Course Chapters

▪ Introduction ▪ Hadoop Fundamentals	Course Introduction
▪ **Introduction to Pig** ▪ Basic Data Analysis with Pig ▪ Processing Complex Data with Pig ▪ Multi-Dataset Operations with Pig ▪ Pig Troubleshooting and Optimization	**Data ETL and Analysis With Pig**
▪ Introduction to Impala and Hive ▪ Querying With Impala and Hive ▪ Impala and Hive Data Management ▪ Data Storage and Performance	Introduction to Impala and Hive
▪ Relational Data Analysis With Impala and Hive ▪ Working with Impala ▪ Analyzing Text and Complex Data with Hive ▪ Hive Optimization ▪ Extending Hive	Data Analysis With Impala and Hive
▪ Choosing the Best Tool for the Job ▪ Conclusion	Course Conclusion

Introduction to Pig

In this chapter, you will learn

- **The key features Pig offers**
- **How organizations use Pig for data processing and analysis**
- **How to use Pig interactively and in batch mode**

Chapter Topics

Introduction to Pig | **Data ETL and Analysis With Pig**

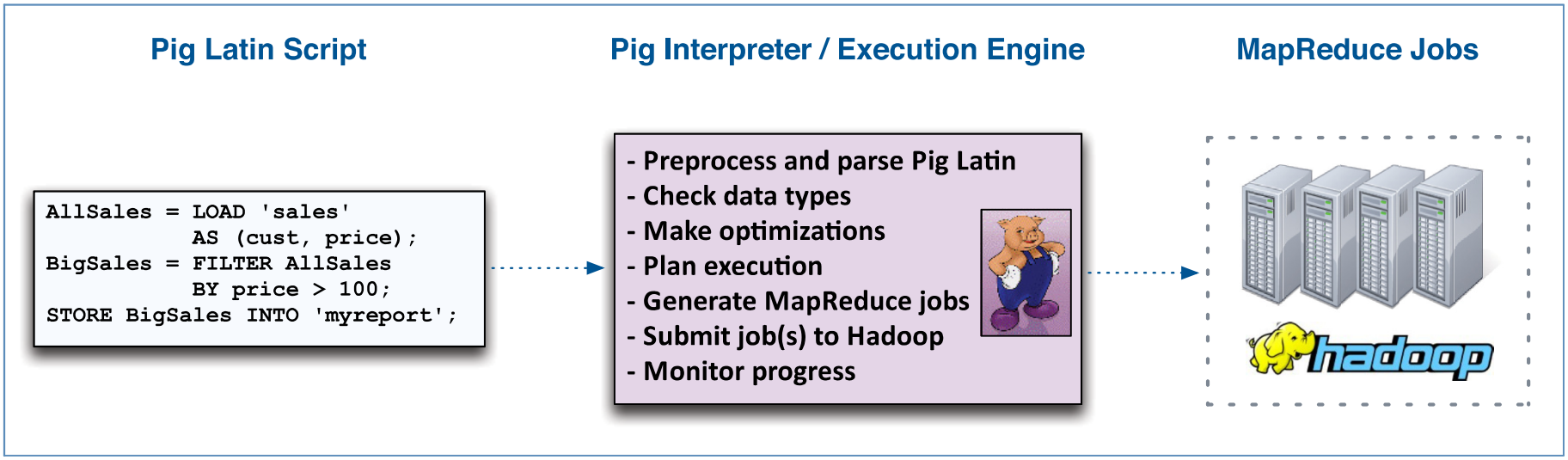
- **What is Pig?**
- Pig's Features
- Pig Use Cases
- Interacting with Pig
- Conclusion

Apache Pig Overview

- **Apache Pig is a platform for data analysis and processing on Hadoop**
  - It offers an alternative to writing MapReduce code directly
- **Originally developed as a research project at Yahoo**
  - Goals: flexibility, productivity, and maintainability
  - Now an open-source Apache project

The Anatomy of Pig

- **Main components of Pig**
  - The data flow language (Pig Latin)
  - The interactive shell where you can type Pig Latin statements (Grunt)
  - The Pig interpreter and execution engine



Where to Get Pig

- **CDH is the easiest way to install Hadoop and Pig**
  - A Hadoop distribution which includes HDFS, MapReduce, Spark, Pig, Hive, Impala, Sqoop, HBase, and other Hadoop ecosystem components
  - Available as RPMs, Ubuntu/Debian/SuSE packages, or a tarball
  - Simple installation
  - 100% free and open source
- **Installation is outside the scope of this course**
  - Cloudera offers a training course for System Administrators, *Cloudera Administrator Training for Apache Hadoop*

Chapter Topics

Introduction to Pig | **Data ETL and Analysis With Pig**

- What is Pig?
- **Pig's Features**
- Pig Use Cases
- Interacting with Pig
- Conclusion

Pig Features

- **Pig is an alternative to writing low-level MapReduce code**
- **Many features enable sophisticated analysis and processing**
  - HDFS manipulation
  - UNIX shell commands
  - Relational operations
  - Positional references for fields
  - Common mathematical functions
  - Support for custom functions and data formats
  - Complex data structures

