the average and median values of all session lengths.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: View the Clickstream Data

1.1. Open a command prompt and change directories to **Lab6.4**.

1.2. View the contents of **clicks.csv**:

```
more clicks.csv
```

The first column is the user's **id**, the second column is the **time** of the click stored as a long, and the third column is the **URL** visited.

1.3. Put the file in HDFS:

```
hadoop fs -put clicks.csv clicks.csv
```

Step 2: Define the Sessionize UDF

2.1. Using a text editor, open the file **sessions.pig** in the **Lab6.4** folder.

2.2. Notice two JAR files are registered: **datafu-0.0.10.jar** and **piggybank.jar**. The datafu JAR contains the **Sessionize** function that you are going to use, and the **piggybank.jar** contains a time utility function named **UnixToISO**, which is already defined for you in this Pig script.

2.3. Add the following **DEFINE** statement to define the **Sessionize** UDF:

```
DEFINE Sessionize datafu.pig.sessions.Sessionize('8m');
```

2.4. What does the '**8m**' mean in the constructor? _____

Step 3: Sessionize the Clickstream

3.1. Notice the **clicks.csv** file is loaded for you in **sessions.pig**:

```
clicks = LOAD 'clicks.csv' USING PigStorage(',')  
AS (id:int, time:long, url:chararray);
```

3.2. Notice also that the **clicks** relation is projected onto **clicks_iso** with the long converted to an ISO time format, then grouped by **id** in the **clicks_group** relation:

```
clicks_iso = FOREACH clicks GENERATE UnixToISO(time)  
AS isotime, time, id;  
clicks_group = GROUP clicks_iso BY id;
```

3.3. Sessionize the clickstream by adding the following nested **FOREACH** loop:

```
clicks_sessionized = FOREACH clicks_group {  
    sorted = ORDER clicks_iso BY isotime;  
    GENERATE FLATTEN(Sessionize(sorted))  
    AS (isotime, time, id, sessionid);  
}
```

3.4. Dump the sessionized data:

```
dump clicks_sessionized;
```

3.5. Save your changes to **sessions.pig**.

Step 4: Run the Script

4.1. Let's verify the **Sessionized** function is working by running the script:

```
pig sessions.pig
```

4.2. Verify the tail of the output looks similar to the following:

```
(2013-01-10T07:15:20.520Z,1357802120520,2,51d89b38-b14a-  
4158-8703-724525d9f787)
```

```
(2013-01-10T07:15:39.797Z,1357802139797,2,51d89b38-b14a-4158-8703-724525d9f787)
(2013-01-10T07:26:30.602Z,1357802790602,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:26:53.357Z,1357802813357,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:26:58.800Z,1357802818800,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:27:05.253Z,1357802825253,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:27:57.844Z,1357802877844,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:28:20.610Z,1357802900610,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:29:01.128Z,1357802941128,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:29:02.190Z,1357802942190,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:29:23.190Z,1357802963190,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:30:04.181Z,1357803004181,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
(2013-01-10T07:30:32.455Z,1357803032455,2,711525c4-eff6-4697-ade7-e2ad5ec555e5)
```

Step 5: Compute the Session Length

5.1. Comment out the **dump** statement:

```
--dump clicks_sessionized
```

5.2. Define a projection named **sessions** that is a projection of only the **time** and **sessionid** fields of the **clicks_sessionized** relation.

5.3. Define a relation named **sessions_group** that is the **sessions** relation grouped by **sessionid**.

5.4. Define a **session_times** relation using the following projection that computes the length of each session:

```
session_times = FOREACH sessions_group
    GENERATE group as sessionid,
        (MAX(sessions.time) - MIN(sessions.time)) / 1000.0 / 60
            as session_length;
```

5.5. Dump the **session_times** relation:

```
dump session_times;
```

5.6. Save your changes to sessions.pig and run the script. The output should look like the following:

```
(01e5259c-c5a6-45b0-8d04-1be86182d12e,0.16571666666666665)
(164be386-1df2-40dd-9331-563e1b8a7275,4.030883333333334)
(16ab9225-28d3-45f6-9d07-f065223046bb,38.809916666666666)
(18362695-d032-424a-a983-33ab45638700,0.0)
(2699ef77-bd37-4611-a239-ddbd8006043,10.398116666666665)
(3077f9d1-a5d5-4bf9-8212-87ae848b4ed8,3.44485)
(3e732d19-e3ed-4cc4-810f-f05c8534fb28,1.140283333333333)
(455183ea-c3bb-43fe-9f07-63e0c019008,14.648516666666666)
(5a65d8dc-1a4e-4355-b86a-f1efc519b084,63.620149999999995)
(5ef45fc4-01df-40d8-805f-a61c60fc421e,0.0317333333333333)
(61e14bcf-1fb4-4f7e-a3b4-2b67b8840756,1.081983333333333)
(63b53f03-31e9-4a01-8029-6334020080e4,4.48765)
(66f58bc2-7aeb-487d-a28e-21090578cfe2,22.9298)
(812a7fc4-9ea2-4c3b-a3da-17bbd740a49a,0.0061833333333333)
(84f8c113-d3c9-4590-83a8-5a9edf44c5c5,86.69525)
(85cd8b8c-644b-4fb9-a6c6-3b5082d32f0c,2.50913333333333)
(8e4cfed7-8500-47bb-a5e9-3744de6b1595,0.0)
(a35be8db-de7b-4b55-a230-66389a4e4b5f,0.971316666666667)
(bcfe9fa-fd71-4962-8a0b-ddcf77ea47a3,0.372466666666667)
(c092d0c4-3c7d-4cfb-b7f9-078baaa7469f,1.645333333333333)
(d1d1b88e-b827-4005-b088-233d56c4ea8f,0.660833333333333)
(e0f48349-1d2a-4cd7-8258-e36b4b6118fc,31.8878833333333)
(e1ccdf96-fc37-4b7e-9a7c-95acb8f52fa7,0.0)
(fd92f410-19fc-4927-917f-0f86b5d7edb2,17.19768333333334)
(fdfecea38-ddf9-477a-bb3e-401e8874e0ac,2.251233333333334)
(ff70c6b5-abb2-4606-b12f-3054501947a4,0.051183333333334)
```

5.7. How long was the longest session? _____

Step 6: Compute the Average Session Length

6.1. Comment out the **dump** statement:

```
--dump session_times;
```

6.2. Define a relation named **sessiontimes_all** that is a grouping of all **session_times**.

6.3. Define **sessiontimes_avg** using the following nested **FOREACH** statement:

```
sessiontimes_avg = FOREACH sessiontimes_all {
    ordered = ORDER session_times BY session_length;
    GENERATE AVG(ordered.session_length)
```

```
        AS avg_session;  
}
```

6.4. Dump the **sessiontimes_avg** relation:

```
dump sessiontimes_avg;
```

6.5. Save your changes to **sessions.pig** and run the script.

6.6. Verify the output, which should be a single value representing the average session time:

```
(11.88608076923077)
```

NOTE: This value is hard to find within the all the logging output. You may need to search carefully for the output!

Step 7: Compute the Median Session Length

7.1. Using the **sessiontimes_avg** relation as an example, compute the median session time. You will need to define the **Median** function from the datafu library, which is named **datafu.pig.stats.Median()**.

7.2. Verify you got the following value for the median session length:

```
(1.948283333333334)
```

RESULT: You have taken clickstream data and sessionized it using Pig to determine statistical information about the sessions, like the length of each session and the average and median lengths of all sessions.

ANSWERS:

2.4: The '8m' stands for 8 minutes, which is the length of the session. You can pick any length of time you want to define your sessions.

5.7: The longest session was 86.69525 minutes.

SOLUTIONS:

Step 5.2:

```
sessions = FOREACH clicks_sessionized  
    GENERATE time, sessionid;
```

Step 5.3:

```
sessions_group = GROUP sessions BY sessionid;
```

Step 6.2:

```
sessiontimes_all = GROUP session_times ALL;
```

Step 7.1: A quick solution for computing the median is to simply add it to the existing nested **FOREACH** statement:

```
sessiontimes_avg = FOREACH sessiontimes_all {  
    ordered = ORDER session_times BY session_length;  
    GENERATE  
        AVG(ordered.session_length) AS avg_session,  
        Median(ordered.session_length) AS median_session;  
}
```

Lab 6.5: Analyzing Stock Market Data using Quantiles

Objective:	Use DataFu to compute quantiles.
Location of Files:	/root/labs/Lab6.5
Successful Outcome:	You will have computed quartiles for the daily high prices of stocks traded on the New York Stock Exchange.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: Review the Stock Market Data

1.1. From the command prompt, change directories to the **Lab6.5** folder.

1.2. View the contents of the **stocks.csv** file, which contains the historical prices for New York Stock Exchange stocks that begin with the letter “Y”:

```
# tail stocks.csv
```

The first column is always “NYSE”. The second column is the stock’s symbol. The third column is the date that the prices occurred. The next columns are the open, high, low, close and trading volume.

1.3. Put **stocks.csv** into your **/user/root** folder in HDFS:

```
# hadoop fs -put stocks.csv stocks.csv
```

Step 2: Define the Quantile Function

2.1. Create a new text file in the **Lab6.5** folder named **quantile.pig**.

2.2. On the first line of the file, register the datafu JAR file.

2.3. Define the **datafu.pig.stats.Quantile** function as **Quantile**, and pass in the values for computing the *quartiles* of a set of numbers:

```
define Quantile datafu.pig.stats.Quantile(  
    '0.0','0.25','0.50','0.75','1.0');
```

Step 3: Load the Stocks

3.1. Enter the following **LOAD** command, which loads the first five values of each row:

```
stocks = LOAD 'stocks.csv' USING PigStorage(',') AS  
    (nyse:chararray,  
     symbol:chararray,  
     closingdate:chararray,  
     low:double,  
     highprice:double);
```

Step 4: Filter Null Values

4.1. The **Quantile** function fails if any of the values passed to it are null. Define a relation named **stocks_filter** that filters the stocks relation where the **highprice** is not null.

Step 5: Group the Values

5.1. We want to compute the quantiles for each individual stock (as opposed to all the stocks prices that start with a "Y"), so define a relation named **stocks_group** that groups the **stock_filter** relation by **symbol**.

Step 6: Compute the Quantiles

6.1. Define the following relation that invokes the Quantile method on the highprice values:

```
quantiles = FOREACH stocks_group {  
    sorted = ORDER stocks_filter BY highprice;  
    GENERATE group AS symbol,  
        Quantile(sorted.highprice) AS quant;  
}
```

6.2. How many times will the Quantile function be invoked in the nested FOREACH statement above? _____

6.3. Add a DUMP statement that outputs the quantiles relation:

```
DUMP quantiles;
```

Step 7: Run the Script

7.1. Save your changes to **quantile.pig**.

7.2. Run the script:

```
pig quantile.pig
```

7.3. There are only five stocks in the input data, so the output will be the quartiles of the high price of these five stocks:

```
(YGE, (3.22,10.97,14.79,19.6,41.5))  
(YPF, (9.0,23.62,31.94,41.47,69.98))  
(YSI, (1.56,8.04,16.43500000000002,19.93,23.61))  
(YUM, (21.9,32.08,37.85,48.91,73.87))  
(YZC, (4.41,14.4,20.795,47.13,116.73))
```

Step 8: Compute the Median

8.1. Now that you have a working Pig script for computing quantiles of the high prices of stocks, see if you can modify the script (you only have to make a few changes) to compute the median value of the high prices.

RESULT: You have used the DataFu library to compute quantiles of a collection of numbers using Pig.

ANSWERS:

6.2: The **FOREACH** statement iterates over the **stocks_group**, which is a grouping by symbol. So the **Quantile** function will be invoked once for each unique stock symbol in the **stocks.csv** file.

SOLUTIONS:

Step 4:

```
stocks_filter = FILTER stocks BY highprice is not null;
```

Step 5:

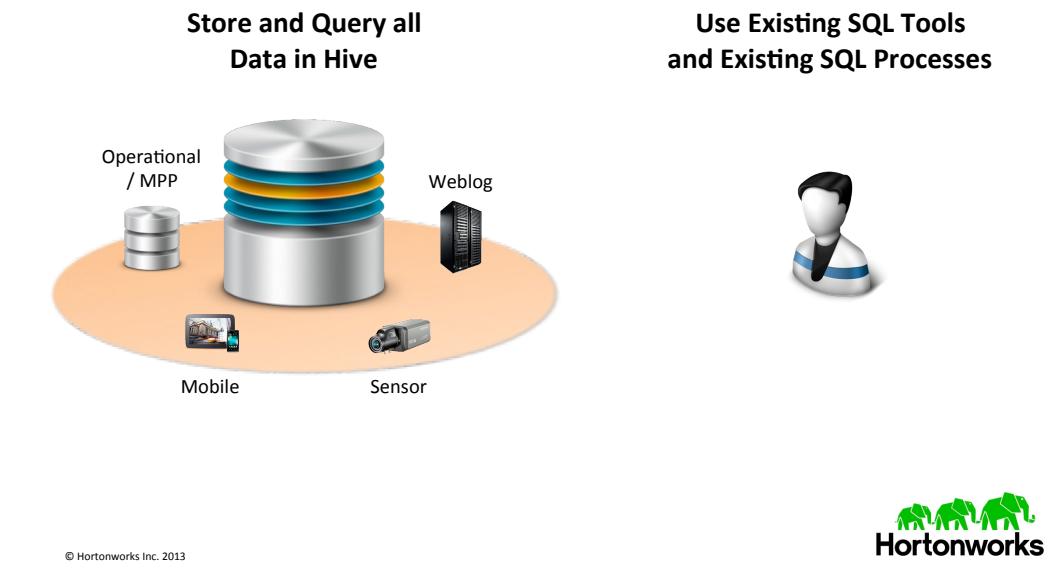
```
stocks_group = GROUP stocks_filter BY symbol;
```

Unit 7: Hive Programming

Topics covered:

- What is Hive?
- Comparing Hive to SQL
- Hive Architecture
- Submitting Hive Queries
- Defining Tables
- Loading Data into Hive
- Performing Queries
- Lab 7.1: Understanding Hive Tables
- Hive Partitions, Buckets and Skewed Tables
- Demonstration: Understanding Partitions and Skew
- Sorting Data
- Using Distribute By
- Storing Results to a File
- Specifying MapReduce Properties
- Lab 7.2: Analyzing Big Data with Hive
- Lab 7.3: Understanding MapReduce in Hive
- Joining Data and Hive Join Strategies
- Invoking a Hive UDF
- Computing ngrams in Hive
- Lab 7.4: Joining Datasets in Hive
- Lab 7.5: Computing ngrams of Emails in Avro Format

What is Hive?



What is Hive?

Apache Hive is a data warehouse system for Hadoop. Hive is not a relational database, it only maintains metadata information about your Big Data stored on HDFS. Hive allows you to treat your Big Data as tables and perform SQL-like operations on the data using a scripting language called **HiveQL**.

- Hive is not a database, but it uses a database (called the **metastore**) to store the tables that you define. Hive uses Derby by default.
- A Hive table consists of a schema stored in the metastore and data stored on HDFS.
- Hive converts HiveQL commands into MapReduce jobs (similar to how Pig Latin scripts execute with Pig).
- One of the key benefits of HiveQL is its similarity to SQL. Data analysts familiar with SQL can run MapReduce jobs by writing SQL-like queries - something they are already comfortable doing!
- You can easily perform ad-hoc, custom queries on HDFS using Hive.

Pig and Hive have quite a few similarities, so you might be wondering which framework to choose for your particular application. For most use cases:

- Pig is a good choice for ETL jobs, where unstructured data is reformatted so that it is easier to define a structure to it.
- Hive is a good choice when you want to query data that has a certain known structure to it.

In other words, you will likely benefit from using both Pig and Hive! Pig is great for moving data around and restructuring it, while Hive is great for performing analyses on the data.

NOTE: Hive does not make any promises regarding performance. The benefit of Hive is its simplicity in being able to define and run a MapReduce job, but the queries are not meant to execute in real-time. Even the simplest of Hive queries can take several minutes to execute (just like any MapReduce job), and large Hive queries can feasibly take hours to run.

Comparing Hive to SQL

SQL Datatypes	SQL Semantics
INT	SELECT, LOAD, INSERT from query
TINYINT/SMALLINT/BIGINT	Expressions in WHERE and HAVING
BOOLEAN	GROUP BY, ORDER BY, SORT BY
FLOAT	CLUSTER BY, DISTRIBUTE BY
DOUBLE	Sub-queries in FROM clause
STRING	GROUP BY, ORDER BY
BINARY	ROLLUP and CUBE
TIMESTAMP	UNION
ARRAY, MAP, STRUCT, UNION	LEFT, RIGHT and FULL INNER/OUTER JOIN
DECIMAL	CROSS JOIN, LEFT SEMI JOIN
CHAR	Windowing functions (OVER, RANK, etc.)
VARCHAR	Sub-queries for IN/NOT IN, HAVING
DATE	EXISTS / NOT EXISTS
	INTERSECT, EXCEPT

© Hortonworks Inc. 2014



Comparing Hive to SQL

Hive provides basic SQL functionality using MapReduce to execute queries. Hive supports standard SQL clauses:

```
INSERT INTO
SELECT
FROM ... JOIN ... ON
WHERE
GROUP BY
HAVING
ORDER BY
LIMIT
```

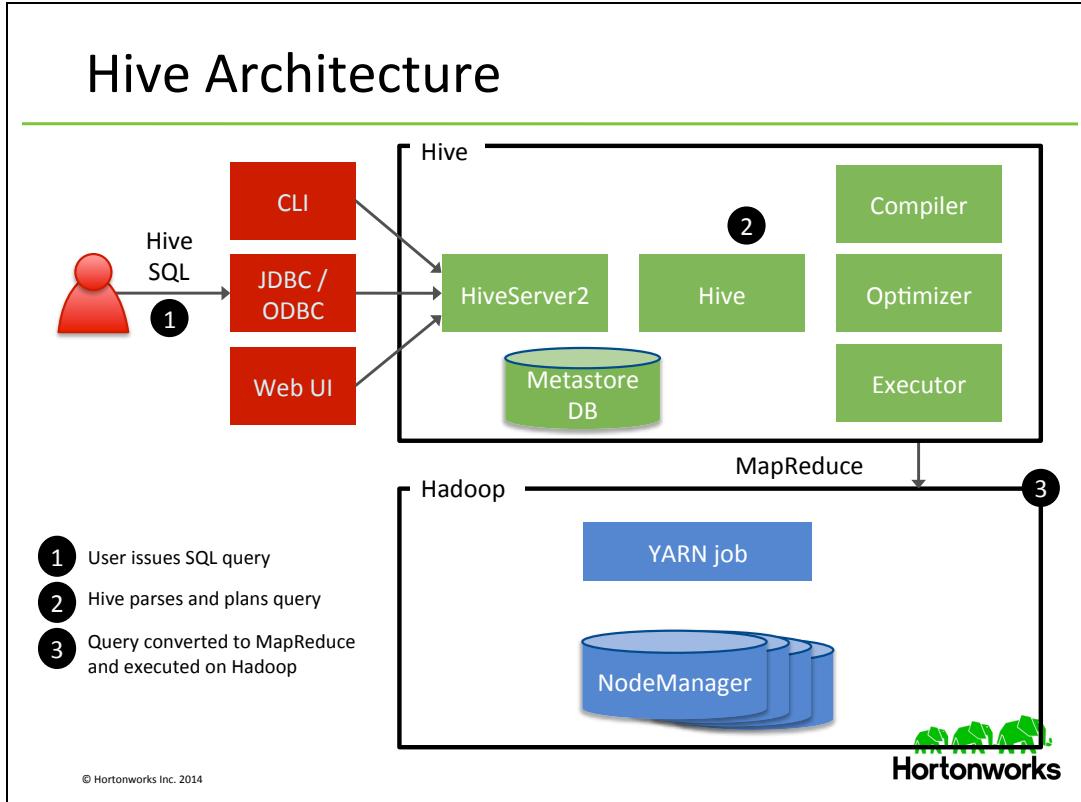
Hive also supports basic DDL commands:

```
CREATE/ALTER/DROP TABLE/DATABASE
```

Some of the limitations of Hive include:

- Limited index and view support (discussed in detail later)
- The data in Hive is read only (no updates)
- Subqueries are only allowed in a **FROM** clause
- Datatypes do not line up with traditional SQL types
- Very limited security
- You can insert new partitions only, but not individual rows

Hive Architecture



Hive Architecture

Step 1: Issuing Commands

- Using the Hive CLI, a Web interface, or a Hive JDBC/ODBC client, a Hive query is submitted to the HiveServer.

Step 2: Hive Query Plan

- The Hive query is compiled, optimized and planned as a MapReduce job.

Step 3: MapReduce Job Executes

- The corresponding MapReduce job is executed on the Hadoop cluster.

NOTES

Submitting Hive Queries

- **Hive CLI**
 - Traditional Hive client that connects to a HiveServer instance
 - ```
$ hive
hive>
```
- **Beeline**
  - A new command line client that connects to a HiveServer2 instance
  - ```
$ beeline
Hive version 0.11.0-SNAPSHOT by Apache
beeline>
```

© Hortonworks Inc. 2013



Submitting Hive Queries

Hive queries are written using the ***HiveQL*** language, a SQL-like scripting language that simplifies the creation of MapReduce jobs. With HiveQL, data analysts can focus on answering questions about the data, and let the Hive framework convert the HiveQL into a MapReduce job.

You have two options for executing HiveQL commands:

- **Hive CLI:** The Hive command line interface allows you to enter commands directly in to the Hive shell, or write the commands in a text file and execute the file.
- **Beeline:** a new JDBC client that works with HiveServer2. The Beeline shell works in embedded mode (just like the Hive CLI) and also remote mode, where you connect to a HiveServer2 process using Thrift.

The Hive CLI shell is started using the **hive** executable:

```
$ hive -h hostname
hive>
```

Use the **-f** flag to specify a file that contains a Hive script:

```
$ hive -f myquery.hive
```

Beeline is started using the **beeline** executable:

```
$ beeline
Hive version 0.11.0-SNAPSHOT by Apache
beeline>
```

Once Beeline is started, you issue a connect command to connect to a specific HiveServer2 instance:

```
beeline> !connect jdbc:hive2://hostname:10000 username
password org.apache.hive.jdbc.HiveDriver
```

Defining a Hive-Managed Table

A Hive table allows you to add structure to your otherwise unstructured data in HDFS. Use the **CREATE TABLE** command to define a Hive table, similar to creating a table in SQL.

For example, the following HiveQL creates a new Hive-managed table named **customer**:

```
CREATE TABLE customer (
    customerID INT,
    firstName STRING,
    lastName STRING,
    birthday TIMESTAMP,
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

- The customer table has four columns.
- **ROW FORMAT** is either **DELIMITED** or **SERDE**.
- Hive supports the following data types: **TINYINT**, **SMALLINT**, **INT**, **BIGINT**, **BOOLEAN**, **FLOAT**, **DOUBLE**, **STRING**, **BINARY** and **TIMESTAMP**.
- Hive also has four complex data types: **ARRAY**, **MAP**, **STRUCT** and **UNIONTYPE**.

Defining an External Table

The following **CREATE** statement creates an external table named **salaries**:

```
CREATE EXTERNAL TABLE salaries (
    gender string,
    age int,
    salary double,
    zip int
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

An external table is just like a Hive-managed table, except that when the table is dropped, Hive will not delete the underlying **/apps/hive/warehouse/salaries** folder.

Defining a Table LOCATION

Hive does not have to store the underlying data in **/apps/hive/warehouse**. Instead, the files for a Hive table can be stored in a folder anywhere in HDFS by defining the **LOCATION** clause. For example:

```
CREATE EXTERNAL TABLE salaries (
    gender string,
    age int,
    salary double,
    zip int
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/user/train/salaries/';
```

In the table above, the table data for **salaries** will be whatever is in the **/user/train/salaries** directory.

IMPORTANT: The sole difference in behavior between external tables and Hive-managed tables is when they are dropped. If you drop a Hive-managed table, then its underlying data is deleted from HDFS. If you drop an external table, then its underlying data remains in HDFS (even if the **LOCATION** was in **/apps/hive/warehouse**).

Loading Data into a Hive Table

The data for a Hive table resides in HDFS. To associate data with a table, use the **LOAD DATA** command. The data does not actually get “loaded” into anything, but the data does get moved:

- For Hive-managed tables, the data is moved into a special Hive subfolders of **/apps/hive/warehouse**.
- For external tables, the data is moved to the folder specified by the **LOCATION** clause in the table’s definition.

The **LOAD DATA** command can load files from the local file system (using the **LOCAL** qualifier) or files already in HDFS. For example, the following command loads a local file into a table named **customers**:

```
LOAD DATA LOCAL INPATH '/tmp/customers.csv' OVERWRITE INTO  
TABLE customers;
```

- The **OVERWRITE** option deletes any existing data in the table and replaces it with the new data. If you want to append data to the table’s existing contents, simply leave off the **OVERWRITE** keyword.

If the data is already in HDFS, then leave off the **LOCAL** keyword:

```
LOAD DATA INPATH '/user/train/customers.csv' OVERWRITE INTO  
TABLE customers;
```

In either case above, the file **customers.csv** is moved either into HDFS in a subfolder of **/apps/hive/warehouse** or to the table’s **LOCATION** folder, and the contents of **customers.csv** are now associated with the **customers** table.

