



HDP Developer: Apache Pig and Hive

Student Guide



HDP Developer: Apache Pig and Hive

HDP Developer: Apache Pig and Hive

Title HDP Developer: Pig and Hive

Version: GA

Date: March, 2015

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. 2015 All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in
any form by any means electronic, photocopy, recording or otherwise without prior written
permission of Hortonworks.

HDP Developer: Apache Pig and Hive

HDP Developer: Apache Pig and Hive

Table of Contents

Understanding Hadoop	11
Lesson Objectives.....	11
Additional Content.....	11
The Three Vs of Big Data.....	12
Six Key Hadoop Data Types	14
About Use Cases.....	15
Sentiment Use Case.....	15
Geolocation Use Case	18
About Hadoop.....	22
Relational Databases vs. Hadoop	23
About Hadoop 2.....	23
New in Hadoop 2.x	24
The Hadoop Ecosystem.....	25
The Hortonworks Data Platform (HDP)	26
The Path to ROI.....	27
Review Questions.....	28
Lab: Start an HDP 2.1 Cluster	29
Objective: Start an HDP cluster in your VM	29
The Hadoop Distributed File System (HDFS)	31
Lesson Objectives.....	31
Additional Content.....	31
Hadoop vs. RDBMS.....	33
An Example of Disk Read Performance	34
HDFS Components	35
Understanding Block Storage	36
Demonstration: Understanding Block Storage	37
Objective: To understand how data is partitioned into blocks and stored in HDFS	37
The NameNode	38
The DataNodes	40
DataNode Failure.....	41
HDFS Commands	42
Examples of HDFS Commands	43
HDFS File Permissions	44
Review Questions.....	45
Lab: Using HDFS Commands	46
Objective: Become familiar with adding, removing, and viewing files in HDFS.....	46
Inputting Data into HDFS	47
Lesson Objectives.....	47
Additional Content.....	47
The Hadoop Client.....	48
WebHDFS	49
Overview of Flume	50
A Flume Example.....	51
Overview of Sqoop	52

HDP Developer: Apache Pig and Hive

The Sqoop Import Tool	53
Importing a Table	53
Importing Specific Columns	55
Importing from a Query	56
The Sqoop Export Tool	57
Exporting to a Table	58
Review Questions	59
Lab: Importing RDBMS Data into HDFS	60
Objective: Import data from a database into HDFS	60
Lab: Exporting HDFS Data to a RDBMS	60
Objective: Export data from HDFS into a MySQL table using Sqoop	60
The MapReduce Framework	61
Lesson Objectives	61
Additional Content	61
Overview of MapReduce	62
Understanding MapReduce	64
The Key/Value Pairs of MapReduce	67
WordCount in MapReduce	68
Demonstration: Understanding MapReduce	69
Objective: To understand how MapReduce works	69
The Map Phase	70
The Reduce Phase	72
Review Questions	74
Lab: Running a MapReduce Job	75
Objective: Run a Java MapReduce job	75
Introduction to Pig	77
Lesson Objectives	77
Additional Content	77
About Pig	78
Pig Latin	79
The Grunt Shell	79
Demonstration: Understanding Pig	80
Objective: To understand Pig scripts and relations	80
Pig Latin Relation Names	81
Pig Latin Field Names	81
Pig Data Types	82
Pig Complex Types	83
Defining a Schema	84
Lab: Getting Started with Pig	85
Objective: Use Pig to navigate through HDFS and explore a dataset	85
The GROUP Operator	86
GROUP ALL	87
Relations without a Schema	88
The FOREACH GENERATE Operator	89
Specifying Ranges in FOREACH	90
Field Names in a FOREACH	90

HDP Developer: Apache Pig and Hive

FOREACH with Groups.....	91
The FILTER Operator.....	92
The LIMIT Operator	94
Review Questions	95
Lab: Exploring Data with Pig	96
Objective: Use Pig to navigate through HDFS and explore a dataset	96
Advanced Pig Programming	97
Lesson Objectives	97
Additional Content	97
The ORDER BY Operator	98
The CASE Operator	100
Parameter Substitution.....	101
The DISTINCT Operator	102
Using PARALLEL	103
The FLATTEN Operator	104
Lab: Splitting a Dataset	106
Objective: Research the White House visitor data and look for members of Congress.....	106
Nested FOREACH	107
About Joins	108
Performing an Inner Join.....	108
Performing an Outer Join	110
Replicated Joins.....	112
The COGROUP Operator	113
Pig User-Defined Functions	115
A UDF Example.....	116
Invoking a UDF	116
Tips for Optimizing Pig Scripts	117
Lab: Joining Datasets	118
Objective: Join two datasets in Pig.....	118
Lab: Preparing Data for Hive	118
Objective: Transform and export a dataset for use with Hive	118
Overview of the DataFu Library	119
Computing Quantiles.....	120
Demonstration: Computing PageRank	121
Objective: To understand how to use the PageRank UDF in DataFu	121
Review Questions	122
Lab: Analyzing Clickstream Data	123
Objective: Become familiar with using the DataFu library to sessionize clickstream data	123
Lab: Analyzing Stock Market Data using Quantiles	123
Objective: Use DataFu to compute quantiles.....	123
Hive Programming	125
Lesson Objectives	125
Additional Content	125
About Hive	126
Comparing Hive to SQL.....	127
Hive Architecture.....	128

HDP Developer: Apache Pig and Hive

Submitting Hive Queries.....	129
Defining a Hive-Managed Table	130
Defining an External Table	130
Defining a Table LOCATION.....	131
Loading Data into a Hive Table.....	132
Performing Queries.....	133
Lab: Understanding Hive Tables.....	134
Objective: Understand how Hive table data is stored in HDFS.....	134
Hive Partitions	135
Hive Buckets	136
Skewed Tables	137
Demonstration: Understanding Partitions and Skew.....	138
Objective: To understand how Hive partitioning and skewed tables work.....	138
Sorting Data	139
Using Distribute By	139
Storing Results to a File	141
Specifying MapReduce Properties	141
Lab: Analyzing Big Data with Hive.....	142
Objective: Analyze the White House visitor data.....	142
Lab: Understanding MapReduce in Hive	142
Objective: To understand how Hive queries get executed as MapReduce jobs	142
Hive Join Strategies	143
Shuffle Joins.....	143
Map (Broadcast) Joins	144
Sort-Merge-Bucket (SMB) Joins	145
Invoking a Hive UDF	146
Computing ngrams in Hive.....	147
Demonstration: Computing ngrams	148
Objective: To understand how to compute ngrams using Hive.....	148
Review Questions.....	149
Lab: Joining Datasets in Hive.....	150
Objective: Perform a join of two datasets in Hive	150
Lab: Computing ngrams of Emails in Avro Format.....	150
Objective: Use Hive to compute ngrams	150
Using HCatalog.....	151
Lesson Objectives.....	151
Additional Content.....	151
About HCatalog	152
HCatalog in the Ecosystem	153
Defining a New Schema	154
Using HCatLoader with Pig.....	154
Using HCatStorer with Pig	154
The Pig SQL Command	154
Review Questions.....	155
Lab: Using HCatalog with Pig.....	156
Objective: Use HCatalog to provide the schema for a Pig relation	156

HDP Developer: Apache Pig and Hive

Advanced Hive Programming.....	157
Lesson Objectives.....	157
Additional Content.....	157
Performing a Multi-Table/File Insert	158
Understanding Views.....	160
Defining Views.....	161
Using Views.....	161
The TRANSFORM Clause.....	162
The OVER Clause.....	163
Using Windows	164
Hive Analytics Functions.....	165
Lab: Advanced Hive Programming.....	166
Objective: To understand how some of the more advanced features of Hive work.....	166
Hive File Formats	167
Hive SerDes	167
Hive ORC Files.....	168
Computing Table Statistics	169
Hive Cost Based Optimization.....	170
Vectorization	171
Using HiveServer2	172
Understanding Hive on Tez	173
Using Tez for Hive Queries	173
Demonstration: Hive Optimizations.....	174
Objective: To become familiar with some ways to optimize Hive	174
Hive Optimization Tips	175
Hive Query Tunings	176
Review Questions.....	177
Lab: Streaming Data with Hive and Python.....	178
Objective: Use a custom reducer script to optimize a Hive query.....	178
Hadoop 2 and YARN	179
Lesson Objectives.....	179
Additional Content.....	179
About HDFS Federation.....	180
Multiple Federated NameNodes	180
Multiple Namespaces.....	181
Overview of HDFS High Availability.....	181
Quorum Journal Manager	182
Configuring Automatic Failover	183
About YARN	184
Open-source YARN Use Cases	184
The Components of YARN	185
Lifecycle of a YARN Application	186
A Cluster View Example.....	187
Review Questions.....	188
Lab: Running a YARN Application.....	189
Objective: To run a YARN application	189

HDP Developer: Apache Pig and Hive

Defining Workflow with Oozie.....	191
Lesson Objectives.....	191
Additional Content.....	191
Overview of Oozie	192
Defining an Oozie Workflow.....	193
Pig Actions	194
Hive Actions	195
MapReduce Actions	196
Submitting a Workflow Job	198
Fork and Join Nodes	199
Defining an Oozie Coordinator Job.....	200
Schedule a Job Based on Time.....	200
Schedule a Job Based on Data Availability.....	201
Review Questions.....	202
Lab: Defining an Oozie Workflow	203
Objective: Define and run an Oozie workflow	203
Hadoop Streaming.....	205
Lesson Objectives.....	205
Hadoop Streaming	206
Running a Hadoop Streaming Job.....	207
Appendix A: Lesson Review Quiz Answers	209
Understanding Hadoop: Review Answers	209
The Hadoop Distributed File System (HDFS): Review Answers.....	210
Inputting Data into HDFS: Review Answers.....	211
The MapReduce Framework: Review Answers	212
Introduction to Pig: Review Answers	213
Advanced Pig Programming: Review Answers	215
Hive Programming: Review Answers.....	217
Using HCatalog: Review Answers.....	219
Advanced Hive Programming: Review Answers	220
Hadoop 2 and YARN: Review Answers.....	221
Defining Workflow with Oozie: Review Answers.....	222

Understanding Hadoop

Lesson Objectives

This lesson covers an overview of big data, Hadoop, and the Hortonworks Data Platform.

After completing this lesson, students should be able to:

- Describe the Three Vs of Big Data
- Describe the Six Key Hadoop Data Types
- Describe Use Cases
- Describe Hadoop
- Describe the Hortonworks Data Platform (HDP)
- Describe the Path to ROI

Additional Content

- Quiz: Lesson Review
- Lab: Start an HDP 2.1 Cluster

The Three Vs of Big Data

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

The characteristics of big data are often defined as the three Vs:

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: 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 an RDBMS and used in online transactions is not big data. However, the three terabytes of web-log files from millions of visits to your website over the last ten years is probably big data.

HDP Developer: Apache Pig and Hive

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

Six 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

6 Key Hadoop Data Types

The type of big data that ends up in Hadoop typically fits into one of the following categories:

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

About Use Cases

Sentiment Use Case



- 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 Cases

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.

HDP Developer: Apache Pig and Hive



Flume was used to input the Twitter feeds into Hadoop. Once the data was in HDFS, HCatalog was used to define a schema:

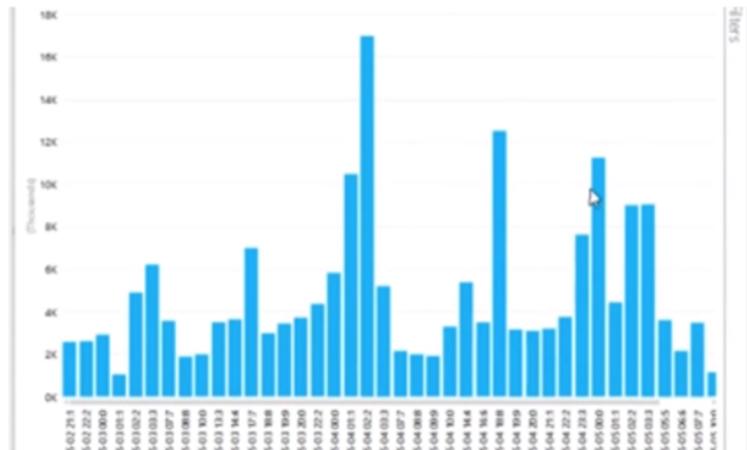
```
CREATE EXTERNAL TABLE tweets_raw (
    id BIGINT,
    created_at STRING,
    source STRING,
    favorited BOOLEAN,
    retweet_count INT,
    text STRING
)
```

There were a lot of Hive queries used to create the final result. Hive looks like SQL. For example:

```
CREATE TABLE tweetsbi
STORED AS RCFile
AS
SELECT
  t.*,
  case s.sentiment
    when 'positive' then 2
    when 'neutral' then 1
    when 'negative' then 0
  end as sentiment
FROM tweets_clean t LEFT OUTER JOIN tweets_sentiment s on t.id = s.id;
```

HDP Developer: Apache Pig and Hive

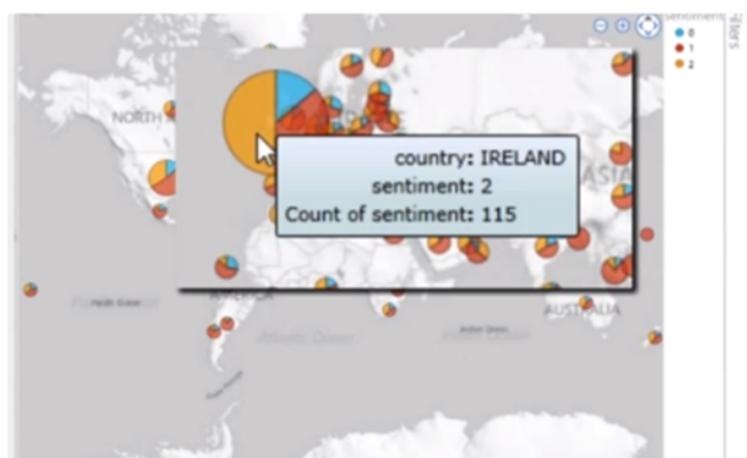
The result was imported into Power View. The following graph shows the volume of tweets over the opening weekend of the movie:



Notice a large spike in tweets around the Thursday midnight opening, and spikes around the Friday evening, Saturday afternoon and Saturday evening showings.

View Spikes in Tweet Volume

The sentiment of the tweets was graphed by country:



Viewing the tweets on a map shows the sentiment of the movie by country. For example, Ireland had 50% positive tweets, while 67% of tweets from Mexico were neutral.

View Sentiment by Country



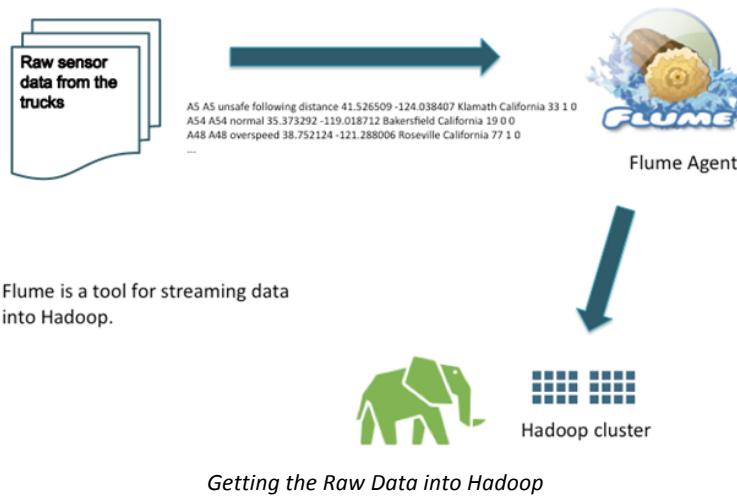
Reference: Visit <http://hortonworks.com/use-cases/sentiment-analysis-hadoop-example/> to watch a video that walks through the steps above.

HDP Developer: Apache Pig and Hive

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, 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.