Chapter Topics

Introduction to Pig | **Data ETL and Analysis With Pig**

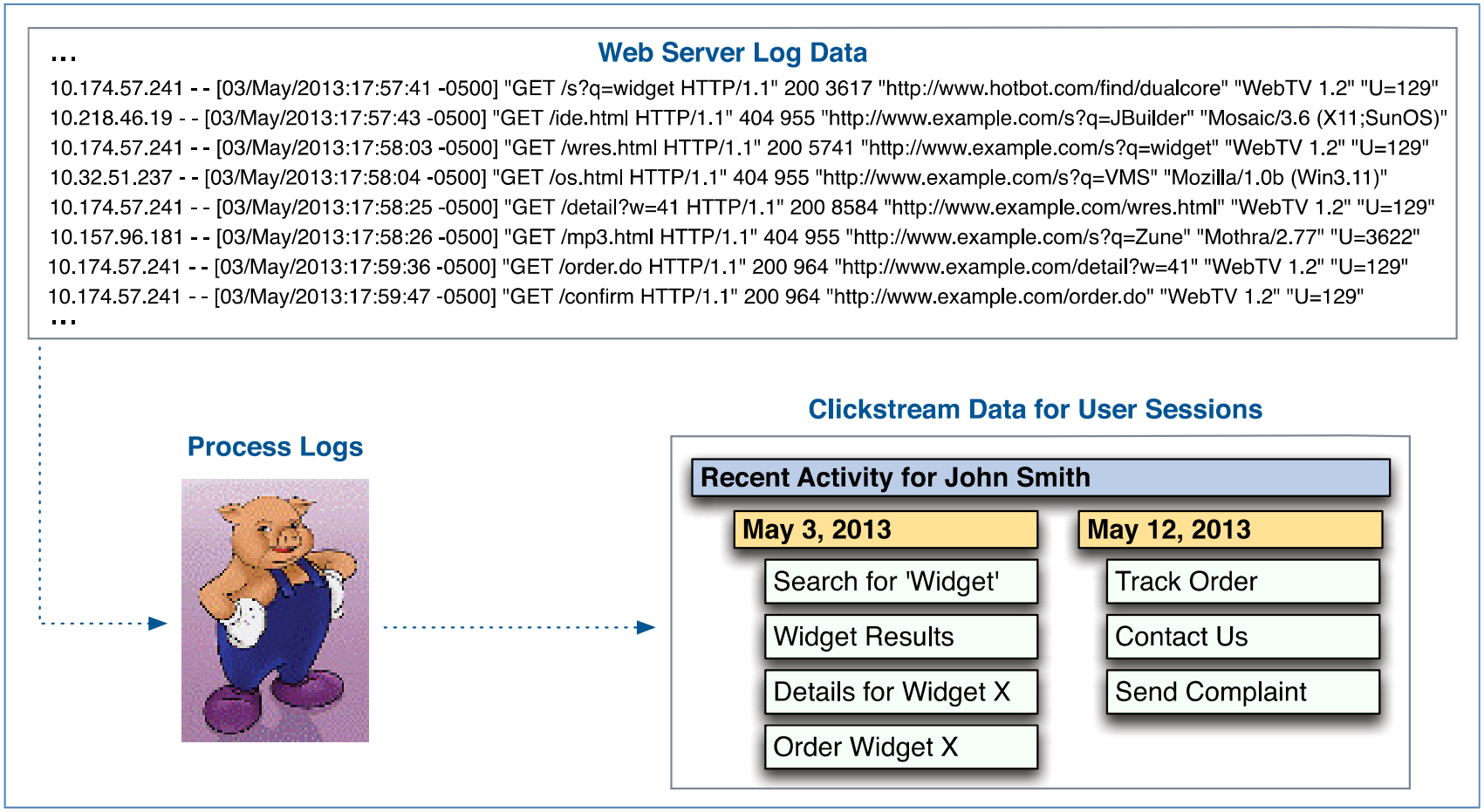
- What is Pig?
- Pig's Features
- **Pig Use Cases**
- Interacting with Pig
- Conclusion

How Are Organizations Using Pig?

- **Many organizations use Pig for data analysis**
  - Finding relevant records in a massive data set
  - Querying multiple data sets
  - Calculating values from input data
- **Pig is also frequently used for data processing**
  - Reorganizing an existing data set
  - Joining data from multiple sources to produce a new data set