You can also insert data into a Hive table that is the result of a query, which is a common technique in Hive. The syntax looks like:

```
INSERT INTO birthdays SELECT firstName, lastName, birthday  
FROM customers WHERE birthday IS NOT NULL;
```

The **birthdays** table will contain all customers whose **birthday** column is not null.

Performing Queries

Let's take a look at some sample queries to demonstrate what HiveQL looks like. The following SELECT statement selects all records from the **customers** table:

```
SELECT * FROM customers;
```

You can use the familiar WHERE clause to specify which rows to select from a table:

```
FROM customers SELECT firstName, lastName, address, zip  
WHERE orderID > 0 GROUP BY zip;
```

NOTE: The **FROM** clause in Hive can appear before or after the **SELECT** clause.

One nice benefit of Hive is its ability to join data in a simple fashion. The JOIN command in HiveQL is similar to its SQL counterpart. For example, the following statement performs an inner join on two tables:

```
SELECT customers.*, orders.* FROM customers JOIN orders ON  
(customers.customerID = orders.customerID);
```

To perform an outer join, use the OUTER keyword:

```
SELECT customers.*, orders.* FROM customers LEFT OUTER JOIN  
orders ON (customers.customerID = orders.customerID);
```

In the SELECT above, a row will be returned for every customer, even those without any orders.

Lab 7.1: Understanding Hive Tables

Objective:	Understand how Hive table data is stored in HDFS.
Location of Files:	/root/labs/7.1
Successful Outcome:	A new Hive table filled with the data from the wh_visits folder.
Before You Begin:	Complete Lab 6.3, or put the data from the solution of Lab 6.3 into HDFS.

Perform the following steps:

Step 1: Review the Data

1.1. Use the **ls** command to view the contents of the **wh_visits** folder. You should see six **part-m** files:

```
# hadoop fs -ls /apps/hive/warehouse/wh_visits/
```

1.2. Recall that the Pig projection to create these files had the following schema:

```
project_potus = FOREACH potus GENERATE
    $0 AS lname:chararray,
    $1 AS fname:chararray,
    $6 AS time_of_arrival:chararray,
    $11 AS appt_scheduled_time:chararray,
    $21 AS location:chararray,
    $25 AS comment:chararray ;
```

In this lab, you will define a Hive table that matches these records and contains the exported data from your Pig script.

Step 2: Define a Hive Script

2.1. In the **Lab7.1** folder, there is a text file named **wh_visits.hive**. View its contents. Notice it defines a Hive table named **wh_visits** with the following schema that matches the data in your **project_potus** folder:

```
# more wh_visits.hive
create table wh_visits (
    lname string,
    fname string,
    time_of_arrival string,
    appt_scheduled_time string,
    meeting_location string,
    info_comment string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' ;
```

NOTE: You cannot use **comment** or **location** as column names because those are reserved Hive keywords, so we changed them slightly.

2.2. Run the script with the following command:

```
# hive -f wh_visits.hive
```

2.3. If successful, you should see “OK” in the output along with the time it took to run the query.

Step 3: Verify the Table Creation

3.1. Start the Hive Shell:

```
# hive
hive>
```

3.2. From the **hive>** prompt, enter the “**show tables**” command:

```
hive> show tables;
```

You should see **wh_visits** in the list of tables.

3.3. Use the **describe** command to view the details of **wh_visits**:

```
hive> describe wh_visits;
```

```

OK
lname          string      None
fname          string      None
time_of_arrival string      None
appt_scheduled_time string      None
meeting_location string      None
info_comment   string      None

```

3.4. Try running a query (even though the table is empty):

```
select * from wh_visits limit 20;
```

You should see 20 rows returned. How is this brand new Hive table already populated with records? _____

3.5. Why did the previous query not require a MapReduce job to execute?

3.6. Try the following query. Make sure the output looks like first names:

```
hive> select fname from wh_visits limit 20;
```

This time a MapReduce job executed. Why? _____

Step 4: Count the Number of Rows in a Table

4.1. Enter the following Hive query, which outputs the number of rows in **wh_visits**:

```
hive> select count(*) from wh_visits;
```

4.2. How many rows are currently in **wh_visits**? _____

Step 5: Selecting the Input File Name

5.1. Hive has two virtual columns that gets created automatically for every table: **INPUT_FILE_NAME** and **BLOCK_OFFSET_INSIDE_FILE**. You can use these column names in your queries just like any other column of the table. To demonstrate, run the following query:

```
hive> select INPUT_FILE_NAME, lname, fname FROM wh_visits
WHERE lname LIKE 'Y%';
```

5.2. The result of this query is visitors to the White House whose last name starts with “Y”. Notice that the output also contains the particular file that the record was found in:

```
hdfs://sandbox.hortonworks.com:8020/apps/hive/warehouse/wh_
visits/part-m-00001      YOUNG LEDISI
hdfs://sandbox.hortonworks.com:8020/apps/hive/warehouse/wh_
visits/part-m-00002      YARNOLD      DAVID
```

Step 6: Drop a Table

6.1. Let’s see what happens when a managed table is dropped. Start by defining a simple table called **names** using the Hive Shell:

```
hive> create table names (id int, name string)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

6.2. Use the Hive **dfs** command to put **Lab7.1/names.txt** into the table’s warehouse folder:

```
hive> dfs -put /root/labs/Lab7.1/names.txt
      /apps/hive/warehouse/names/;
```

6.3. View the contents of the table’s warehouse folder:

```
hive> dfs -ls /apps/hive/warehouse/names;
Found 1 items
root hdfs      78 /apps/hive/warehouse/names/names.txt
```

6.4. From the Hive Shell, run the following query:

```
hive> select * from names;
OK
0      Rich
1      Barry
2      George
3      Ulf
4      Danielle
5      Tom
6      manish
7      Brian
8      Mark
```

6.5. Now drop the **names** table:

```
hive> drop table names;
```

6.6. View the contents of the table's warehouse folder again. Notice the **names** folder is gone:

```
hive> dfs -ls /apps/hive/warehouse/names;
ls: '/apps/hive/warehouse/names': No such file or directory
```

IMPORTANT: Be careful when you drop a managed table in Hive. Make sure you either have the data backed up somewhere else, or that you no longer want the data.

Step 7: Define an External Table

7.1. In this step you will see how external tables work in Hive. Start by putting **names.txt** into HDFS:

```
hive> dfs -put /root/labs/Lab7.1/names.txt names.txt;
```

7.2. Create a folder in HDFS for the external table to store its data in:

```
hive> dfs -mkdir hivedemo;
```

7.3. Define the **names** table as external this time:

```
hive> create external table names (id int, name string)
  > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
  > LOCATION '/user/root/hivedemo';
```

7.4. Load data into the table:

```
hive> load data inpath '/user/root/names.txt' into table
names;
```

7.5. Verify the load worked:

```
hive> select * from names;
```

7.6. Notice the **names.txt** file has been moved to **/user/root/hivedemo**:

```
hive> dfs -ls hivedemo;
Found 1 items
-rw-r--r-- 1 root hdfs      78  hivedemo/names.txt
```

7.7. Similarly, verify **names.txt** is no longer in your **/user/root** folder in HDFS. Why is it gone? _____

7.8. Use the **ls** command to verify that the **/apps/hive/warehouse** folder does not contain a subfolder for the **names** table.

7.9. Now drop the **names** table:

```
hive> drop table names;
```

7.10. View the contents of **/user/root/hivedemo**. Notice that **names.txt** is still there.

RESULT: As you just verified, the data for external tables is not deleted when the corresponding table is dropped. Aside from this behavior, managed tables and external tables in Hive are essentially the same. You now have a table in Hive named **wh_visits** that was loaded from the result of a Pig job. You also have an external table called **names** that stores its data in **/user/root/hivedemo**. At this point, you should have a pretty good understanding of how Hive tables are created and populated.

Hive Partitions

- Use the **partitioned by** clause to define a partition when creating a table:

```
create table employees (id int, name string, salary double)
partitioned by (dept string);
```

- Subfolders are created based on the partition values:

```
/apps/hive/warehouse/employees
  /dept=hr/
  /dept=support/
  /dept=engineering/
  /dept=training/
```

© Hortonworks Inc. 2013



Hive Partitions

Hive manages the data in its tables using files in HDFS. You can define a table to have a **partition**, which results in the underlying data being stored in files partitioned by a specified column (or columns) of the table. Partitioning the data can greatly improve the performance of queries because the data is already separated into files based on the column value, which can decrease the number of mappers and greatly decrease the amount of shuffling and sorting of data in the resulting MapReduce job.

Use the **partitioned by** clause to define a partition when creating a table:

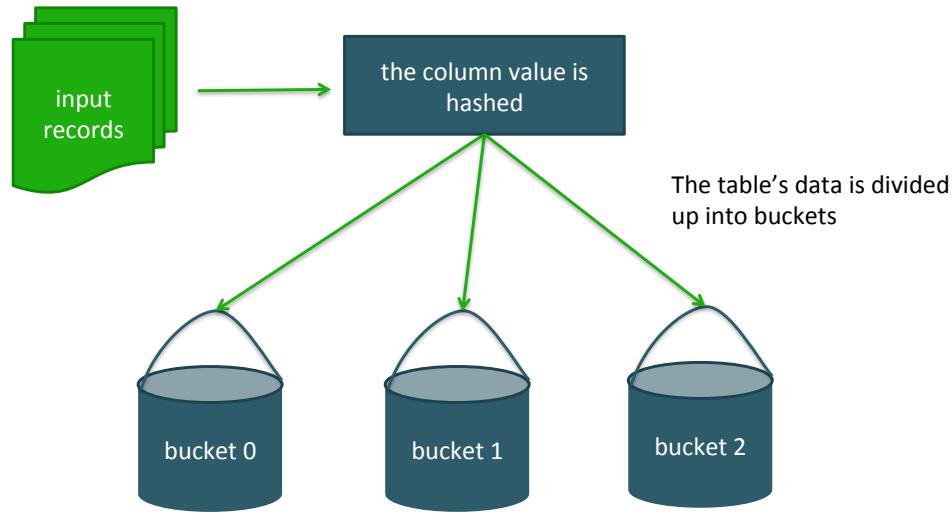
```
create table employees (id int, name string, salary double)
partitioned by (dept string);
```

This will result in each department having its own subfolder in the underlying **warehouse** folder for the table:

```
/apps/hive/warehouse/employees
  /dept=hr/
  /dept=support/
  /dept=engineering/
  /dept=training/
```

NOTE: You can partition by multiple columns, which results in subfolders within the subfolders of the table's warehouse directory.

Hive Buckets



© Hortonworks Inc. 2013



Hive Buckets

Hive tables can be organized into **buckets**, which imposes extra structure on the table and how the underlying files are stored. Bucketing has two key benefits:

- **More efficient queries:** especially when performing joins on the same bucketed columns.
- **More efficient sampling:** because the data is already split up into smaller pieces.

Buckets are created using the **clustered by** clause. For example, the following table has 16 buckets that are clustered by the **id** column:

```
create table employees (id int, name string, salary double)
clustered by (id) into 16 buckets;
```

How does Hive determine which bucket to put a record into? If you have n buckets, then the buckets are numbered 0 to $n-1$, and Hive *hashes* the column value and then uses the modulo operator on the hash value.

Skewed Tables

```
CREATE TABLE Customers (
    id int,
    username string,
    zip int
)
SKEwed BY (zip) ON (57701, 57702)
STORED as DIRECTORIES;
```

© Hortonworks Inc. 2013



Skewed Tables

In Hive, ***skew*** refers to one or more columns in a table that have values that appear very often. If you know a column is going to have heavy skew, you can specify this in the table's schema:

```
CREATE TABLE Customers (
    id int,
    username string,
    zip int
)
SKEwed BY (zip) ON (57701, 57702)
STORED as DIRECTORIES;
```

By specifying the values with heavy skew, Hive will split those out into separate files automatically and take this fact into account during queries so that it can skip whole files if possible.

In the **Customers** table above, records with a **zip** of 57701 or 57702 will be stored in separate files because the assumption is there will be a large number of customers in those two ZIP codes.

Demonstration: Understanding Partitions and Skew

Objective:	To understand how Hive partitioning works, as well as skewed tables.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: View the Data

- 1.1. Review the **hivedata_<>.txt** files in **/root/labs/demos**. This will be the data added to the table.

Step 2: Define the Table in Hive

- 2.1. View the **create table** statement in **partitiondemo.sql**:

```
# more partitiondemo.sql
create table names (id int, name string)
    partitioned by (state string)
    row format delimited fields terminated by '\t';
```

- 2.2. Run the query to define the **names** table:

```
# hive -f partitiondemo.sql
```

- 2.3. Show the partitions (there won't be any yet):

```
hive> show partitions names;
```

Step 3: Load Data into the Table

- 3.1. When you load data into a partitioned table, you specify which partition the data goes in to. For example:

```
load data local inpath '/root/labs/demos/hivedata_ca.txt'
into table names partition (state = 'CA');
```

3.2. Load the CO and SD files also:

```
load data local inpath '/root/labs/demos/hivedata_co.txt'
into table names partition (state = 'CO');
load data local inpath '/root/labs/demos/hivedata_sd.txt'
into table names partition (state = 'SD');
```

3.3. Verify all of the data made it into the **names** table:

```
hive> select * from names;
OK
1      Ulf      CA
2      Manish   CA
3      Brian    CA
4      George   CO
5      Mark     CO
6      Rich     SD
```

Step 4: View the Directory Structure

4.1. View the contents of **/apps/hive/warehouse/names**:

```
hive> dfs -ls -R /apps/hive/warehouse/names/
0  /apps/hive/warehouse/names/state=CA
24 /apps/hive/warehouse/names/state=CA/hivedata_ca.txt
0  /apps/hive/warehouse/names/state=CO
16 /apps/hive/warehouse/names/state=CO/hivedata_co.txt
0  /apps/hive/warehouse/names/state=SD
6  /apps/hive/warehouse/names/state=SD/hivedata_sd.txt
```

4.2. Notice that each partition has its own subfolder for storing its contents.

Step 5: Perform a Query

5.1. When you specify a **where** clause that includes a partition, Hive is smart enough to only scan the files in that partition. For example:

```
hive> select * from names where state = 'CA';
OK
1      Ulf      CA
2      Manish   CA
3      Brian    CA
```

5.2. Notice a MapReduce job was not executed. Why? _____

5.3. Notice you can select the partition field, even though it is not actually in the data file. Hive uses the directory name to retrieve the value. For example:

```
hive> select name, state from names where state = 'CA';
```

5.4. You can still run queries across the entire dataset. For example, the following query spans multiple partitions:

```
hive> select name, state from names where state = 'CA' or  
state = 'SD';
```

Step 6: Create a Skewed Table

6.1. Put **labs/demos/salaries.txt** into the **/user/root/salarydata/** folder in HDFS.

6.2. View the contents of **demos/skewdemo.hive**, which defines a skewed table named **skew_demo** using the **salaries.txt** data:

```
# more skewdemo.hive
```

6.3. Which values is this table being skewed by? _____

6.4. Run the commands in **skewdemo.hive**:

```
# hive -f skewdemo.hive
```

6.5. View the contents of the underlying Hive warehouse folder:

```
# hadoop fs -ls -R /apps/hive/warehouse/skew_demo
```

6.6. Select a few records to make sure the table has data behind it:

```
# hive -f show_skewdemo.hive
```

Sorting Data

- Hive has two sorting clauses:
 - **order by**: a complete ordering of the data
 - **sort by**: data output is sorted per reducer

© Hortonworks Inc. 2013



Sorting Data

HiveQL has two sorting clauses:

- **ORDER BY**: a complete ordering of the data, which is accomplished by using a single reducer.
- **SORT BY**: data output is sorted per reducer.

The syntax for the two clauses looks like:

```
select * from table_name [order | sort] by column_name;
```

The syntax for both is identical, only the behavior is different. If there is more than one reducer, than **sort by** only provides a partial sorting of the data by reducer, but not a total ordering. **Order by** implements total ordering by using only one reducer.

IMPORTANT: The **ORDER BY** clause requires a **LIMIT** clause if the **hive.mapred.mode** property is set to **strict**. If the property is **nonstrict** then no **LIMIT** clause is required, but be aware that a single reducer with a large number of records can take an extremely long time to finish.

Using Distribute By

```
from dataset  
  select *  
  distribute by age;
```

```
from dataset  
  select *  
  distribute by age  
  sort by age;
```

```
from dataset  
  select *  
  cluster by age;
```

© Hortonworks Inc. 2013



Using Distribute By

Hive uses the columns in **distribute by** to distribute the rows among reducers. In other words, all rows with the same **distribute by** columns will go to the same reducer. For example, suppose you have the following dataset with the schema (**year, name, age**) with **year** being the key:

1995	Dexter	24
2012	Carlos	33
2002	Michael	38
1995	Troy	33
2000	Megan	20
2002	Emma	24

The following command demonstrates **distribute by** on the **age** column:

```
hive> from dataset select *  
  > distribute by age;
```

- Dexter and Emma will go to the same reducer even though they have a different key.
- Similarly, Carlos and Troy will go to the same reducer.

The **distribute by** does not guarantee any type of clustering of the records. For example, a reducer might get:

1995	Troy	33
2000	Megan	20
2012	Carlos	33

Carlos and Troy were sent to the same reducer, but they are not adjacent. You can use **sort by** to cluster together records with the same **distribute by** column:

```
hive> from dataset select *
  > distribute by age
  > sort by age;
```

The records with the same **age** will now appear together in the reducer's output:

2000	Megan	20
1995	Troy	33
2012	Carlos	33

NOTE: If you use **distribute by** followed with a **sort by** on the same column, you can use **cluster by** and get the same result. For example, the following statement has the same result as the previous Hive statement above:

```
hive> from dataset select *
  > cluster by age;
```

Storing Results to a File

INSERT OVERWRITE DIRECTORY

```
'/user/train/ca_or_sd/'  
from names  
    select name, state  
    where state = 'CA'  
    or state = 'SD';
```

INSERT OVERWRITE LOCAL DIRECTORY

```
'/tmp/myresults/'  
SELECT * FROM bucketnames  
ORDER BY age;
```

© Hortonworks Inc. 2013



Storing Results to a File

The following command outputs the results of a query to a file in HDFS. For example:

```
INSERT OVERWRITE DIRECTORY '/user/train/ca_or_sd/' select  
name, state from names where state = 'CA' or state = 'SD';
```

You can also output the results of a query to a file on the local file system by adding the **LOCAL** keyword:

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/myresults/' SELECT *  
FROM bucketnames ORDER BY age;
```

Specifying MapReduce Properties

```
SET mapreduce.job.reduces = 12
```

```
hive -f myscript.hive  
-hiveconf mapreduce.job.reduces=12
```

```
SELECT * FROM names  
WHERE age = ${age}  
hive -f myscript.hive -hivevar age=33
```

© Hortonworks Inc. 2014



Specifying MapReduce Properties

Keep in mind that a Hive query is actually a MapReduce job behind the scenes. You can specify some of the properties of that underlying MapReduce job in Hive using the **SET** command.

You can either set the property in the Hive script:

```
SET mapreduce.job.reduces = 12
```

Or, you can set properties at the command line using the **hiveconf** flag:

```
hive -f myscript.hive -hiveconf mapreduce.job.reduces =12
```

You can use **hivevar** for parameter substitution. For example:

```
SELECT * FROM names WHERE age = ${age}
```

Specify age using either **SET** or with the **hivevar** flag:

```
hive -f myscript.hive -hivevar age=33
```

Lab 7.2: Analyzing Big Data with Hive

Objective:	Analyze the White House visitor data.
Location of Files:	n/a
Successful Outcome:	You will have discovered several useful pieces of information about the White House visitor data.
Before You Begin:	Complete Lab 7.1.

Perform the following steps:

Step 1: Find the First Visit

1.1. Create a new text file named **whitehouse.hive** and save it in your **Lab7.2** folder.

1.2. In this step, you will find the first visitor to the White House (based on our dataset). This will involve some clever handling of timestamps. This will be a long query, so enter it on multiple lines. Start by selecting all columns where the **time_of_arrival** is not empty:

```
select * from wh_visits where time_of_arrival != ""
```

1.3. To find the first visit, we need to sort the result. This requires converting the **time_of_arrival** string into a timestamp. We will use the **unix_timestamp** function to accomplish this. Add the following **order by** clause:

```
order by unix_timestamp(time_of_arrival,  
'MM/dd/yyyy hh:mm')
```

1.4. Since we are only looking for one result, we certainly don't need to return every row. Let's limit the result to 10 rows, so we can view the first 10 visitors:

```
limit 10;
```

1.5. Save your changes to **whitehouse.hive**.

1.6. Execute the script **whitehouse.hive** and wait for the results to be displayed:

```
# hive -f whitehouse.hive
```

1.7. The results should be 10 visitors, and the first visit should be in 2009 since that is when the dataset begins. The first visitor is Charles Kahn on 3/5/2009.

Step 2: Find the Last Visit

2.1. This one is easy - just take the previous query and reverse the order by adding **desc** to the **order by** clause:

```
order by unix_timestamp(time_of_arrival,  
'MM/dd/yyyy hh:mm') desc
```

2.2. Run the query again, and you should see that the most recent visit was Jackie Walker on 3/18/2011.

Step 3: Find the Most Common Comment

3.1. In this step, you will explore the **info_comment** field and try to determine the most common comment. You will use some of Hive's aggregate functions to accomplish this. Start by creating a new text file named **comments.hive**.

3.2. You will now create a query that displays the 10 most frequently occurring comments. Start with the following select clause:

```
from wh_visits  
select count(*) as comment_count, info_comment
```

This runs the aggregate **count** function on each group (which you will define later in the query) and names the result **comment_count**. For example, if "OPEN HOUSE" occurs 5 times, then **comment_count** will be 5 for that group.

Notice we are also selecting the **info_comment** column so we can see what the comment is.

3.3. Group the results of the query by the **info_comment** column:

```
group by info_comment
```

3.4. Order the results by **comment_count**, because we are only interested in comments that appear most frequently:

```
order by comment_count DESC
```

3.5. We only want the top results, so limit the result set to 10:

```
limit 10;
```

3.6. Save your changes to **comments.hive** and execute the script. Wait for the MapReduce job to execute.

3.7. The output will be 10 comments and should look like:

```
9036  
1253 HOLIDAY BALL ATTENDEES/  
894 WHO EOP RECEP 2  
700 WHO EOP 1 RECEPTION/  
601 RESIDENCE STAFF HOLIDAY RECEPTION/  
586 PRESS RECEPTION ONE (1) /  
580 GENERAL RECEPTION 1  
540 HANUKKAH RECEPTION./  
540 GEN RECEP 5/  
516 GENERAL RECEPTION 3
```

3.8. It appears that a blank comment is the most frequent comment, followed by the HOLIDAY BALL, then a variation of other receptions.

3.9. OPTIONAL: Modify the query so that it ignores empty comments. If it works, the comment “GEN RECEP 6/” will show up in your output.

Step 4: Least Frequent Comment

4.1. Run the previous query again, but this time find the 10 least occurring comments. The output should look like:

```
1 merged to u59031  
1 WHO EOP/  
1 WHO EOP RECLEAR  
1 WAITING FOR SUPERMAN VISIT  
1 ST. PATRICK'S RECEPTION GUESTS  
1 SCIENCE FAIR  
1 RES PARTY/  
1 PRIVATE MEETING  
1 PRIVATE LUNCH  
1 POTUS PHOTO W/ US ATTORNEYS/
```

This seems accurate since 1 is the least number of times a comment can appear. Plus this query reveals that Superman has visited the President at least once!

Step 5: Analyze the Data Inconsistencies

5.1. Analyzing the results of the most and least frequent comments, it appears that several variations of GENERAL RECEPTION occur. In this step, you will try to determine the number of visits to the POTUS involving a general reception by trying to clean up some of these inconsistencies in the data.

NOTE: Inconsistencies like these are very common in big data, especially when human input is involved. In this dataset, we like have different people entering similar comments but using their own abbreviations.

5.2. Modify the query in **comments.hive**. Instead of searching for empty comments, search for comments that contain the string “RECEP”.

```
where info_comment like "%RECEP%"
```

5.3. Change the limit clause from 10 to 30:

```
limit 30;
```

5.4. Run the query again.

5.5. Notice there are several GENERAL RECEPTION entries that only differ by a number at the end, or use the GEN RECEP abbreviation:

```
580  GENERAL RECEPTION 1
540  GEN RECEP 5/
516  GENERAL RECEPTION 3
498  GEN RECEP 6/
438  GEN RECEP 4
31   GENERAL RECEPTION 2
23   GENERAL RECEPTION 3
20   GENERAL RECEPTION 6
20   GENERAL RECEPTION 5
13   GENERAL RECEPTION 1
```

5.6. Let's try one more query to try and narrow GENERAL RECEPTION visits. Modify the WHERE clause in comments.hive to include “%GEN%”:

```
where info_comment like "%RECEP%"  
    and info_comment like "%GEN%"
```

5.7. Leave the limit at 30, and run the query again.

5.8. The output this time reveals all the variations of GEN and RECEP. Let's add up the total number of them by running the following query:

```
from wh_visits  
      select count(*)  
        where info_comment like "%RECEP%"  
          and info_comment like "%GEN%";
```

5.9. Notice there are 2,697 visits to the POTUS with GEN RECEP in the comment field, which is about 12% of the 21,819 total visits to the POTUS in our dataset.