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



Flume was used to input the data into HDFS. The data collected from the trucks looks like:

```
truckid  
driverid  
event  
latitude  
longitude  
city  
state  
velocity  
event_indicator (0 or 1)  
idling_indicator (0 or 1)
```

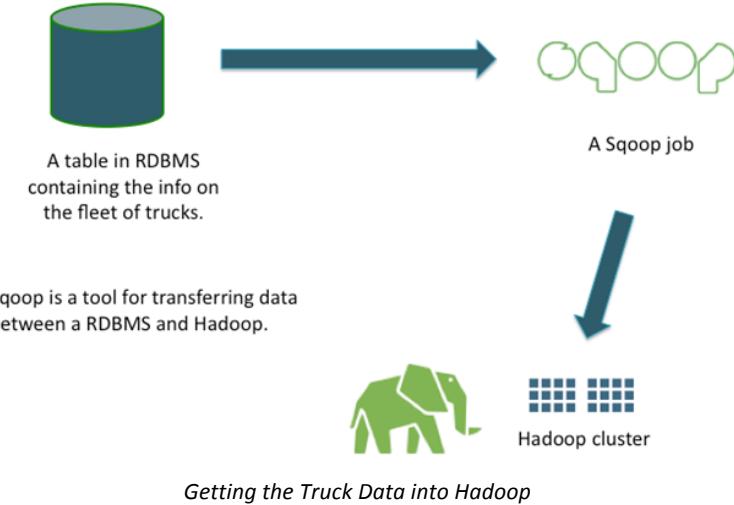
HDP Developer: Apache Pig and Hive

For example:

```
A5 A5 unsafe following distance 41.526509 -124.038407 Klamath California 33 1 0  
A54 A54 normal 35.373292 -119.018712 Bakersfield California 19 0 0  
A48 A48 overspeed 38.752124 -121.288006 Roseville California 77 1 0
```

The details of the trucks and drivers are stored in a relational database. Sqoop was used to import the relational data into HDFS, and HCatalog was used to define schemas for this data:

```
create table trucks (  
  driverid string,  
  truckid string,  
  model string,  
  monthyear_miles int,  
  monthyear_gas int,  
  total_miles int,  
  total_gas double,  
  mileage double  
);
```



HDP Developer: Apache Pig and Hive

Lots of Hive queries were used to evaluate the data. Hive looks like SQL:

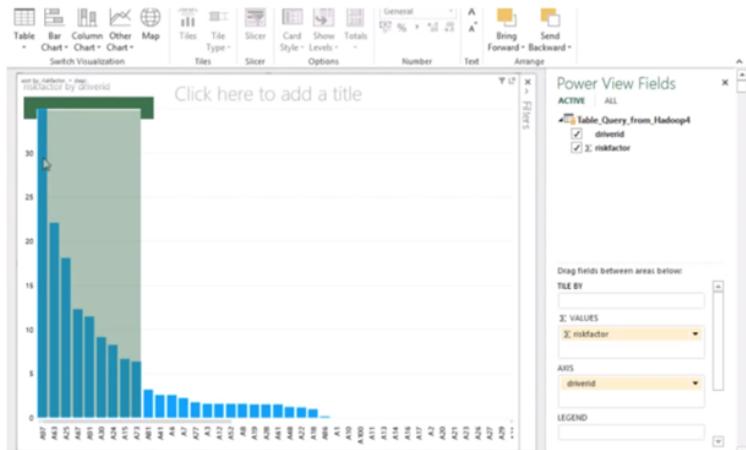
```
CREATE TABLE truck_mileage AS
  SELECT truckid, rdate, miles, gas,
         miles/gas mpg
    FROM trucks
   LATERAL VIEW stack(54,
'jun13', jun13_miles, jun13_gas, 'may13', may13_miles, may13_gas, 'apr13', apr13_miles, apr13_gas, ...
 ) dummyalias AS rdate, miles, gas;
```

Pig is a scripting language that has an SQL-like look to it. Pig was used to compute the risk factor of each driver:

```
a = LOAD 'events' using org.apache.hcatalog.pig.HCatLoader();
b = filter a by event != 'Normal';
c = foreach b
      generate driverid, event, (int) '1' as occurrence;
d = group c by driverid;
e = foreach d generate group as driverid,
      SUM(c.occurrence) as t_occ;
f = LOAD 'trucks' using org.apache.hcatalog.pig.HCatLoader();
g = foreach f generate driverid,
      ((int) apr09_miles + (int) apr10_miles) as t_miles;
join_d = join e by (driverid), g by (driverid);
final_data = foreach join_d generate
              $0 as driverid, (float) $1/$3*1000 as riskfactor;
store final_data into 'riskfactor'
      using org.apache.hcatalog.pig.HCatStorer();
```

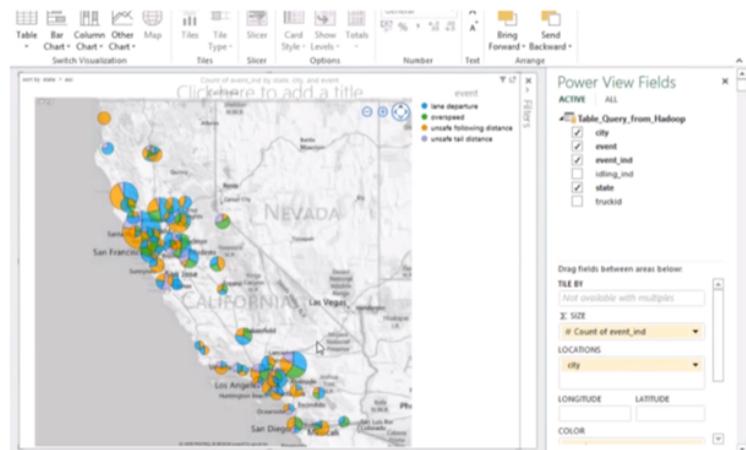
HDP Developer: Apache Pig and Hive

Using Power View, the risks were displayed in a bar graph:



Risk Factors Viewed in a Graph

The risk factors were also plotted on a map:



Risk Factors Viewed on a Map



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

About 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. 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	To scale massively, it is important things are as simple as possible, 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
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
Variety of processing engines	Big data on Hadoop can be processed using multiple different processing engines, including Tez, Spark and Storm.
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

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

HDP Developer: Apache Pig and Hive

Relational Databases vs. Hadoop

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

Relational Database 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.

About Hadoop 2

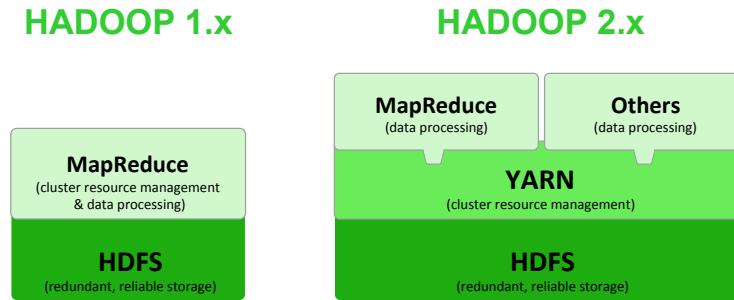
Hadoop 2.x refers to the next generation of Hadoop. As expected, the Hadoop framework has grown to meet the demands of its own popularity and usage, and 2.x reflects the natural maturing of the open-source project.

The Apache Hadoop 2.x 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

HDP Developer: Apache Pig and Hive

New in Hadoop 2.x



What's New in Hadoop 2.x?

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

HDFS HA and Federation

Provides a name service that is scalable and reliable

YARN

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. Hadoop 2.x provides an `HA NameNode`.

Federation provides the ability to configure multiple `NameNodes`, and therefore multiple namespaces, to provide a distribution of workloads 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.

HDP Developer: Apache Pig and Hive

The Hadoop Ecosystem



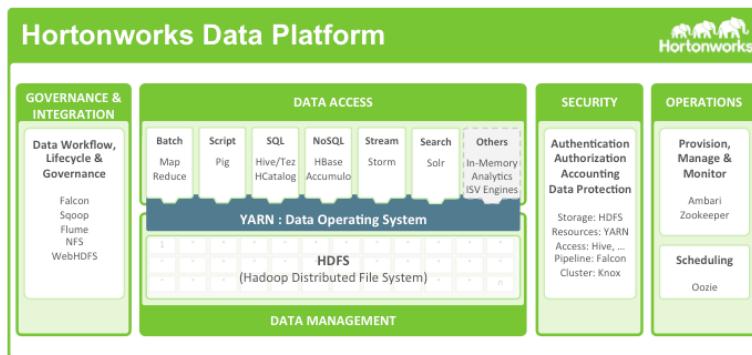
The Hadoop Ecosystem

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
Accumulo	A robust, scalable, high-performance data-storage and retrieval system, built on Hadoop and Zookeeper
Ambari	Provisioning, managing, and monitoring Apache Hadoop clusters
Sqoop	Efficiently transfers bulk data between Hadoop and RDBMS
Falcon	A data processing and management solution, designed for pipeline coordination, lifecycle management, and data discovery
Oozie	A workflow scheduler system to manage Apache Hadoop jobs
Solr	A standalone enterprise search server with a REST-like API
Flume	Efficiently collects, aggregates, and moves log data
ZooKeeper	An open-source server that enables highly reliable distributed coordination
Mahout	An Apache project whose goal is to build scalable-machine learning libraries
Storm	Framework that provides real-time processing of streams of data
Spark	A fast and general engine for large-scale data processing

HDP Developer: Apache Pig and Hive

The Hortonworks Data Platform (HDP)



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 distributor. It delivers the cost effectiveness of Hadoop and the advanced services required for enterprise deployments.

The key features of HDP include:

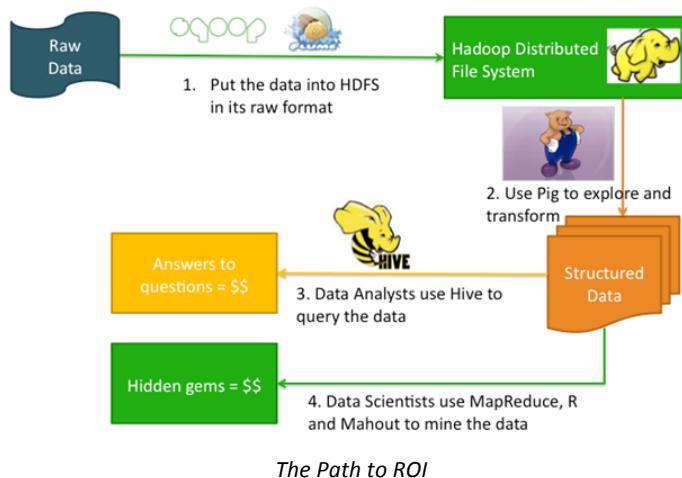
High Availability	HA is now achievable in HDP 2.x without the use of an outside technology
Open-Source Cluster Management	HDP includes Apache Ambari, the only open-source operations tool that allows you to provision, manage, and monitor a Hadoop cluster of any size
Metadata Services & HCatalog	HCatalog 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 an 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 designed to easily inter-operate so you can extend your existing investments in applications, tools, and processes with Hadoop.

HDP Developer: Apache Pig and Hive

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 to 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

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.

HDP Developer: Apache Pig and Hive

Review Questions

- 1) What are 1,024 petabytes known as? _____
- 2) What are 1,024 exabytes known as? _____
- 3) List the three Vs of big data: _____
- 4) Sentiment is one of the six key types of big data. List the other five:

- 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.x?

Lab: Start an HDP 2.1 Cluster

Objective: Start an HDP cluster in your VM

See page 7 of the HDP Developer: Apache Pig and Hive Lab Booklet.

HDP Developer: Apache Pig and Hive

The Hadoop Distributed File System (HDFS)

Lesson Objectives

This lesson covers the details of how files are stored and maintained in the Hadoop Distributed File System (HDFS).

After completing this lesson, students should be able to:

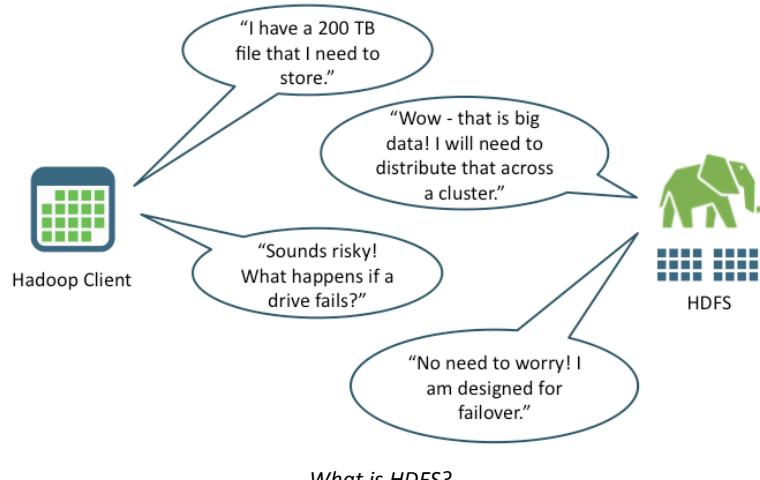
- Describe HDFS
- Describe How to Understand Block Storage
- Describe the NameNode
- Describe the DataNodes
- Describe HDFS Commands

Additional Content

- Demonstration: Understanding Block Storage
- Quiz: Lesson Review
- Lab: Using HDFS Commands

HDP Developer: Apache Pig and Hive

About HDFS



Data in Hadoop is stored on a filesystem referred to as HDFS or 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 data-storage mechanism for Hadoop. In Hadoop 2.x, YARN is referred to as the data operating system.

HDP Developer: Apache Pig and Hive

Hadoop vs. RDBMS

Hadoop	RDBMS
<ul style="list-style-type: none">Assumes a task will require reading a significant amount of data off of a diskIt does not maintain any data structureSimply reads the entire fileScales well (increase the cluster size to decrease the read time of each task)	<ul style="list-style-type: none">Uses indexes to avoid reading an entire file (very fast lookups)Maintains a data structure in order to provide a fast execution layerWorks well as long as the index fits in RAM
<ul style="list-style-type: none">- 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	<p>500 GB data file</p> <p>61 minutes to read this data off of a disk (assuming a transfer rate of 1,030 Mbps)</p>

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 RDBMS 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 the expense of disk access

Using indexes optimizes a relational database's performance by avoiding disk access. In order to store a lot of data and access it efficiently, RDBMS uses a smaller, organized representation of the data (an index) that can be loaded into memory and can 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.

HDP Developer: Apache Pig and Hive

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 eight disks each (so a total of 320 disks) will get an average six or seven 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 seven times 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.

HDP Developer: Apache Pig and Hive

HDFS Components

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

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.
DataNode	Stores the chunks of data and is responsible for replicating the chunks across other DataNodes.

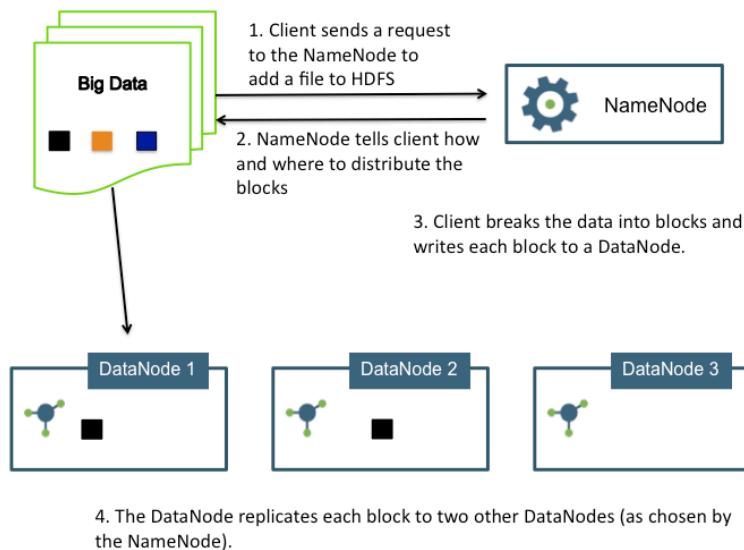
The NameNode and DataNode are daemon processes running in the cluster. Some important concepts involving the NameNode and DataNodes:

- 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 three (and is also configurable), which means each chunk of data is replicated across three 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



Understand 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 filesystem.
- 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, the global value of `dfs.blocksize` defined in `hdfs-site.xml` is used.



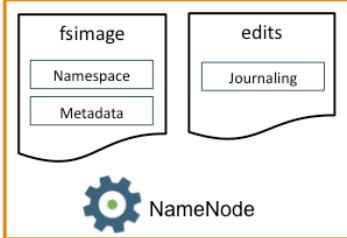
Important: Notice that the data never actually passes through the NameNode. The client program that is uploading the data into HDFS performs I/O directly with the DataNodes. The NameNode only stores the metadata of the filesystem, but is not responsible for storing or transferring the data.