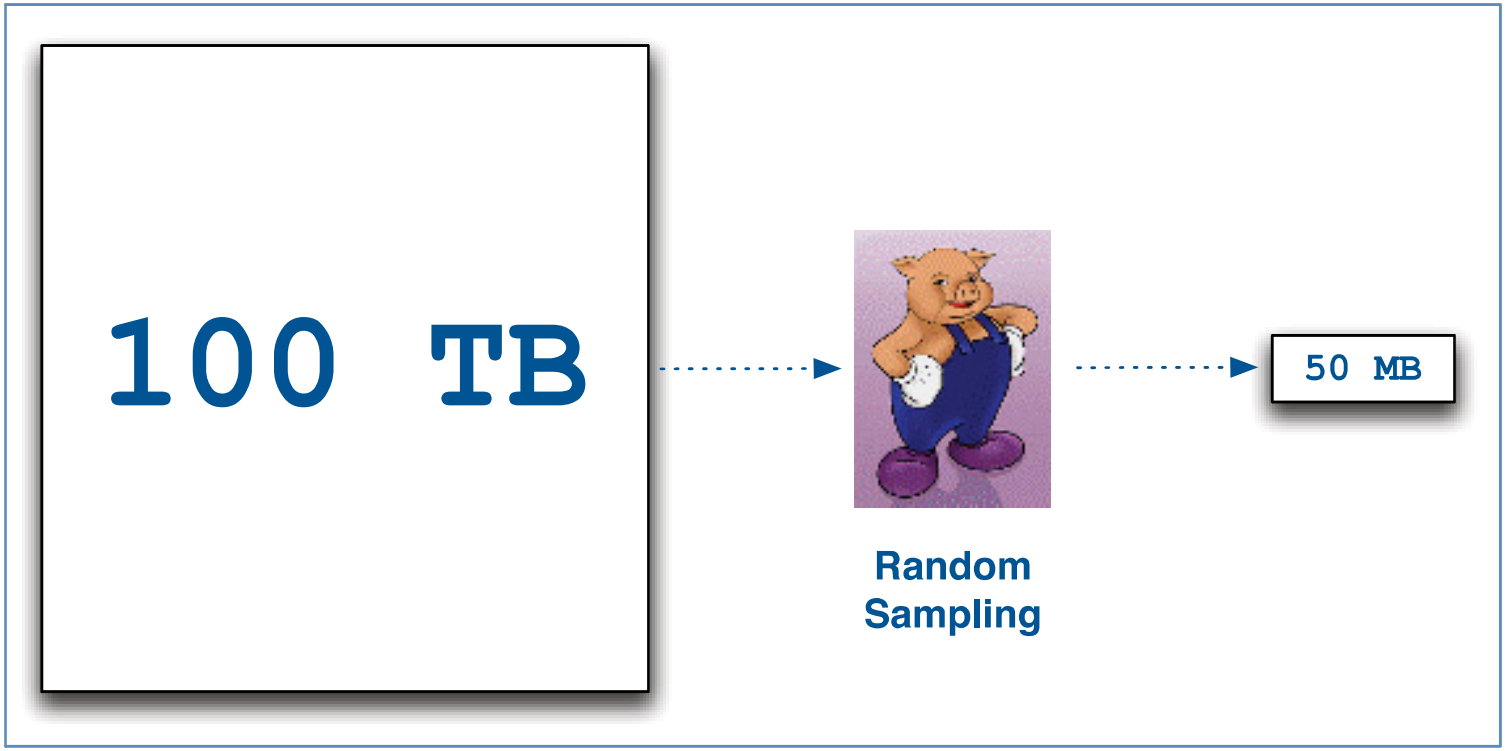
Use Case: Web Log Sessionization

- **Pig can help you extract valuable information from Web server log files**



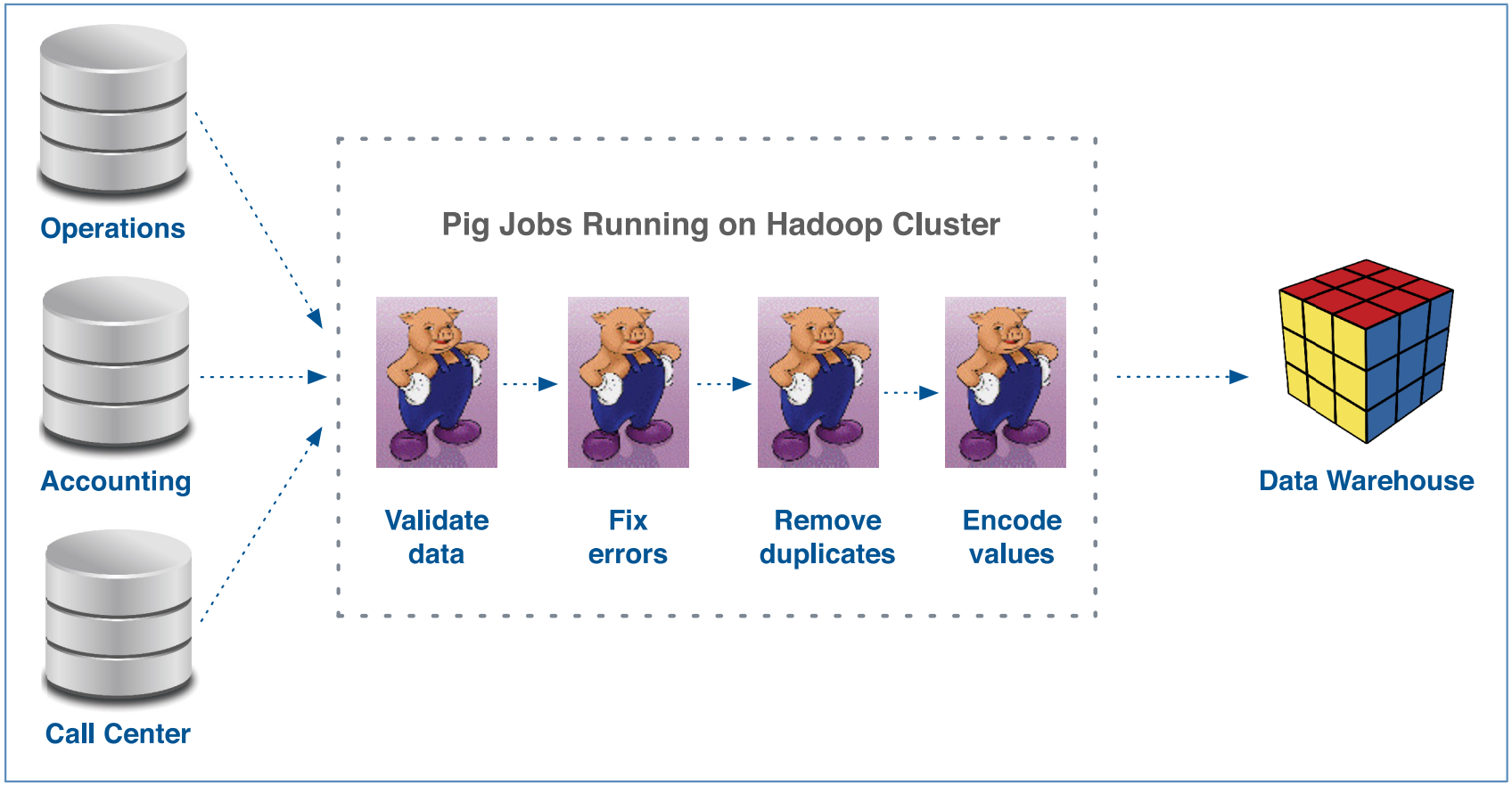
Use Case: Data Sampling

- **Sampling can help you explore a representative portion of a large data set**
  - Allows you to examine this portion with tools that do not scale well
  - Supports faster iterations during development of analysis jobs



Use Case: ETL Processing

- **Pig is also widely used for Extract, Transform, and Load (ETL) processing**



Chapter Topics

Introduction to Pig | **Data ETL and Analysis With Pig**

- What is Pig?
- Pig's Features
- Pig Use Cases
- **Interacting with Pig**
- Conclusion

Using Pig Interactively

- **You can use Pig interactively, via the Grunt shell**
  - Pig interprets each Pig Latin statement as you type it
  - Execution is delayed until output is required
  - Very useful for ad hoc data inspection
- **Example of how to start, use, and exit Grunt**

```
$ pig
grunt> allsales = LOAD 'sales' AS (name, price);
grunt> bigsales = FILTER allsales BY price > 100;
grunt> STORE bigsales INTO 'myreport';
grunt> quit;
```

- **Can also execute a Pig Latin statement from the UNIX shell via the `-e` option**

Interacting with HDFS

- **You can manipulate HDFS with Pig, via the `fs` command**

```
grunt> fs -mkdir sales/;
grunt> fs -put europe.txt sales/;
grunt> allsales = LOAD 'sales' AS (name, price);
grunt> bigsales = FILTER allsales BY price > 100;
grunt> STORE bigsales INTO 'myreport';
grunt> fs -getmerge myreport/ bigsales.txt;
```

Interacting with UNIX

- **The `sh` command lets you run UNIX programs from Pig**

```
grunt> sh date;
Wed Nov 12 06:39:13 PST 2014
grunt> fs -ls;                    -- lists HDFS files
grunt> sh ls;                     -- lists local files
```

Running Pig Scripts

- **A Pig script is simply Pig Latin code stored in a text file**
  - By convention, these files have the `.pig` extension
- **You can run a Pig script from within the Grunt shell via the `run` command**
  - This is useful for automation and batch execution

```
grunt> run salesreport.pig;
```

- **It is common to run a Pig script directly from the UNIX shell**

```
$ pig salesreport.pig
```

MapReduce and Local Modes

- **As described earlier, Pig turns Pig Latin into MapReduce jobs**
  - Pig submits those jobs for execution on the Hadoop cluster