NOTE: More importantly, these results show that our first query of 1,253 attendees to the HOLIDAY BALL does not mean that the holiday ball is the most likely reason to visit the President. More than twice as many visitors are there for a general reception. This type of analysis is common in big data, and it shows how Data Analysts need to be creative when researching their data.

Step 6: Verify the Result

6.1. We have 12% of visitors to the POTUS going for a general reception, but there were a lot of statements in the comments that contained WHO and EOP. Modify the query from the last step and display the top 30 comments that contain "WHO" and "EOP". The result should look like:

```
894 WHO EOP RECEP 2  
700 WHO EOP 1 RECEPTION/  
43 WHO EOP RECEP/  
20 WHO EOP HOLIDAY RECEP/  
13 WHO/EOP #2/  
8 WHO EOP RECEPTION  
7 WHO EOP RECEP  
1 WHO EOP/  
1 WHO EOP RECLEAR
```

6.2. Run a query that counts the number of records with WHO and EOP in the comments:

```
from wh_visits
  select count(*)
    where info_comment like "%WHO%"
      and info_comment like "%EOP%";
```

You should get 1,687 visits, or 7.7% of the visitors to the POTUS. So GENERAL RECEPTION still appears to be the most frequent comment.

Step 7: Find the Most Visits

7.1. See if you can write a Hive script that finds the top 20 individuals who visited the POTUS most. Use the Hive command from Step 3 earlier in this lab as a guide.

HINT: Use a grouping by both **fname** and **lname**.

7.2. To verify your script worked, here are the top 20 individuals who visited the POTUS, along with the number of visits:

16	ALAN PRATHER
15	CHRISTOPHER FRANKE
15	ANNAMARIA MOTTOLA
14	ROBERT BOGUSLAW
14	CHARLES POWERS
12	SARAH HART
12	JACKIE WALKER
12	JASON FETTIG
12	SHENGTSUNG WANG
12	FERN SATO
12	DIANA FISH
11	JANET BAILEY
11	PETER WILSON
11	GLENN DEWEY
11	MARCIO BOTELHO
11	DONNA WILLINGHAM
10	DAVID AXELROD
10	CLAUDIA CHUDACOFF
10	VALERIE JARRETT
10	MICHAEL COLBURN

RESULT: You have written several Hive queries to analyze the White House visitor data. The goal is for you to become comfortable with working with Hive, so hopefully you now feel like you can tackle a Hive problem and be able to answer questions about your big data stored in Hive.

Lab 7.3: Understanding MapReduce in Hive

Objective:	To understand better how Hive queries get executed as MapReduce jobs.
Location of Files:	n/a
Successful Outcome:	No specific outcome - you will answer various questions about Hive queries and run a few examples.
Before You Begin:	Start the Hive Shell.

Step 1: The Describe Command

1.1. Run the **describe** command on the **wh_visits** table:

```
hive> describe wh_visits;
OK
lname          string      None
fname          string      None
time_of_arrival string      None
appt_scheduled_time string      None
meeting_location string      None
info_comment   string      None
Time taken: 0.677 seconds
```

1.2. Did this query require a MapReduce job? _____

1.3. What is the name of the Hive resource that was accessed to retrieve this schema information? _____

Step 2: A Simple Query

2.1. Run the following query:

```
select * from wh_visits where fname = "JOE";
```

2.2. Does Hive run a MapReduce job to generate the result? _____

2.3. Open your browser and point it to the JobHistory UI:

```
http://ipaddress:19888/
```

2.4. Notice the most recent MapReduce job executed is your “JOE” query:

2.5. How many map tasks were used to execute this query? _____

2.6. How many reduce tasks were used to execute this query? _____

2.7. How many attempts did it take for this task to succeed? _____

2.8. How long did it take for this query to execute? _____

Step 3: A Sorted Query

3.1. Run the following query:

```
hive> select * from wh_visits where fname = "JOE" sort by  
lname;
```

3.2. When the MapReduce job completes, find its job details page from the Job Browser.

3.3. How many map tasks were used to execute this query? _____

3.4. How many reduce tasks were used to execute this query? _____

3.5. The map task outputs <key,value> pairs and sends them to the reducer. What do you think this MapReduce job chose as the key for the mapper’s output?

Step 4: A Select Query

4.1. Run the following query:

```
hive> select * from wh_visits limit 5;
```

4.2. Does Hive run a MapReduce job to generate the results? _____

4.3. What data is read from HDFS? _____

4.4. Now select a single column from **wh_visits**:

```
hive> select fname from wh_visits limit 5;
```

4.5. Why did this require a MapReduce job but “**select ***” did not? _____

Step 5: Using the EXPLAIN Command

5.1. The **EXPLAIN** command shows the execution plan of a query, without actually executing the query. To demonstrate, add **EXPLAIN** to the beginning of the following query that you ran earlier in this lab:

```
hive> explain select * from wh_visits where fname = "JOE"  
sort by lname;
```

5.2. Notice the query is executed in two stages. **Stage-0** performs the limit operator. This was the first mapper that executed the query.

5.3. Notice **Stage-1** is a MapReduce job that has one mapper (look for **Map Operator Tree**) and one reducer (under **Reduce Operator Tree**). As you can see from this execution plan, the mapper is doing most of the work.

Step 6: Use EXPLAIN EXTENDED

6.1. Run the previous **EXPLAIN** again, except this time add the **EXTENDED** command:

```
hive> explain extended select * from wh_visits where fname  
= "JOE" sort by lname;
```

6.2. Compare the two outputs. Notice the **EXTENDED** command adds a lot of additional information about the underlying execution plan.

ANSWERS:

1.2: No

1.3: The Hive metastore contains the schema information of all tables.

2.3: Yes

2.5: 1 map task

2.6: 0 reduce tasks

2.7: It probably succeeded on the first attempt. If not, you will see multiple entries in the list of attempts.

2.8: Subtract the Execution Finish Time from the Execution Start Time. It should have executed in about 15-30 seconds.

3.3: 1 map task

3.4: 1 reduce task

3.5: It makes sense for the mapper to use **Iname** as the key, which would mean the visitors would already be sorted by last name when they got to the reducer.

4.2: No

4.3: Hive simply reads the data directly from the underlying file in HDFS.

4.5: Selecting specific columns requires actual processing of the contents of each record to split out the required columns.

Hive Join Strategies

Type	Approach	Pros	Cons
Shuffle Join	Join keys are shuffled using MapReduce and joins are performed on the reduce side.	Works regardless of data size or layout.	Most resource-intensive and slowest join type.
Map (Broadcast) Join	Small tables are loaded into memory in all nodes, mapper scans through the large table and joins.	Very fast, single scan through largest table.	All but one table must be small enough to fit in RAM.
Sort-Merge-Bucket Join	Mappers take advantage of co-location of keys to do efficient joins.	Very fast for tables of any size.	Data must be sorted and bucketed ahead of time.

© Hortonworks Inc. 2013



Hive Join Strategies

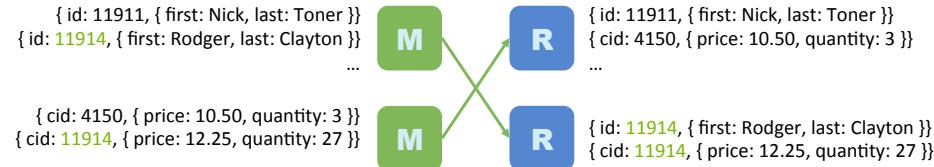
Some important concepts to understand when performing joins and laying out your Hive data:

- Shuffle joins always work in the sense that if you cannot perform a more efficient type of join, two tables can always be joined using a shuffle join.
- A map join is very efficient and ideal if one side of the join is a small enough dataset to fit into memory.
- If a map join is not an option, then the next best option is a sort-merge-bucket join, which we will discuss in more detail.

Shuffle Joins

customer			order		
first	last	id	cid	price	quantity
Nick	Toner	11911	4150	10.50	3
Jessie	Simonds	11912	11914	12.25	27
Kasi	Lamers	11913	3491	5.99	5
Rodger	Clayton	11914	2934	39.99	22
Verona	Hollen	11915	11914	40.50	10

```
SELECT * FROM customer JOIN order ON customer.id = order.cid;
```



© Hortonworks Inc. 2013



Shuffle Joins

A **shuffle join** is the default join technique for Hive, and it works with any data sets (no matter how large). Identical keys are shuffled to the same reducer, and the join is performed on the reduce side. This is the most expensive join from a network utilization standpoint because all records from both sides of the join need to be processed by a mapper and then shuffled and sorted, even the records that are not a part of the result set.

Map (Broadcast) Joins

customer			order		
first	last	id	cid	price	quantity
Nick	Toner	11911	4150	10.50	3
Jessie	Simonds	11912	11914	12.25	27
Kasi	Lamers	11913	3491	5.99	5
Rodger	Clayton	11914	2934	39.99	22
Verona	Hollen	11915	11914	40.50	10

```
SELECT * FROM customer JOIN order ON customer.id = order.cid;
```

```
{ id: 11914, { first: Rodger, last: Clayton } }  
{ cid: 11914, { price: 12.25, quantity: 27 } }  
cid: 11914, { price: 12.25, quantity: 27 }
```



Records are joined during
the Map phase.

© Hortonworks Inc. 2014



Map (Broadcast) Joins

If one of the datasets is small enough to fit into memory, then it can be distributed (broadcast) to each Mapper and perform the join in the map phase. This greatly reduces the number of records being shuffled and sorted because only records that appear in the result set will be passed on to a reducer.

A **map join** has a special C-style comment syntax for providing a hint to the Hive engine:

```
select /*+ MAPJOIN(states) */ customers.* , states.*  
from customers  
join states on (customers.state = states.state);
```

IMPORTANT: In HDP 2.0, Hive joins are automatically optimized without the need for providing hints.

Sort-Merge-Bucket Joins

customer			order		
first	last	id	cid	price	quantity
Nick	Toner	11911	4150	10.50	3
Jessie	Simonds	11912	11914	12.25	27
Kasi	Lamers	11913	11914	40.50	10
Rodger	Clayton	11914	12337	39.99	22
Verona	Hollen	11915	15912	40.50	10

```
SELECT * FROM customer join order ON customer.id = order.cid;
```

Distribute and sort by the most common join key.

```
CREATE TABLE order (cid int, price float, quantity int)  
CLUSTERED BY(cid) INTO 32 BUCKETS;
```

```
CREATE TABLE customer (id int, first string, last string)  
CLUSTERED BY(id) INTO 32 BUCKETS;
```

© Hortonworks Inc. 2014



Sort-Merge-Bucket (SMB) Joins

If you have two datasets that are too large for a map side join, an efficient technique for joining them is to sort the two datasets into buckets. The trick is to cluster and sort by the same join key.

This provides two major optimization benefits:

- Sorting by the join key makes joins easy. All possible matches reside in the same area on disk.
- Hash bucketing a join key ensures all matching values reside on the same node. Equi-joins can then run with no shuffle.

For this to work properly, the number of bucket columns has to equal the number of join columns. This means, in general, you will need to specifically define your Hive tables to fit the requirements of a sort-merge-bucket join, which implies you are aware at design time of the columns that will be most commonly used in join statements.

NOTE: An SMB join can be converted to an SMB map join. This requires the following configuration settings enabled. (Note that these settings are already set to true in HDP 2.0):

```
hive.auto.convert.sortmerge.join=true;  
hive.optimize.bucketmapjoin = true;  
hive.optimize.bucketmapjoin.sortedmerge = true;  
hive.auto.convert.sortmerge.join.noconditionaltask  
= true;
```

Invoking a Hive UDF

```
ADD JAR /myapp/lib/myhiveudfs.jar;
CREATE TEMPORARY FUNCTION
ComputeShipping
AS 'hiveudfs.ComputeShipping';

FROM orders SELECT
address,
description,
ComputeShipping(zip, weight);
```

© Hortonworks Inc. 2013



Invoking a Hive UDF

Similar to Pig, Hive has the ability to use **User-Defined Functions** written in Java to perform computations that would otherwise be difficult (or impossible) to perform using the built-in Hive functions and SQL commands.

To invoke a UDF from within a Hive script, you need to:

1. Register the JAR file that contains the UDF class, and
2. Define an alias for the function using the CREATE TEMPORARY FUNCTION command.

For example, the following Hive commands demonstrate how to invoke the ComputeShipping UDF defined above:

```
ADD JAR /myapp/lib/myhiveudfs.jar;
CREATE TEMPORARY FUNCTION ComputeShipping
AS 'hiveudfs.ComputeShipping';
FROM orders SELECT address, description,
ComputeShipping(zip, weight);
```

Computing ngrams in Hive

```
select ngrams(sentences(val), 2, 100)  
from mytable;
```

```
select context_ngrams(sentences(val),  
                      array("error", "code", null),  
                      100)  
from mytable;
```

© Hortonworks Inc. 2013



Computing ngrams in Hive

An **ngram** is a subsequence of text within a large document. The “n” represents the length of the subsequence. The result of an ngram is a frequency distribution

For example, when **n** is 2 it's called a *bigram*, and it represents the occurrence of two adjacent terms. A *trigram* is when **n** is 3 and represents three adjacent terms, and so on.

Hive contains an **ngram** function for computing the frequency distribution. For example:

```
select ngrams(sentences(val), 2, 100) from mytable;
```

The above command computes a bigram of the data in the **val** column of **mytable**, returning a frequency distribution of the top 100 results.

Hive also contains a **context_ngram** function, which computes ngrams based on a context string that appears around the subsequence of text. For example:

```
select context_ngrams(sentences(val),  
                      array("error", "code", null),  
                      100)  
   from mytable;
```

The above command generates a frequency distribution of the top 100 words that follow the expression “error code”.

Demonstration: Computing ngrams

Objective:	To understand how to compute ngrams using Hive.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Create a Hive Table for the Data

- 1.1.** This demonstration computes ngrams on the U.S. Constitution, which is in a text file in the **/root/labs/demos** folder:

```
# cd ~/labs/demos/  
# more constitution.txt
```

- 1.2.** Start the Hive shell and define the following table:

```
hive> create table constitution (  
      line string  
)  
ROW FORMAT DELIMITED;
```

Each line of text in the text file is going to be a record in our Hive table.

Step 2: Load the Hive Table

- 2.1.** Load **constitution.txt** into the **constitution** table:

```
hive> load data local inpath  
'/root/labs/demos/constitution.txt' into table  
constitution;
```

- 2.2.** Verify the data is loaded:

```
hive> select * from constitution;
```

You should see the contents of **constitution.txt** again.

Step 3: Compute a Bigram

3.1. Enter the following Hive command, which computes a bigram for the Constitution and shows the top 15 results:

```
hive> select explode(ngrams(sentences(line),2,15)) as x  
from constitution;
```

The result should look like:

```
{"ngram": ["of", "the"], "estfrequency": 194.0}  
{"ngram": ["shall", "be"], "estfrequency": 100.0}  
{"ngram": ["the", "United"], "estfrequency": 76.0}  
{"ngram": ["United", "States"], "estfrequency": 76.0}  
{"ngram": ["to", "the"], "estfrequency": 57.0}  
{"ngram": ["shall", "have"], "estfrequency": 44.0}  
{"ngram": ["the", "President"], "estfrequency": 30.0}  
{"ngram": ["shall", "not"], "estfrequency": 29.0}  
{"ngram": ["in", "the"], "estfrequency": 28.0}  
{"ngram": ["by", "the"], "estfrequency": 25.0}  
{"ngram": ["the", "Congress"], "estfrequency": 22.0}  
{"ngram": ["and", "the"], "estfrequency": 21.0}  
{"ngram": ["for", "the"], "estfrequency": 21.0}  
{"ngram": ["Vice", "President"], "estfrequency": 21.0}  
{"ngram": ["the", "Senate"], "estfrequency": 21.0}  
{"ngram": ["States", "and"], "estfrequency": 20.0}  
{"ngram": ["States", "shall"], "estfrequency": 19.0}  
{"ngram": ["any", "State"], "estfrequency": 18.0}  
{"ngram": ["Congress", "shall"], "estfrequency": 18.0}  
{"ngram": ["on", "the"], "estfrequency": 17.0}
```

Step 4: Compute a Trigram

4.1. Run the previous query again, but this time compute a trigram:

```
hive> select explode(ngrams(sentences(line),3,20)) as  
result from constitution;
```

4.2. The result should look like:

```
{"ngram": ["the", "United", "States"], "estfrequency": 68.0}  
{"ngram": ["of", "the", "United"], "estfrequency": 51.0}  
{"ngram": ["shall", "not", "be"], "estfrequency": 16.0}
```

```
{
  "ngram": ["of", "the", "Senate"], "estfrequency": 14.0}
  {"ngram": ["States", "shall", "be"], "estfrequency": 13.0}
  {"ngram": ["House", "of", "Representatives"], "estfrequency": 13.0}
  {"ngram": ["United", "States", "shall"], "estfrequency": 13.0}
  {"ngram": ["shall", "have", "been"], "estfrequency": 12.0}
  {"ngram": ["the", "several", "States"], "estfrequency": 12.0}
  {"ngram": ["President", "of", "the"], "estfrequency": 11.0}
  {"ngram": ["United", "States", "and"], "estfrequency": 11.0}
  {"ngram": ["The", "Congress", "shall"], "estfrequency": 10.0}
  {"ngram": ["the", "House", "of"], "estfrequency": 10.0}
  {"ngram": ["United", "States", "or"], "estfrequency": 10.0}
  {"ngram": ["Congress", "shall", "have"], "estfrequency": 10.0}
  {"ngram": ["the", "Vice", "President"], "estfrequency": 9.0}
  {"ngram": ["of", "the", "President"], "estfrequency": 8.0}
  {"ngram": ["Consent", "of", "the"], "estfrequency": 8.0}
  {"ngram": ["shall", "be", "the"], "estfrequency": 7.0}
  {"ngram": ["by", "the", "Congress"], "estfrequency": 7.0}
}
```

Step 5: Compute a Contextual ngram

5.1. Let's find the 20 most frequent words that follow "the":

```
hive> select explode(context_ngrams(sentences(line),
      array("the", null), 20)) as result
    from constitution;
```

5.2. The result looks like:

```
{
  "ngram": ["United"], "estfrequency": 76.0}
  {"ngram": ["President"], "estfrequency": 30.0}
  {"ngram": ["Congress"], "estfrequency": 22.0}
  {"ngram": ["Senate"], "estfrequency": 21.0}
  {"ngram": ["several"], "estfrequency": 15.0}
  {"ngram": ["Vice"], "estfrequency": 12.0}
  {"ngram": ["State"], "estfrequency": 11.0}
  {"ngram": ["same"], "estfrequency": 10.0}
  {"ngram": ["Constitution"], "estfrequency": 10.0}
  {"ngram": ["States"], "estfrequency": 10.0}
  {"ngram": ["House"], "estfrequency": 10.0}
  {"ngram": ["whole"], "estfrequency": 10.0}
  {"ngram": ["office"], "estfrequency": 9.0}
  {"ngram": ["right"], "estfrequency": 8.0}
  {"ngram": ["Legislature"], "estfrequency": 8.0}
  {"ngram": ["Consent"], "estfrequency": 6.0}
  {"ngram": ["powers"], "estfrequency": 6.0}
  {"ngram": ["supreme"], "estfrequency": 6.0}
  {"ngram": ["people"], "estfrequency": 6.0}
  {"ngram": ["first"], "estfrequency": 6.0}
}
```

Unit 7 Review

1. A Hive table consists of a schema stored in the Hive _____ and data stored in _____.
2. True or False: The Hive metastore requires an underlying SQL database.
3. What happens to the underlying data of a Hive-managed table when the table is dropped? _____
4. True or False: A Hive external table must define a **LOCATION**.
5. List three different ways data can be loaded into a Hive table: _____

6. When would you use a skewed table? _____
7. Suppose you have the following table definition:

```
create table movies (title string, rating string,  
length double) partitioned by (genre string);
```

What will the folder structure in HDFS look like for the **movies** table?

8. Explain the output of the following query:

```
select * from movies order by title;
```

9. What does the following Hive query compute?

```
from mytable  
    select explode(ngrams(sentences(val), 3, 100)) as myresult;
```

-
10. What does the following Hive query compute?

```
from mytable  
    select explode(context_ngrams(sentences(val),  
        array("I", "liked", null), 10)) as myresult;
```

Lab 7.4: Joining Datasets in Hive

Objective:	Perform a join of two datasets in Hive.
Location of Files:	/root/labs/Lab7.4
Successful Outcome:	A table named stock_aggregates that contains a join of NYSE stock prices with the stock's dividend prices.
Before You Begin:	SSH into your VM.

Perform the following steps:

Step 1: Load the Data into Hive

- 1.1. View the contents of the file **setup.hive** in /root/labs/Lab7.4:

```
# more setup.hive
```

- 1.2. Notice this script creates three tables in Hive. The **nyse_data** table is filled with the daily stock prices of stocks that start with the letter "K", and the **dividends** table that contains the quarterly dividends of those stocks. The **stock_aggregates** table is going to be used for a join of these two datasets and contain the stock price and dividend amount on the date the dividend was paid.

- 1.3. Run the **setup.hive** script from the **Lab7.4** folder:

```
# hive -f setup.hive
```

- 1.4. To verify the script worked, enter the following query from the Hive Shell:

```
hive> select * from nyse_data limit 20;
hive> select * from dividends limit 20;
```

You should see daily stock prices and dividends from stocks that start with the letter “K”.

1.5. The **stock_aggregates** table should be empty, but view its schema to verify it was created successfully:

```
hive> describe stock_aggregates;
OK
symbol          string      None
year            string      None
high             float       None
low              float       None
average_close   float       None
total_dividends float       None
```

Step 2: Join the Datasets

2.1. The **join** statement is going to be fairly long, so let’s create it in a text file. Create a new text in the **Lab7.4** folder named **join.hive**, and open the file with a text editor.

2.2. We will break the join statement down into sections. First, the result of the **join** is going to put into the **stock_aggregates** table, which requires an **insert**:

```
insert overwrite table stock_aggregates
```

The **overwrite** causes any existing data in **stock_aggregates** to be deleted.

2.3. The data being inserted is going to be the result of a select query that contains various insightful indicators about each stock. The result is going to contain the stock symbol, date traded, maximum high for the stock, minimum low, average close, and sum of dividends, as shown here:

```
select a.symbol, year(a.trade_date), max(a.high),
min(a.low), avg(a.close), sum(b.dividend)
```

2.4. The **from** clause is the **nyse_data** table:

```
from nyse_data a
```

2.5. The join is going to be a left outer join of the **dividends** table:

```
left outer join dividends b
```

2.6. The join is by stock symbol and trade date:

```
on (a.symbol = b.symbol and a.trade_date = b.trade_date)
```

2.7. Let's group the result by symbol and trade date:

```
group by a.symbol, year(a.trade_date);
```

2.8. Save your changes to join.hive.

Step 3: Run the Query

3.1. Run the query and wait for the MapReduce jobs to execute:

```
# hive -f join.hive
```

3.2. How many MapReduce jobs does it take to perform this query?

Step 4: Verify the Results

4.1. Run a **select** query to view the contents of **stock_aggregates**:

```
hive> select * from stock_aggregates;
```

The output should look like:

KYO	2004	90.9	66.25	75.79952	0.544
KYO	2005	78.45	62.58	72.042656	0.91999996
KYO	2006	98.01	71.73	85.80327	0.851
KYO	2007	110.01		81.0	93.737686
KYO	2008	100.78		45.41	79.6098
KYO	2009	93.2	52.98	77.04389	NULL
KYO	2010	93.83	85.94	90.71	NULL
stock_symbol		NULL	NULL	NULL	NULL

4.2. List the contents of the **stock_aggregates** directory in HDFS. The **000000_0** file was created as a result of the **join** query:

```
# hadoop fs -ls -R /apps/hive/warehouse/stock_aggregates/
41109 /apps/hive/warehouse/stock_aggregates/000000_0
```

4.3. View the contents of the **stock_aggregates** table using the **cat** command:

```
# hadoop fs -cat  
/apps/hive/warehouse/stock_aggregates/000000_0
```

RESULT: The **stock_aggregates** table is a joining of the daily stock prices and the quarterly dividend amounts on the date the dividend was announced, and the data in table is an aggregate of various statistics like max high, min low, etc.

Lab 7.5: Computing ngrams of Emails in Avro Format

Objective:	Use Hive to compute ngrams and histograms.
Location of Files:	/root/labs/Lab7.5
Successful Outcome:	A bi-gram of words found in a collection of Avro-formatted emails.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: View an Avro Schema

1.1. Change directories to the **Lab7.5** folder. Notice this folder contains an Avro file named **sample.avro**.

1.2. Enter the following command to view the schema of the contents of **sample.avro**:

```
avro cat --print-schema sample.avro
```

1.3. How many fields do records in **sample.avro** have? _____

1.4. Create a schema file for **sample.avro**:

```
avro cat --print-schema sample.avro > /tmp/sample.avsc
```

Step 2: Create a Hive Table from an Avro Schema

2.1. View the contents of the **CREATE TABLE** query defined in the **create_sample_table.hive** file in your **Lab7.5** folder.