Demonstration: Understanding Block Storage

Objective: To understand how data is partitioned into blocks and stored in HDFS

See page 15 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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.
- 

The NameNode

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

The `NameNode` has the following characteristics:

- Acts as the master of the `DataNodes`
- Executes filesystem `namespace` operations, like opening, closing, and renaming files and directories
- Determines the mapping of blocks to `DataNodes`
- Maintains the filesystem `namespace`

The `NameNode` performs these tasks by maintaining two files:

<code>fsimage_N</code>	Contains the entire filesystem <code>namespace</code> , including the mapping of blocks to files and filesystem properties
<code>edits_N</code>	A transaction log that persistently records every change that occurs to filesystem 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.

HDP Developer: Apache Pig and Hive

For example, initially you will have an `fsimage_0` file and an `edits_inprogress_1` file. When the merging occurs, the transactions in `edits_1` are merged with `fsimage_0` and a new `fsimage_1` file is created. In addition, a new empty `edits_2` file is created for all future transactions that occur after the creation of `fsimage_1`.

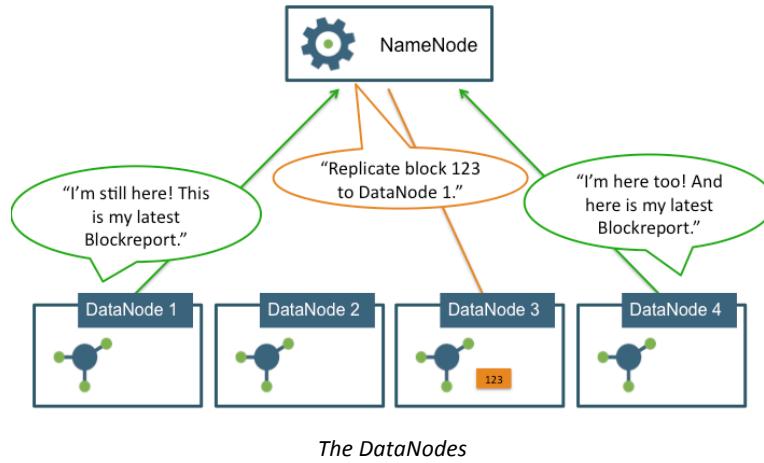
This process is called a checkpoint. Once the NameNode has successfully checkpointed, it will leave `safemode`, thus enabling writes.



Note: On your classroom VM, you can view the `fsimage` and `edit` files in the `/hadoop/hdfs/namenode/current` folder on the namenode machine:

```
# ssh namenode
[namenode ~]# ls -la /hadoop/hdfs/namenode/current/
total 1048
-rw-r--r-- 1 hdfs hdfs 1048576 edits_inprogress_00000000000000000001
-rw-r--r-- 1 hdfs hdfs 336 fsimage_00000000000000000000
-rw-r--r-- 1 hdfs hdfs 62 fsimage_00000000000000000000.md5
-rw-r--r-- 1 hdfs hdfs 2 seen_txid
-rw-r--r-- 1 hdfs hdfs 202 VERSION
```

The DataNodes



HDFS exposes a filesystem 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:

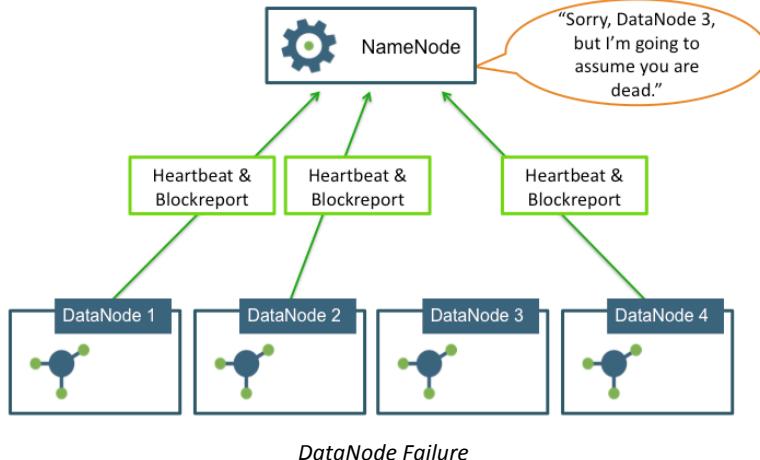
- The DataNode has no knowledge about HDFS files
- It stores each block of HDFS data in a separate file on its local filesystem
- The DataNode does not create all files in the same local directory. Instead, it uses a discovery technique to determine the optimal number of files per directory and creates 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)

HDP Developer: Apache Pig and Hive



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

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 the 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 anymore
- 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

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>
```

A command is one of the following:

```
# hadoop fs
Usage: hadoop fs [generic options]
      [-appendToFile <localsrc> ... <dst>]
      [-cat [-ignoreCrc] <src> ...]
      [-checksum <src> ...]
      [-chgrp [-R] GROUP PATH...]
      [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
      [-chown [-R] [OWNER][:[GROUP]] PATH...]
      [-copyFromLocal [-f] [-p] <localsrc> ... <dst>]
      [-copyToLocal [-p] [-ignoreCrc] [-crc] <src>...<localdst>]
      [-count [-q] <path> ...]
      [-cp [-f] [-p] <src> ... <dst>]
      [-createSnapshot <snapshotDir> [<snapshotName>]]
      [-deleteSnapshot <snapshotDir> <snapshotName>]
      [-df [-h] [<path> ...]]
      [-du [-s] [-h] <path> ...]
      [-expunge]
      [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
      [-getfacl [-R] <path>]
      [-getmerge [-nl] <src> <localdst>]
      [-help [cmd ...]]
      [-ls [-d] [-h] [-R] [<path> ...]]
      [-mkdir [-p] <path> ...]
      [-moveFromLocal <localsrc> ... <dst>]
      [-moveToLocal <src> <localdst>]
      [-mv <src> ... <dst>]
      [-put [-f] [-p] <localsrc> ... <dst>]
      [-renameSnapshot <snapshotDir> <oldName> <newName>]
      [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
      [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
      [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>] | [--set <acl_spec> <path>]]
      [-setrep [-R] [-w] <rep> <path> ...]
      [-stat [format] <path> ...]
      [-tail [-f] <file>]
      [-test [-defsz] <path>]
      [-text [-ignoreCrc] <src> ...]
      [-touchz <path> ...]
      [-usage [cmd ...]]
```

Use the help option for a description of a command. For example:

```
# hadoop fs -help put
-put [-f] [-p] <localsrc> ... <dst>: Copy files from the local file system
    into fs. Copying fails if the file already
    exists, unless the -f flag is given. Passing
    -p preserves access and modification times,
    ownership and the mode. Passing -f overwrites
    the destination if it already exists.
```

HDP Developer: Apache Pig and Hive

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.



Note: The `hdfs dfs` command is the same command as `hadoop fs`, and you may see the two used interchangeably.

HDP Developer: Apache Pig and Hive

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/worksaces/unmanaged/shell  
drwxr-xr-x - hue    hue      77 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/shell/hello.py  
drwxr-xr-x - hue    hue      0 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/sleep  
drwxr-xr-x - hue    hue      0 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/sleep/empty  
drwxr-xr-x - hue    hue      0 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/sqoop  
drwxr-xr-x - hue    hue     7175 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/sqoop/TT.java  
drwxr-xr-x - hue    hue     422 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/sqoop/db.hsqldb.properties  
drwxr-xr-x - hue    hue     276 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/sqoop/db.hsqldb.script  
drwxr-xr-x - hue    hue      0 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/ssh  
drwxr-xr-x - hue    hue      0 2013-08-27 23:00 /user/hue/oozie/worksaces/unmanaged/ssh/empty  
drwxr-xr-x - root   root     0 2013-08-29 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]#
```

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 `x` 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 /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
```



Note: HDFS also supports ACLs, which provide even finer-grained authorization capabilities.

HDP Developer: Apache Pig and Hive

Review Questions

- 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: Using HDFS Commands

Objective: Become familiar with adding, removing, and viewing files in HDFS

See page 19 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Inputting Data into HDFS

Lesson Objectives

This lesson covers the various ways to input data into the Hadoop Distributed File System, including the Sqoop and Flume frameworks.

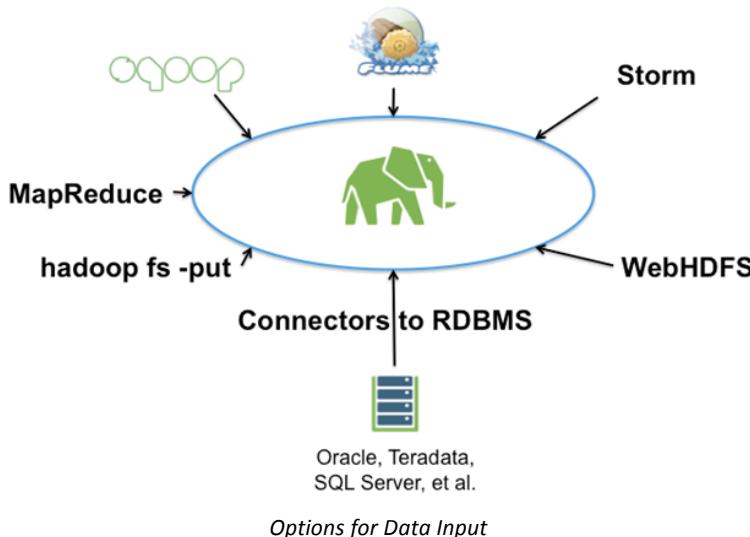
After completing this lesson, students should be able to:

- Describe the Options for Data Input
- Describe Flume
- Describe Sqoop
- Use Sqoop to transfer data between HDFS and a relational database

Additional Content

- Quiz: Lesson Review
- Lab: Importing RDBMS Data into HDFS
- Lab: Exporting HDFS Data to a RDBMS

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

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
```

HDP Developer: Apache Pig and Hive

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	Redirects the file read and file write calls to the corresponding Datanodes. It uses the full bandwidth of the Hadoop cluster for streaming data
Built into Hadoop	Runs inside NameNode and DataNodes, so there 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:

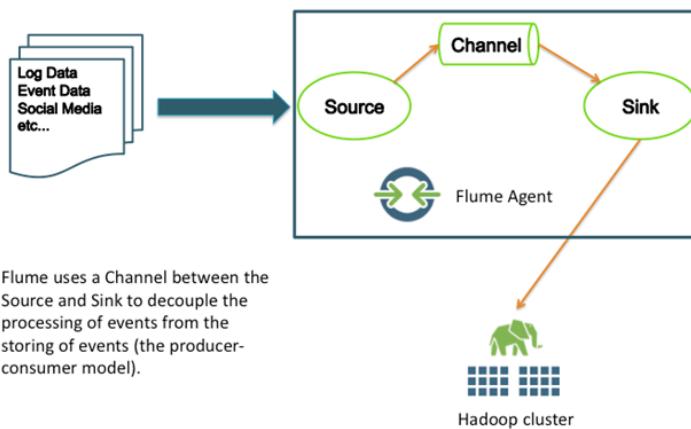
```
http://host:port/webhdfs/v1/user/root/data?op=MKDIRS
```

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
```

HDP Developer: Apache Pig and Hive

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.

HDP Developer: Apache Pig and Hive

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
```

The code `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:

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

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

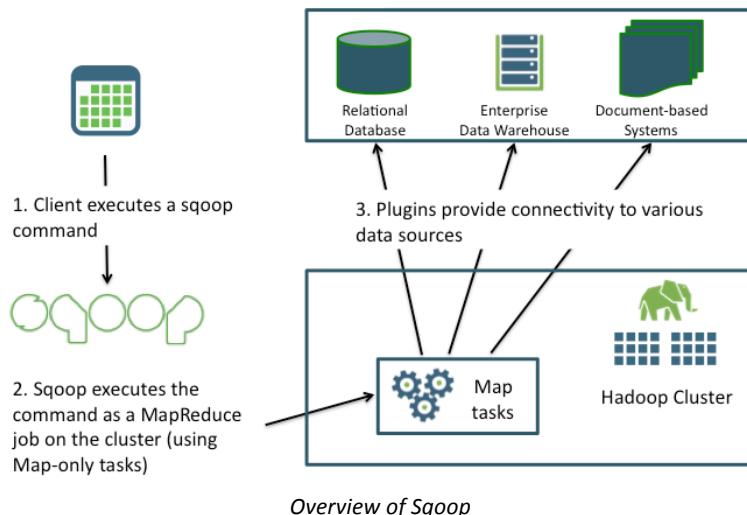
my_agent.channels.memoryChannel.type = memory
my_agent.channels.memoryChannel.capacity = 10000

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

- The name of the Agent is `my_agent`
- 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. Notice that 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`

HDP Developer: Apache Pig and Hive

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 diagram, Sqoop uses MapReduce to distribute its work across the Hadoop cluster:

- The `sqoop` command line executes a Sqoop job
- Map tasks (4 by default) execute the command in Sqoop
- Plugins are used to communicate with the outside data source. The data source provides the schema, 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 the SQL Server is also available from Microsoft: SQL Server R2 connector

HDP Developer: Apache Pig and Hive

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
- Can include credentials in the `connect` string, using the `--username` and `--password` arguments
- Must specify either a table to import using `--table` or the result of an SQL query using `--query`

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 the data will be imported into HDFS
- The default number of `map` tasks for Sqoop is four, so the result of this import will be in four 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.

HDP Developer: Apache Pig and Hive

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 <code>split-by</code> column, then the primary key column is used
-m	The number of <code>map</code> tasks to use
--query	Use instead of <code>-table</code> . The imported data are 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 was 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/> for a list of all arguments available for the `import` command.

HDP Developer: Apache Pig and Hive

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?

Answer: The `StockSymbol` column.

HDP Developer: Apache Pig and Hive

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.

HDP Developer: Apache Pig and Hive

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:

Insert Mode	The records being exported are inserted into the table using an SQL <code>INSERT</code> statement
Update Mode	An <code>UPDATE</code> SQL statement is executed for existing rows, and an <code>INSERT</code> can be used for new rows
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 <code>--update-key</code> is defined, 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 <code>--call</code> , do not define the <code>--table</code> 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	Specifies how updates are performed when new rows are found with non-matching keys in the database. Values are <code>updateonly</code> (the default) and <code>allowinsert</code>

HDP Developer: Apache Pig and Hive

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 `mylogs` 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 four `mappers`, but you can specify the number to use with the `-m` argument

HDP Developer: Apache Pig and Hive

Review Questions

1) What tool would work best for importing data from a relational database into HDFS?

2) What tool would work best for putting a file on your local `filesystem` into HDFS?

3) List the three main components of a typical `Flume` agent: _____

4) What is the default number of `map` tasks for a Sqoop job? _____

5) How do you specify a different number of `mappers` in a Sqoop job?

6) What is the purpose of the `$CONDITIONS` value in the `WHERE` clause of a Sqoop query?

Lab: Importing RDBMS Data into HDFS

Objective: Import data from a database into HDFS

See page 25 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Lab: Exporting HDFS Data to a RDBMS

Objective: Export data from HDFS into a MySQL table using Sqoop

See page 29 of the HDP Developer: Apache Pig and Hive Lab Booklet.

The MapReduce Framework

Lesson Objectives

This lesson covers the details of the MapReduce programming paradigm.

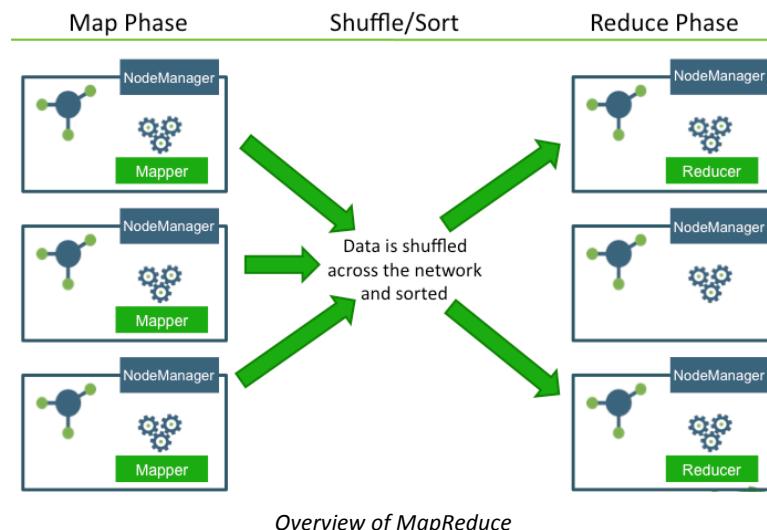
After completing this lesson, students should be able to:

- Describe MapReduce
- Describe the Map Phase
- Describe the Reduce Phase

Additional Content

- Demo: Understanding MapReduce
- Quiz: Lesson Review
- Lab: Running a MapReduce Job

Overview of MapReduce



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:

Map phase Data is input into the `mapper`, where it is transformed and prepared for the `reducer`

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.