- **It is also possible to run Pig in 'local mode' using the `-x` flag**
  - This runs jobs on the *local machine* instead of the cluster
  - Local mode uses the local filesystem instead of HDFS
  - Can be helpful for testing before deploying a job to production

```
$ pig -x local                          -- interactive

$ pig -x local salesreport.pig  -- batch
```

Client-Side Log Files

- **If a job fails, Pig may produce a log file to explain why**
  - These log files are typically produced in your current working directory
    - On the local (client) machine

Chapter Topics

Introduction to Pig | **Data ETL and Analysis With Pig**

- What is Pig?
- Pig's Features
- Pig Use Cases
- Interacting with Pig
- **Conclusion**

Essential Points

- **Pig offers an alternative to writing MapReduce code directly**
  - Pig interprets Pig Latin code in order to create MapReduce jobs
  - It then submits these jobs to the Hadoop cluster
- **You can execute Pig Latin code interactively through Grunt**
  - Pig delays job execution until output is required
- **It is also common to store Pig Latin code in a script for batch execution**
  - Allows for automation and code reuse

Bibliography

The following offer more information on topics discussed in this chapter

- **Apache Pig Web Site**
  - `http://pig.apache.org/`
- **Process a Million Songs with Apache Pig**
  - `http://tiny.cloudera.com/dac03a`
- **Powered By Pig**
  - `http://tiny.cloudera.com/poweredbypig`
- **LinkedIn: User Engagement Powered By Apache Pig and Hadoop**
  - `http://tiny.cloudera.com/dac03c`
- ***Programming Pig*** **(O'Reilly book)**
  - `http://tiny.cloudera.com/programmingpig`

cloudera®
Ask Bigger Questions

Basic Data Analysis with Pig

Chapter 4

Course Chapters

▪ Introduction ▪ Hadoop Fundamentals	Course Introduction
▪ Introduction to Pig ▪ **Basic Data Analysis with Pig** ▪ Processing Complex Data with Pig ▪ Multi-Dataset Operations with Pig ▪ Pig Troubleshooting and Optimization	**Data ETL and Analysis With Pig**
▪ Introduction to Impala and Hive ▪ Querying With Impala and Hive ▪ Impala and Hive Data Management ▪ Data Storage and Performance	Introduction to Impala and Hive
▪ Relational Data Analysis With Impala and Hive ▪ Working with Impala ▪ Analyzing Text and Complex Data with Hive ▪ Hive Optimization ▪ Extending Hive	Data Analysis With Impala and Hive
▪ Choosing the Best Tool for the Job ▪ Conclusion	Course Conclusion