2.2. Make sure the **avro.schema.file** property points to the schema file you created in the previous step:

```
WITH SERDEPROPERTIES (
    'avro.schema.url'='file:///tmp/sample.avsc')
```

2.3. Run the CREATE TABLE query:

```
hive -f create_sample_table.hive
```

Step 3: Verify the Table

3.1. Start the Hive shell.

3.2. Run the **show tables** command and verify that you have a table named **sample_table**.

3.3. Run the describe command on **sample_table**. Notice the schema for **sample_table** matches the Avro schema from **sample.avsc**.

3.4. Let's associate some data with **sample_table**. Copy **sample.avro** into the Hive **warehouse** folder by running the following command (all on a single line):

```
hive> dfs -put /root/labs/Lab7.5/sample.avro
      /apps/hive/warehouse/sample_table
```

3.5. View the contents of **sample_table**:

```
hive> select * from sample_table;
OK
Foo    19    10, Bar Eggs Spam    800
```

Note there is only one record in **sample.avro**.

Step 4: Create Email User Table

4.1. There is an Avro file in your **Lab7.5** folder named **mbox7.avro**, which represents emails in an Avro format from a Hive mailing list for the month of July. Use the **--print-schema** option of **avro** to view the schema of this file.

4.2. How many fields do records in **mbox7.avro** have? _____

4.3. Run the **--print-schema** command again, but this time output the schema to a file named **mbox.avsc**:

```
avro cat --print-schema mbox7.avro > /tmp/mbox.avsc
```

4.4. View the contents of the **create_email_table.hive** script in your **Lab7.5** folder. Verify the **avro.schema.url** property is correct.

4.5. Run the script to create the **hive_user_email** table:

```
hive -f create_email_table.hive
```

4.6. Copy **mbox7.avro** into the warehouse directory:

```
hadoop fs -put mbox7.avro  
/apps/hive/warehouse/hive_user_email/
```

4.7. Verify the table has data in it:

```
select * from hive_user_email limit 20;
```

Step 5: Compute a Bigram

5.1. Start the Hive shell.

5.2. Use the Hive **ngrams** function to create a bigram of the words in **mbox7.avro**:

```
select  
    ngrams(sentences(content),2 ,10)  
  from hive_user_email;
```

The output will be kind of a jumbled mess:

```
[{"ngram": ["2013", "at"], "estfrequency": 802.0}, {"ngram": ["of", "the"], "estfrequency": 391.0}, {"ngram": ["I", "am"], "estfrequency": 368.0}, {"ngram": ["I", "have"], "estfrequency": 340.0}, {"ngram": ["J", "E9r"], "estfrequency": 306.0}, {"ngram": ["for", "the"], "estfrequency": 291.0}, {"ngram": ["you", "are"], "estfrequency": 289.0}, {"ngram": ["user", "hive.apache.org"], "estfrequency": 289.0}, {"ngram": ["to", "the"], "estfrequency": 276.0}, {"ngram": ["E9r", "F4me"], "estfrequency": 270.0}]
```

5.3. To clean this up, use the Hive **explode** function to display the output in a more readable format:

```
select  
    explode(ngrams(sentences(content),2 ,10)) as x  
  from hive_user_email;
```

You should see the a nice, readable list of 10 bigrams:

```
{"ngram": ["2013", "at"], "estfrequency": 802.0}
{"ngram": ["of", "the"], "estfrequency": 391.0}
{"ngram": ["I", "am"], "estfrequency": 368.0}
{"ngram": ["I", "have"], "estfrequency": 340.0}
{"ngram": ["J", "E9r"], "estfrequency": 306.0}
{"ngram": ["for", "the"], "estfrequency": 291.0}
{"ngram": ["you", "are"], "estfrequency": 289.0}
{"ngram": ["user", "hive.apache.org"], "estfrequency": 289.0}
{"ngram": ["to", "the"], "estfrequency": 276.0}
{"ngram": ["E9r", "F4me"], "estfrequency": 270.0}
```

5.4. Typically when working with word comparison we ignore case. Run the query again, but this time add the Hive **lower** function and compute 20 bigrams:

```
select
    explode(ngrams(sentences(lower(content)), 2, 20)) as x
    from hive_user_email;
```

The output shoud look like the following:

```
{"ngram": ["2013", "at"], "estfrequency": 802.0}
{"ngram": ["i", "have"], "estfrequency": 409.0}
{"ngram": ["of", "the"], "estfrequency": 391.0}
 {"ngram": ["i", "am"], "estfrequency": 372.0}
 {"ngram": ["if", "you"], "estfrequency": 347.0}
 {"ngram": ["in", "hive"], "estfrequency": 337.0}
 {"ngram": ["for", "the"], "estfrequency": 309.0}
 {"ngram": ["j", "e9r"], "estfrequency": 306.0}
 {"ngram": ["you", "are"], "estfrequency": 289.0}
 {"ngram": ["user", "hive.apache.org"], "estfrequency": 289.0}
 {"ngram": ["to", "the"], "estfrequency": 276.0}
 {"ngram": ["outer", "join"], "estfrequency": 271.0}
 {"ngram": ["2013", "06"], "estfrequency": 270.0}
 {"ngram": ["e9r", "f4me"], "estfrequency": 270.0}
 {"ngram": ["left", "outer"], "estfrequency": 270.0}
 {"ngram": ["in", "the"], "estfrequency": 252.0}
 {"ngram": ["gmail.com", "wrote"], "estfrequency": 248.0}
 {"ngram": ["17", "16"], "estfrequency": 248.0}
 {"ngram": ["06", "17"], "estfrequency": 246.0}
 {"ngram": ["wrote", "hi"], "estfrequency": 234.0}
```

Step 6: Compute a Context n-gram

6.1. From the Hive shell, run the following query, which uses the context_ngrams function to find the top 20 terms that follow the word “**error**”:

```
select
    explode(context_ngrams(sentences(lower(content)),
                           array("error", null) ,20)) as x
  from hive_user_email;
```

The output should look like the following:

```
{"ngram": ["in"], "estfrequency": 102.0}
{"ngram": ["return"], "estfrequency": 97.0}
{"ngram": ["org.apache.hadoop.hive.ql.exec.udfargumenttypeexception"], "estfrequency": 49.0}
{"ngram": ["failed"], "estfrequency": 49.0}
{"ngram": ["is"], "estfrequency": 41.0}
{"ngram": ["message"], "estfrequency": 40.0}
{"ngram": ["when"], "estfrequency": 39.0}
{"ngram": ["please"], "estfrequency": 36.0}
 {"ngram": ["while"], "estfrequency": 28.0}
 {"ngram": ["org.apache.thrift.transport.ttransportexception"], "estfrequency": 28.0}
 {"ngram": ["datanucleus.plugin"], "estfrequency": 26.0}
 {"ngram": ["during"], "estfrequency": 18.0}
 {"ngram": ["query"], "estfrequency": 16.0}
 {"ngram": ["hive"], "estfrequency": 16.0}
 {"ngram": ["could"], "estfrequency": 16.0}
 {"ngram": ["java.lang.runtimeexception"], "estfrequency": 13.0}
 {"ngram": ["13"], "estfrequency": 12.0}
 {"ngram": ["error"], "estfrequency": 12.0}
 {"ngram": ["exec.execdriver"], "estfrequency": 10.0}
 {"ngram": ["exec.task"], "estfrequency": 10.0}
```

6.2. What is the most likely word to follow “**error**” in these emails? _____

6.3. Run a Hive query that finds the top 20 results for words in mbox7.avro that follow the phrase “error in”.

RESULT: You have written several Hive queries that computed bigrams based on the data in the mbox7.avro file, and also computed histograms using the data in the orders table. You should also be familiar with working with Avro files, a popular file format in Hadoop.

SOLUTIONS:

Step 6.3:

```
select
    explode(context_ngrams(sentences(lower(content)),
                           array("error", "in", null) ,20)) as x
  from hive_user_email;
```

Unit 8: Using HCatalog

Topics covered:

- What is HCatalog?
- HCatalog in the Ecosystem
- Defining a New Schema
- Using HCatalogLoader with Pig
- Using HCatStorer with Pig
- Lab 8.1: Using HCatalog with Pig

What is HCatalog?

This is programmer Bob, he uses Pig to crunch data.



Photo Credit: totalAldo via Flickr

This is analyst Joe, he uses Hive to build reports and answer ad-hoc queries.



Bob, I need today's data

Ok

Hmm, is it done yet? Where is it? What format did you use to store it today? Is it compressed? And can you help me load it into Hive?

Dude, we need HCatalog!

© Hortonworks Inc. 2013

Page 127

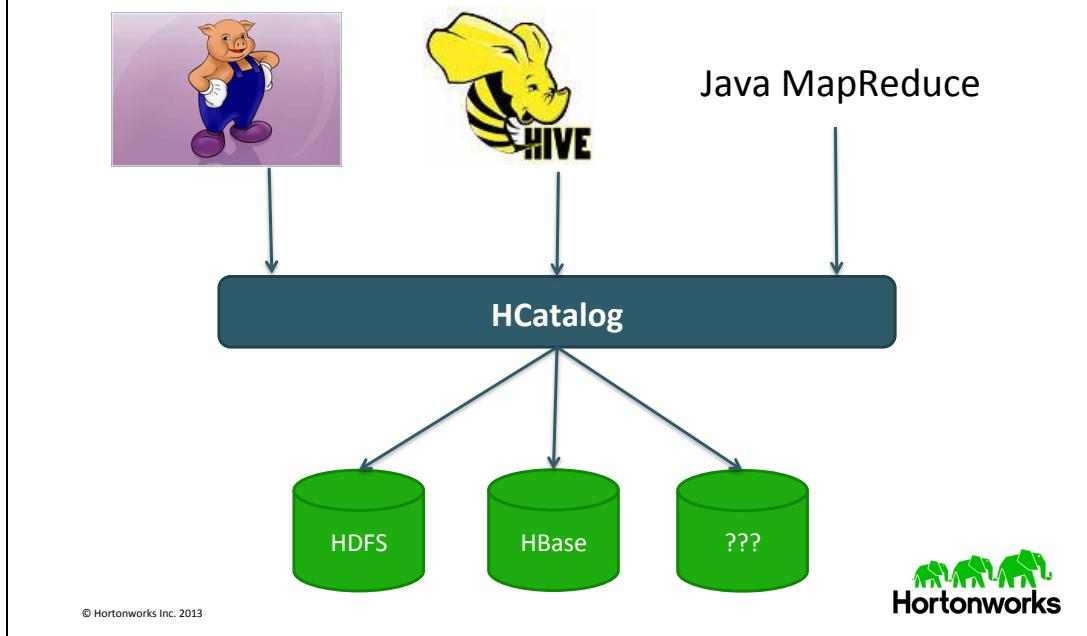
What is HCatalog?

One of the most attractive qualities of Hadoop is its flexibility to require schema on read, not on write. **HCatalog** helps Hadoop deliver on this promise. It is a metadata and table management system for Hadoop. HCatalog has the following features:

- Makes the Hive metastore available to users of other tools on Hadoop.
- Provides connectors for Map Reduce and Pig so that users of those tools can read data from and write data to Hive's warehouse.
- Allows users to share data and metadata across Hive, Pig, and MapReduce.
- Provides a relational view through a SQL-like language (HiveQL) to data within Hadoop.
- Allows users to write their applications without being concerned how or where the data is stored.
- Insulates users from schema and storage format changes.

NOTES

HCatalog in the Ecosystem



HCatalog in the Ecosystem

HCatalog provides a consistent data model for the various tools that use Hadoop. It also provides **table abstraction**, which abstracts some of the details about your data like:

- How the data is stored.
- Where the data resides on the filesystem
- What format that data is in
- What the schema is of the data

Having this information available to Hadoop tools in a consistent fashion can simplify the software development process and also bring consistency of algorithms and results across all the tools and frameworks used in your Hadoop environment.

Defining a New Schema

```
create table mytable (
    id                  int,
    firstname           string,
    lastname            string
)
comment 'An example of an HCatalog
table'
partitioned by (birthday string)
stored as sequencefile;
```

© Hortonworks Inc. 2013



Defining a New Schema

HCatalog is an extension of Hive that exposes the Hive metadata to other tools and frameworks. To define a new HCatalog schema, you simply define a table in Hive.

This means you already have HCatalog schemas defined! The benefit of HCatalog is not in the defining of schemas, but in its ability to expose the schemas and make them available to frameworks outside of Hive.

Using HCatLoader with Pig

- **HCatLoader:** to read data from HCatalog managed tables.

```
emp_relation = LOAD 'employees' USING  
org.apache.hcatalog.pig.HCatLoader();
```

© Hortonworks Inc. 2013



Using HCatLoader with Pig

HCatalog provides two interfaces for use by Pig scripts to read and write data in HCatalog managed tables:

- **HCatLoader:** to read data from HCatalog managed tables.
- **HCatStorer:** to write data to HCatalog managed tables.

For example, the following Pig Latin command loads a table named **employees** managed by HCatalog:

```
emp_relation = LOAD 'employees' USING  
org.apache.hcatalog.pig.HCatLoader();
```

Notice you do not provide a schema when loading a relation with HCatalog. The schema of the relation **emp_relation** is whatever the schema is of the **employees** table.

Using HCatStorer with Pig

- **HCatStorer:** to write data to HCatalog managed tables.

```
STORE customer_projection INTO  
'customers' USING  
org.apache.hcatalog.pig.HCatStorer();
```

© Hortonworks Inc. 2013



Using HCatStorer with Pig

Similarly, if you have a relation that you want to store into an HCatalog managed table, you use the **STORE** command along with the **USING** clause with **HCatStorer**:

```
STORE customer_projection INTO 'customers' USING  
org.apache.hcatalog.pig.HCatStorer();
```

IMPORTANT: For the above command to execute successfully, the field names of the **customer_projection** relation must match the column names of the **customers** table. You will see how this works in the upcoming lab.

Unit 8 Review

1. Where does HCatalog store its schema information? _____
2. List three programming frameworks that can readily access an HCatalog schema:

3. What Java class does Pig use to load data from an HCatalog table?

4. True or False: HCatalog is now merged with Hive.

Lab 8.1: Using HCatalog with Pig

Objective:	Use HCatalog to provide the schema for a Pig relation.
Location of Files:	n/a
Successful Outcome:	You will have written a Pig script that uses HCatLoader to retrieve a schema from an HCatalog table, and HCatStorer to write data to a table managed by HCatalog.
Before You Begin:	You will need to have completed the previous lab that created the wh_visits table in Hive.

Perform the following steps:

Step 1: Start the Grunt Shell

1.1. SSH into your HDP 2.0 virtual machine.

1.2. Start the Grunt shell for use with HCatalog:

```
# pig -useHCatalog
```

Step 2: Load a HCatalog Table

2.1. Define a relation for the **wh_visits** table in Hive using the **HCatLoader()**:

```
grunt> visits = LOAD 'wh_visits' USING  
org.apache.hcatalog.pig.HCatLoader();
```

2.2. View the schema of the **visits** relation to verify it matches the schema of the **wh_visits** table:

```
visits: {lname: chararray, fname: chararray, time_of_arrival:  
chararray, appt_scheduled_time: chararray, meeting_location:  
chararray, info_comment: chararray}
```

Step 3: Run a Pig Query

- 3.1.** Let's execute a query to verify everything is working. Define the following relation:

```
grunt> joe = FILTER visits BY (fname == 'JOE');
```

- 3.2.** Dump the relation:

```
grunt> DUMP joe;
```

The output should be visitors from **wh_visits** with the firstname "JOE".

Step 4: Create an HCatalog Schema

- 4.1.** Quit the Grunt shell and start the Hive shell.

- 4.2.** An HCatalog schema is essentially just a table in the Hive metastore. To define a schema for use with HCatalog, create a table in Hive:

```
hive> create table joes (fname string, lname string,  
comments string);
```

- 4.3.** Verify the table was created successfully using '**show tables**'.

- 4.4.** Use the **describe** command to view the schema of **joes**:

```
hive> describe joes;  
OK  
fname          string          None  
lname          string          None  
comments       string          None
```

Step 5: Using HCatStorer

- 5.1.** Exit the Hive shell and start the Grunt shell again. Be sure to use the **useHCatalog** option:

```
# pig -useHCatalog
```

5.2. Define the **visits** and **joe** relations again (using the up arrow to browse through the history of Pig commands).

5.3. In this step, you will use **HCatStorer** in Pig to input records into the **joes** table. To do this, you need a relation whose fields match the schema of **joes**. You can accomplish this using a projection. Define the following relation:

```
grunt> project_joe = FOREACH joe GENERATE fname, lname,  
info_comment;
```

5.4. Store the projection into the HCatalog table using the **STORE** command:

```
grunt> STORE project_joe INTO 'joes' USING  
org.apache.hcatalog.pig.HCatStorer();
```

This command failed. Why? _____

5.5. Notice the projection has fields named **fname**, **lname** and **info_comment**, but the **joes** table in HCatalog has a schema with **fname**, **lname** and **comments**. The **fname** and **lname** fields match, but **info_comment** needs to be renamed to **comments**. Modify your projection by using the **AS** keyword:

```
project_joe = FOREACH joe GENERATE fname, lname,  
info_comment AS comments;
```

5.6. Now run the **STORE** command again:

```
grunt> STORE project_joe INTO 'joes' USING  
org.apache.hcatalog.pig.HCatStorer();
```

This time the command should work and a MapReduce job will execute.

Step 6: Verify the STORE Worked

6.1. Quit the Grunt shell and start the Hive shell again.

6.2. View the contents of the **joes** table:

```
hive> select * from joes;
```

You should see visitors all named “JOE”, along with their last name and the comments.

Step 7: View the Files

7.1. You can also check the file system to see if a **STORE** command worked. From the command line, view the contents of **/apps/hive/warehouse/joes**:

```
# hadoop fs -ls /apps/hive/warehouse/joes/
Found 1 items
root hdfs      896 /apps/hive/warehouse/joes/part-m-00000
```

Notice that the file for the **joes** table is named **part-m-00000**. Where did that name come from? _____

7.2. Use the **cat** command to view the contents of **part-m-00000**:

```
# hadoop fs -cat /apps/hive/warehouse/joes/part-m-00000
```

As you can see, this is the same list of names from the Hive **select *** query, which should be no surprise at this point in the course!

RESULT: You have seen how to run a Pig script that uses HCatalog to provide the schema using HCatLoader and HCatStorer.

ANSWERS:

5.4: The initial **STORE** command failed because the field names in the relation you were trying to store did not match the column names of the underlying table's schema.

7.1: The **part-m-00000** file is a result of the Pig MapReduce job that executed when you ran the **STORE** command with **HCatStorer**.

Unit 9: Advanced Hive Programming

Topics covered:

- Performing a Multi Table/File Insert
- Understanding Views
- Defining Views
- Using Views
- Overview of Indexes
- Defining Indexes
- The OVER Clause
- Using Windows
- Hive Analytics Functions
- Lab 9.1: Advanced Hive Programming
- Hive File Formats
- Hive SerDes
- Hive ORC Files
- Computing Table Statistics
- Vectorization
- Using HiveServer2
- Understanding Hive on Tez
- Using Tez for Hive Queries
- Hive Optimization Tips
- Hive Query Tunings
- Lab 9.2: Streaming Data with Hive and Python

Performing a Multi Table/File Insert

```
insert overwrite directory '2013_visitors' select * from wh_visits  
where visit_year='2013'  
insert overwrite directory 'ca_congress' select * from congress  
where state='CA';
```

No semi-colon

```
from visitors  
INSERT OVERWRITE TABLE gender_sum  
    SELECT visitors.gender, count_distinct(visitors.userid)  
    GROUP BY visitors.gender  
INSERT OVERWRITE DIRECTORY '/user/tmp/age_sum'  
    SELECT visitors.age, count_distinct(visitors.userid)  
    GROUP BY visitors.age;
```

© Hortonworks Inc. 2013



Performing a Multi Table/File Insert

Hive queries are converted into one or more MapReduce jobs and executed on a Hadoop cluster. Your Hive query might result in a map-only job, a single mapper and a single reducer, or multiple mappers and reducers. Each MapReduce job requires a lot of work on the cluster, and some Hive queries can take a very long time (hours) to execute.

One clever trick you can do when querying Big Data using Hive is to perform a multi table or file insert, where you essentially run multiple queries within a single MapReduce job. The queries do not even need to process the same tables.

Consider the following simple Hive query that selects all White House visitors for the year 2013.

```
insert overwrite directory '2013_visitors' select * from  
wh_visits where visit_year='2013' ;
```

Now suppose we have the following query on a different table named **congress**:

```
insert overwrite directory 'ca_congress' select * from  
congress where state='CA' ;
```

Each query above requires a MapReduce job, as expected.

Notice in the following Hive query that we perform both selects in the same query:

```
insert overwrite directory '2013_visitors' select * from  
wh_visits where visit_year='2013'  
insert overwrite directory 'ca_congress' select * from  
congress where state='CA' ;
```

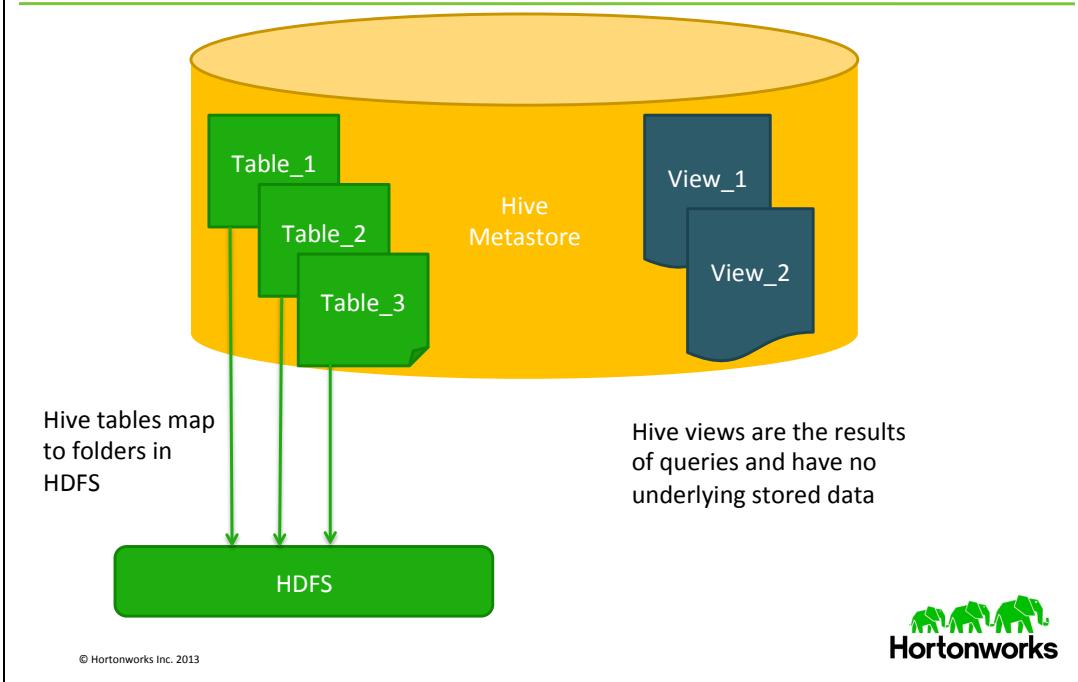
Notice the only difference is that the semi-colon was removed after the first query - which means the Hive code above is a single statement. The important result is that the above Hive command runs as a single MapReduce job! Two output folders are created (**2013_visitors** and **ca_congress**) and the data from two separate Hive tables are processed, but all in a single MapReduce job.

NOTE: Using a multi-file insert may seem a bit odd, but it is important to understand how Hive queries relate to underlying MapReduce jobs. In general, you can gain a lot of performance by running two tasks at the same time, instead of running two separate MapReduce jobs

The following example demonstrates querying from the same table, with one result being output to another table and the other result getting written to HDFS:

```
from visitors  
INSERT OVERWRITE TABLE gender_sum  
    SELECT visitors.gender, count_distinct(visitors.userid)  
    GROUP BY visitors.gender  
  
INSERT OVERWRITE DIRECTORY '/user/tmp/age_sum'  
    SELECT visitors.age, count_distinct(visitors.userid)  
    GROUP BY visitors.age;
```

Understanding Views



Understanding Views

A **view** in Hive is defined by a **SELECT** statement and allows you to treat the result of the query like a table. The table does not actually exist, and the query does not execute until the statement that refers to the view is executed.

Use cases for using views include:

- Define a view to reduce the complexity of a query. For example, a nested SELECT statement can be defined separately as a view.
- Restrict a user's access to subset of an actual Hive table by defining a view that contains only the columns and rows that the user needs.

NOTE: Depending on the query, a view gets combined (optimized) into the query that is using the view, or the view may have to be executed in its own MapReduce job. For example, if the view query contains an **ORDER BY**, then it will execute in its own MapReduce job.

NOTE: Views in Hive are non-materialized, so you can use them without concern of creating more work for the resulting MapReduce job.

Defining Views

```
CREATE VIEW 2010_visitors AS
  SELECT fname, lname,
         time_of_arrival, info_comment
    FROM wh_visits
   WHERE
     cast(substring(time_of_arrival,6,4)
AS int) >= 2010
      AND
     cast(substring(time_of_arrival,6,4)
AS int) < 2011;
```

© Hortonworks Inc. 2013



Defining Views

A view is defined using the **CREATE VIEW** statement. For example, the following Hive statement defines a view named **2010_visitors**:

```
CREATE VIEW 2010_visitors AS
  SELECT fname, lname, time_of_arrival, info_comment
    FROM wh_visits
   WHERE
     cast(substring(time_of_arrival,6,4) AS int) >= 2010
      AND
     cast(substring(time_of_arrival,6,4) AS int) < 2011;
```

The **2010_visitors** is a view that represents people that visited the White House in the year 2010.