HDP Developer: Apache Pig and Hive

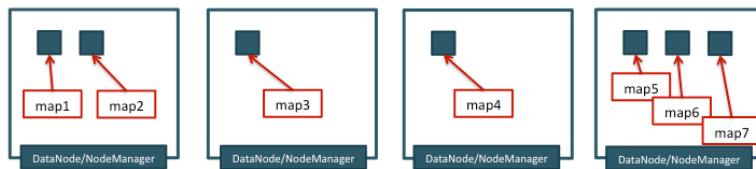
Some important concepts to understand about MapReduce:

- 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 of 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

HDP Developer: Apache Pig and Hive

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.

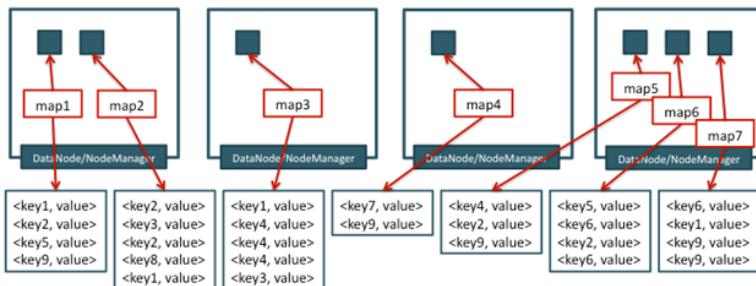
Understanding MapReduce

The `map` phase involves running `map` tasks on `NodeManagers`. The main purpose of the `map` phase is to read all of 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

HDP Developer: Apache Pig and Hive

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



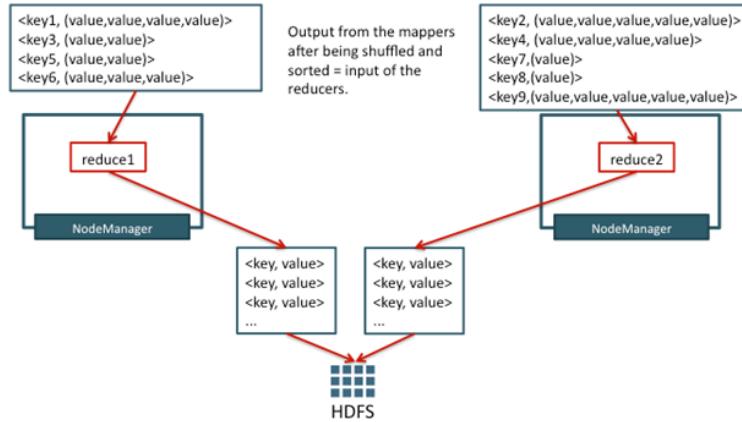
4. The `<key,value>` pairs go through a shuffle/sort phase, where records with the same key end up at the same reducer. The specific pairs sent to a reducer are sorted by key, and the values are aggregated into a collection.

Understanding MapReduce - continued

- 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

HDP Developer: Apache Pig and Hive

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.



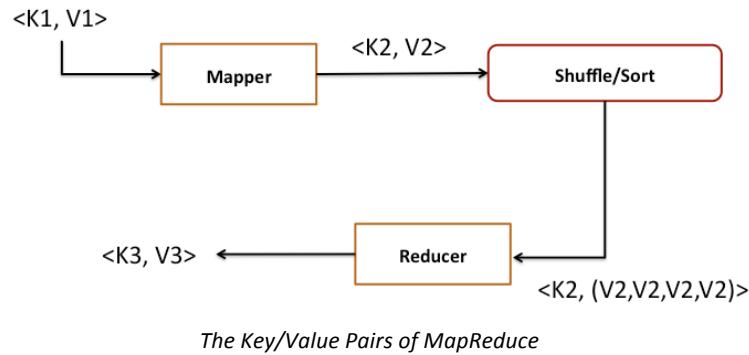
Understanding MapReduce - continued

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 into 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 five reducers, the output will be five different files
- The number of `reduce` tasks in a MapReduce job is a setting that you get to choose

HDP Developer: Apache Pig and Hive

The Key/Value Pairs of MapReduce



The data types of the `<key, value>` pairs in a MapReduce job look like:

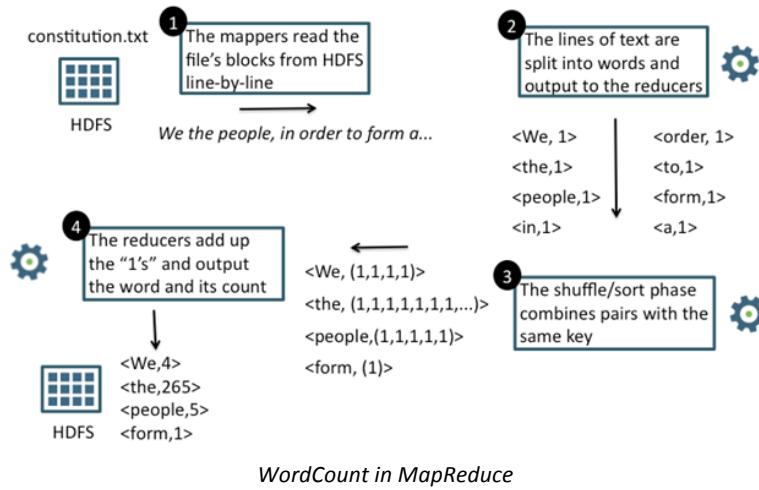
<code><K1, V1></code>	Input to the mapper
<code><K2, V2></code>	Output from the mapper
<code><K2, Iterable<V2>></code>	Input to the reducer
<code><K3, V3></code>	Output from the reducer



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.

HDP Developer: Apache Pig and Hive

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.

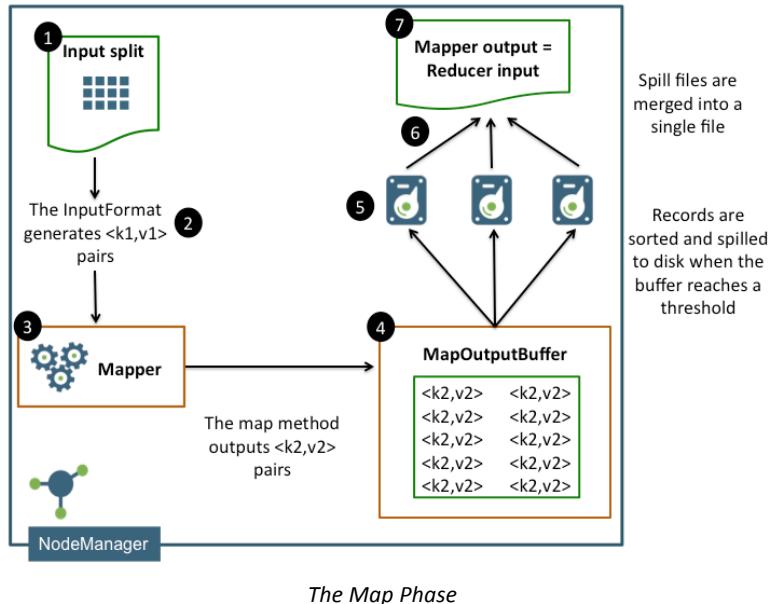
The diagram above shows how the `<key, value>` pairs of the word-count application are passed through the MapReduce job.

Demonstration: Understanding MapReduce

Objective: To understand how MapReduce works

See page 33 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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.

HDP Developer: Apache Pig and Hive

Here is how data flows through the `map` phase:

- The `InputFormat` determines where the input data needs to be split between the `mappers`, and then it generates an `InputSplit` instance for each `split`
- MapReduce spawns a `map` task for each `InputSplit` generated by the `InputFormat`
- Each `<key, value>` pair generated by the `InputFormat` is passed to the `map` method of the `mapper class`
- The `map` method outputs a `<key, value>` pair that is serialized into an unsorted buffer in memory
- 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
- If more than one spill file was created, these files are merged into a single file of sorted `<key, value>` pairs
- The sorted records in the `spill` file wait to be retrieved by a `reducer`

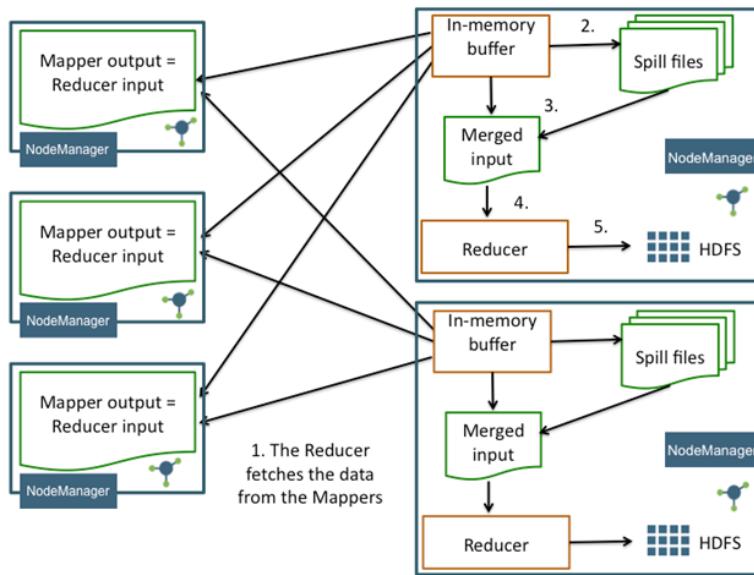


Note: The size of the `mapper's` output memory buffer is configurable with the `mapreduce.task.io.sort.mb` property. A `spill` occurs when the buffer reaches a certain capacity configured by the `mapreduce.map.sort.spill.percent` property.



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 rewritten one more time, which results in three times the disk I/O for those spilled records

The Reduce Phase



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).

HDP Developer: Apache Pig and Hive

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

Shuffle	Also referred to as the <code>fetch</code> phase, this is when <code>reducers</code> retrieve the output of the <code>mappers</code> . All records with the same <code>key</code> are combined and sent to the same <code>reducer</code>
Sort	This phase happens simultaneously with the <code>shuffle</code> phase. As the records are fetched and merged, they are sorted by <code>key</code>
Reduce	The <code>reduce</code> method is invoked for each <code>key</code> , with the records combined into an iterable collection

Here is how data flows through the `reduce` phase:

- As `mappers` finish their tasks, the `reducers` start fetching the records and storing them into a buffer in their JVM's memory
- If the buffer fills, it is spilled to disk
- 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)
- The `reduce` method is invoked on the `reducer` for each `key`
- 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 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 zero 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

Review Questions

- 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: Running a MapReduce Job

Objective: Run a Java MapReduce job

See page 35 of the HDP Developer: Apache Pig and Hive Lab Booklet.

HDP Developer: Apache Pig and Hive

Introduction to Pig

Lesson Objectives

This lesson covers the Pig framework and describes how to load and transform data using the Pig programming language.

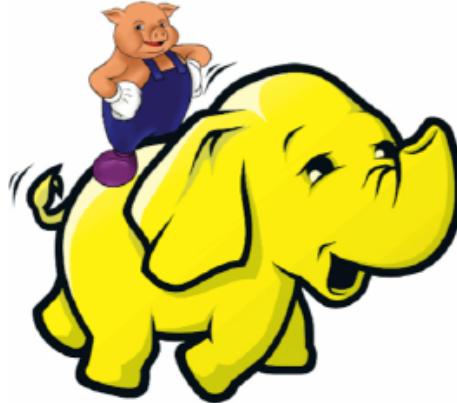
After completing this lesson, students should be able to:

- Describe Pig
- Define a Schema
- Describe the GROUP Operator
- Describe the FOREACH GENERATE Operator
- Describe the FILTER Operator
- Describe the LIMIT Operator

Additional Content

- Demo: Understanding Pig
- Lab: Getting Started with Pig
- Quiz: Lesson Review
- Lab: Exploring Data with Pig

About Pig



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 the ability to:

- 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)
- Use a familiar SQL-like syntax in Pig Latin
- Read and write data from HDFS with Pig scripts
- Create code with 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

HDP Developer: Apache Pig and Hive

Pig Latin

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

Pig script	Write a Pig Latin program in a text file and execute it using the <code>pig</code> 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 <code>PigServer</code> 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 is used

The Grunt Shell



Grunt shell

A screenshot of a terminal window titled "rich — root@sandbox:~ — ssh — 59x5". The window shows the following Pig Latin commands and their results:

```
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;
```

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, which 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

See page 37 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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 reassigned an alias to a different relation, but that is not recommended.

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:

- Must start with an alphabetic character
- Can contain alphabetic and numeric characters, as well as the underscore (`_`) character
- Can only contain ASCII characters



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.

HDP Developer: Apache Pig and Hive

Pig Data Types

Pig has six built-in scalar data types:

int	A 32-bit signed integer
long	A 64-bit signed integer
float	A 32-bit floating-point number
double	A 64-bit floating-point number
chararray	Strings of Unicode characters (represented as <code>java.lang.String</code> objects)
bytearray	A blob or array of bytes
boolean	Can be either true or false (case-sensitive)
datetime	A date and time stored in the format 1970-01-01T00:00:00.000+00:00
bigdecimal and biginteger	For performing precision arithmetic

HDP Developer: Apache Pig and Hive

Pig Complex Types

Pig has three complex types:

Tuple	Ordered set of fields. A tuple is analogous to a row in an 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 commas separate the tuples within the bag. 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. The following map has three key#value pairs:

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

As you can see 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

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, the extra fields will be truncated. If you load a `customer` record that has fewer than six fields, 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{(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`.

Pig is very lenient when it comes to schemas:

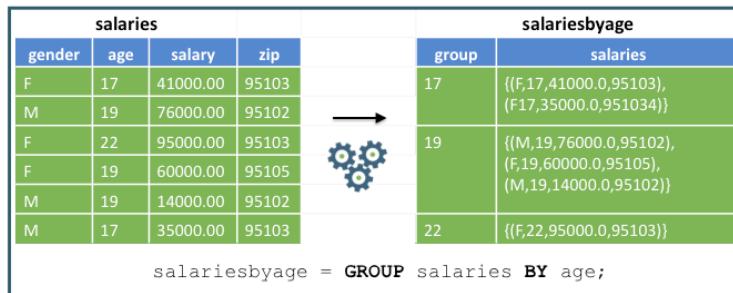
- If you define a schema, then Pig will perform error-checking with it
- If you do not define a schema, Pig will make its best guess as to how the data should be treated

Lab: Getting Started with Pig

Objective: Use Pig to navigate through HDFS and explore a dataset

See page 41 of the HDP Developer: Apache Pig and Hive Lab Booklet.