Basic Data Analysis with Pig

In this chapter, you will learn

- **The basic syntax of Pig Latin**
- **How to load and store data using Pig**
- **Which simple data types Pig uses to represent data**
- **How to sort and filter data in Pig**
- **How to use many of Pig's built-in functions for data processing**

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- **Pig Latin Syntax**
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly-used Functions
- Conclusion
- Hands-On Exercise: Using Pig for ETL Processing

Pig Latin Overview

- **Pig Latin is a *data flow* language**
  - The flow of data is expressed as a sequence of statements
- **The following is a simple Pig Latin script to load, filter, and store data**

```
allsales = LOAD 'sales' AS (name, price);

bigsales = FILTER allsales BY price > 999; -- in US cents

/*
 * Save the filtered results into a new
 * directory, below my home directory.
 */
STORE bigsales INTO 'myreport';
```

Pig Latin Grammar: Keywords

- **Pig Latin keywords are highlighted here in blue text**
  - Keywords are reserved – you cannot use them to name things

```
allsales = LOAD 'sales' AS (name, price);

bigsales = FILTER allsales BY price > 999; -- in US cents

/*
 * Save the filtered results into a new
 * directory, below my home directory.
 */
STORE bigsales INTO 'myreport';
```

Pig Latin Grammar: Identifiers (1)

- **Identifiers are the names assigned to fields and other data structures**

```
allsales = LOAD 'sales' AS (name, price);

bigsales = FILTER allsales BY price > 999; -- in US cents

/*
 * Save the filtered results into a new
 * directory, below my home directory.
 */
STORE bigsales INTO 'myreport';
```

Pig Latin Grammar: Identifiers (2)

- **Identifiers must conform to Pig's naming rules**
- **An identifier must always begin with a letter**
  - This may only be followed by letters, numbers, or underscores

Valid	`x`	`q1`	`q1_2013`	`MyData`
Invalid	`4`	`price$`	`profit%`	`_sale`

Pig Latin Grammar: Comments

- **Pig Latin supports two types of comments**
  - Single line comments begin with `--`
  - Multi-line comments begin with `/*` and end with `*/`

```
allsales = LOAD 'sales' AS (name, price);

bigsales = FILTER allsales BY price > 999; -- in US cents

/*
 * Save the filtered results into a new
 * directory, below my home directory.
 */
STORE bigsales INTO 'myreport';
```

Case-Sensitivity in Pig Latin

- **Whether case is significant in Pig Latin depends on context**
- **Keywords (shown here in blue text) *are not* case-sensitive**
  - Neither are operators (such as `AND`, `OR`, or `IS NULL`)
- **Identifiers and paths (shown here in red text) *are* case-sensitive**
  - So are function names (such as `SUM` or `COUNT`) and constants

```
allsales = LOAD 'sales' AS (name, price);

bigsales = FILTER allsales BY price > 999;

STORE bigsales INTO 'myreport';
```

Common Operators in Pig Latin

- **Many commonly-used operators in Pig Latin are familiar to SQL users**
  - Notable difference: Pig Latin uses == and != for comparison

Arithmetic	Comparison	Null	Boolean
+	==	IS NULL	AND
-	!=	IS NOT NULL	OR
*	<		NOT
/	>		
%	<=		
	>=		

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- **Loading Data**
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly-used Functions
- Conclusion
- Hands-On Exercise: Using Pig for ETL Processing

Basic Data Loading in Pig

- **Pig's default loading function is called `PigStorage`**
  - The name of the function is implicit when calling `LOAD`
  - `PigStorage` assumes text format with tab-separated columns
- **Consider the following file in HDFS called `sales`**
  - The two fields are separated by tab characters

```
Alice      2999
Bob        3625
Carlos     2764
```

- **This example loads data from the above file**

```
allsales = LOAD 'sales' AS (name, price);
```

Data Sources: File and Directories

- **The previous example loads data from a file named `sales`**

```
allsales = LOAD 'sales' AS (name, price);
```

- **Since this is not an absolute path, it is relative to your home directory**
  - Your home directory in HDFS is typically `/user/youruserid/`
  - Can also specify an absolute path (e.g., `/dept/sales/2012/q4`)
- **The path can also refer to a directory**
  - In this case, Pig will recursively load all files in that directory
  - File patterns ("globs") are also supported

```
allsales = LOAD 'sales_200[5-9]' AS (name, price);
```

Specifying Column Names During Load

- **The previous example also assigns names to each column**

```
allsales = LOAD 'sales' AS (name, price);
```

- **Assign column names is not required**
  - This can be useful when exploring a new dataset
  - Refer to fields by position (`$0` is first, `$1` is second, `$53` is 54th, etc.)

```
allsales = LOAD 'sales';
```

Using Alternate Column Delimiters

- **You can specify an alternate delimiter as an argument to `PigStorage`**
- **This example shows how to load comma-delimited data**
  - Note that this is a single statement

```
allsales = LOAD 'sales.csv' USING PigStorage(',') AS
(name, price);
```