A view is not a table in Hive with actual data, but a view can be treated like a table. For example, you can run the DESCRIBE command on a view to see its schema:

```
hive> describe 2010_visitors;
OK
fname          string      None
lname          string      None
time_of_arrival string      None
info_comment   string      None
```

A view will also show up in your list of tables. For example, notice the output of the **SHOW TABLES** command:

```
hive> show tables;
OK
2010_visitors
wh_visits
```

Similar to a table, you can delete a view using the **DROP VIEW** command. For example:

```
DROP VIEW 2010_visitors;
```

NOTE: Views can also contain partitions, just like tables. This allows you to define views that behave exactly like your underlying tables, even tables that are partitioned.

Using Views

- You use a view just like a table:

```
from 2010_visitors
select *
where info_comment like "%CONGRESS%"
order by lname;
```

© Hortonworks Inc. 2013



Using Views

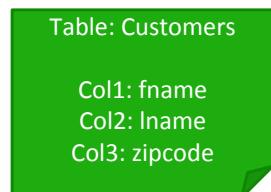
You can use a view in a query just like you would use a table. For example, the following query uses the 2010_visitors view to find visitors the President:

```
from 2010_visitors
select *
where info_comment like "%CONGRESS%"
order by lname;
```

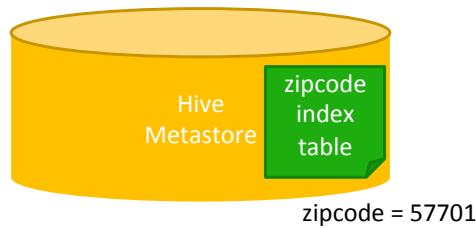
Notice you could have performed the above query without using a view. Instead, you could have defined a longer WHERE clause or a nested SELECT statement. However, using a view keeps the SQL easier to read. This is obviously a simple example, but it demonstrates the power and usefulness of views. Hive will determine at runtime the best way to convert the above command into one or more MapReduce jobs.

Overview of Indexes

1. An index is defined on the zipcode column of a table named Customers.



2. An index table is created in the Hive metastore for the zipcode column



3. The index table now knows which blocks in HDFS contain each zipcode



© Hortonworks Inc. 2013



Overview of Indexes

Indexing was added to Hive in version 0.7.0 and allows you to create an index for a column. Hive indexes have the following properties:

- An index can improve the speed of a query lookup on certain columns. If a column is indexed, then the entire table does not need to be loaded and processed.
- The index data for a table is stored in another table in the Hive metastore known as the **index table**.
- Indexes can be a good alternative to partitioning, especially when the partitioned column would result in a large number of small partitions.

The index table keeps track of the blocks in HDFS that contain each occurrence of a particular value. When a query is executed on the indexed column, entire blocks of data can be skipped if the block does not contain the requested column value.

In the example above, suppose I run a query on the **zipcode** column of the **Customers** table:

```
SELECT * FROM Customers WHERE zipcode = 57701
```

Only the blocks in HDFS that contain customers with a zipcode of 57701 will be processed. Without indexing, all blocks of the Customers table would need to be processed.

NOTE: In this course we discuss indexes because many of you will be familiar with them in RDBMS. However, it is important to be realistic about their usage, as indexes are a bit of a work in progress in Hive, especially in terms of performance.

At this point in time, it is a better practice in Hive to take advantage of buckets or skew instead of using indexes.

Defining Indexes

```
CREATE INDEX zipcode_index ON TABLE  
Customers (zipcode) AS 'COMPACT' WITH  
DEFERRED REBUILD;
```

```
ALTER INDEX zipcode_index ON  
Customers REBUILD;
```

```
SHOW INDEX ON Customers;
```

```
DROP INDEX zipcode_index ON  
Customers;
```

© Hortonworks Inc. 2013



Defining Indexes

Use the **CREATE INDEX** statement to define an index. For example:

```
CREATE INDEX zipcode_index ON TABLE Customers (zipcode) AS  
'COMPACT' WITH DEFERRED REBUILD;
```

- The **AS** clause specifies which index handler to use. The **COMPACT** option is a shortcut for specifying the **CompactIndexHandler** Java class. The other option is **BITMAP** for using a bitmap index. You can also define your own index handler class by writing a custom Java class.
- The **WITH DEFERRED REBUILD** clause specifies that the index table not be built immediately. You need to run the **REBUILD** command to actually create the index table:

```
ALTER INDEX zipcode_index ON Customers REBUILD;
```

The above command triggers a MapReduce job that builds the index table for zipcode.

IMPORTANT: If you modify the **Customers** table by appending data to it, you need to rebuild the zipcode index table using the **REBUILD** command.

Use the **SHOW INDEX** statement to view the indexes of a table:

```
SHOW INDEX ON Customers;
```

Use the **DROP INDEX** statement to delete an index and its underlying index table:

```
DROP INDEX zipcode_index ON Customers;
```

The OVER Clause

orders			result set	
cid	price	quantity	cid	max(price)
4150	10.50	3	2934	39.99
11914	12.25	27	4150	10.50
4150	5.99	5	11914	40.50
2934	39.99	22		
11914	40.50	10		

```
SELECT cid, max(price) FROM orders GROUP BY cid;
```

orders			result set	
cid	price	quantity	cid	max(price)
4150	10.50	3	2934	39.99
11914	12.25	27	4150	10.50
4150	5.99	5	4150	10.50
2934	39.99	22	11914	40.50
11914	40.50	10	11914	40.50

```
SELECT cid, max(price) OVER (PARTITION BY cid) FROM orders;
```

The OVER Clause

Hive 0.11 introduced windowing capabilities to the Hive QL. Similar to an aggregate function (like **GROUP BY**), a **window function** performs a calculation across a set of table rows that are somehow related, except that a window function does not cause rows to become grouped into a single output row - the rows retain their separate identities.

This is best demonstrated by the **OVER** clause, as you can see in the result above. The **GROUP BY** statement finds the maximum price of each order, and the results are aggregated into a single row for each unique **cid**.

The **OVER** clause does not aggregate the result, but instead maintains each row of data and outputs the maximum price of the each **cid** group.

Using Windows

orders			result set	
cid	price	quantity	cid	sum(price)
4150	10.50	3	4150	5.99
11914	12.25	27	4150	16.49
4150	5.99	5	4150	36.49
4150	39.99	22	4150	70.49
11914	40.50	10	11914	12.25
4150	20.00	2	11914	52.75

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY price  
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) FROM orders;
```

© Hortonworks Inc. 2013



Using Windows

The OVER clause allows you to define a window over which to perform a specific calculation. For example, the following Hive statement computes the sum of each order, but the sum is not computed over all prices in an order. Instead, the sum is computed over a window that includes the current row and the two preceding rows, as ordered by the **price** column.

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY  
price ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) FROM  
orders;
```

Study the output carefully and see if you verify that the result is what you expected based on the query.

The **FOLLOWING** statement is used to specify rows after the current row:

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY
price ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING) FROM
orders;
```

Use the **UNBOUNDED** statement to specify all prior or following rows:

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY
price ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM orders;
```

NOTE: Hive window functions also include the **LEAD** and **LAG** functions for specifying the number of rows to lead ahead or lag behind in the window. Their usage is identical to SQL-92.

Hive Analytics Function

orders			result set		
cid	price	quantity	cid	quantity	rank
4150	10.50	3	4150	2	1
11914	12.25	27	4150	3	2
4150	5.99	5	4150	5	3
4150	39.99	22	4150	22	4
11914	40.50	10	11914	10	1
4150	20.00	2	11914	27	2

SELECT cid, quantity, rank() OVER (PARTITION BY cid ORDER BY quantity) FROM orders;

© Hortonworks Inc. 2013



Hive Analytics Functions

Hive 0.11 also added the following SQL standard analytics functions:

- **RANK**: Returns the rank of each row within the partition of a result set.
- **DENSE_RANK**: Returns the rank of rows within the partition of a result set, without any gaps in the ranking.
- **PERCENT_RANK**: Calculates the relative rank of a row within a group of rows.
- **ROW_NUMBER**: Returns the sequential number of a row within a partition of a result set.
- **CUME_DIST**: For a row **r**, the **CUME_DIST** of **r** is the number of rows with values lower than or equal to the value of **r**, divided by the number of rows evaluated in the partition.
- **NTILE**: Distributes the rows in an ordered partition into a specified number of groups. For each row, **NTILE** returns the number of the group to which the row belongs.

Lab 9.1: Advanced Hive Programming

Objective:	To understand how some of the more advanced features of Hive work, including multi-table inserts, views and windowing.
Location of Files:	/root/labs/Lab9.1
Successful Outcome:	You will have executed numerous Hive queries on customer order data.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: Create and Populate a Hive Table

- 1.1. From the command line, change directories to the **Lab9.1** folder:

```
# cd ~/labs/Lab9.1
```

- 1.2. View the contents of the **orders.hive** file in the **Lab9.1** folder:

```
# more orders.hive
```

Notice it defines a Hive table named **orders** that has 7 columns. Notice it also loads the contents of **/tmp/shop.tsv** into the **orders** table.

- 1.3. Copy **shop.tsv** into the **/tmp** folder:

```
# cp shop.tsv /tmp/
```

- 1.4. Execute the contents of **orders.hive**:

```
# hive -f orders.hive
```

1.5. From the Hive shell, verify the script worked by running the following two commands:

```
hive> describe orders;
hive> select count(*) from orders;
```

Your **orders** table should contain 99,999 records.

Step 2: Analyze the Customer Data

2.1. Let's run a few queries to see what this data looks like. Start by verifying that the **username** column actually looks like names:

```
hive> SELECT username FROM orders LIMIT 10;
```

You should see ten first names.

2.2. The orders table contains orders placed by customers. Run the following query, that shows the 10 lowest-price orders:

```
hive> SELECT username, ordertotal FROM orders ORDER BY
ordertotal LIMIT 10;
```

The smallest orders are each \$10, as you can see from the output:

```
Jeremy 10
Christina 10
Jasmine 10
Hannah 10
Thomas 10
Michelle 10
Brian 10
Amber 10
Maria 10
Victoria 10
```

2.3. Run the same query, but this time use descending order:

```
hive> SELECT username, ordertotal FROM orders ORDER BY
ordertotal DESC LIMIT 10;
```

The output this time is the 10 highest-priced orders:

```
Mark 612
Jordan 612
```

```
Anthony      612
Brandon      612
Sean        612
Paul        611
Nathan      611
Eric        611
Jonathan     611
Andrew      610
```

2.4. Let's find out if men or women spent more money:

```
hive> SELECT sum(ordertotal), gender
FROM orders GROUP BY gender;
```

Based on the output, which gender has spent more money on purchases? _____

2.5. The **orderdate** column is a string with the format **yyyy-mm-dd**. Use the **year** function to extract the various parts of the date. For example, run the following query, which computes the sum of all orders for each year:

```
SELECT sum(ordertotal), year(order_date) FROM orders GROUP
BY year(order_date);
```

The output should look like:

```
4082780      2009
4404806      2010
4399886      2011
4248950      2012
2570749      2013
```

Step 3: Multi-File Insert

3.1. In this step, you will run two completely different queries, but in a single MapReduce job. The output of the queries will be in two separate directories in HDFS. Start by creating a new text file in the **Lab9.1** folder named **multifile.hive**.

3.2. Within the text file, enter the following query. Notice there is no semi-colon between the two **INSERT** statements:

```
FROM ORDERS o
INSERT OVERWRITE DIRECTORY '2010_orders'
    SELECT o.* WHERE year(order_date) = 2010
INSERT OVERWRITE DIRECTORY 'software'
    SELECT o.* WHERE itemlist LIKE '%Software%';
```

3.3. Save your changes to **multifile.hive**.

3.4. Run the query from the command line:

```
# hive -f multifile.hive
```

3.5. The above query executes in a single MapReduce job. Even more interesting, it only requires a map phase. Why did this job not require a reduce phase?

3.6. Verify the two queries executed successfully by viewing the folders in HDFS:

```
# hadoop fs -ls
```

You should see two new folders: **2010_orders** and **software**.

3.7. View the output files in these two folders. Verify the **2010_orders** directory contains orders from only the year 2010, and verify the **software** directory contains only orders that included '**Software**'.

Step 4: Define a View

4.1. Define a view named **2013_orders** that contains the **orderid**, **order_date**, **username**, and **itemlist** columns of the **orders** table where the **order_date** was in the year 2013.

4.2. Run the **show tables** command:

```
hive> show tables;
```

You should see **2013_orders** in the list of tables.

4.3. To verify your view is defined correctly, run the following query:

```
hive> SELECT COUNT(*) FROM 2013_orders;
```

The **2013_orders** view should contain 13,104 records.

Step 5: Find the Maximum Order of Each Customer

5.1. Suppose you want to find the maximum order of each customer. This can be done easily enough with the following Hive query. Run this query now:

```
hive> SELECT max(ordertotal), userid
```

```
FROM orders GROUP BY userid;
```

5.2. How many different customers are in the **orders** table? _____

5.3. Suppose you want to add the itemlist column to the previous query. Try adding it to the **SELECT** clause:

```
hive> SELECT max(ordertotal), userid, itemlist  
FROM orders GROUP BY userid;
```

Notice this query is not valid because **itemlist** is not in the **GROUP BY** key.

5.4. We can join the result set of the max-total query with the **orders** table to add the **itemlist** to our result. Start by defining a view named **max_ordertotal** for the maximum order of each customer:

```
hive> CREATE VIEW max_ordertotal AS  
SELECT max(ordertotal) AS maxtotal, userid  
FROM orders GROUP BY userid;
```

5.5. Now join the **orders** table with your **max_ordertotal** view:

```
hive> SELECT ordertotal, orders.userid, itemlist  
FROM orders  
JOIN max_ordertotal ON  
max_ordertotal.userid = orders.userid  
AND  
max_ordertotal.maxtotal = orders.ordertotal;
```

5.6. How many MapReduce jobs did this query need? _____

5.7. The end of your output should look like:

```
600    98  
Grill,Freezer,Bedding,Headphones,DVD,Table,Grill,Software,D  
ishwasher,DVD,Microwave,Adapter  
600    99    Washer,Cookware,Vacuum,Freezer,2-Way  
Radio,Bicycle,Washer & Dryer,Coffee  
Maker,Refrigerator,DVD,Boots,DVD  
600    100   Bicycle,Washer,DVD,Wrench Set,Sweater,2-Way  
Radio,Pants,Freezer,Blankets,Grill,Adapter,pillows
```

NOTE: In the next lab, you will optimize this query using a custom Python script to avoid the need for two MapReduce jobs.

Step 6: Fixing the GROUP BY Key Error

6.1. Let's compute the sum of all of the orders of all customers. Start by entering the following query:

```
SELECT sum(ordertotal), userid FROM orders GROUP BY userid;
```

Notice the output is the sum of all orders, but displaying just the **userid** is not very exciting.

6.2. Try to add the **username** column to the **SELECT** clause:

```
SELECT sum(ordertotal), userid, username  
FROM orders  
GROUP BY userid;
```

This generates the infamous “Expression not in GROUP BY key” error, because **username** column is not being aggregated but the **ordertotal** is.

6.3. An easy fix is to aggregate the **username** values using the **collect_set** function, but output only one of them:

```
SELECT sum(ordertotal), userid, collect_set(username) [0]  
FROM orders  
GROUP BY userid;
```

You should get the same output as before, but this time the **username** is included.

Step 7: Using the **OVER** Clause

7.1. Now let's compute the sum of all orders for each customer, but this time use the **OVER** clause to not group the output and to also display the **itemlist** column:

```
SELECT userid, itemlist, sum(ordertotal)  
OVER (PARTITION BY userid)  
FROM orders;
```

Notice the output contains every order, along with the items they purchased and the sum of all the orders ever placed from that particular customer.

Step 8: Using the Window Functions

- 8.1.** It is not difficult to compute the sum of all orders for each day using the **GROUP BY** clause:

```
select order_date, sum(ordertotal)
FROM orders
GROUP BY order_date;
```

Run the query above and the tail of the output should look like:

```
2013-07-28 18362
2013-07-29 3233
2013-07-30 4468
2013-07-31 4714
```

- 8.2.** Suppose you want to compute the sum for each day that includes each order. This can be done using a window that sums all previous orders along with the current row:

```
SELECT order_date, sum(ordertotal)
OVER
(PARTITION BY order_date ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW)
FROM orders;
```

To verify it worked, your tail of your output should look like:

```
2013-07-31 3163
2013-07-31 3415
2013-07-31 3607
2013-07-31 4146
2013-07-31 4470
2013-07-31 4610
2013-07-31 4714
```

Step 9: Using the Hive Analytics Functions

- 9.1.** Run the following query, which displays the rank of the **ordertotal** by day:

```
SELECT order_date, ordertotal, rank()
OVER
(PARTITION BY order_date ORDER BY ordertotal)
FROM orders;
```

9.2. To verify it worked, the output of July 31, 2013, should look like:

2013-07-31	48	1
2013-07-31	104	2
2013-07-31	119	3
2013-07-31	130	4
2013-07-31	133	5
2013-07-31	135	6
2013-07-31	140	7
2013-07-31	147	8
2013-07-31	156	9
2013-07-31	192	10
2013-07-31	192	10
2013-07-31	196	12
2013-07-31	240	13
2013-07-31	252	14
2013-07-31	296	15
2013-07-31	324	16
2013-07-31	343	17
2013-07-31	500	18
2013-07-31	528	19
2013-07-31	539	20

9.3. As a challenge, see if you can run a query similar to the previous one except compute the rank over months, instead of each day.

Step 10: Histograms

10.1. Run the following Hive query, which uses the **histogram_numeric** function to compute 20 (x,y) pairs of the frequency distribution of the total order amount from customers who purchased a microwave (using the **orders** table):

```
select
    explode(histogram_numeric(ordertotal,20)) as x
    from orders
    where itemlist LIKE "%Microwave%";
```

The output should look like the following:

{"x":14.33333333333332,"y":3.0}
{"x":33.87755102040816,"y":441.0}
{"x":62.52577319587637,"y":679.0}
{"x":89.37823834196874,"y":965.0}
{"x":115.1242236024843,"y":1127.0}
{"x":142.646885672939,"y":1382.0}
{"x":174.07664233576656,"y":1370.0}
{"x":208.06909090909105,"y":1375.0}
{"x":242.55486381322928,"y":1285.0}

```
{"x":275.8625954198475,"y":1048.0}
{"x":304.71100917431284,"y":872.0}
{"x":333.1514423076924,"y":832.0}
{"x":363.7630208333335,"y":768.0}
{"x":397.51587301587364,"y":756.0}
{"x":430.9072847682117,"y":604.0}
{"x":461.68715083798895,"y":537.0}
 {"x":494.1598360655734,"y":488.0}
 {"x":528.5816326530613,"y":294.0}
 {"x":555.516666666672,"y":180.0}
 {"x":588.7979797979801,"y":198.0}
```

10.2. Write a similar Hive query that computes 10 frequency distribution pairs for the **ordertotal** from the **orders** table where **ordertotal** is greater than \$200. The output should look like:

```
{"x":218.8195174551819,"y":7419.0}
 {"x":254.10237580993478,"y":6945.0}
 {"x":293.4231618807192,"y":6338.0}
 {"x":334.57302573203015,"y":5635.0}
 {"x":379.79714934930786,"y":4841.0}
 {"x":428.1165628891644,"y":4015.0}
 {"x":473.1484734420741,"y":2391.0}
 {"x":511.2576946288467,"y":1657.0}
 {"x":549.0106899902812,"y":1029.0}
 {"x":589.0761194029857,"y":670.0}
```

RESULT: You should now be comfortable running Hive queries and using some of the more advanced features of Hive like views and the window functions.

ANSWERS:

2.4: Men spent \$9,919,847 and women spent \$9,787,324.

3.5: Because the query only does a **SELECT ***, no reduce phase was needed.

5.2: There are 100 unique customers in the **orders** table.

5.6: The query resulted in two MapReduce jobs.

SOLUTIONS:

Step 4.1: The **2013_orders** view:

```
CREATE VIEW 2013_orders AS
SELECT orderid, order_date, username, itemlist
FROM orders
WHERE year(order_date) = '2013';
```

Step 9.3: The rank query by month:

```
select substr(order_date,0,7), ordertotal, rank()
OVER
(PARTITION BY substr(order_date,0,7) ORDER BY ordertotal)
FROM orders;
```

Step 10.2:

```
select
    explode(histogram_numeric(ordertotal,10)) as x
  from orders
 where ordertotal > 200;
```

Hive File Formats

- Text file
- SequenceFile
- RCFile
- ORC File

```
CREATE TABLE names  
  (fname string, lname string)  
STORED AS RCFile;
```

© Hortonworks Inc. 2014



Hive File Formats

As you have seen, Hive does not store data. The data for a table is stored in HDFS in one of the following formats:

- **Text file:** comma, tab or other delimited file types.
- **SequenceFile:** stores serialized key/value pairs that can quickly be deserialized in Hadoop.
- **RCFile:** a *record columnar file* that organizes data by columns (as opposed to the traditional database row format).
- **ORC File:** the optimized row columnar format that improves the efficiency of Hive by a considerable amount (discussed in more detail on the next slide).

You specify a file format when you create the table, using the STORED AS clause:

```
CREATE TABLE tablename (
  ...
) STORED AS fileformat;
```

For example, the following table is for data using the RCFile format:

```
CREATE TABLE names
  (fname string, lname string)
STORED AS RCFile;
```

Hive SerDes

- SerDe = serializer/deserializer
- Determines how records are read from a table and written to HDFS.

```
CREATE TABLE names
  (fname string, lname string)
ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "([0-9]+\.[0-9]+\.[0-9]+\.
[0-9]+)[^[]+\[(\[^ ]+)\]\[^/]+(\[^ ]+)\.+",
  "output.format.string" = "%1$s %2$s %3$s"
)
STORED AS RCFile;
```

© Hortonworks Inc. 2014



Hive SerDes

SerDe is short for serializer/deserializer, and refers to how records are read in from a table (deserialized) and written back out to HDFS (serialized). Records can be stored in any custom format you want by writing Java classes, or you can use one of the several built-in SerDes, including:

- **AvroSerDe**: for reading and writing files using an Avro schema.
- **RegexSerDe**: uses a regular expression to deserialize data.
- **ColumnarSerDe**: for columnar based storage supported by RCFiles.
- **OrcSerDe**: for reading and writing to ORC files.

There are quite a few built-in SerDes, so check the documentation for a complete list.

NOTE: There are third-party SerDes available as well, so do a search online before attempting to develop a custom SerDe that might already be available!

Using SerDes requires the **ROW FORMAT SERDE** clause. For example, the following table is for data stored in the Avro format:

```
CREATE TABLE names (fname string, lname string)
  ROW FORMAT SERDE
  'org.apache.hadoop.hive.serde2.RegexSerDe'
  WITH SERDEPROPERTIES (
    "input.regex" = "([^\s]) . . [([^\s]+)]",
    "output.format.string" = "%1$s %2$s %3$s"
  )
  STORED AS RCfile;
```

Hive ORC Files

- The **Optimized Row Columnar** (ORC) file format provides a highly efficient way to store Hive data.

```
CREATE TABLE tablename (
  ...
) STORED AS ORC;

ALTER TABLE tablename SET FILEFORMAT
ORC;

SET hive.default.fileformat=Orc
```

© Hortonworks Inc. 2013



Hive ORC Files

The **Optimized Row Columnar** (ORC) file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data.

File formats in Hive are specified at the table level. Use the **AS** keyword and specify the **ORC** file format:

```
CREATE TABLE tablename (
  ...
) AS ORC;
```

You can also modify the file format of an existing table:

```
ALTER TABLE tablename SET FILEFORMAT ORC;
```

You can also specify ORC as the default file format of new tables:

```
SET hive.default.fileformat=Orc
```

ORC files have three main components:

- Stripe
- Footer
- Postscript

Here are the features of these components:

- An ORC file is broken down into sets of rows called ***stripes***.
- The default stripe size is 250 MB. Large stripe sizes enable efficient reads of columns.
- An ORC file contains a ***footer*** that contains the list of stripe locations.
- The footer also contains column data like the count, min, max, and sum.
- At the end of the file, the ***postscript*** holds compression parameters and the size of the compressed footer.

Demonstration: ORC Files

Objective:	To learn how to create a table that uses the ORC file format.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Define and Populate the **salaries** Table

1.1. Review the contents of the file **salaries.hive** located in the **demos** folder.

1.2. Run the script:

```
# cd ~/labs/demos  
# hive -f salaries.hive
```

1.3. Verify it worked by running the following query from the Hive shell:

```
hive> select * from salaries;
```

You should see 50 rows.

Step 2: Define the ORC Table

2.1. Define the following Hive table named **orctest**. The following command is in **demos/orctest.hive**:

```
create table orctest (  
    gender string,  
    age int,  
    salary double,  
    zip int  
)  
stored as ORC;
```

Notice it uses **ORC** as the file format.

Step 3: Verify the Table Definition

- 3.1.** From the Hive shell, run the **describe formatted** command on **orctest**:

```
hive> describe formatted orctest;
```

- 3.2.** Notice the **OrcSerDe** is being used, along with the ORC input and output formats.