The GROUP Operator



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

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) })
```

HDP Developer: Apache Pig and Hive

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

The diagram illustrates the GROUP ALL operation. On the left, a table labeled "salaries" contains six rows of data with columns: gender, age, salary, and zip. An arrow points from this table to a second table on the right labeled "salariesbyage". The "salariesbyage" table has two columns: group and salaries. The "group" column contains the value "all". The "salaries" column contains a bag of tuples, each representing a record from the original "salaries" table. Below the tables is the Pig Latin command: "salariesbyage = GROUP salaries 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	all	{(F,17,41000.0,95103), (M,19,76000.0,95102), (F,22,95000.0,95103), (F,19,60000.0,95105), (M,19,14000.0,95102), (F,17,35000.0,95103)}
M	19	14000.00	95102		
M	17	35000.00	95103		

salariesbyage = GROUP salaries ALL;

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)})
```

HDP Developer: Apache Pig and Hive

Relations without a Schema

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

`salariesgroup = GROUP salaries BY $3;`

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;
```

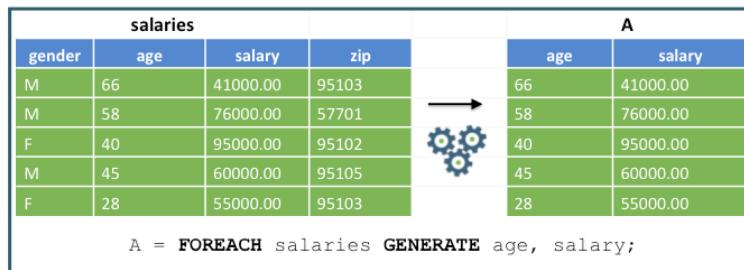
Notice the `salariesgroup` relation does not have a schema for its `salaries` field:

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

Why is the datatype of group `bytearray`? _____

Answer: Because the `salaries` relation does not have a schema, the data type of the field used for grouping is the default `bytearray` type.

The FOREACH GENERATE Operator



The FOREACH GENERATE Operator

The `FOREACH...GENERATE` operator transforms records based on a set of expressions that 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)
```

HDP Developer: Apache Pig and Hive

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:

```
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

A relation created from a `FOREACH` statement is a new tuple. Pig infers the data types of the fields in the new tuple, but sometimes the names of the fields are not inferred. 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}
```

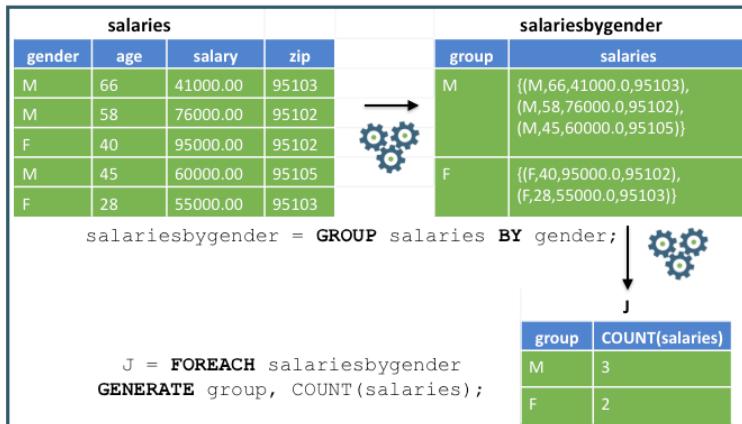
Notice the second field in `E` has the name `bonus`.



Note: You can use the `AS` keyword for any of the fields in the `GENERATE` clause, even if Pig can infer the field name.

HDP Developer: Apache Pig and Hive

FOREACH with Groups



FOREACH with Groups

Let's look at an example of a Pig script that performs a FOREACH operation on a group:

```
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}
```

The 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 the 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)
```

HDP Developer: Apache Pig and Hive

The FILTER Operator

salaries				G			
gender	age	salary	zip	gender	age	salary	zip
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				

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

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. 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;
```

HDP Developer: Apache Pig and Hive

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

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 that there is no guarantee which three 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 lesson) immediately before the `LIMIT`, then you will be guaranteed to get the same results each time.

Review Questions

- 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 a `volume` greater than 3,000:

Lab: Exploring Data with Pig

Objective: Use Pig to navigate through HDFS and explore a dataset

See page 45 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Advanced Pig Programming

Lesson Objectives

This lesson covers some of the more advanced features of Pig, including sorting, parallelization, joins, and user-defined functions.

After completing this lesson, students should be able to:

- Describe the ORDER BY Operator
- Describe the CASE Operator
- Describe the DISTINCT Operator
- Describe How to Use PARALLEL
- Describe the FLATTEN Operator
- Describe Nested FOREACH
- Describe Joins
- Describe the COGROUP Operator
- Describe Pig User-Defined Functions

Additional Content

- Lab: Splitting a Dataset
- Lab: Joining Datasets
- Lab: Preparing Data for Hive
- Demo: Computing PageRank
- Quiz: Lesson Review
- Lab: Analyzing Clickstream Data
- Lab: 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	F	28	55000.0	95103
M	58	76000.00	95102	F	40	95000.0	95102
F	40	95000.00	95102	M	45	60000.0	95105
M	45	60000.00	95105	M	58	76000.0	95102
F	28	55000.00	95103	M	66	41000.0	95103

byage = ORDER salaries BY age ASC;

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:

```
agesalary = ORDER salaries BY age ASC, salary ASC;
```

The output is similar to byage, except the salary field is sorted in ascending order. 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)
```

HDP Developer: Apache Pig and Hive



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, 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` in descending order.

The CASE Operator

The diagram illustrates the transformation of a **salaries** table into a **bonuses** table using the CASE operator. An arrow points from the **salaries** table to the **bonuses** table, accompanied by three interlocking gears, symbolizing data processing.

salaries				bonuses	
gender	age	salary	zip	salary	bonus
M	66	41000.0	95103	41000.0	2050.0
M	58	76000.0	95102	76000.0	7600.0
F	40	95000.0	95102	95000.0	9500.0
M	45	20000.0	95105	20000.0	0.0
F	28	55000.0	95103	55000.00	2750.0

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

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
```

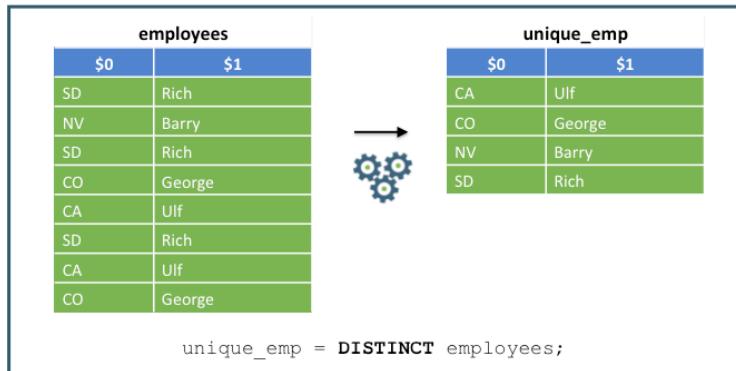
Use the -param_file switch if your properties are stored in a text file:

```
> pig -param_file stock.params myscript.pig
```

The text file stock.params looks like this:

```
INPUTFILE=NYSE_daily_prices_A.csv
```

The DISTINCT Operator



The DISTINCT Operator

The DISTINCT operator removes duplicate tuples in a relation. The syntax is:

```
DISTINCT alias;
```

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

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;
```

In this clause, `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 five reducers.

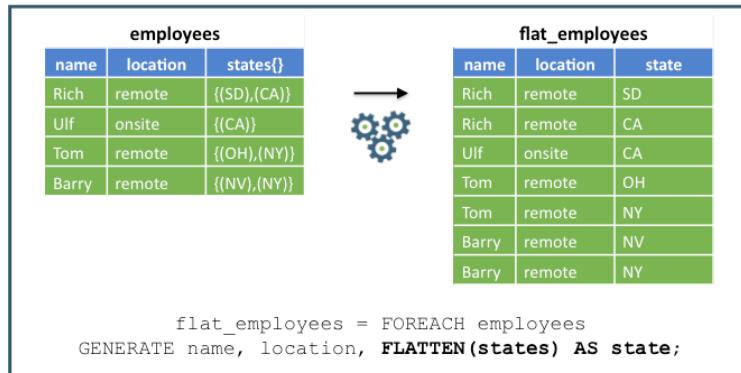
You can use the `default_parallel` property to set the number of reducers at the script level. As an example, there will be eight reducers for each reduce task in the following Pig script:

```
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; these are `LOAD`, `FOREACH`, `FILTER`, and `SAMPLE`. For those types of operators, it does not make sense to specify a `PARALLEL` value.

The FLATTEN Operator



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 Pig Latin statements below 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)})

```

HDP Developer: Apache Pig and Hive

Notice that each record has a `bag` containing one or more states. If you flatten the `states` field in the `employees` relation, each entry in the `bag` becomes its own full record:

```
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. Suppose a `tuple` looks like:

```
(1, (2, 3))
```

After this `tuple` was flattened, it would look like:

```
(1,2,3)
```

Lab: Splitting a Dataset

Objective: Research the White House visitor data and look for members of Congress

See page 51 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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 shows how to count unique entries in a group using a nested FOREACH. The data is daily stock prices from the New York Stock Exchange (NYSE); 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 only have the field `symbol`
- The `unique_symbol` relation is also a list of symbols but with all of the duplicates removed
- The GENERATE statement projects the group (which is “NYSE” in this example) and the number of values in `unique_symbol`

The output is:

```
(NYSE,203)
```

This means there are 203 unique stock symbols in the `NYSE_daily_prices_A.csv` file.



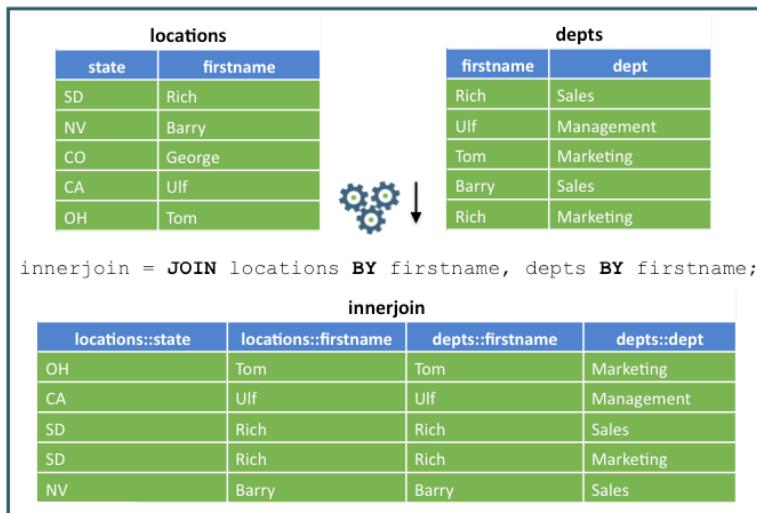
Note: Another common task inside a nested FOREACH is ORDER BY. For example:

```
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;
    first_traded_date = LIMIT sorted 1;
    GENERATE group, first_traded_date;
};
```

HDP Developer: Apache Pig and Hive

About Joins

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
```

HDP Developer: Apache Pig and Hive

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)
```

HDP Developer: Apache Pig and Hive

Performing an Outer Join

locations		depts	
state	firstname	firstname	dept
SD	Rich	Rich	Sales
NV	Barry	Ulf	Management
CO	George	Tom	Marketing
CA	Ulf	Barry	Sales
OH	Tom	Rich	Marketing

outerjoin			
locations::state	locations::firstname	depts::firstname	depts::dept
OH	Tom	Tom	Marketing
CA	Ulf	Ulf	Management
SD	Rich	Rich	Sales
SD	Rich	Rich	Marketing
NV	Barry	Barry	Sales
CO	George		

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 in the outer join. Pig inserts null values into the missing fields.

HDP Developer: Apache Pig and Hive

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 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, ,)
```

HDP Developer: Apache Pig and Hive

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 the memory, the underlying MapReduce will generate an error and fail.

The COGROUP Operator



locations		departments	
state	firstname	firstname	dept
SD	Rich	Rich	Sales
NV	Barry	Ulf	Management
CO	George	Tom	Marketing
CA	Ulf	Barry	Sales
OH	Tom	Rich	Marketing

cgroup		
group	locations	departments
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)}	{}

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)}, {}{})
```

HDP Developer: Apache Pig and Hive

You could use the `cgroup` relation to count the number of records that would occur in the join's result:

```
counters = FOREACH cgroup GENERATE group, COUNT(locations),  
          COUNT(departments);
```

The first number is the inner join count, and the second number is the outer join count:

```
(Tom,1,1)  
(Ulf,1,1)  
(Rich,1,2)  
(Barry,1,1)  
(George,1,0)
```



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

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 tasks. Pig UDFs can be written in six languages:

- Java
- Jython
- Python
- JRuby
- JavaScript
- Groovy

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.



Reference: The Pig API Javadocs are at:

<http://pig.apache.org/docs/r0.12.1/api/>

HDP Developer: Apache Pig and Hive

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_COMMMA 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;
    }
}
```

- The `CONCAT_COMMMA` 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.
- The `get` method of `Tuple` is used 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, call the UDF using its fully qualified class name:

```
x = FOREACH logevents
    GENERATE com.hortonworks.udfs.CONCAT_COMMMA(level, code);
```

As an option, you can use the `DEFINE` command to define an alias that simplifies the syntax for invoking the UDF:

```
DEFINE CONCAT_COMMMA com.hortonworks.udfs.CONCAT_COMMMA();
```

Now you can invoke the UDF using the alias:

```
x = FOREACH logevents GENERATE CONCAT_COMMMA(level, code);
```

HDP Developer: Apache Pig and Hive

Tips for Optimizing Pig Scripts

Here are a few best practices that can make the difference in the performance of 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 <code>FOREACH</code> to remove unwanted or unused fields in your records as soon as possible
Drop nulls before a join	Filter out null records before the <code>JOIN</code> . The gain can be significant, even if you have a small percentage of null values
Use replicated joins whenever possible	A <code>map-side</code> join is always much more efficient than a <code>reduce-side</code> join
Use <code>PARALLEL</code> 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 <code>map/reduce</code> tasks and jobs by setting <code>mapreduce.map.output.compress</code> to true and specifying a compression codec with <code>mapreduce.map.output.compress.codec</code> . Enable compression of the output files between MapReduce jobs within a Pig processing pipeline by setting the <code>pig.tmpfilecompression</code> and <code>pig.tmpfilecompression.codec</code> properties
Choose the right data types	If you are treating a field as a specific data type, define the type in the <code>LOAD</code> 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 perform any other task that can be applied globally to all of your Pig scripts.

Lab: Joining Datasets

Objective: Join two datasets in Pig

See page 55 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Lab: Preparing Data for Hive

Objective: Transform and export a dataset for use with Hive

See page 61 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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-1.2.0.jar;
```

HDP Developer: Apache Pig and Hive

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 four-quantile (commonly called a quartile). If $n = 5$, you have a five-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 five-quantile function would be defined as:

```
define Quintile datafu.pig.stats.Quantile('0.0','0.20',
                                             '0.40','0.60','0.80','1.0');
```

You can also instantiate a `Quantile` by passing in the number of evenly-spaced ranges. For example, the above `Quintile` could also be defined as:

```
define Quintile datafu.pig.stats.Quantile('6');
```

Quintiles are what five-quantiles are called, 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, (-7.22,-3.48,13.6,16.05,19.49,24.5))
(Moscow, (-9.0,-2.04,5.5,18.975,21.205,24.98))
(NorthPole, (-20.5,-14.6,-8.76,-2.57,1.475,2.445,3.61))
(Houston, (40.9,51.12,69.41,82.75,94.55,105.87))
(IntlFalls, (-14.41,-4.25,-1.15,12.15,17.6,21.73))
```

For example, in Toronto you have an equal number of days where the high temperature was between -7.22 and -3.48 degrees Celsius, between -3.48 and 13.6 degrees Celsius, between 13.6 and 16.05 degrees Celsius, and so on.

Demonstration: Computing PageRank

Objective: To understand how to use the PageRank UDF in DataFu

See page 63 of the HDP Developer: Apache Pig and Hive Lab Booklet.

HDP Developer: Apache Pig and Hive

Review Questions

- 1) If a relation is sorted using `ORDER BY` and the resulting MapReduce job runs with three 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

```
(CASE
    WHEN volume > 3000 THEN volume
    WHEN volume <= 3000 THEN -1
END) AS high_volume;
```

- 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: Analyzing Clickstream Data

Objective: Become familiar with using the DataFu library to sessionize clickstream data

See page 67 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Lab: Analyzing Stock Market Data using Quantiles

Objective: Use DataFu to compute quantiles

See page 71 of the HDP Developer: Apache Pig and Hive Lab Booklet.

HDP Developer: Apache Pig and Hive

Hive Programming

Lesson Objectives

This lesson covers the details of the Hive framework and HiveQL programming language.

After completing this lesson, students should be able to:

- Describe Hive
- Describe How to Define Tables
- Describe How to Perform Queries
- Describe How to Sort Data
- Describe How to Join Data and Hive Join Strategies

Additional Content

- Lab: Understanding Hive Tables
- Demo: Understanding Partitions and Skew
- Lab: Analyzing Big Data with Hive
- Lab: Understanding MapReduce in Hive
- Demo: Computing ngrams
- Lab: Joining Datasets in Hive
- Lab: Computing ngrams of Emails in Avro Format
- Quiz: Lesson Review

About 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.

HDP Developer: Apache Pig and Hive

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

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
```

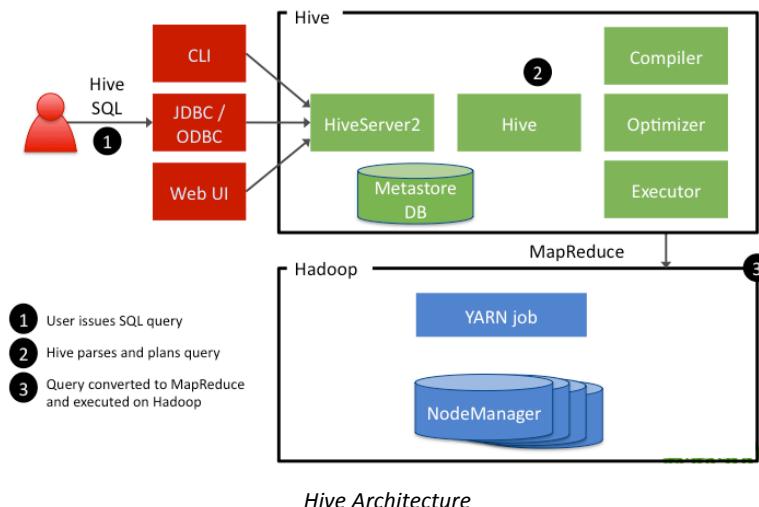
Some of the limitations of Hive include:

- Index and view support are limited (discussed in detail later)
- The data in Hive is read only (no updates)
- Datatypes do not line up with traditional SQL types
- New partitions can be inserted, but not individual rows

HDP Developer: Apache Pig and Hive

Hive Architecture

Hive queries are submitted to a HiveServer2 process that typically runs on a master node in the cluster.