- **Or to load pipe-delimited data without specifying column names**

```
allsales = LOAD 'sales.txt' USING PigStorage('|');
```

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- **Simple Data Types**
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly-used Functions
- Conclusion
- Hands-On Exercise: Using Pig for ETL Processing

Simple Data Types in Pig

- **Pig supports several basic data types**
  - Similar to those in most databases and programming languages
- **Pig treats fields of unspecified type as an array of bytes**
  - Called the `bytearray` type in Pig

```
allsales = LOAD 'sales' AS (name, price);
```

List of Simple Data Types

- **There are eight data types in Pig for simple values**

Name	Description	Example Value
`int`	Whole numbers	`2013`
`long`	Large whole numbers	`5,365,214,142L`
`float`	Decimals	`3.14159F`
`double`	Very precise decimals	`3.141592653589793238 46`
`boolean`*	True or false values	`true`
`datetime`*	Date and time	`2013-05-30T14:52:39.000-04:00`
`chararray`	Text strings	`Alice`
`bytearray`	Raw bytes (e.g. any data)	`N/A`

* Not available in older versions of Pig

Specifying Data Types in Pig

- **Pig will do its best to determine data types based on context**
  - For example, you can calculate sales commission as `price * 0.1`
  - In this case, Pig will assume that this value is of type `double`
- **However, it is better to specify data types explicitly when possible**
  - Helps with error checking and optimizations
  - Easiest to do this upon load using the format *fieldname* **:** *type*

```
allsales = LOAD 'sales' AS (name:chararray, price:int);
```

- **Choosing the right data type is important to avoid loss of precision**
- **Important: Avoid using floating point numbers to represent money!**

How Pig Handles Invalid Data

- **When encountering invalid data, Pig substitutes `NULL` for the value**
  - For example, an `int` field containing the value `Q4`
- **The `IS NULL` and `IS NOT NULL` operators test for null values**
  - Note that `NULL` is not the same as the empty string `''`
- **You can use these operators to filter out bad records**

```
hasprices = FILTER Records BY price IS NOT NULL;
```

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- **Field Definitions**
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly-used Functions
- Conclusion
- Hands-On Exercise: Using Pig for ETL Processing

Key Data Concepts in Pig

- **Relational databases have tables, rows, columns, and fields**
- **We will use the following data to illustrate Pig's equivalents**
  - Assume this data was loaded from a tab-delimited text file as before

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Fields

- **A *single* element of data is called a field**
  - It corresponds to one of the eight data types seen earlier

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Tuples

- **A *collection* of values is called a *tuple***
  - Fields within a tuple are ordered, but need not all be of the same type

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Bags

- **A *collection* of tuples is called a *bag***
- **Tuples within a bag are unordered by default**
  - The field count and types may vary between tuples in a bag

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Relations

- **A relation is simply a bag with an assigned name (alias)**
  - Most Pig Latin statements create a new relation
- **A typical script loads one or more datasets into relations**
  - Processing creates new relations instead of modifying existing ones
  - The final result is usually also a relation, stored as output

```
allsales = LOAD 'sales' AS (name, price);
bigsales = FILTER allsales BY price > 999;
STORE bigsales INTO 'myreport';
```

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- **Data Output**
- Viewing the Schema
- Filtering and Sorting Data
- Commonly-used Functions
- Conclusion
- Hands-On Exercise: Using Pig for ETL Processing

Data Output in Pig

- **The command used to handle output depends on its destination**
  - `DUMP`: sends output to the screen
  - `STORE`: sends output to disk (HDFS)
- **Example of `DUMP` output, using data from the file shown earlier**
  - The parentheses and commas indicate tuples with multiple fields

```
(Alice,2999,us)
(Bob,3625,ca)
(Carlos,2764,mx)
(Dieter,1749,de)
(Étienne,2368,fr)
(Fredo,5637,it)
```

Storing Data with Pig

- **The `STORE` command is used to store data to HDFS**
  - Similar to `LOAD`, but *writes* data instead of reading it
  - The output path is the name of a directory
    - The directory must not yet exist
- **As with `LOAD`, the use of PigStorage is implicit**
  - The field delimiter also has a default value (tab)

```
STORE bigsales INTO 'myreport';
```

  - You may also specify an alternate delimiter

```
STORE bigsales INTO 'myreport' USING PigStorage(',');
```

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- **Viewing the Schema**
- Filtering and Sorting Data
- Commonly-used Functions
- Conclusion
- Hands-On Exercise: Using Pig for ETL Processing

Viewing the Schema with DESCRIBE

- The **DESCRIBE** command shows the structure of the data, including names and types
- The following Grunt session shows an example

```
grunt>  allsales = LOAD 'sales' AS (name:chararray,
        price:int);
grunt>  DESCRIBE allsales;

allsales: {name: chararray,price: int}
```

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- **Filtering and Sorting Data**
- Commonly-used Functions
- Conclusion
- Hands-On Exercise: Using Pig for ETL Processing