Step 4: Populate the ORC Table

- 4.1.** The orctest table is currently empty, as you can see by viewing the contents of **/apps/hive/warehouse/orctest**.

- 4.2.** Run the following query, which populates **orctest** with the contents of **salaries**:

```
hive> insert overwrite table orctest
      > select * from salaries;
```

A series of 3 MapReduce jobs will execute.

- 4.3.** Verify the table is populated by performing the following query:

```
hive> select * from orctest;
```

You should see the same 50 rows that are in the **salaries** table.

Step 5: View the ORC File

- 5.1.** View the contents of the Hive **warehouse** folder:

```
# hadoop fs -ls /apps/hive/warehouse/orctest
Found 1 items
1 root hdfs 720 /apps/hive/warehouse/orctest/000000_0
```

- 5.2.** The file in the **orctest** folder is a binary file in the ORC format, as you can see by viewing its contents:

```
# hadoop fs -cat /apps/hive/warehouse/orctest/000000_0
```

As you can see, the output is just binary characters.

Computing Table Statistics

```
ANALYZE TABLE tablename COMPUTE  
STATISTICS;
```

```
DESCRIBE FORMATTED tablename
```

```
DESCRIBE EXTENDED tablename
```

© Hortonworks Inc. 2013



Computing Table Statistics

Hive can store table and partition statistics in its metastore. There are two types of table statistics currently supported by Hive:

- **Table and partition statistics:** number of rows, number of files, raw data size, number of partitions.
- **Column Level Top K statistics:** number of null values, number of true/false values, maximum and minimum values, estimate of number of distinct values, average column length, maximum column length, and height balanced histograms.

IMPORTANT: Column statistics are computed using the *top K algorithm*; hence the name Top K statistics. Column statistics is still a work in progress and has not been included in the current stable release of Hive.

The **ANALYZE TABLE** command gathers statistics for a table, partition and columns and writes them to the Hive metastore. To compute table statistics, the syntax looks like:

```
ANALYZE TABLE tablename COMPUTE STATISTICS;
```

For computing stats on partitions, use the **PARTITION** command:

```
ANALYZE TABLE tablename PARTITION(part1, part2,..) COMPUTE STATISTICS
```

The **ANALYZE** command runs a MapReduce job that processes the entire table. The table and partition stats are output to the command window:

```
Table default.customers stats: [num_partitions: 0,  
num_files: 11, num_rows: 891048, total_size: 4605775,  
raw_data_size: 0]
```

You can also view these stats of a table by running the **DESCRIBE** command:

```
DESCRIBE FORMATTED tablename  
DESCRIBE EXTENDED tablename
```

You can also specify one or more partitions to view details at the partition level:

```
DESCRIBE EXTENDED tablename PARTITION(part1=value1,  
part2=value2);
```

Vectorization

Before



After



Vectorization + ORC files = huge breakthrough in Hive query performance

© Hortonworks Inc. 2013



Vectorization

Vectorization is a new feature that allows Hive to process a batch of up to 1,024 rows together, instead of processing one row at a time. Each batch consists of a **column vector**, which is usually an array of primitive types. Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage.

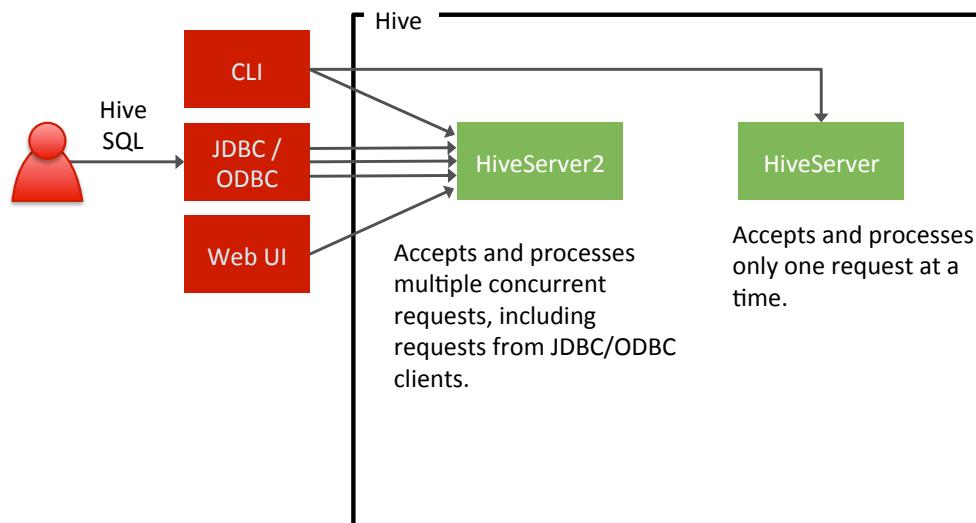
To take advantage of vectorization, your table needs to be in the ORC format and you need to enable vectorization with the following property:

```
hive.vectorized.execution.enabled=true
```

When vectorization is enabled, Hive examines the query and the data to determine whether vectorization can be supported. If it cannot be supported, Hive will execute the query with vectorization turned off.

NOTE: Vectorization is a joint effort between Hortonworks and Microsoft. The improvements from vectorization, in addition to the new ORC file format, have helped increase the speed of Hive queries by a large magnitude.

Using HiveServer2



© Hortonworks Inc. 2013



Using HiveServer2

As we discussed earlier, Hive queries are submitted to a HiveServer process. Older versions of Hive used the hiveserver process, which can only process one request at a time. HDP 2.0 ships with **HiveServer2**, a Thrift-based implementation that allows multiple concurrent connections and also supports Kerberos authentication.

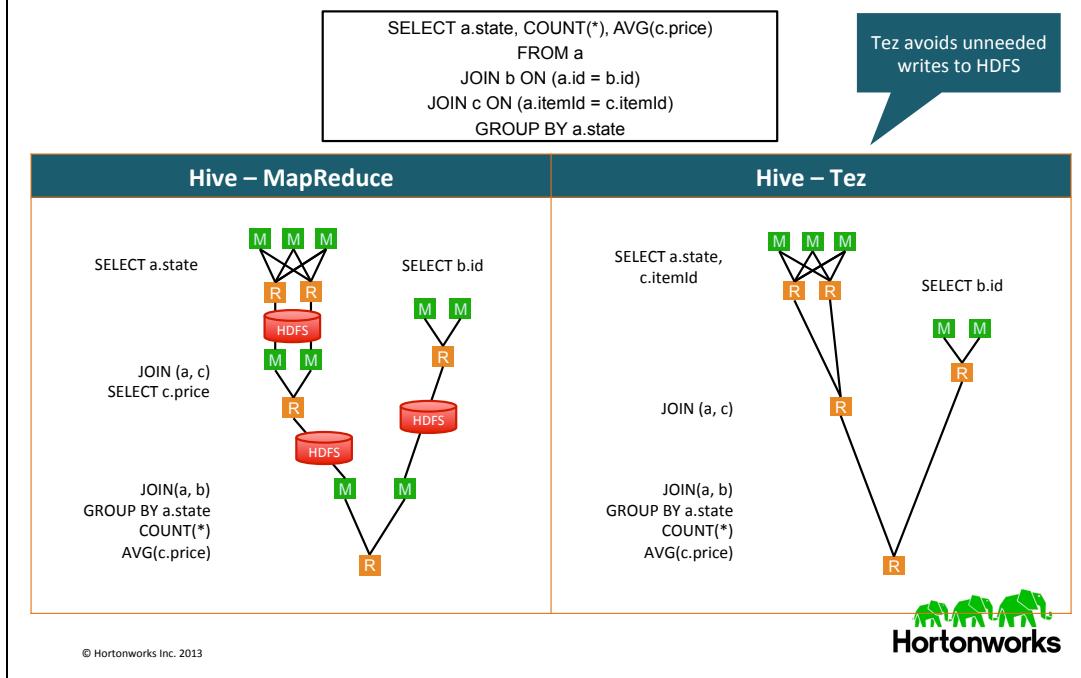
- A new HiveServer2 instance is started with the **hiveserver2** binary, or it can be run as a service.
- Settings are defined in **hive-site.xml**, except for the bind host and port, which can be defined using the **HIVE_SERVER2_THRIFT_BIND_HOST** and **HIVE_SERVER2_THRIFT_PORT** environment variables. This allows you to run multiple HiveServer2 instances on the same machine.
- You need a HiveServer2 instance running if you are going to use Beeline.

For example:

```
set HIVE_SERVER2_THRIFT_PORT=12345
hive --service hiveserver2
```

The above command runs a **hiveserver2** instance on port 12345.

Understanding Hive on Tez



Understanding Hive on Tez

Tez provides a general-purpose, highly customizable framework that simplifies data-processing tasks across both small scale (low-latency) and large-scale (high throughput) workloads in Hadoop. It generalizes the MapReduce paradigm to a more powerful framework by providing the ability to execute a complex DAG of tasks for a single job.

As you can see in the diagram above, a Hive query without Tez can consist of multiple MapReduce jobs. Tez performs a Hive query in a single job, avoiding the intermediate writes to disk that were a result of the multiple MapReduce jobs.

Using Tez for Hive Queries

- Set the following properties in either **hive-site.xml** or in your script:

```
set hive.use.tez.natively=true;  
set hive.enable.mrr=true;
```

© Hortonworks Inc. 2013



Using Tez for Hive Queries

To use Tez for a Hive query, you need to define the following properties by adding them to your Hive script or in **hive-site.xml**:

```
set hive.use.tez.natively=true;  
set hive.enable.mrr=true;
```

Hive Optimization Tips

- Divide data amongst different files which can be pruned out by using partitions, buckets and skews
- Sort data ahead of time to simplify joins
- Use the ORC file format
- Sort and Bucket on common join keys
- Use map (broadcast) joins whenever possible
- Increase the replication factor for hot data (which reduces latency)
- Take advantage of Tez

© Hortonworks Inc. 2013



Hive Optimization Tips

Above are some helpful design tips for improving the speed of Hive queries.

NOTE: Hive has a special file called the `.hiverc` file that gets executed each time you launch a Hive shell. This makes the `.hiverc` file a great place for adding custom configuration settings that you use all the time, or loading JAR files that contain frequently-used UDFs. The file is saved in the Hive conf directory, which is `/etc/hive/conf` for an HDP installation.

Hive Query Tunings

- `mapreduce.input.fileinputformat.split.maxsize`
- `mapreduce.input.fileinputformat.split.minsize`
- `mapreduce.tasks.io.sort.mb`
- In addition, set the important join and bucket properties to **true** in **hive-site.xml** or by using the **set** command.

© Hortonworks Inc. 2014



Hive Query Tunings

Hive has a lot of parameters that can be set globally in **hive-site.xml** or at the script level using the **set** command. Here are some of the more important parameters to improve the performance of your Hive queries:

- **mapreduce.input.fileinputformat.split.maxsize** and **mapreduce.input.fileinputformat.split.minsize**: if the min is too large, you will have too few mappers. If the max is too small, you will have too many mappers.
- **mapreduce.tasks.io.sort.mb**: increase this value to avoid disk spills.

Set the following properties all the time:

```
hive.optimize.mapjoin.mapreduce=true;
hive.optimize.bucketmapjoin=true;
hive.optimize.bucketmapjoin.sortedmerge=true;
hive.auto.convert.join=true;
hive.auto.convert.sortmerge.join=true;
hive.auto.convert.sortmerge.join.noconditionaltask=true;
```

When bucketing data, set the following properties:

```
hive.enforce.bucketing=true;  
hive.enforce.sorting=true;
```

IMPORTANT: In HDP, these values are set to true by default. You can verify by viewing the properties in **hive-site.xml**. If a property is not set, just use the **set** command in your Hive script. For example:

```
set hive.optimize.mapjoin.mapreduce=true;
```

Unit 9 Review

1. What is the benefit of performing two **insert** queries in the same Hive command? _____
2. True or False: Hive views are materialized when they are defined. _____
3. True or False: Using indexes in Hive is considered a best practice. _____
4. Suppose an **employees** table has 200 rows, and its **department** column has 15 distinct values. How many rows would be in the result set of the following query? _____

```
from employees
    select fname, lname, MAX(salary)
    over (partition by department);
```

5. Explain what the following query is computing:

```
from employees
    select fname, lname, AVG(salary)
    over (partition by department order by salary
          rows between 5 preceding and current row);
```

6. Which Hive file format provides the best performance? _____
7. What does DAG stand for? _____
8. True or False: Tez improves the performance of all MapReduce jobs, including those written in Pig, Hive and Java.

Lab 9.2: Streaming Data with Hive and Python

Objective:	Use a custom Reducer script to optimize a Hive query.
Location of Files:	/root/labs/Lab9.2
Successful Outcome:	The join query from the previous lab that executed in two MapReduce jobs will now execute in one MapReduce job.
Before You Begin:	Start the Hive CLI.

Step 1: Create the max_ordertotal View

- 1.1.** In the previous lab, you defined a view named **max_ordertotal**. Use the **describe** command to verify:

```
hive> describe max_ordertotal;
OK
maxtotal          int          None
userid            int          None
```

If you do not have this view, define it now as:

```
CREATE VIEW max_ordertotal AS
SELECT max(ordertotal) AS maxtotal, userid
FROM orders GROUP BY userid;
```

Step 2: Think in MapReduce

- 2.1.** Consider the following join statement that you executed in the previous lab:

```
SELECT ordertotal, orders.userid, itemlist
FROM orders
JOIN max_ordertotal ON
max_ordertotal.userid = orders.userid
AND
max_ordertotal.maxtotal = orders.ordertotal;
```

Recall this join statement required two MapReduce jobs to execute.

2.2. What if we could send all the orders by a particular customer to the same reducer? How could we accomplish this? _____

2.3. Suppose we have distributed the records so that we know the same reducer handles all orders from a customer. Then we could sort the orders by **totalorder** descending, and the first order would be their maximum order. Run the following query to understand the logic here:

```
SELECT *
FROM orders
DISTRIBUTE BY userid
SORT BY userid, ordertotal DESC;
```

2.4. Look closely at the output. Each customer's largest order should appear first in his or her respective list of orders. For example, Caitlin F's largest order was \$600 on April 25, 2012:

```
72094 2012-04-25 100 Caitlin F 600 ...
87194 2013-01-05 100 Caitlin F 588 ...
53034 2011-06-11 100 Caitlin F 588 ...
56003 2011-07-30 100 Caitlin F 588 ...
```

The reducer gets all orders from a customer, and the first order the reducer receives is the largest one (which is what we are trying to find!). In the next step, you will use a custom reducer using Python that pulls off this top value.

Step 3: Use a Custom Reducer

3.1. Using a text editor, open the file **max_order.py** in the **Lab9.2** folder.

3.2. Notice this Python script prints the first line that it processes. Then it hangs on to the userid and skips all subsequent lines until the userid changes.

3.3. Copy **max_order.py** into the **/tmp** folder and make it executable:

```
# cp max_order.py /tmp
# chmod +x /tmp/max_order.py
```

3.4. Start the Hive shell.

3.5. Add **max_order.py** as a resource using the **add file** command:

```
hive> add file /tmp/max_order.py;
Added resource: max_order.py
```

NOTE: The **add file** command makes the file available to all mappers and reducers of this Hive query.

3.6. Specify three reducers so we can verify the logic of our query:

```
hive> set mapreduce.job.reduces=3;
```

3.7. Now run the following join query, which uses the Python script as its reducer. You may want to type this in a text file so you can rerun it easier if you have a typo, and make sure you use the proper path to **max_order.py**.

```
from (
    select userid,ordertotal,itemlist
        from orders
        distribute by userid
        sort by userid,ordertotal DESC)
    orders
insert overwrite directory 'maxorders'
reduce userid,ordertotal,itemlist
using 'max_order.py';
```

The query should execute a single MapReduce job this time, and you should also notice three reducers.

Step 4: View the Results

4.1. From the command line, list the contents of the **maxorders** folder in HDFS. You should see three files, one from each reducer:

```
# hadoop fs -ls maxorders
Found 3 items
root hdfs      3611  maxorders/000000_0
root hdfs      3708  maxorders/000001_0
root hdfs      3714  maxorders/000002_0
```

4.2. View the contents of one of the files:

```
# hadoop fs -cat maxorders/000000_0
...
```

```
90588 Boots,Grill,Spark Plugs,Vacuum,Coffee Maker,DVD,2-Way  
Radio,Dolls,Games,DVD,pillows,Pants  
93600 Dishwasher,Table,Grill,DVD,DVD,Keychain,Dryer,  
Washer & Dryer,Grill,Coffee Maker,pillows  
96600 Table,Jeans,Washer,Wrench Set,Grill,Color Laser  
Printer,Dryer,Air Compressor,DVD,Dolls,2-Way Radio,Sweater  
99600 Washer,Cookware,Vacuum,Freezer,2-Way  
Radio,Bicycle,Washer & Dryer,Coffee Maker,Refrigerator,  
DVD,Boots,DVD
```

The output shows the userid, ordertotal and itemlist of the largest order placed by each customer.

RESULT: You used a custom reducer (a Python script) to modify a Hive query that originally took two MapReduce jobs to execute so that it can now be executed in a single MapReduce job. You also learned how to assign a custom reducer (or mapper) to a Hive query.

ANSWERS:

2.2: Use the **DISTRIBUTE BY** clause and distribute the records by the **userid** column.

Unit 10: Hadoop 2.0 and YARN

Topics covered:

- What is HDFS Federation?
- Multiple Federated NameNodes
- Multiple Namespaces
- Overview of HDFS High Availability
- Quorum Journal Manager
- Configuring Automatic Failover
- What is YARN?
- Open-source YARN Use Cases
- The Components of YARN
- Lifecycle of a YARN Application
- A Cluster View Example
- Lab 10.1: Running a YARN Application

What is HDFS Federation?

- According to Webster's: a **federation** is an organization or group within which smaller divisions have some degree of internal autonomy
- **HDFS Federation** refers to the ability of NameNodes to work independently of each other

© Hortonworks Inc. 2013

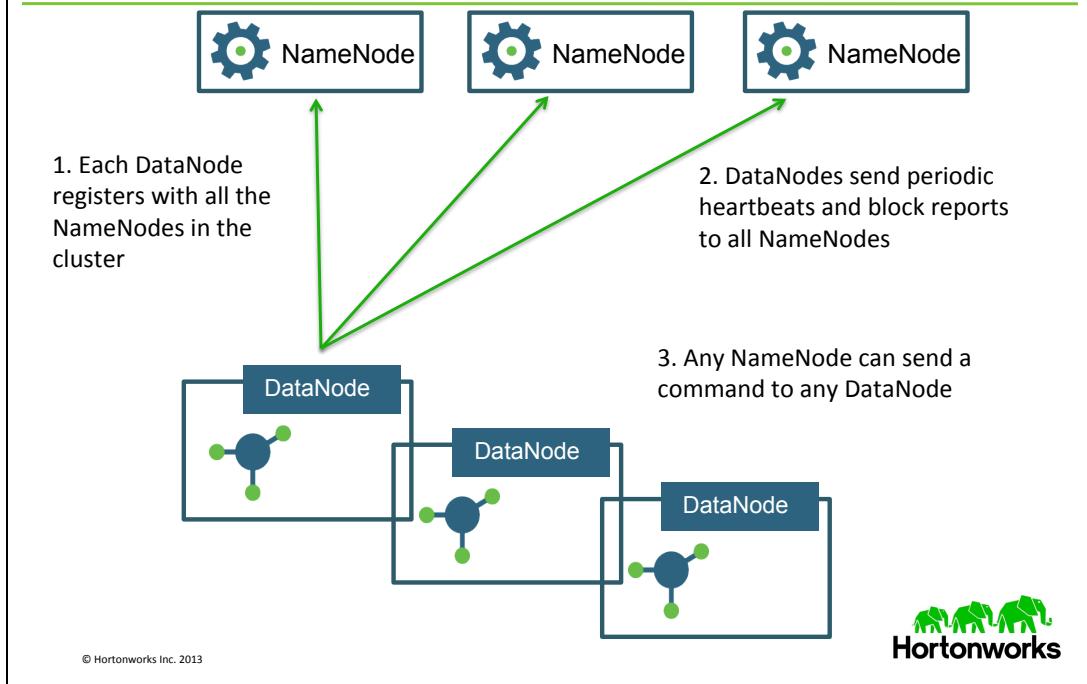


What is HDFS Federation?

Hadoop 2.0 introduces a failover and scaling mechanism for the NameNode referred to as *HDFS Federation*. As opposed to a single NameNode (which was used in Hadoop 1.x), the new Hadoop infrastructure provides for multiple NameNodes that run independently of each other, providing:

- **Scalability:** NameNodes can now scale horizontally, allowing you to improve the performance of NameNode tasks by distributing reads and writes across a cluster of NameNodes.
- **Namespaces:** the ability to define multiple Namespaces allows for the organizing and separating of your big data.

Multiple Federated NameNodes



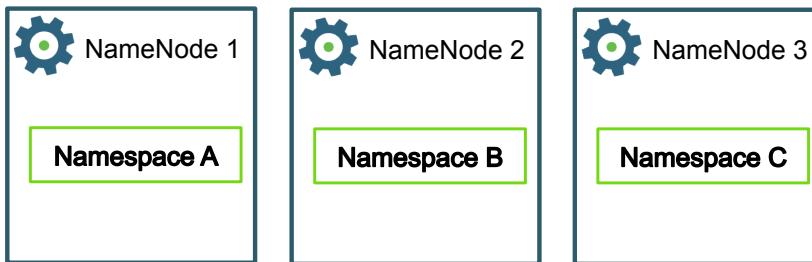
Multiple Federated NameNodes

The NameNodes are federated: that is, the NameNodes are independent and don't require coordination with each other.

The DataNodes are used as common storage for blocks by all the NameNodes. The NameNodes and DataNodes communicate as follows:

1. Each DataNode registers with all the NameNodes in the cluster.
2. DataNodes send periodic heartbeats and block reports to the NameNodes.
3. NameNodes send commands to the DataNodes.

Multiple Namespaces



- Files and directories belong to a Namespace
- Prior versions of Hadoop only had a single Namespace
- Hadoop 2.x allows for multiple Namespaces
- A NameNode manages a single Namespace Volume



© Hortonworks Inc. 2013

Multiple Namespaces

Benefits of multiple Namespaces include:

- **Scalability:** having multiple, independent Namespaces is what makes scaling possible in Hadoop 2.0.
- **File Management:** You can now associate your Big Data with a Namespace, making it easier to manage and maintain files.

NOTE: A NameNode can only define one namespace.

Overview of HDFS HA

- In prior versions of Hadoop, the NameNode was a single point of failure that required additional tools to achieve HA
- In Hadoop 2.0, NameNode HA is now achieved using the built-in Quorum Journal Manager framework

© Hortonworks Inc. 2014



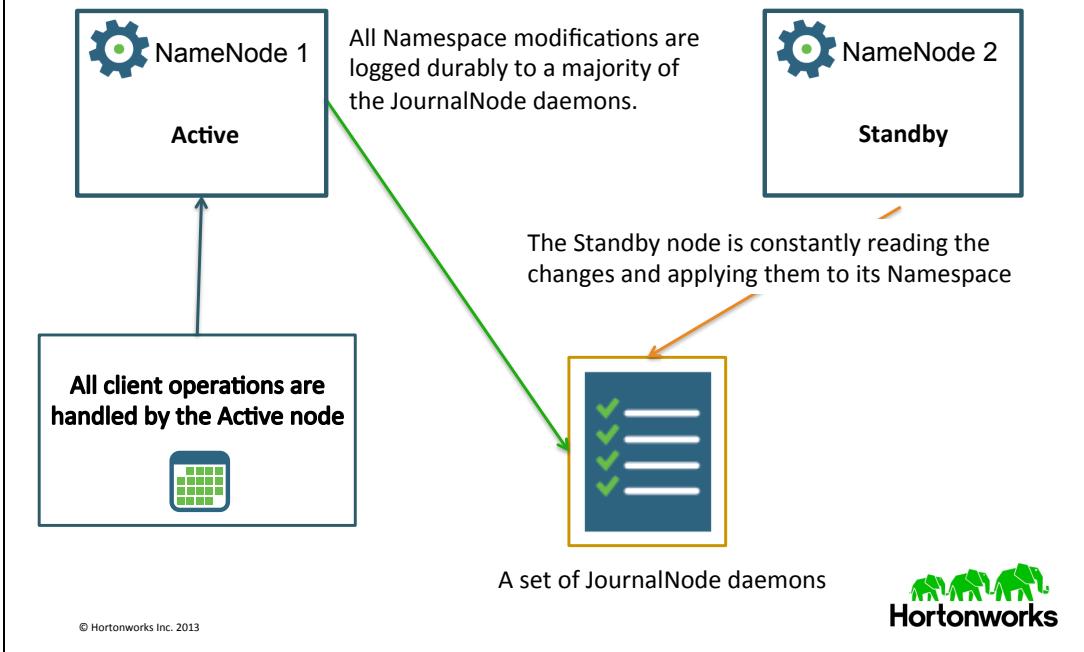
Overview of HDFS High Availability

Prior to Hadoop 2.0, the NameNode was a single point of failure in an HDFS cluster. Each cluster had a single NameNode, and if that machine or process became unavailable, the cluster as a whole would be unavailable until the NameNode was either restarted or brought up on a separate machine.