Issuing Commands	Using the Hive CLI, a Web interface, or a Hive JDBC/ODBC client, a Hive query is submitted to the <code>HiveServer</code>
Hive Query Plan	The Hive query is compiled, optimized, and planned as a MapReduce job
MapReduce Job Executes	The corresponding MapReduce job is executed on the Hadoop cluster

HDP Developer: Apache Pig and Hive

Submitting Hive Queries

Hive queries are written using the `HiveQL` language, an 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 into 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  
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 -u url -n username -p password  
beeline>
```

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.

HDP Developer: Apache Pig and Hive

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/`).

HDP Developer: Apache Pig and Hive

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 TABLE 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
 ORDER 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: Understanding Hive Tables

Objective: Understand how Hive table data is stored in HDFS

See page 75 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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) in 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:

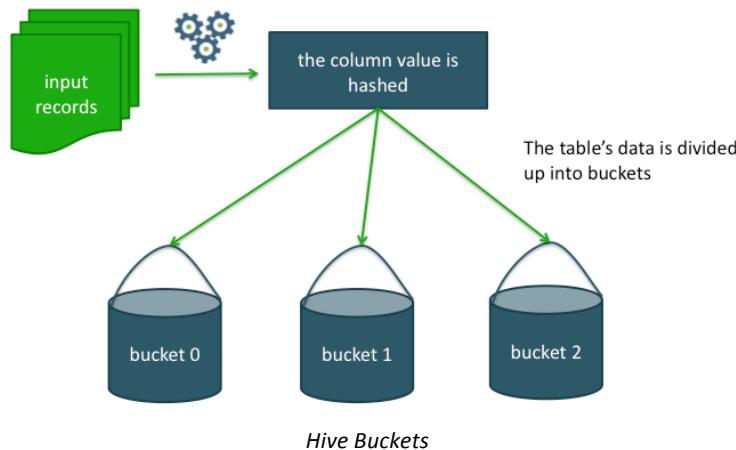
```
/apps/hive/warehouse/employees
  /dept=hr/
  /dept=support/
  /dept=engineering/
  /dept=training/
```



Note: You can partition by multiple columns, which results in subfolders within the subfolders of the table's warehouse directory.

HDP Developer: Apache Pig and Hive

Hive Buckets



Hive tables can be organized into buckets, which imposes extra structure on the table and the way 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, the buckets are numbered `0` to `n-1` and Hive hashes the column value and then uses the `modulo` operator on the `hash` value.

HDP Developer: Apache Pig and Hive

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 that there will be a large number of customers in those two ZIP codes.

Demonstration: Understanding Partitions and Skew

Objective: To understand how Hive partitioning and skewed tables work

See page 81 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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, `sort by` provides a partial sorting of the data by `reducer` but not a total ordering.

`Order by` implements a total ordering across all `reducers`. To obtain a parallel total ordering across multiple `reducers` in Hive, you have to set the following property:

```
hive.optimize.sampling.orderby=true
```

If you do not set the property above then the total ordering is achieved by using one `reducer`. In that situation, you must add a `LIMIT` clause to the Hive query to limit the size of the output so that it can be managed by a single `reducer`.

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 table named `salaries` with the schema (gender, age, salary, zip):

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
...			

Note that `distribute by` is typically used in conjunction with an `insert` statement (or also when using Hadoop streaming with custom `mappers` and/or `reducers`). The following command demonstrates `distribute by` on the `age` column:

```
set mapreduce.job.reduces=2;
insert overwrite table mytable
  select gender, age, salary from salaries
  distribute by age;
```

Records with the same `age` will go to the same `reducer`.

HDP Developer: Apache Pig and Hive

The `distribute by` does not guarantee any type of clustering of the records. For example, a reducer might get:

```
M,66,84000.0  
F,58,95000.0  
M,40,76000.0  
F,66,41000.0
```

The two records with `age = 66` are sent to the same reducer, but they are not adjacent. You can use `sort by` to cluster records with the same `distribute by` column together:

```
insert overwrite table mytable  
    select gender, age, salary from salaries  
    distribute by age  
    sort by age;
```

The records with the same age will now appear together in the reducer's output:

```
F,58,95000.0  
M,66,84000.0  
F,66,41000.0  
M,68,15000.0  
F,68,60000.0  
M,72,83000.0
```



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:

```
insert overwrite table myoutput_table  
    from dataset  
        select *  
    cluster by age;
```

HDP Developer: Apache Pig and Hive

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

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 the `hivevar` flag:

```
hive -f myscript.hive -hivevar age=33
```

Lab: Analyzing Big Data with Hive

Objective: Analyze the White House visitor data

See page 85 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Lab: Understanding MapReduce in Hive

Objective: To understand how Hive queries get executed as MapReduce jobs

See page 91 of HDP Developer: Apache Pig and Hive Lab Booklet.

Hive Join Strategies

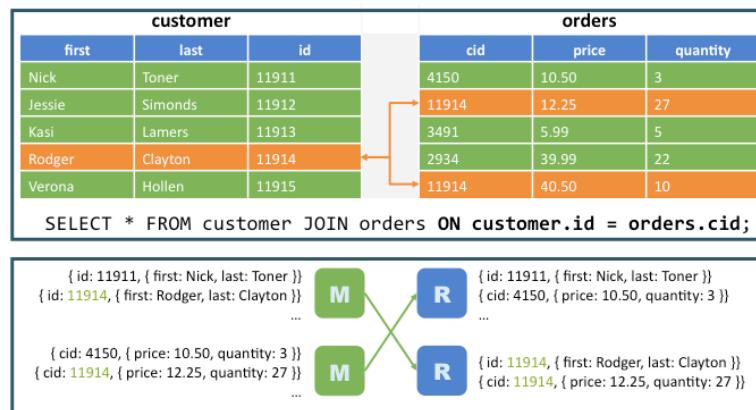
Type	Approach	Pros	Cons
Shuffle Join	Join keys are shuffled using MapReduce and joins are performed on the reduce side.	Works regardless of data size or layout.	Most resource-intensive and slowest join type.
Map (Broadcast) Join	Small tables are loaded into memory in all nodes, mapper scans through the 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.

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



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.

HDP Developer: Apache Pig and Hive

Map (Broadcast) Joins

customer			orders		
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 orders ON customer.id = orders.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.

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.x, Hive joins are automatically optimized without the need for providing hints.

HDP Developer: Apache Pig and Hive

Sort-Merge-Bucket (SMB) Joins

customer			orders		
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 orders ON customer.id = orders.cid;

Distribute and sort by the most common join key.

```
CREATE TABLE orders (cid int, price float, quantity int)
CLUSTERED BY(cid) SORTED BY(cid) INTO 32 BUCKETS;

CREATE TABLE customer (id int, first string, last string)
CLUSTERED BY(id) SORTED BY(cid) INTO 32 BUCKETS;
```

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 that, 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.x):

```
hive.auto.convert.sortmerge.join=true;
hive.optimize.bucketmapjoin = true;
hive.optimize.bucketmapjoin.sortedmerge = true;
hive.auto.convert.sortmerge.join.noconditionaltask = true;
```

HDP Developer: Apache Pig and Hive

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:

- Register the JAR file that contains the UDF class and
- 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);
```

HDP Developer: Apache Pig and Hive

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

See page 95 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Review Questions

- 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: Joining Datasets in Hive

Objective: Perform a join of two datasets in Hive

See page 99 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Lab: Computing ngrams of Emails in Avro Format

Objective: Use Hive to compute ngrams

See page 103 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Using HCatalog

Lesson Objectives

This lesson covers the details of how HCatalog is used to provide a central repository for defining and sharing schemas for data stored in Hadoop.

After completing this lesson, students should be able to:

- About HCatalog
- HCatalog in the Ecosystem
- Defining a New Schema
- Using HCatalogLoader with Pig
- Using HCatStorer with Pig

Additional Content

- Quiz: Lesson Review
- Lab: Using HCatalog with Pig

HDP Developer: Apache Pig and Hive

About 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?

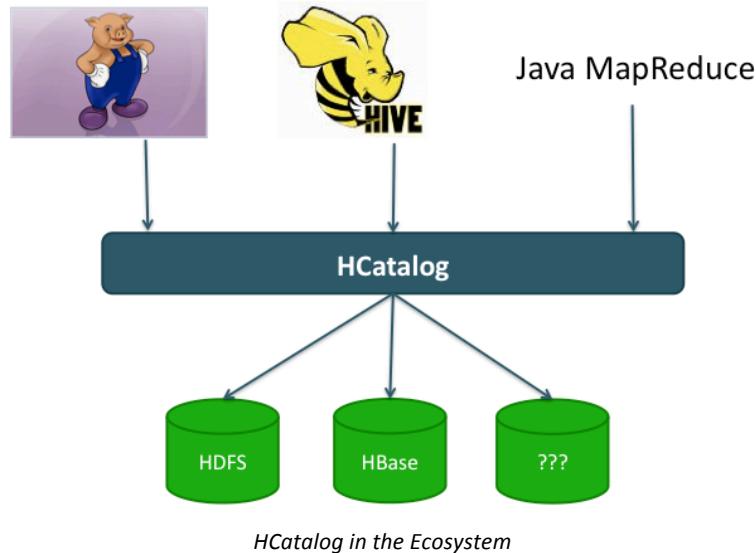
Dude, we need HCatalog

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 MapReduce 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 an SQL-like language (HiveQL) to data within Hadoop
- Allows users to write their applications without being concerned about how or where the data is stored
- Insulates users from schema- and storage-format changes

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 of the tools and frameworks used in your Hadoop environment.

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

`HCatalog` provides two interfaces for use by `Pig` scripts to read and write data in `HCatalog`-managed tables:

<code>HCatLoader</code>	To read data from <code>HCatalog</code> -managed tables
<code>HCatStorer</code>	To write data to <code>HCatalog</code> -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 that 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

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.

The Pig SQL Command

`Pig` has an `SQL` command that you can use to run `Hive` DDL commands. For example, you could create a table from within a `Pig` script (or the `Grunt` shell) using the following command:

```
grunt> sql create table movies (  
    title string,  
    rating string,  
    length double)  
partitioned by (genre string)  
stored as ORC;
```

HDP Developer: Apache Pig and Hive

Review Questions

- 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: Using HCatalog with Pig

Objective: Use HCatalog to provide the schema for a Pig relation

See page 109 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Advanced Hive Programming

Lesson Objectives

This lesson covers some of the more advanced features of Hive programming, including views, the windowing functions, and the various optimization capabilities of Hive.

After completing this lesson, students should be able to:

- Describe How to Perform a Multi-Table/File Insert
- Describe Views
- Describe How to Use Windows
- Describe How to Compute Table Statistics
- Describe How to Use HiveServer2
- Describe How to Understand Hive on Tez

Additional Content

- Lab: Advanced Hive Programming
- Demo: Hive Optimizations
- Quiz: Lesson Review
- Lab: Streaming Data with Hive and Python

Performing a Multi-Table/File Insert

```
insert overwrite directory '2014_visitors' select * from wh_visits  
where visit_year='2014'  
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;
```

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, in a single `mapper` and a single `reducer`, or in multiple `mappers` and multiple `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 use when querying Big Data using Hive is to perform a multi-table or multi-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' ;
```

As expected, each query above requires a MapReduce job.

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' ;
```

HDP Developer: Apache Pig and Hive

Notice the only difference is that the semicolon 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.



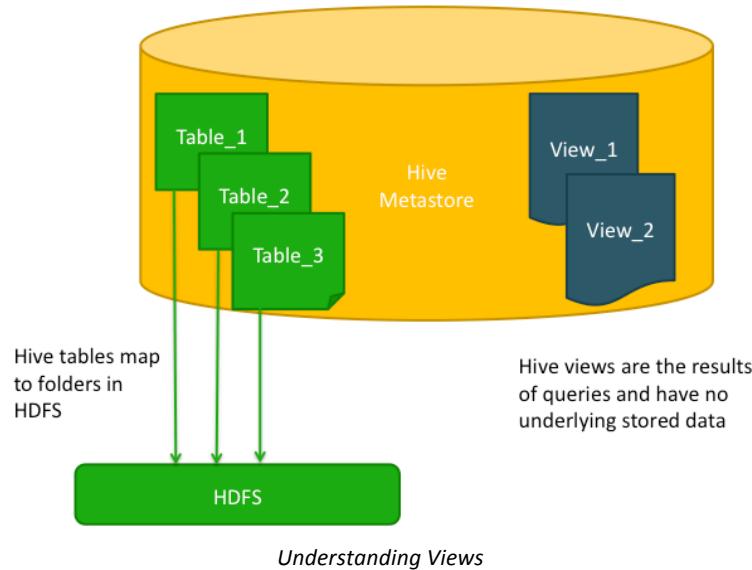
Note: Using a multi-file insert may seem a bit odd, but it is important to understand how Hive queries relate to underlying MapReduce jobs. In general, you can gain a lot of performance by running two tasks at the same time instead of running two separate MapReduce jobs.

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



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 the 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.

HDP Developer: Apache Pig and Hive

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. 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:

```
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 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 to the President:

```
from 2010_visitors
  select *
  where info_comment like "%CONGRESS%"
  order by lname;
```

HDP Developer: Apache Pig and Hive

Notice that 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 the best way to convert the above command into one or more MapReduce jobs at runtime .

The TRANSFORM Clause

You can write your own custom mappers or reducers and use them in Hive using the TRANSFORM clause. For example, the following example shows data being processed by a Python script named splitwords.py in a SELECT clause and then that result being processed by countwords.py.