Filtering in Pig Latin

- The `FILTER` keyword extracts tuples matching the specified criteria

```
bigsales = FILTER allsales BY price > 3000;
```

allsales

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

`price > 3000` →

bigsales

name	price	country
Bob	3625	ca
Fredo	5637	it

Filtering by Multiple Criteria

- **You can combine criteria with AND and OR**

```
somesales = FILTER allsales BY name == 'Dieter' OR (price >
3500 AND price < 4000);
```

allsales

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

somesales

name	price	country
Bob	3625	ca
Dieter	1749	de

Name is Dieter, or price is greater than 3500 and less than 4000

Aside: String Comparisons in Pig Latin

- **The == operator is supported for *any* type in Pig Latin**
  - This operator is used for exact comparisons

```
alices = FILTER allsales BY name == 'Alice';
```

- **Pig Latin supports pattern matching through Java's *regular expressions***
  - This is done with the `MATCHES` operator

```
a_names = FILTER allsales BY name MATCHES 'A.*';
```

```
spammers = FILTER senders BY email_addr
           MATCHES '.*@example\\.com$';
```

Field Selection in Pig Latin

- **Filtering extracts rows, but sometimes we need to extract columns**
  - This is done in Pig Latin using the `FOREACH` and `GENERATE` keywords

```
twofields = FOREACH allsales GENERATE amount, trans_id;
```

allsales

salesperson	amount	trans_id
Alice	2999	107546
Bob	3625	107547
Carlos	2764	107548
Dieter	1749	107549
Étienne	2368	107550
Fredo	5637	107550

twofields

amount	trans_id
2999	107546
3625	107547
2764	107548
1749	107549
2368	107550
5637	107550

Generating New Fields in Pig Latin

- **The `FOREACH` and `GENERATE` keywords can also be used to *create* fields**
  - For example, you could create a new field based on price

```
t = FOREACH allsales GENERATE price * 0.07;
```

- **It is possible to name such fields**

```
t = FOREACH allsales GENERATE price * 0.07 AS tax;
```

- **And you can also specify the data type**

```
t = FOREACH allsales GENERATE price * 0.07 AS tax:float;
```

Eliminating Duplicates

- **`DISTINCT` eliminates duplicate records in a bag**
  - **All fields** must be equal to be considered a duplicate

```
unique_records = DISTINCT all_alices;
```

`all_alices`

firstname	lastname	country
Alice	Smith	us
Alice	Jones	us
Alice	**Brown**	**us**
Alice	**Brown**	**us**
Alice	Brown	ca

`unique_records`

firstname	lastname	country
Alice	Smith	us
Alice	Jones	us
Alice	**Brown**	**us**
Alice	Brown	ca

Controlling Sort Order

- **Use `ORDER...BY` to sort the records in a bag in ascending order**
  - Add **`DESC`** to sort in descending order instead
  - Take care to specify a schema – data type affects how data is sorted!

```
sortedsales = ORDER allsales BY country DESC;
```

allsales

name	price	country
Alice	29.99	us
Bob	36.25	ca
Carlos	27.64	mx
Dieter	17.49	de
Étienne	23.68	fr
Fredo	56.37	it

sortedsales

name	price	country
Alice	29.99	**us**
Carlos	27.64	**mx**
Fredo	56.37	**it**
Étienne	23.68	**fr**
Dieter	17.49	**de**
Bob	36.25	**ca**

Limiting Results

- **As in SQL, you can use `LIMIT` to reduce the number of output records**

```
somesales = LIMIT allsales 10;
```

- **Beware! Record ordering is random unless specified with `ORDER BY`**
  - Use `ORDER BY` and `LIMIT` together to find top-N results

```
sortedsales = ORDER allsales BY price DESC;
top_five = LIMIT sortedsales 5;
```

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- **Commonly-used Functions**
- Conclusion
- Hands-On Exercise: Using Pig for ETL processing

Built-in Functions

- **These are just a sampling of Pig's many built-in functions**

Function Description	Example Invocation	Input	Output
Convert to uppercase	`UPPER(country)`	`uk`	`UK`
Remove leading/trailing spaces	`TRIM(name)`	`_Bob_`	`Bob`
Return a random number	`RANDOM()`		`0.4816132 6652569`
Round to closest whole number	`ROUND(price)`	`37.19`	`37`
Return chars between two positions	`SUBSTRING(name, 0, 2)`	`Alice`	`Al`

- **You can use these with the `FOREACH..GENERATE` keywords**

```
rounded = FOREACH allsales GENERATE ROUND(price);
```

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly-used Functions
- **Conclusion**
- Hands-On Exercise: Using Pig for ETL processing

Essential Points

- **Pig Latin supports many of the same operations as SQL**
  - Though Pig's approach is quite different
  - Pig Latin loads, transforms, and stores data in a series of steps
- **The default delimiter for both input and output is the tab character**
  - You can specify an alternate delimiter as an argument to `PigStorage`
- **Specifying the names and types of fields is not required**
  - But it can improve performance and readability of your code

Bibliography

The following offer more information on topics discussed in this chapter

- **Pig Latin Basics**
  - `http://tiny.cloudera.com/piglatinbasics`
- **Pig Latin Built-In Functions**
  - `http://tiny.cloudera.com/piglatinbuiltin`
- **Documentation for Java Regular Expression Patterns**
  - `http://tiny.cloudera.com/javaregex`

Chapter Topics

Basic Data Analysis with Pig | **Data ETL and Analysis With Pig**

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly-used Functions
- Conclusion
- **Hands-On Exercise: Using Pig for ETL processing**

Hands-On Exercise: Using Pig for ETL processing

- **In this Hands-On Exercise, you will write Pig Latin code to perform basic ETL processing tasks on data related to Dualcore's online advertising campaigns**
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®
Ask Bigger Questions