The HDFS High Availability (HA) feature addresses this issue by providing the option of running two redundant NameNodes in the same cluster in an Active/Passive configuration with a *hot standby*. This allows a fast failover to a new NameNode in the case that a machine crashes, or a graceful administrator-initiated failover for the purpose of planned maintenance.

You can now achieve NameNode HA by configuring your cluster to use the Quorum Journal Manager (QJM), which we will discuss next.

Quorum Journal Manager



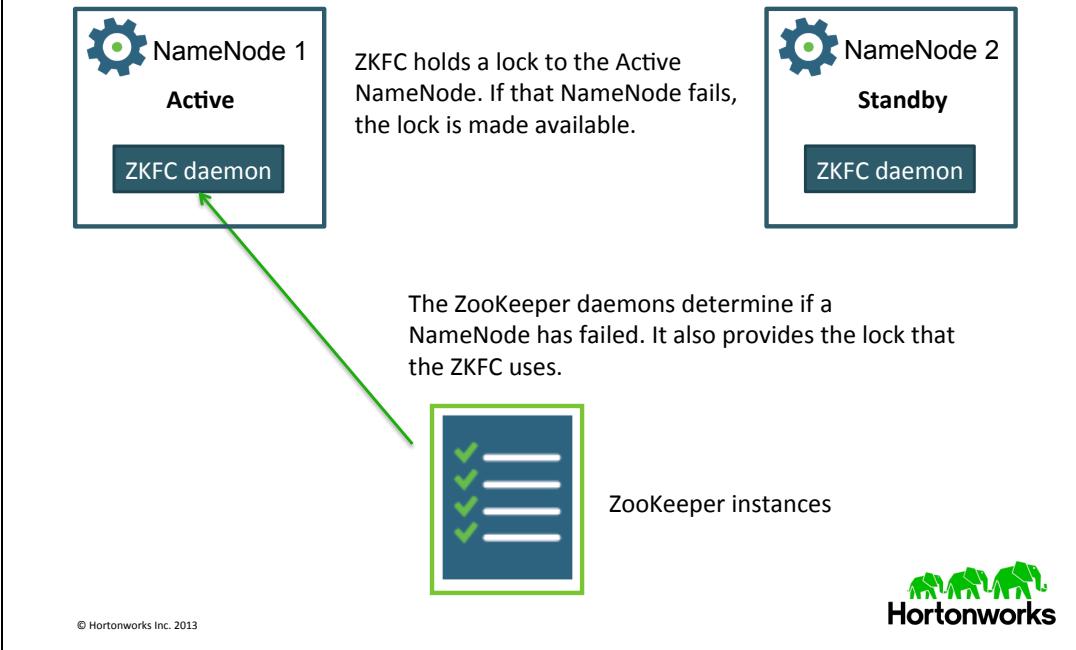
Quorum Journal Manager

Two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an **Active** state, and the other is in a **Standby** state. The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary.

- Both nodes communicate with a group of separate daemons called **JournalNodes**.
- All Namespace modifications are logged durably to a majority of the JournalNode daemons (hence the name *quorum*).
- As the Standby Node sees the edits in the JournalNodes, it applies them to its own namespace.

NOTE: In the event of a failover, the Standby must read all of the edits from the JournalNodes before promoting itself to the Active state. This ensures that the namespace state is fully synchronized before a failover occurs.

Configuring Automatic Failover



Configuring Automatic Failover

Up to this point, you have a Quorum Journal Manager, but note that it requires manual failover. If you want your HA NameNodes to failover automatically, you need to configure ZooKeeper.

More specifically, you need the following within your cluster:

- **ZooKeeper**: an odd number of ZooKeeper daemons that monitor when a NameNode fails
- **ZKFailoverController (ZKFC)**: a new component that is a ZooKeeper client that monitors and manages the state of a NameNode.

Each of the machines that runs a NameNode also runs a ZKFC. The ZKFC pings its local NameNode on a periodic basis with a health-check command. So long as the NameNode responds in a timely fashion with a healthy status, the ZKFC considers the node healthy. If the NameNode has crashed, frozen, or otherwise entered an unhealthy state, the ZKFC will mark it as unhealthy.

What is YARN?

- YARN = Yet Another Resource Negotiator
- YARN splits up the functionality of the JobTracker in Hadoop 1.x into two separate processes:
 - **ResourceManager**: for allocating resources and scheduling applications
 - **ApplicationMaster**: for executing applications and providing failover

© Hortonworks Inc. 2013



What is YARN?

YARN takes Hadoop beyond just MapReduce for data processing. You will still be able to execute MapReduce jobs across your Hadoop cluster, but YARN provides a generic framework that allows for any type of application to execute on the big data across your clusters.

Open-source YARN Use Cases

- **Tez:** improves the execution of MapReduce jobs
- **HOYA:** HBase on YARN
- **Storm and Apache S4:** for real-time computing
- **Spark:** a MapReduce-like cluster computing framework designed for low-latency iterative jobs and interactive use from an interpreter
- **Open MPI:** a high performance Message Passing Library that implements MPI-2
- **Apache Giraph:** a graph processing platform

© Hortonworks Inc. 2014



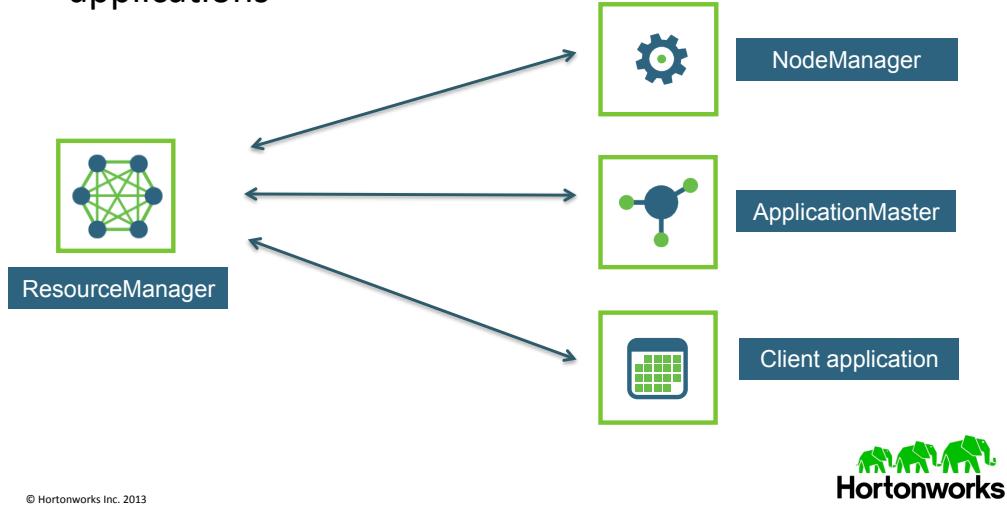
Open-source YARN Use Cases

Now that Hadoop can run applications beyond MapReduce, there are countless possibilities for the type of processing that can be done on data stored in HDFS. Above are some open-source projects that are currently being ported onto YARN for use in Hadoop 2.0.

You can expect other computing frameworks to be developed once YARN becomes prevalent.

The Components of YARN

- The **ResourceManager** communicates with the **NodeManagers**, **ApplicationMasters**, and **Client** applications



The Components of YARN

YARN consists of the following main components:

- ResourceManager
- NodeManager
- ApplicationsMaster

The ResourceManager typically runs on its own machine and is responsible for scheduling and allocating resources. The two main components of the ResourceManager are:

- Scheduler
- Applications Manager (AsM)

The ResourceManager is the central controlling authority for resource management and makes allocation decisions:

- It has a pluggable scheduler that allows for different algorithms (such as capacity and fair scheduling) to be used as necessary.

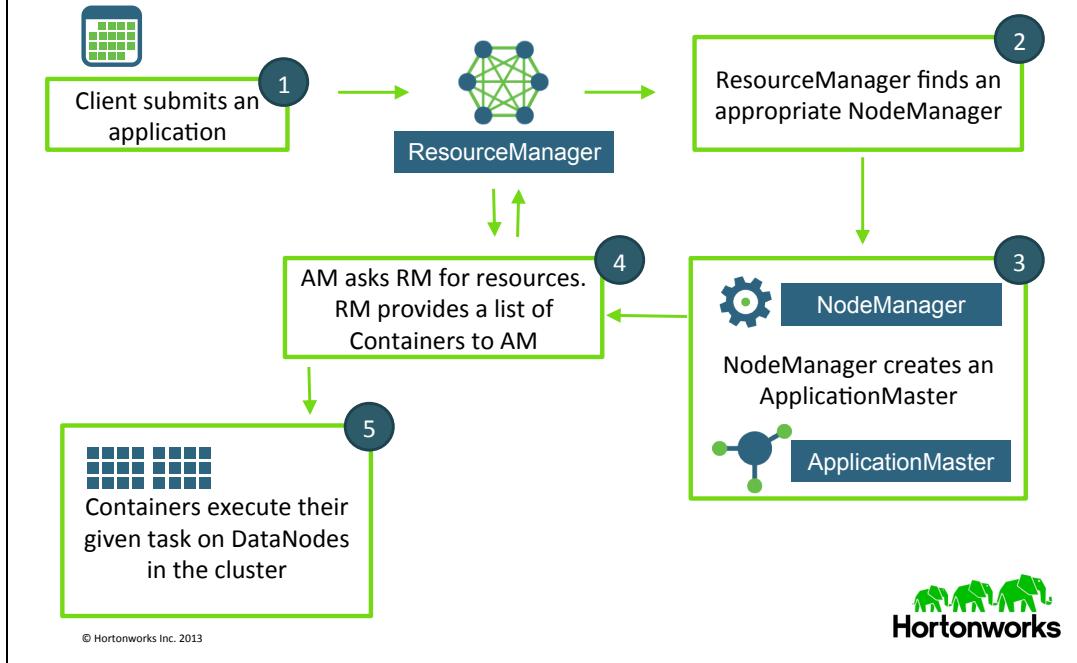
- It tries to optimize the cluster (i.e. use all resources all the time) based on the constraints of the scheduler.

NOTE: If you are familiar with Hadoop 1.x, note that YARN splits up the functionality of the JobTracker into two separate processes:

1. **ResourceManager**: a daemon process that allocates cluster resources to applications.
2. **ApplicationMaster**: a per-application process that provides the runtime for executing applications

The ResourceManager allocates resources for applications, but does not manage the lifecycle of applications. Instead, applications are managed by an ApplicationMaster that runs on a node in the cluster. Each application running in the cluster requires its own ApplicationMaster.

Lifecycle of a YARN Application

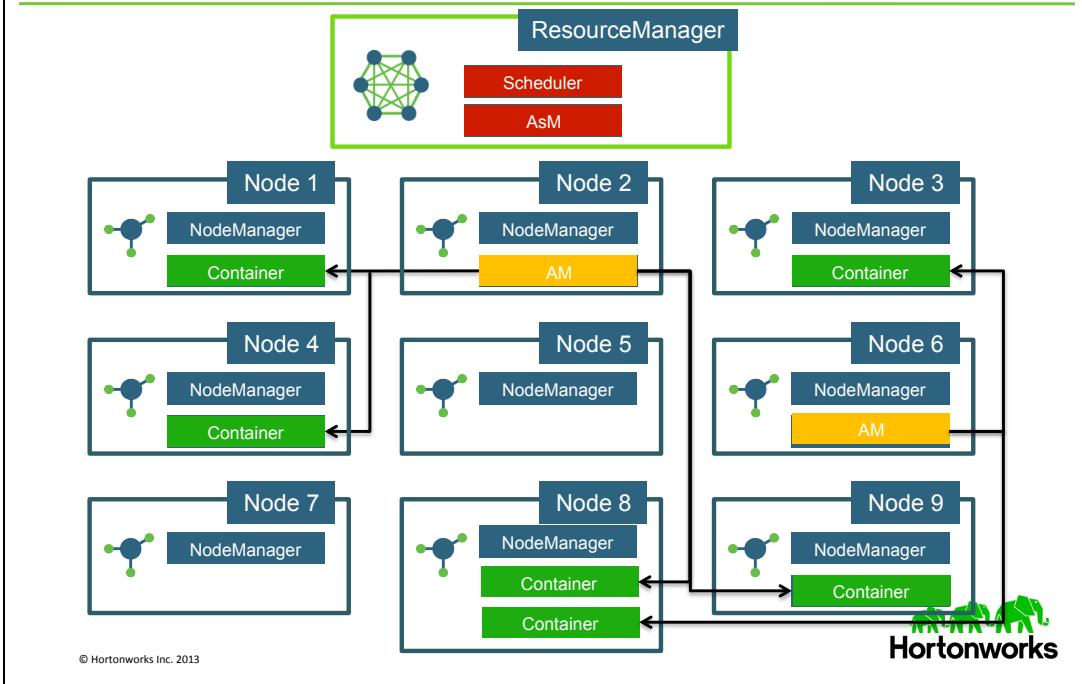


Lifecycle of a YARN Application

1. It all starts with a client submitting a new Application Request to the Resource Manager (RM).
2. The ApplicationsManager (AsM) finds an available DataNode on the cluster that is not too busy.
3. That node's NodeManager (NM) creates an instance of the ApplicationMaster (AM).
4. The AM then sends a request to the RM, asking for specific resources like memory and CPU requirements. The RM replies with a list of Containers, which includes the specific DataNodes to start the Containers on.
5. The AM starts a Container on each DataNode as instructed by the RM. The Container performs a task, as directed by the AM.

As the tasks are being performed by the Containers, the client application can request status updates directly from the ApplicationMaster.

A Cluster View Example



A Cluster View Example

Answer the following questions:

1. How many applications are running on the cluster above? _____
2. How many containers are being used by the application controlled by the AM on Node 2? _____
3. Node 8 appears to have two Containers running on it. Is this allowed in YARN?

4. Is it possible that a Container could be executed on the same node as its corresponding AM? _____

Unit 10 Review

1. True or False: A NameNode can contain multiple namespaces. _____

2. What is the key benefit of the new YARN framework?

3. What are the 3 main components of YARN? _____

4. What happens if a Container fails to complete its task in a YARN application?

Lab 10.1: Running a YARN Application

Objective:	To run a YARN application.
Location of Files:	/root/labs/Lab10.1
Successful Outcome:	You will have executed the Distributed Shell YARN application.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Perform the following steps:

Step 1: Run a DistributedShell Application

1.1. Change directories to **Lab10.1**.

1.2. View the contents of the **simpleyarn.sh** file, which contains a command for running the sample YARN application that ships with HDP 2.0:

```
# more simpleyarn.sh

yarn jar /usr/lib/hadoop-yarn/hadoop-yarn-applications-
distributedshell-2.2.0.2.0.6.0-76.jar
org.apache.hadoop.yarn.applications.distributedshell.Client
-jar /usr/lib/hadoop-yarn/hadoop-yarn-applications-
distributedshell-2.2.0.2.0.6.0-76.jar -shell_command cal
```

The Distributed Shell command allows you to run a script or shell command on your cluster. The example above runs the Unix “**cal**” command, which displays a text calendar.

1.3. Run the **simpleyarn.sh** script:

```
# ./simpleyarn.sh
```

Step 2: Verify the Result

- 2.1. From the command prompt, enter the following command:

```
# yarn application -list -appStates FINISHED
```

You should see a list of recent applications executed, including the DistributedShell command that you just ran:

```
application_1378331467073_0004 DistributedShell      YARN  
yarn      default      FINISHED      SUCCEEDED      100%
```

- 2.2. Copy-and-paste the application ID of your DistributedShell command and check its status using the following command (but replacing the ID shown here with your actual application ID):

```
# yarn application -status application_1378331467073_0004
```

Notice the details of the job are displayed. This was a simple application, so there is not a lot of information to analyze:

```
Application Report :  
  Application-Id : application_1389172320052_0001  
  Application-Name : DistributedShell  
  Application-Type : YARN  
  User : yarn  
  Queue : default  
  Start-Time : 1389177296819  
  Finish-Time : 1389177305487  
  Progress : 100%  
  State : FINISHED  
  Final-State : SUCCEEDED  
  Tracking-URL : N/A  
  RPC Port : -1  
  AM Host : sandbox.hortonworks.com/192.168.215.139
```

NOTE: The **yarn application** command also has a **-kill** option (followed by the application's ID) that kills a running YARN job. This is a great tool when you have submitted a job and then need to stop it before it runs to completion.

Step 3: View the Log File

- 3.1.** Enter the following command to view the output for this YARN application that you just executed:

```
# yarn logs -applicationId application_1389172320052_0001
```

Near the top of the log file, you should see a text calendar of the current month. For example:

```
LogType: stdout
LogLength: 145
Log Contents:
    January 2014
Su Mo Tu We Th Fr Sa
      1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Step 4: OPTIONAL: Run the Job in Two Containers

- 4.1.** The DistributedShell application allows you to specify how many containers that the ApplicationMaster uses. Add the following arguments to the end of the command in **simpleyarn.sh**:

```
-num_containers 2 -container_memory 20
```

- 4.2.** Now view the aggregated log file:

```
# yarn logs -applicationId application_id
```

You should see two calendars this time - one from each container. Notice this also demonstrates how the log output from multiple containers is aggregated into a single, convenient log file.

RESULT: In this lab you ran a simple YARN application called the DistributedShell (that ships with HDP 2.0). You also saw how to view the output of the aggregated log file using the **yarn logs** command.

Appendix A: Defining Workflow with Oozie

Topics covered:

- Overview of Oozie
- Defining an Oozie Workflow
- Pig Actions
- Hive Actions
- MapReduce Actions
- Submitting a Workflow Job
- Making Decisions
- Defining an Oozie Coordinator Job
- Schedule a Job Based on Time
- Schedule a Job Based on Data Availability
- Lab: Defining an Oozie Workflow

Overview of Oozie

- Oozie has two main capabilities:
 - **Oozie Workflow:** a collection of actions
 - **Oozie Coordinator:** a recurring workflow

© Hortonworks Inc. 2012

Overview of Oozie

Oozie is an open-source Apache project that provides a framework for coordinating and scheduling Hadoop jobs. Oozie is not restricted to just MapReduce jobs; you can use Oozie to schedule Pig, Hive, Sqoop, Streaming jobs, and even Java programs.

Oozie has two main capabilities:

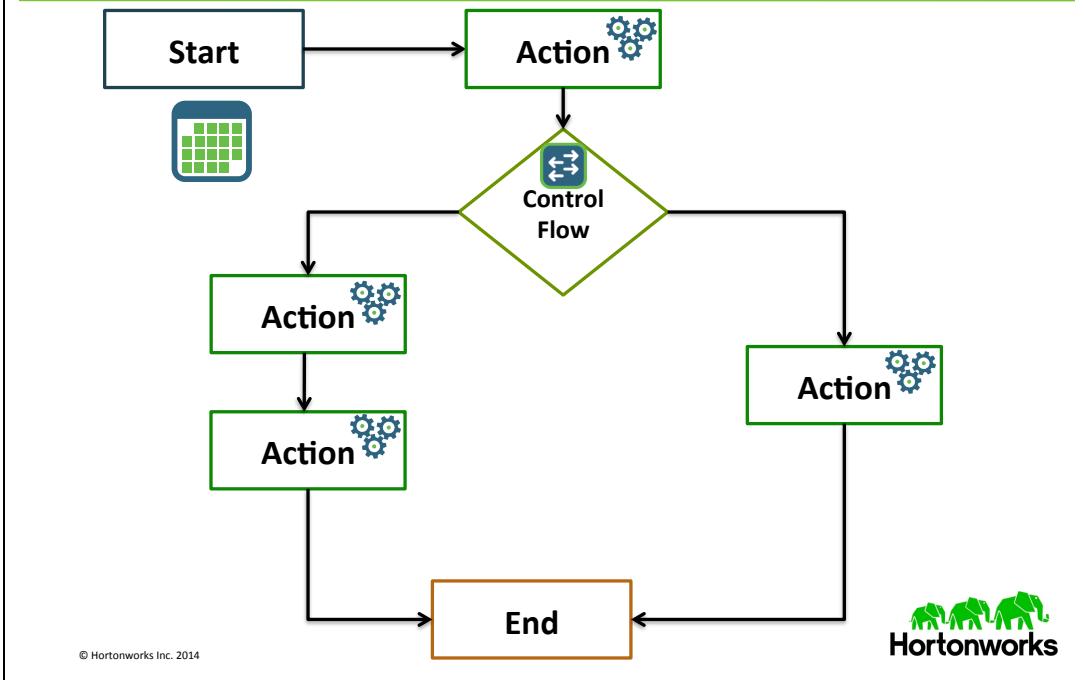
1. **Oozie Workflow:** a collection of actions (defined in a **workflow.xml** file).
2. **Oozie Coordinator:** a recurring workflow (defined in a **coordinator.xml** file).

Behind-the-scenes, Oozie is a Java web application that runs in a Tomcat instance. You run Oozie as a service, and then start workflows using the **oozie** command.

We will now discuss the details of defining an Oozie **workflow.xml** file.

REFERENCE: For more information on the Apache Oozie project, visit their website at <http://oozie.apache.org/>.

Defining an Oozie Workflow



Defining an Oozie Workflow

An Oozie workflow consists of a **workflow.xml** file and the necessary files required by the workflow. The workflow is put into HDFS with the following directory structure:

```
/appdir/workflow.xml  
/appdir/config-default.xml  
/appdir/lib/files.jar
```

- The **config-default.xml** file is optional and contains properties shared by all workflows.
- Each workflow can also have a **job.properties** file (not put into HDFS) for job-specific properties.

As you will soon discover, most of your time spent defining an Oozie workflow is in writing **workflow.xml**. A workflow definition consists of two main entries:

- **Control flow nodes:** for determining the execution path.
- **Action nodes:** for executing a job or task.

Pig Actions

The **pig** action starts a Pig job. The workflow job will wait until the **pig** job completes before continuing to the next action. Here is an example of a simple workflow that only contains a single Pig script as one of its actions:

```
<workflow-app xmlns="uri:oozie:workflow:0.2"
               name="whitehouse-workflow">

    <start to="transform_whitehouse_visitors"/>

    <action name="transform_whitehouse_visitors">
        <pig>
            <job-tracker>${resourceManager}</job-tracker>
            <name-node>${nameNode}</name-node>
            <prepare>
                <delete path="wh_visits"/>
            </prepare>
            <script>whitehouse.pig</script>
        </pig>
        <ok to="end"/>
        <error to="fail"/>
    </action>
    <kill name="fail">
        <message>Job failed, error
            message[$wf:errorMessage(wf:lastErrorNode())]
        </message>
    </kill>
    <end name="end"/>
</workflow-app>
```

- Every workflow must define a `<start>` and `<end>`.
- This workflow has one action named `transform_whitehouse_visitors`.
- Notice a workflow looks almost identical to a `run` method of a MapReduce job - except the job properties are specified in XML.
- The `<delete>` function is a convenient way to delete an existing output folder.
- The `<ok>` tag determines where flow should go if the job completes successfully. The `<error>` tag defines where to go if the job fails.
- Parameters use the `${ }` syntax and represent values defined outside of **workflow.xml**. For example, `${resourceManager}` is the server name and port number where the ResourceManager is running. Instead of hard-coding this value, you define it in an external properties file (the **job.properties** file).

- The Oozie framework provides functions also, like `wf:user()` which returns the name of the user running the job, and `wf:lastErrorNode()` which returns the DataNode where the most recent error occurred on. View the Oozie Documentation for a complete lis of functions.

Hive Actions

The **hive** action runs a Hive job. It looks similar to a pig action:

```
<action name="find_congress_visits">
    <hive xmlns="uri:oozie:hive-action:0.5">
        <job-tracker>${resourceManager}</job-tracker>
        <name-node>${nameNode}</name-node>
        <prepare>
            <delete path="congress_visits"/>
        </prepare>
        <job-xml>hive-site.xml</job-xml>
        <configuration>
            <property>
                <name>mapreduce.map.output.compress</name>
                <value>true</value>
            </property>
        </configuration>
        <script>congress.hive</script>
    </hive>
    <ok to="end"/>
    <error to="fail"/>
</action>
```

- The **congress.hive** script will execute when this action is executed.
- The **hive-site.xml** file needs to be packaged in the workflow and contain the various properties for connecting to Hive.
- Notice this action compresses the output of the map tasks.

MapReduce Actions

Let's take a look at an example of a **map-reduce** action:

```
<action name="payroll-job">
  <map-reduce>
    <job-tracker>${resourceManager}</job-tracker>
    <name-node>${nameNode}</name-node>
    <prepare>
      <delete
path="${nameNode}/user/${wf:user()}/payroll/result"/>
    </prepare>
    <configuration>
      <property>
        <name>mapreduce.job.queuename</name>
        <value>${queueName}</value>
      </property>
      <property>
        <name>mapred.mapper.new-api</name>
        <value>true</value>
      </property>
      <property>
        <name>mapred.reducer.new-api</name>
        <value>true</value>
      </property>
      <property>
        <name>mapreduce.job.map.class</name>
        <value>payroll.PayrollMapper</value>
      </property>
      <property>
        <name>mapreduce.job.reduce.class</name>
        <value>payroll.PayrollReducer</value>
      </property>
      <property>
        <name>mapreduce.job.inputformat.class</name>
        <value>
          org.apache.hadoop.mapreduce.lib.input.TextInputFormat
        </value>
      </property>
      <property>
        <name>mapreduce.job.outputformat.class</name>
        <value>
          org.apache.hadoop.mapreduce.lib.output.NullOutputFormat
        </value>
      </property>
      <property>
        <name>mapreduce.job.output.key.class</name>
        <value>payroll.EmployeeKey</value>
      </property>
      <property>
```