```
add file splitwords.py;
add file countwords.py;

FROM (
    FROM mytable
    SELECT TRANSFORM(keywords) USING 'python splitwords.py'
    AS word, count
    CLUSTER BY word
) wc
INSERT OVERWRITE TABLE word_count
SELECT TRANSFORM (wc.word, wc.count)
USING 'python countwords.py'
AS word, count;
```

By default, columns will be transformed to STRING and delimited by a tab before being fed to the user script. The output of the script will be treated as tab-separated STRING columns.

You can achieve a similar result using the MAP and REDUCE clauses:

```
add file splitwords.py;
add file countwords.py;

FROM (
    FROM mytable
    MAP keywords USING 'python splitwords.py'
    AS word, count
    CLUSTER BY word
) wc
INSERT OVERWRITE TABLE word_count
REDUCE wc.word, wc.count USING 'python countwords.py'
AS word, count;
```



Note: Using MAP and REDUCE as an alias to SELECT TRANSFORM may not have the exact affect that you desire, since there is no guarantee that your specified script will be executed during a map or reduce phase. The end result of your query will likely be the same, but MAP does not force a map phase, and REDUCE does not force a reduce phase.

HDP Developer: Apache Pig and Hive

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;

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 can 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 the SQL standard.

HDP Developer: Apache Pig and Hive

Hive Analytics Functions

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;

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	Calculates 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 for a row r
NTILE	Distributes the rows in an ordered partition into a specified number of groups. For each row, <code>NTILE</code> returns the number of the group to which the row belongs

Lab: Advanced Hive Programming

Objective: To understand how some of the more advanced features of Hive work

See page 113 of the HDP Developer: Apache Pig and Hive Lab Booklet.

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	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 in the next section)

Using the `STORED AS` clause, you specify a file format when you create the table:

```
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` is short for serializer/deserializer and refers to how records 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 <code>Avro</code> schema
RegexSerDe	For using a regular expression to deserialize data
ColumnarSerDe	For columnar-based storage supported by <code>RCFiles</code>
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.

HDP Developer: Apache Pig and Hive

Using `SerDes` requires the `ROW FORMAT SERDE` clause. For example, the following table is for data stored in the `Avro` format:

```
CREATE TABLE emails (
  from_field string,
  sender string,
  email_body string)
ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS
INPUTFORMAT

'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
TBLPROPERTIES (
  'avro.schema.url'='hdfs://nn:8020/emailschema.avsc');
```

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;
```

And you can 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 64 MB in Hive 0.14. 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

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, and 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, a 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 column statistics, use the following syntax:

```
ANALYZE TABLE tablename COMPUTE STATISTICS FOR COLUMNS column_name_1, column_name_2,  
...
```

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 outputted 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 for a table by running the `DESCRIBE` command:

```
DESCRIBE FORMATTED tablename  
DESCRIBE EXTENDED tablename
```

You can also specify one or more partitions to view details for at the partition level:

```
DESCRIBE EXTENDED tablename PARTITION(part1=value1, part2=value2);
```

HDP Developer: Apache Pig and Hive

Hive Cost Based Optimization

In the first phase of Optiq and CBO in Hive, Optiq is used to reorder joins and to pick the right join algorithm to reduce query latency. Table cardinality and boundary statistics are used for this cost-based optimization.

Suppose you want to use CBO on a table named `tweets` that has columns named `sender` and `topic` that are commonly used in your Hive `JOIN` queries. First you need to analyze the table:

```
analyze table tweets compute statistics;
```

Second, compute the column statistics for `sender` and `topic`:

```
analyze table tweets compute statistics
  for columns sender, topic;
```

Third, set the following properties to enable CBO:

```
set hive.compute.query.using.stats=true;
set hive.cbo.enable=true;
set hive.stats.fetch.column.stats=true;
```

HDP Developer: Apache Pig and Hive

Vectorization



Vectorization + ORC files = huge breakthrough in Hive query performance

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, improving 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:

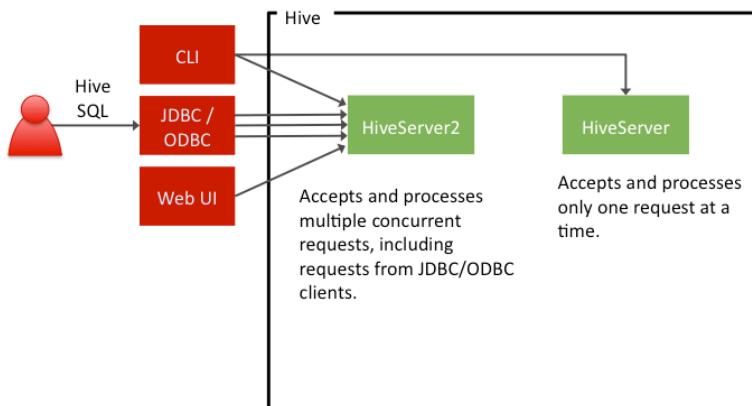
```
hive.vectorized.execution.enabled=true
```

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 magnitude.

Using HiveServer2



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.x ships with HiveServer2, a Thrift-based implementation that allows multiple concurrent connections and also supports Kerberos authentication.

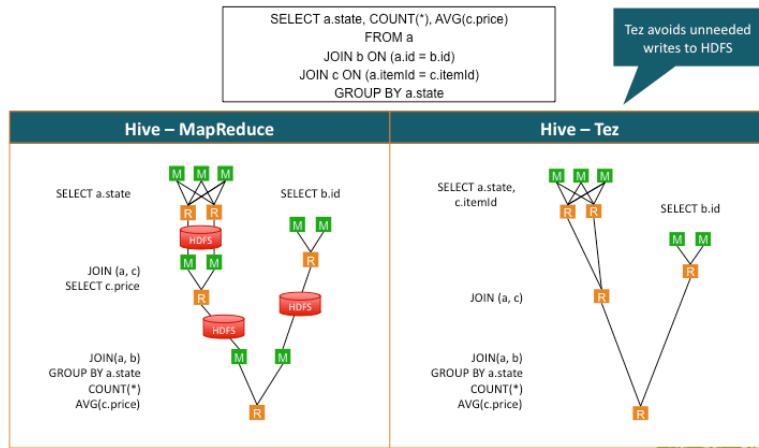
- A new HiveServer2 instance is started with the `hiveserver2` binary, or it can be run as a service
- Settings are defined in `hive-site.xml`, except for the bind host and port, which can be defined using the `HIVE_SERVER2_THRIFT_BIND_HOST` and `HIVE_SERVER2_THRIFT_PORT` environment variables. This allows you to run multiple HiveServer2 instances on the same machine

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

To use Tez for a Hive query, you need to define the following property in your Hive script or in `hive-site.xml`:

```
set hive.execution.engine=tez;
```

Note that this property is set to `mr` by default.

Demonstration: Hive Optimizations

Objective: To become familiar with some ways to optimize Hive

See page 121 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Hive Optimization Tips

- Divide data amongst different files that can be pruned out by using partitions, buckets, and skews
- 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

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 for 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.

HDP Developer: Apache Pig and Hive

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

Always set the following properties:

```
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;
```

HDP Developer: Apache Pig and Hive

Review Questions

- 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) 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);
```

- 4) 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);
```

-
-
- 5) Which Hive file format provides the best performance? _____
 - 6) What does `DAG` stand for? _____

Lab: Streaming Data with Hive and Python

Objective: Use a custom reducer script to optimize a Hive query

See page 125 of the HDP Developer: Apache Pig and Hive Lab Booklet.

Hadoop 2 and YARN

Lesson Objectives

This lesson covers the newer features of Hadoop 2, like YARN, HDFS Federation, and NameNode high availability.

After completing this lesson, students should be able to:

- Define HDFS Federation
- Explain how NameNode HA is implemented
- Define YARN

Additional Content

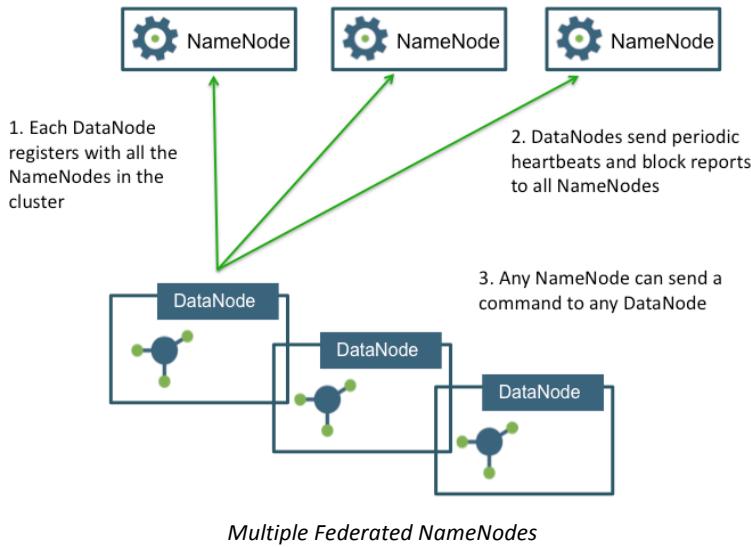
- Quiz: Lesson Review
- Lab: Running a YARN Application

About HDFS Federation

Hadoop 2.x introduces a 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



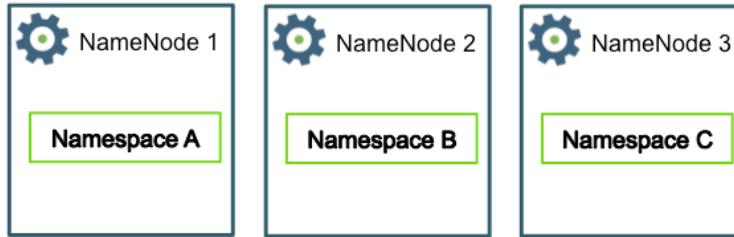
The NameNodes are federated: that is, the NameNodes are independent and don't require coordination with each other.

The DataNodes are used as common storage for blocks by all of the NameNodes. The NameNodes and DataNodes communicate as follows:

- Each DataNode registers with all of the NameNodes in the cluster
- DataNodes send periodic heartbeats and block reports to the NameNodes
- NameNodes send commands to the DataNodes

HDP Developer: Apache Pig and Hive

Multiple Namespaces



Multiple Namespaces

Benefits of multiple Namespaces include:

Scalability Having multiple independent Namespaces is what makes scaling possible in Hadoop 2.x

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 High Availability

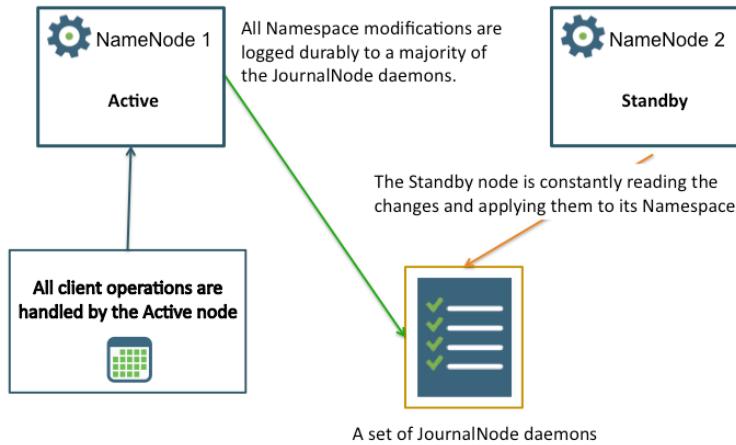
Prior to Hadoop 2.0, the NameNode was a single point of failure in an HDFS cluster. Each cluster had a single NameNode, and if that machine or process became unavailable, the cluster as a whole would be unavailable until the NameNode was either restarted or brought up on a separate machine.

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 occurs 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.

HDP Developer: Apache Pig and Hive

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.

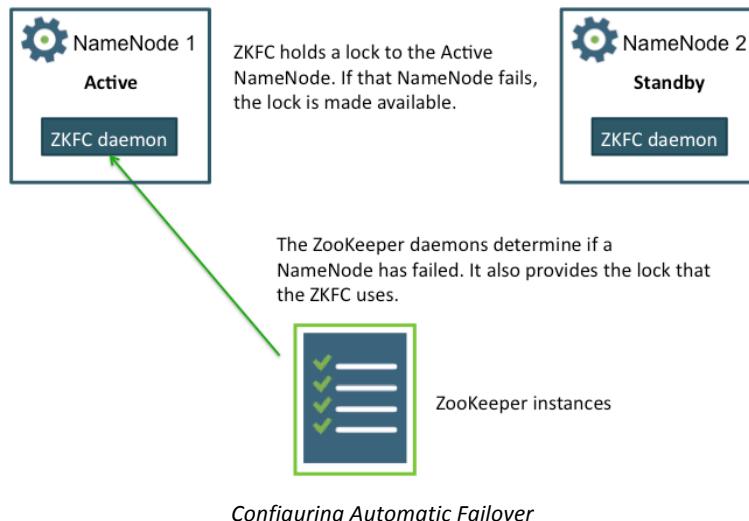
- Both nodes communicate with a group of separate daemons called JournalNodes
- All Namespace modifications are logged durably to a majority of the JournalNode daemons (hence the name Quorum)
- As the Standby Node sees the edits in the JournalNodes, it applies them to its own namespace



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.

HDP Developer: Apache Pig and Hive

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

About 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

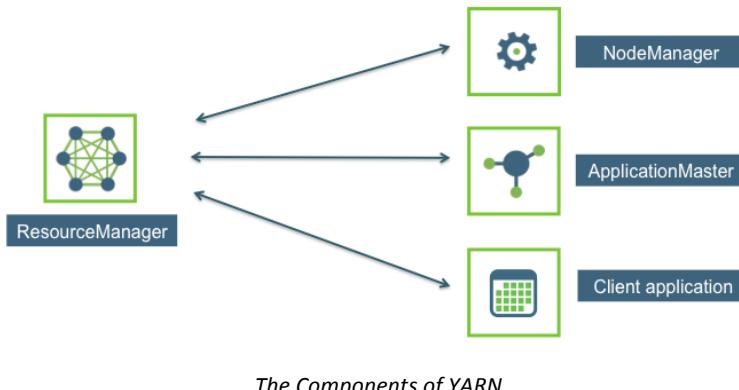
Tez	Improves the execution of MapReduce jobs
Slider	Deploy existing frameworks on YARN
Storm and Apache S4	For real-time computing
Spark	A MapReduce-like cluster computing framework designed for low-latency iterative jobs and interactive use from an interpreter
Open MPI	A high performance Message Passing Library that implements MPI-2
Apache Giraph	A graph-processing platform

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.x.

You can expect other computing frameworks to be developed once `YARN` becomes prevalent.

HDP Developer: Apache Pig and Hive

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:

ResourceManager A daemon process that allocates cluster resources to applications

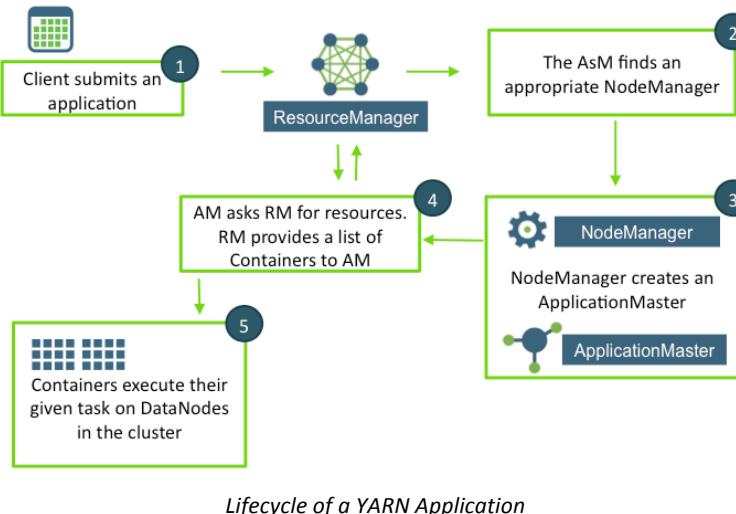
ApplicationMaster A per-application process that provides the runtime for executing applications

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.

HDP Developer: Apache Pig and Hive

Lifecycle of a YARN Application

A YARN application has the following lifecycle:

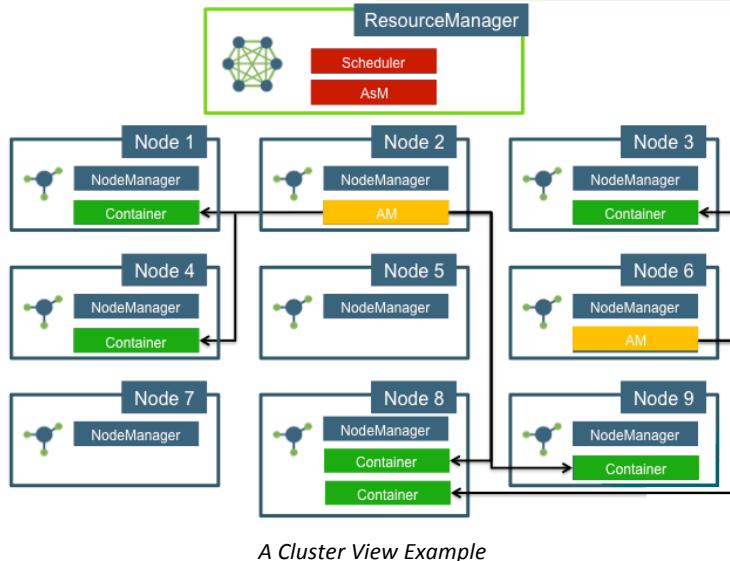


- It all starts with a client submitting a new Application Request to the Resource Manager (RM)
- The ApplicationsManager (AsM) finds an available DataNode on the cluster that is not too busy
- That node's NodeManager (NM) creates an instance of the ApplicationMaster (AM)
- The AM then sends a request 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
- 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.

HDP Developer: Apache Pig and Hive

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**? _____

Answers:

- 1) Two
- 2) Four
- 3) Certainly
- 4) Yes. It all depends on the availability of resources on a node.

Review Questions

- 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 three main components of YARN? _____

- 4) What happens if a Container fails to complete its task in a YARN application?

Lab: Running a YARN Application

Objective: To run a YARN application

See page 129 of the HDP Developer: Apache Pig and Hive Lab Booklet.

HDP Developer: Apache Pig and Hive

Defining Workflow with Oozie

Lesson Objectives

This lesson covers how to implement a Hadoop workflow using the Apache Oozie framework.

After completing this lesson, students should be able to:

- Describe Oozie
- Describe an Oozie Coordinator Job

Additional Content

- Quiz: Lesson Review
- Lab: Defining an Oozie Workflow

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:

Oozie Workflow A collection of actions (defined in a `workflow.xml` file)

Oozie Coordinator A recurring workflow (defined in a `coordinator.xml` file)

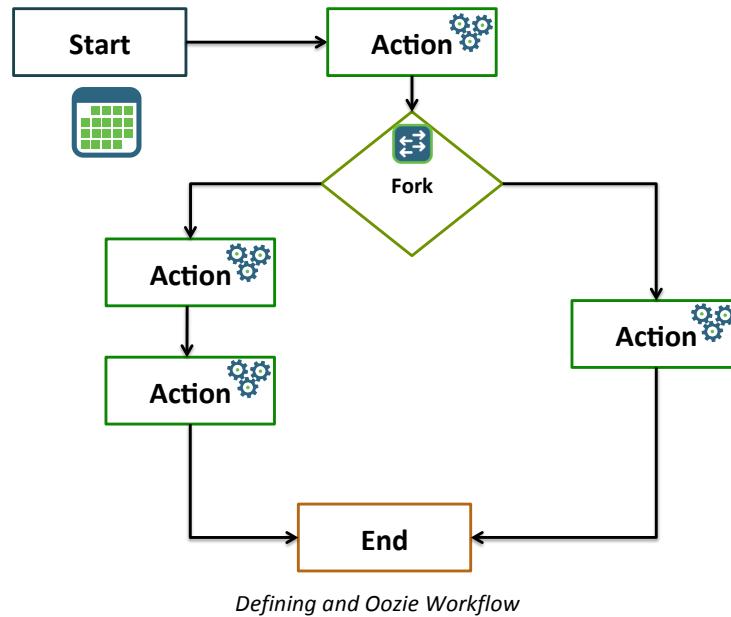
Behind the scenes, Oozie is a Java web application that runs in a Tomcat instance. You run Oozie as a service then start workflows using the `oozie` command.

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



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. A fork node splits one path into multiple paths. A join node waits until every path of a previous fork node arrives to it
Action nodes	For executing a job or task

HDP Developer: Apache Pig and Hive

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`
- A workflow looks almost identical to the `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 the 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. View the Oozie Documentation for a complete list of functions

HDP Developer: Apache Pig and Hive

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 needs to contain the various properties for connecting to Hive
- This action compresses the output of the `map` tasks

HDP Developer: Apache Pig and Hive

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>
      </property>
```

HDP Developer: Apache Pig and Hive

```
        <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.

HDP Developer: Apache Pig and Hive

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
```

The `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`

Fork and Join Nodes

Oozie has fork and join nodes for controlling workflow. For example:

```
<fork name="forking">
    <path start="firstparalleljob"/>
    <path start="secondparalleljob"/>
</fork>
<action name="firstparalleljob">
    <map-reduce>
        ...
        </map-reduce>
        <ok to="joining"/>
        <error to="kill"/>
    </action>
<action name="secondparalleljob">
    <map-reduce>
        ...
        </map-reduce>
        <ok to="joining"/>
        <error to="kill"/>
    </action>
<join name="joining" to="nextaction"/>
```

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:

`coordinator.xml` The definition of the Coordinator application

`coordinator.properties` For defining the job's properties

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 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, the `/myapp` workflow will execute.

HDP Developer: Apache Pig and Hive

Review Questions

- 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

See page 133 of the HDP Developer: Apache Pig and Hive Lab Booklet.

HDP Developer: Apache Pig and Hive

Hadoop Streaming

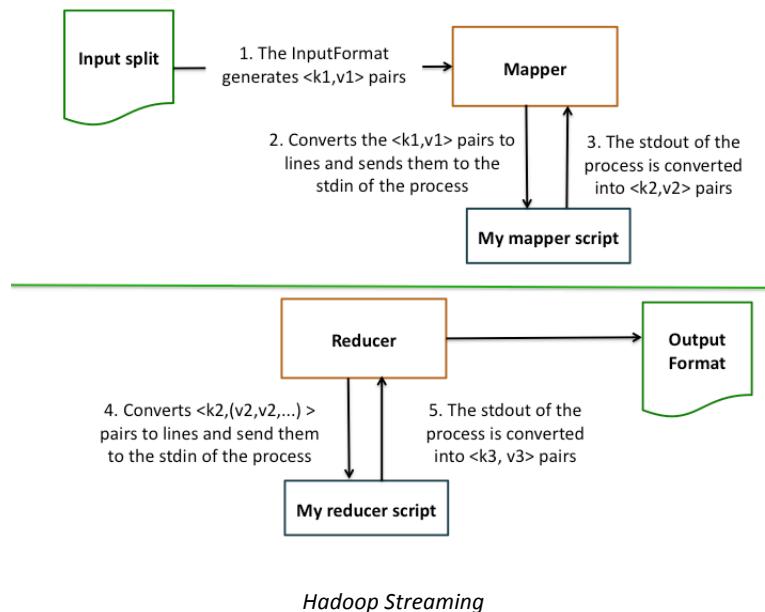
Lesson Objectives

This lesson covers an overview of the streaming capabilities of Hadoop.

After completing this lesson, students should be able to:

- Hadoop Streaming
- Running a Hadoop Streaming Job

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:

- The MapReduce job starts as any other job, with the input splits sending key/value pairs to a `map` task
- 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
- 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
- Similarly, the Streaming reducer converts the input key/values pairs into lines of text and sends them to the `stdin` of the reducer process
- 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

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'
```

HDP Developer: Apache Pig and Hive

Appendix A: Lesson Review Quiz Answers

Understanding Hadoop: Review Answers

- 1) What are 1,024 petabytes known as?

Answer: 1,024 petabytes = 1 Exabyte

- 2) What are 1,024 exabytes known as?

Answer: 1,024 Exabytes = 1 Zettabyte

And for what it's worth:

1,024 Zettabytes = 1 Yottabyte

1,024 Yottabytes = 1 Brontobyte

1,024 Brontabytes = 1 Geopbyte

- 3) List the three Vs of big data:

Answer: Variety, Volume, and Velocity

- 4) Sentiment is one of the six key types of big data. List the other five:

Answer: Clickstream

Sensor and machine data

Location-based (geographic) data

Server logs

Text (web pages, emails, documents, etc.)

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

Answer: Flume is commonly used for importing Twitter feeds into a Hadoop cluster

- 6) What technology might you use to define, store, and share the schemas of your big data stored in Hadoop?

Answer: HCatalog is designed to easily store and share schemas for your big data.

- 7) What are the two main new components in Hadoop 2.x?

Answer: HDFS Federation and YARN.

The Hadoop Distributed File System (HDFS): Review Answers

- 1) Which component of HDFS is responsible for maintaining the namespace of the distributed filesystem?

Answer: NameNode

- 2) What is the default file-replication factor in HDFS?

Answer: 3

- 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.

Answer: False. A file's data in HDFS never passes through the NameNode. Client applications read and write directly from the DataNodes.

- 4) Which property is used to specify the block size of a file stored in HDFS?

Answer: dfs.blocksize

- 5) The NameNode maintains the namespace of the filesystem using which two sets of files?

Answer: The fsimage_N and edits_N files

- 6) What does the following command do?

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

Answer: Recursively lists the contents of /user/thomas in HDFS and all of its subfolders

- 7) What does the following command do?

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

Answer: Lists the file and folders in /user/thomas, but not recursively. (The files in the subfolders of /user/thomas are not listed.)

Inputting Data into HDFS: Review Answers

- 1) What tool would work best for importing data from a relational database into HDFS?

Answer: Sqoop

- 2) What tool would work best for putting a file on your local filesystem into HDFS?

Answer: The Hadoop client (`hadoop fs -put` command)

- 3) List the three main components of a typical Flume agent:

Answer: A Flume agent consists of a source, channel, and sink

- 4) What is the default number of map tasks for a Sqoop job?

Answer: Four map tasks by default

- 5) How do you specify a different number of mappers in a Sqoop job?

Answer: The `-m` option is for specifying the number of mappers.

- 6) What is the purpose of the \$CONDITIONS value in the WHERE clause of a Sqoop query?

Answer: 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

The MapReduce Framework: Review Answers

- 1) What are the three main phases of a MapReduce job?

Answer: Map phase, shuffle/sort phase, and reduce phase

- 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?

Answer: The pairs coming into the reducer will look like `<integer, (string, string, string, ...)>`

- 3) What happens if all the `<key, value>` pairs output by a mapper do not fit into the memory of the mapper?

Answer: 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) What determines the number of mappers of a MapReduce job?

Answer: The number of mappers is determined by the input splits.

- 5) What determines the number of reducers of a MapReduce job?

Answer: You get to choose the number of reducers.

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

Answer: False. The keys are sorted, but the values are not.

Introduction to Pig: Review Answers

- 1) List two Pig commands that cause a logical plan to execute:

Answer: STORE, DUMP, and ILLUSTRATE all cause a logical plan to execute

- 2) Which Pig command stores the output of a relation into a folder in HDFS?

Answer: STORE

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;

Answer: prices: {symbol: chararray, date: chararray, price: double, volume: int}

- 4) A = group prices by symbol; _____

Answer: 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) B = foreach prices generate symbol as x, volume as y;

Answer: 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}
```

HDP Developer: Apache Pig and Hive

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

Answer: 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 = foreach prices generate symbol..price;

Answer: 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) Write a Pig relation that only contains prices with a volume greater than 3,000:

Answer: E = filter prices by volume > 3000;

Advanced Pig Programming: Review Answers

- 1) If a relation is sorted using `ORDER BY` and the resulting MapReduce job runs with three reducers, how is the output actually sorted?

Answer: The `ORDER BY` command generates a total ordering, meaning that the records will be sorted across all three reducers, with the output of reducer 1 containing the first set of sorted records, reducer 2 containing the second set, and so on.

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`

```
(CASE  
WHEN volume > 3000 THEN volume  
WHEN volume <= 3000 THEN -1  
END) AS high_volume;
```

Answer: The output of `F` looks like:

```
(-1)  
(400000)  
(-1)  
(3800)  
(-1)  
(-1)
```

- 3) `G = distinct prices;`

Answer: 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.

HDP Developer: Apache Pig and Hive

```
4) H = GROUP prices BY symbol;  
I = foreach H {  
    J = filter prices by volume > 3000;  
    GENERATE group, SUM(J.price);  
};
```

Answer: 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) What is the benefit of the using 'replicated' clause in a Pig join?

Answer: 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) Why is filtering and projecting a relation early a performance benefit in Pig?

Answer: 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.

Hive Programming: Review Answers

- 1) A Hive table consists of a schema stored in the Hive _____ and data stored in _____.

Answer: A Hive table consists of a schema stored in the Hive *metastore* and data stored in *HDFS*.

- 2) True or False: The Hive metastore requires an underlying SQL database.

Answer: True. Hive uses an in-memory database called Derby by default, but you can configure Hive to use any SQL database.

- 3) What happens to the underlying data of a Hive-managed table when the table is dropped?

Answer: The data and folders are deleted from HDFS.

- 4) True or False: A Hive external table must define a LOCATION.

Answer: False. An external table can use an external location, but it can also use the Hive warehouse folder.

- 5) List three different ways data can be loaded into a Hive table:

Answer: 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) When would you use a skewed table?

Answer: Skewed tables make sense when your data is naturally skewed, where a small number of columns contain a disproportionate amount of records.

- 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?

Answer: Within /apps/hive/warehouse/movies will be subfolders named /genre=value. For example, /genre=scifi, /genre=comedy, /genre=drama, etc.

HDP Developer: Apache Pig and Hive

- 8) Explain the output of the following query:

```
select * from movies order by title;
```

Answer: The `order by` clause causes the output to be totally ordered by title across all output files.

- 9) What does the following Hive query compute?

```
from mytable
  select explode(ngrams(sentences(val),3,100)) as myresult;
```

Answer: The `ngram` output from this query is called a trigram, because the result will be sets of three words. The 100 argument specifies you want the top 100 trigrams from this dataset.

- 10) What does the following Hive query compute?

```
from mytable
  select explode(context_ngrams(sentences(val),
    array("I","liked",null),10)) as myresult;
```

Answer: The output of this query is the top 10 words in the dataset that follow the phrase "I liked."

Using HCatalog: Review Answers

- 1) Where does HCatalog store its schema information?

Answer: In the Hive metastore.

- 2) List three programming frameworks that can readily access an HCatalog schema:

Answer: Pig, Hive, and Java MapReduce programs can all easily use the schemas shared by HCatalog.

- 3) What Java class does Pig use to load data from an HCatalog table?

Answer: The HCatLoader class; more specifically,
org.apache.hcatalog.pig.HCatLoader.

- 4) True or False: HCatalog is now merged with Hive.

Answer: True. HCatalog is now a part of Hive.

Advanced Hive Programming: Review Answers

- 1) What is the benefit of performing two insert queries in the same Hive command?

Answer: Two queries that would normally require two MapReduce jobs can be combined and accomplished in a single MapReduce job.

- 2) True or False: Hive views are materialized when they are defined.

Answer: False. Hive views are not materialized until they are used in another query.

- 3) 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);
```

Answer: 200. The `OVER` clause causes the group aggregation to not occur, so each `employees` row will be output. There will only be 15 salary values, the maximum salary in each department.

- 4) 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);
```

Answer: The result will contain the `fname`, `lname`, and the average salary of this employee and the five preceding employees whose salaries are less than or equal to the current employee.

- 5) Which Hive file format provides the best performance?

Answer: ORC files are a part of the Stinger Initiative and provide the best performance for Hive queries.

- 6) What does DAG stand for?

Answer: DAG = Directed Acyclic Graph. Hive queries are processed into a series of jobs that look like a DAG.

Hadoop 2 and YARN: Review Answers

- 1) True or False: A NameNode can contain multiple namespaces.

Answer: False. A NameNode can represent only a single namespace.

- 2) What is the key benefit of the new YARN framework?

Answer: Hadoop jobs are no longer restricted to MapReduce. With YARN, any type of computing paradigm can be implemented to run on Hadoop.

- 3) What are the three main components of YARN?

Answer: ResourceManager, NodeManager, and ApplicationMaster

- 4) What happens if a Container fails to complete its task in a YARN application?

Answer: It is up to the ApplicationMaster to request a new Container from the ResourceManager and attempt the task again.

Defining Workflow with Oozie: Review Answers

- 1) What are the two main capabilities of Oozie?

Answer: Oozie Workflow, for defining Hadoop job workflows; and the Oozie coordinator, for scheduling recurring workflows.

- 2) What file is required to be a part of an Oozie workflow?

Answer: Each Oozie workflow must contain a `workflow.xml` configuration file.

- 3) List three common Oozie workflow actions:

Answer: `<pig>`, `<hive>`, and `<map-reduce>`

- 4) What two types of events can be used to trigger an Oozie coordinator job?

Answer: 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.