```

<name>mapreduce.job.output.value.class</name>
<value>payroll.Employee</value>
</property>
<property>
    <name>mapreduce.job.reduces</name>
    <value>20</value>
</property>
<property>
    <name>
        mapreduce.input.fileinputformat.inputdir
        </name>
        <value>
            ${nameNode}/user/${wf:user()}/payroll/input
        </value>
    </property>
    <property>
        <name>
            mapreduce.output.fileoutputformat.outputdir
        </name>
        <value>
            ${nameNode}/user/${wf:user()}/payroll/result</value>
        </property>
        <property>
            <name>taxCode</name>
            <value>${taxCode}</value>
        </property>
    </configuration>
</map-reduce>
<ok to="compute-tax"/>
<error to="fail"/>
</action>

```

- Notice a <map-reduce> job consists of properties you would expect, like the map class, reduce class, input and output formats, number of reduce tasks, etc.

Submitting a Workflow Job

```
oozie.wf.application.path=hdfs://node:8020/path/to/app

#Hadoop ResourceManager
resourceManager=node:8050

#Hadoop fs.default.name
nameNode=hdfs://node:8020/

#Hadoop mapred.queue.name
queueName=default

#Application-specific properties
taxCode=2012
```

© Hortonworks Inc. 2014



Submitting a Workflow Job

Oozie has a command-line tool named **oozie** for submitting and executing workflows. The command looks like:

```
# oozie job -config job.properties -run
```

where `job.properties` contains the properties passed in to the workflow. Note that the workflow is typically deployed in HDFS, and `job.properties` is typically kept on the local filesystem.

Notice the command above does not specify which Oozie Workflow to execute. This is specified by the `oozie.wf.application.path` property:

```
oozie.wf.application.path=hdfs://node:8020/path/to/app
```

Here is an example of a **job.properties** file:

```
oozie.wf.application.path=hdfs://node:8020/path/to/app

#Hadoop ResourceManager
resourceManager=node:8050

#Hadoop fs.default.name
nameNode=hdfs://node:8020/

#Hadoop mapred.queue.name
queueName=default

#Application-specific properties
taxCode=2012
```

- The `resourceManager` property was used in **workflow.xml** for the `<job-tracker>` value.
- Similarly, the `nameNode` property became the `<name-node>` value and the `queueName` property ended up as the value of `mapreduce.job.queuename` in **workflow.xml**.
- You define your application-specific properties in **job.properties** as well.

Making Decisions

```
<start to="action1"/>
<action name="action1">
  <capture-output />
  <ok to="decision1" />
  <error to="fail" />
</action>
<decision name="decision1">
  <switch>
    <case to="action2">
      ${wf:actionData('action1')['key1'] == "value1"}
    </case>
    <case to="action3">
      ${wf:actionData('action1')['key1'] == "value2"}
    </case>
    <default to="fail" />
  </switch>
</decision>
```

© Hortonworks Inc. 2014



Making Decisions

Decisions are made with a `<decision>` control flow node. A `<decision>` node contains a `<switch>` with `<case>` statements, similar to a `switch` in Java:

```
<decision name="decision-name">
  <switch>
    <case to="node-name">
      Boolean_expression
    </case>
    <case to="node-name">
      Boolean_expression
    </case>
    <default to="node-name" />
  </switch>
</decision>
```

Decisions are typically made based on the result of an action. The trick is communicating the result of the action with the Oozie Workflow.

Defining an Oozie Coordinator Job

- An Oozie Coordinator job consists of two files:
 - **coordinator.xml**: the definition of the Coordinator application
 - **coordinator.properties**: for defining the job's properties

© Hortonworks Inc. 2012

Defining an Oozie Coordinator Job

Oozie Coordinator is a component of Oozie that allows you to define jobs that are recurring Oozie Workflows. These recurring jobs can be triggered by two types of events:

- **time**: similar to a cron job.
- **data availability**: the job triggers when a specified directory is created.

An Oozie Coordinator job consists of two files:

1. **coordinator.xml**: the definition of the Coordinator application.
2. **coordinator.properties**: for defining the job's properties.

Schedule a Job Based on Time

```
<coordinator-app name="tf-idf"  
    frequency="1440"  
    start="2014-01-01T00:00Z"  
    end="2014-12-31T00:00Z"  
    timezone="UTC"  
    xmlns="uri:oozie:coordinator:0.1">  
    <action>  
        <workflow>  
            <app-path>  
                hdfs://node:8020/home/train/tfidf/workflow  
            </app-path>  
        </workflow>  
    </action>  
</coordinator-app>
```

© Hortonworks Inc. 2014



Schedule a Job Based on Time

Let's take a look at an example of a **coordinator.xml** file. The following Coordinator is triggered based on time:

```
<coordinator-app name="tf-idf"  
    frequency="1440"  
    start="2013-01-01T00:00Z"  
    end="2013-12-31T00:00Z"  
    timezone="UTC"  
    xmlns="uri:oozie:coordinator:0.1">  
    <action>  
        <workflow>  
            <app-path>  
                hdfs://node:8020/home/train/tfidf/workflow  
            </app-path>  
        </workflow>  
    </action>  
</coordinator-app>
```

- The frequency is in minutes, so this job runs once a day.
- Note the Oozie Coordinator has utility functions (similar to the Oozie Workflow) like \${coord:days(1)} for specifying the frequency in days.

- The job starts at midnight on Jan 1, 2013 and runs every day for a year.
- The <app-path> specifies the job to run, which is an Oozie Workflow job.

You submit an Oozie Coordinator job similar to submitting a Workflow job:

```
# oozie job -config coordinator.properties -run
```

The **coordinator.properties** file contains the path to the coordinator app:

```
oozie.coord.application.path=hdfs://node:8020/path/to/app
```

NOTE: Oozie also supports the scheduling of jobs similar to how cron jobs are scheduled.

Schedule Based on Data Availability

```
<coordinator-app name="file_check" frequency="1440"
start="2012-01-01T00:00Z" end="2015-12-31T00:00Z" timezone="UTC"
xmlns="uri:oozie:coordinator:0.1">
<datasets>
<dataset name="input1">
<uri-template>
    hdfs://node:8020/job/result/
</uri-template>
</dataset>
</datasets>
<action>
<workflow>
<app-path>hdfs://node:8020/myapp/</app-path>
</workflow>
</action>
</coordinator-app>
```

© Hortonworks Inc. 2013



Schedule a Job Based on Data Availability

The following Coordinator application triggers a Workflow job when a directory named `hdfs://node:8020/job/result/` gets created:

```
<coordinator-app name="file_check"
frequency="1440" start="2012-01-01T00:00Z"
end="2015-12-31T00:00Z" timezone="UTC"
xmlns="uri:oozie:coordinator:0.1">
<datasets>
<dataset name="input1">
<uri-template>
    hdfs://node:8020/job/result/
</uri-template>
</dataset>
</datasets>
<action>
<workflow>
<app-path>hdfs://node:8020/myapp/</app-path>
</workflow>
</action>
</coordinator-app>
```

- This Coordinator app is scheduled to run once a day.

- If the folder **hdfs://node:8020/job/result/** exists, the <action> executes, which in this example is an Oozie Workflow deployed in the **hdfs://node:8020/myapp** folder.
- The assumption here is that some MapReduce job executes once a day at an unspecified time. When that job runs, it deletes the `hdfs://node:8020/job/result` directory and then creates a new one, which triggers the Coordinator to run.
- This Coordinator runs once a day, and if **/job/result** exists, then the **/myapp** workflow will execute.

Oozie Review

1. What are the two main capabilities of Oozie? _____

2. What file is required to be a part of an Oozie workflow? _____

3. List three common Oozie workflow actions: _____

4. What two types of events can be used to trigger an Oozie coordinator job?

Lab: Defining an Oozie Workflow

Objective:	Define and run an Oozie workflow.
Location of Files:	/root/labs/Oozie
Successful Outcome:	You will run an Oozie job that executes a Pig script and a Hive script.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: Store the Job Data in HDFS

- 1.1. Make sure you have **whitehouse/visits.txt** in HDFS:

```
# hadoop fs -ls whitehouse
```

If not, the file can be found in the **Lab5.1** folder. The Oozie job assumes there is a file named **visits.txt** in a folder named **whitehouse** in HDFS.

Step 2: Configure Oozie User Permissions

- 2.1. View the file **/etc/hadoop/conf/core-site.xml**:

```
# less /etc/hadoop/conf/core-site.xml
```

2.2. The Oozie workflow you defined is going to be executed by the **root** user, so **root** needs permission to communicate with the Oozie server. Notice all users are added to the **hadoop.proxyuser.oozie.groups** property:

```
<property>
    <name>hadoop.proxyuser.oozie.groups</name>
    <value>*</value>
</property>
```

2.3. The following property was also added to **core-site.xml**, which gives the **root** user permission to connect to Oozie on any host.

```
<property>
    <name>hadoop.proxyuser.root.hosts</name>
    <value>*</value>
</property>
```

2.4. Type “**q**” to close the **core-site.xml** file.

Step 3: Deploy the Oozie Workflow

3.1. View the file **workflow.xml** in the **Oozie** folder.

3.2. How many actions are in this workflow? _____

3.3. Which action will execute first? _____

3.4. If the first action is successful, which action will execute next? _____

3.5. To deploy this workflow, we need a directory in HDFS:

```
# hadoop fs -mkdir congress
```

3.6. Put **congress_visits.hive** and **whitehouse.pig** from the **Oozie** folder into the new **congress** folder in HDFS.

3.7. Also, put **workflow.xml** into the **congress** folder.

3.8. Verify you have three files now in your **congress** folder in HDFS:

```
# hadoop fs -ls congress
Found 3 items
1 root hdfs      419 congress/congress_visits.hive
1 root hdfs      580 congress/whitehouse.pig
1 root hdfs     1618 congress/workflow.xml
```

Step 4: Deploy the Hive Configuration File

4.1. If you look at the Hive action in **workflow.xml**, you will notice that it references a file named **hive-site.xml** within the **<job-xml>** tag. This file represents the settings Oozie needs to connect to your Hive instance, and the file needs to be deployed in HDFS (using a relative path to the workflow directory). Put **hive-site.xml** into the workflow directory:

```
# hadoop fs -put /etc/hive/conf/hive-site.xml congress
```

Step 5: Define the OOZIE_URL Environment Variable

5.1. Although not required, you can simplify **oozie** commands by defining the **OOZIE_URL** environment variable. From the command line, enter the following command:

```
# export OOZIE_URL=http://127.0.0.1:11000/oozie
```

Step 6: Run the Workflow

6.1. Open a command line and change directories to **/root/labs/Oozie**.

6.2. Run the workflow with the following command:

```
# oozie job -config job.properties -run
```

If successful, the job ID should be displayed at the command prompt.

Step 7: Monitor the Workflow

7.1. Point your Web browser to the Oozie Web Console:

```
http://ip_address_of_VM:11000/
```

You should see your Oozie job in the list of Workflow Jobs:

The screenshot shows the Oozie Web Console interface. At the top, there's a navigation bar with tabs for 'Workflow Jobs', 'Coordinator Jobs', 'Bundle Jobs', 'System Info', 'Instrumentation', and 'Settings'. Below the navigation bar, there's a filter section with buttons for 'All Jobs', 'Active Jobs', 'Done Jobs', and 'Custom Filter'. The main area displays a table of workflow jobs. The table has columns for Job Id, Name, Status, Run, User, and Group. There is one entry in the table:

	Job Id	Name	Status	Run	User	Group
1	0000000-130904102944019-oozie...	whitehouse-wor...	RUNNING	0	root	

7.2. Double-click on the Job Id to the Job Info page:

Job (Name: whitehouse-workflow/JobId: 0000000-130904102944019-oozie-oozi-W)

Job Info	Job Definition	Job Configuration	Job Log	Job DAG
Job Id: 0000000-130904102944019-oozie Name: whitehouse-workflow App Path: hdfs://sandbox:8020/user/root/con Run: 0 Status: RUNNING User: root Group: Create Time: Wed, 04 Sep 2013 17:34:36 GMT Nominal Time: Start Time: Wed, 04 Sep 2013 17:34:37 GMT Last Modified: Wed, 04 Sep 2013 17:35:31 GMT End Time:				

Actions

Action Id	Name	Type	Status	Transition	StartTime
1 0000000-130904102944019-oozie-oozi-W@...	:start:	:START:	OK	export_con...	Wed, 04 S
2 0000000-130904102944019-oozie-oozi-W@...	export_con... pig	START_RETRY			Wed, 04 S

7.3. Notice you can view the status of each Action within the workflow.

Step 8: Verify the Results

8.1. Once the Oozie job is completed successfully, start the Hive Shell.

8.2. Run a **select** statement on **congress_visits** and verify the table is populated:

```
hive> select * from congress_visits;
...
WATERS MAXINE          12/8/2010 17:00 POTUS      OEOB
    MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
WATT MEL               12/8/2010 17:00 POTUS      OEOB
    MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
WEGNER DAVID L         12/8/2010 16:46 12/8/2010 17:00 POTUS
    OEOB MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
WILLOUGHBY JEANNE     P   12/8/2010 17:07 12/8/2010 17:00
    POTUS      OEOB MEMBERS OF CONGRESS AND CONGRESSIONAL
    STAFF
WILSON ROLLIE          E   12/8/2010 16:49 12/8/2010 17:00
    POTUS      OEOB MEMBERS OF CONGRESS AND CONGRESSIONAL
    STAFF
YOUNG DON              12/8/2010 17:00 POTUS      OEOB
    MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
MCCONNELL MITCH        12/14/2010 9:00 POTUS      WH
    MEMBER OF CONGRESS MEETING WITH POTUS.
Time taken: 1.082 seconds, Fetched: 102 row(s)
```

RESULT: You have just executed an Oozie workflow that consists of a Pig script followed by a Hive script.

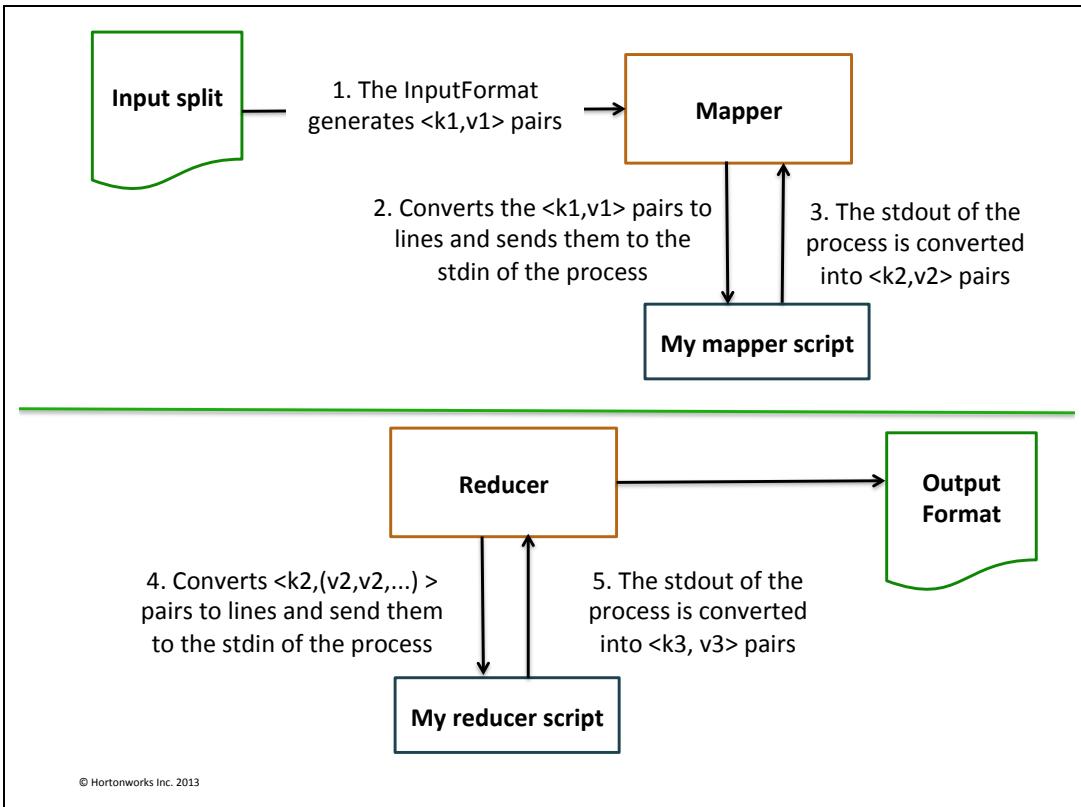
ANSWERS:

Step 3.2: Two

Step 3.3: The Pig action named **export_congress**

Step 3.4: The Hive action named **define_congress_table**

Appendix B: Hadoop Streaming



Hadoop Streaming

Hadoop Streaming is a part of HDP and it allows you to create and run MapReduce jobs with any executable or script as the mapper and/or the reducer. Streaming allows you to take advantage of the benefits of MapReduce while using any scripting language you like.

Here is how Hadoop Streaming works:

1. The MapReduce job starts as any other job, with the input splits sending key/value pairs to a map task.
2. The Streaming mapper converts the key/value pairs into lines of text and sends each line of text to the stdin of the mapper process.
3. The Streaming mapper reads each line of text from the stdout of the process and converts the line to a key/value pair, using a tab as the delimiter between the key and the value.
4. Similarly, the Streaming reducer converts the input key/values pairs into lines of text and sends them to the stdin of the reducer process.
5. The output from stdout of the process is converted to key/value pairs (using a tab as the delimiter) and output by the Streaming reducer.

Running a Hadoop Streaming Job

- A Streaming job is a MapReduce job defined in the `hadoop-streaming.jar` file:

```
hadoop jar $HADOOP_HOME/lib/hadoop-streaming.jar  
-input input_directories  
-output output_directories  
-mapper mapper_script  
-reducer reducer_script
```

© Hortonworks Inc. 2014



Running a Hadoop Streaming Job

The command to run a Hadoop Streaming job looks like the following (entered on a single command line):

```
> hadoop jar hadoop-streaming.jar  
-input input_directories  
-output output_directories  
-mapper mapper_script  
-reducer reducer_script
```

For example, the following command executes a Streaming job that uses `cat` as the mapper and `grep` as its reducer:

```
hadoop jar hadoop-streaming.jar  
-input test/data.txt  
-output streamtest  
-mapper /bin/cat  
-reducer 'grep -i hadoop'
```

Appendix C: Unit Review Answers

Unit 1 Review Answers

1. 1024 petabytes = Exabyte
2. 1024 Exabytes = 1 Zettabyte
And for what it's worth:
1024 Zettabytes = 1 Yottabyte
1024 Yottabytes = 1 Brontobyte
1024 Brontobytes = 1 Geopbyte
3. Variety, Volume and Velocity
4. Clickstream
Sensor and machine data
Location-based (geographic) data
Server logs
Text (web pages, emails, documents, etc)
5. Flume is commonly used for importing Twitter feeds into a Hadoop cluster.
6. HCatalog is designed to easily store and share schemas for your Big Data.
7. HDFS Federation and YARN.

Unit 2 Review Answers

1. NameNode
2. 3
3. False. A file's data in HDFS never passes through the NameNode. Client applications read and write directly from the DataNodes.
4. `dfs.blocksize`
5. The `fsimage_N` and `edits_N` files

6. Recursively lists the contents of **/user/thomas** in HDFS and all of its subfolders.
7. Lists the file and folders in **/user/thomas**, but not recursively. (The files in the subfolders of **/user/thomas** are not listed.)

Unit 3 Review Answers

1. Flume
2. Sqoop
3. The Hadoop client (**hadoop fs -put** command)
4. A Flume agent consists of a source, channel and sink.
5. 4 map tasks by default
6. The **-m** option is for specifying the number of mappers.
7. The **\$CONDITIONS** value is used internally by Sqoop to specify **LIMIT** and **OFFSET** clauses so the data can be split up amongst the map tasks.

Unit 4 Review Answers

1. Map phase, shuffle/sort phase, and reduce phase
2. The pairs coming in to the Reducer will look like **<integer, (string,string,string,...)>**
3. When the map output buffer reaches a threshold, the **<key,value>** pairs are spilled to disk, meaning they are written to a temporary file on the local filesystem.
4. The number of Mappers is determined by the input splits.
5. You get to choose the number of Reducers.
6. False. The keys are sorted, but the values are not.

Unit 5 Review Answers

1. **STORE**, **DUMP** and **ILLUSTRATE** all cause a logical plan to execute.
2. **STORE**
3.

```
prices: {symbol: chararray, date: chararray, price: double, volume: int}
```
4. The result is a collection of bags, with a bag for each distinct symbol. The **A** relation looks like:

```
A: {group: chararray, prices: { (symbol: chararray, date: chararray, price: double, volume: int) }}
```
5. The **B** relation is a projection of the **symbol** and **volume** fields of **prices**. The schema was also changed. **B** looks like:

```
B: {x: chararray, y: int}
```
6. **C** is a projection of **A**. The **group** field is the **symbol** field of **prices**, and the **SUM** function adds up the **volume** field of each group of symbols. The **C** relation looks like:

```
C: {group: chararray, long}
```

The output of **C** looks like:
(XFR, 411900)
7. **D** is a projection of all fields of **prices** between **symbol** and **price**. The **D** relation looks like:

```
D: {symbol: chararray, date: chararray, price: double}
```
8. E = filter prices by volume > 3000;

Unit 6 Review Answers

1. The **ORDER BY** command generate a total ordering, meaning that the records will be sorted across all 3 reducers, with the output of reducer 1 containing the first set of sorted records, reducer 2 containing the second set, and so on.

2. The output of **F** looks like:

```
(-1)  
(400000)  
(-1)  
(3800)  
(-1)  
(-1)
```

3. The **DISTINCT** operator removes duplicate records, but the **prices** relation does not contain any duplicates, so in this example the **G** relation is identical to the **prices** relation.
4. The output of **I** is **(XFR, 45.6)**, which is the sum of the **prices** fields for each record where the **volume** is greater than 3,000.
5. The result is a map-side join, which greatly improves the resulting join operation in MapReduce by limiting network traffic in the shuffle/sort phase to only records that will appear in the result.
6. Filtering limits the number of records, and projecting limits the size of the records, which greatly improves both network traffic and processing time of the resulting MapReduce job.

Unit 7 Review Answers

1. A Hive table consists of a schema stored in the Hive metastore and data stored in HDFS.
2. True. Hive uses an in-memory database called Derby by default, but you can configure Hive to use any SQL database.
3. The data and folders are deleted from HDFS.
4. False. An external table can use an external location, but it can also use the Hive warehouse folder.
5. There are several ways to load data into a Hive table, including manually copying files into the table's folder in HDFS; using the **LOAD DATA** command; and inserting data as the result of a query.

6. Skewed tables make sense when your data is naturally skewed, where a small number of columns contain a disproportionate amount of records.
7. Within `/apps/hive/warehouse/movies` will be subfolders named `/genre=value`. For example, `/genre=scifi`, `/genre=comedy`, `/genre=drama`, etc.
8. The `order by` clause causes the output to be totally ordered by title across all output files.
9. The ngram output from this query is called a trigram, because the result will be sets of 3 words. The 100 argument specifies you want the top 100 trigrams from this dataset.
10. The output of this query is the top 10 words in the dataset that follow the phrase "I liked".

Unit 8 Review Answers

1. In the Hive metastore.
2. Pig, Hive and Java MapReduce programs can all easily use the schemas shared by HCatalog.
3. The `HCatLoader` class; more specifically, `org.apache.hcatalog.pig.HCatLoader`.
4. True. HCatalog is now a part of Hive.

Unit 9 Review Answers

1. Two queries that would normally require two MapReduce jobs can be combined and accomplished in a single MapReduce job.
2. False. Hive views are not materialized until they are used in another query.
3. False. Indexes may cause issues in real-world deployments, and you are better suited using partitioned or skewed tables to improve performance.

4. 200. The **OVER** clause causes the group aggregation to not occur, so each **employees** row will be output, but there will only be 15 **salary** values - the maximum **salary** in each **department**.
5. The result will contain the **fname**, **Iname** and the average **salary** of this employee and the 5 preceding employees whose salaries are less than or equal to the current employee.
6. ORC files are a part of the Stinger Initiative and provide the best performance for Hive queries.
7. DAG = Directed Acyclic Graph. Hive queries are processed into a series of MapReduce jobs that look like a DAG.
8. True. Tez works on any DAG of MapReduce jobs, not just Hive.

Unit 10 Review Answers

1. False. A NameNode can represent only a single namespace.
2. Hadoop jobs are no longer restricted to MapReduce. With YARN, any type of computing paradigm can be implemented to run on Hadoop.
3. ResourceManager, NodeManager and ApplicationMaster
4. It is up to the ApplicationMaster to request a new Container from the ResourceManager and attempt the task again.

Oozie Review Answers

1. Oozie Workflow, for defining Hadoop job workflows; and the Oozie Coordinator, for scheduling recurring workflows.
2. Each Oozie workflow must contain a **workflow.xml** configuration file.
3. <pig>, <hive> and <map-reduce>
4. Time-based, where a job executes at a specific time; or data-based, where a job executes if data is available in a specific location.