Processing Complex Data with Pig

Chapter 5

Course Chapters

▪ Introduction ▪ Hadoop Fundamentals	Course Introduction
▪ Introduction to Pig ▪ Basic Data Analysis with Pig ▪ **Processing Complex Data with Pig** ▪ Multi-Dataset Operations with Pig ▪ Pig Troubleshooting and Optimization	**Data ETL and Analysis With Pig**
▪ Introduction to Impala and Hive ▪ Querying With Impala and Hive ▪ Impala and Hive Data Management ▪ Data Storage and Performance	Introduction to Impala and Hive
▪ Relational Data Analysis With Impala and Hive ▪ Working with Impala ▪ Analyzing Text and Complex Data with Hive ▪ Hive Optimization ▪ Extending Hive	Data Analysis With Impala and Hive
▪ Choosing the Best Tool for the Job ▪ Conclusion	Course Conclusion

Processing Complex Data with Pig

In this chapter, you will learn

- **How Pig uses bags, tuples, and maps to represent complex data**
- **The techniques Pig provides for grouping and ungrouping data**
- **How to use aggregate functions in Pig Latin**
- **How to iterate through records in complex data structures**

Chapter Topics

Processing Complex Data with Pig | **Data ETL and Analysis With Pig**

- **Storage Formats**
- Complex/Nested Data Types
- Grouping
- Built-in Functions for Complex Data
- Iterating Grouped Data
- Conclusion
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Storage Formats

- **We have seen that `PigStorage` loads and stores data**
  - Uses a delimited text file format

```
allsales = LOAD 'sales' AS (name, price);
```

  - The default delimiter (tab) can be easily changed

```
allsales = LOAD 'sales' USING PigStorage(',')
AS (name, price) ;
```

- **Sometimes you need to load or store data in other formats**

Other Supported Formats

- **Here are a few of Pig's built-in functions for loading data in other formats**
  - It is also possible to implement a custom loader by writing Java code

Name	Loads Data From
`TextLoader`	Text files (entire line as one field)
`JsonLoader`	Text files containing JSON-formatted data
`BinStorage`	Files containing binary data
`HBaseLoader`	HBase, a scalable NoSQL database built on Hadoop

Load and Store Functions

- **Some functions load data, some store data, and some do both**

Load Function	Equivalent Store Function
`PigStorage`	`PigStorage`
`TextLoader`	`None`
`JsonLoader`	`JsonStorage`
`BinStorage`	`BinStorage`
`HBaseLoader`	`HBaseStorage`

Chapter Topics

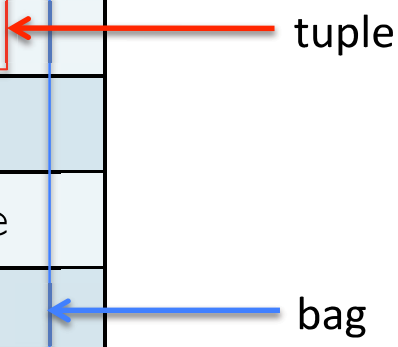
Processing Complex Data with Pig | **Data ETL and Analysis With Pig**

- Storage Formats
- **Complex/Nested Data Types**
- Grouping
- Built-in Functions for Complex Data
- Iterating Grouped Data
- Conclusion
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Pig's Complex Data Types: Tuple and Bag

- **We have already seen two of Pig's three complex data types**
  - A tuple is a collection of values
  - A bag is a collection of tuples

trans_id	total	salesperson
107546	2999	Alice
107547	3625	Bob
107548	2764	Carlos
107549	1749	Dieter
107550	2368	Étienne
107551	5637	Fredo



Pig's Complex Data Types: Map

- **Pig also supports another complex type: Map**
  - A map associates a `chararray` (key) to another data element (value)

trans_id	amount	salesperson	sales_details
107546	2498	Alice	date → 12-02-2013 SKU → 40155 store → MIA01
107547	3625	Bob	date → 12-02-2013 SKU → 3720 store → STL04 coupon → DEC13
107548	2764	Carlos	date → 12-03-2013 SKU → 76102 store → NYC15

Representing Complex Types in Pig

- **It is important to know how to define and recognize these types in Pig**

Type	Definition
Tuple	Comma-delimited list inside parentheses: `('107546', 2498, 'Alice')`
Bag	Braces surround comma-delimited list of tuples: `{('107546', 2498, 'Alice'), ('107547', 3625, 'Bob')}`
Map	Brackets surround comma-delimited list of pairs; keys and values separated by #: `['store'#'MIA01','location'#'Coral Gables']`

Loading and Using Complex Types (1)

- **Complex data types can be used in any Pig field**
- **The following example show how a bag is stored in a text file**

Example: Transaction ID, amount, items sold (a bag of tuples)

```
107550    2498    {('40120', 1999), ('37001', 499)}
```

- **Here is the corresponding LOAD statement specifying the schema**

```
details = LOAD 'salesdetail' AS (
trans_id:chararray, amount:int,
items_sold:bag
   {item:tuple (SKU:chararray, price:int)});
```

Loading and Using Complex Types (2)

- **The following example show how a map is stored in a text file**

Example: Customer name, credit account details (map) , year account opened

```
Eva     [creditlimit#5000,creditused#800]     2012
```

- **Here is the corresponding `LOAD` statement specifying the schema**

```
credit = LOAD 'customer_accounts' AS (
name:chararray, account:map[], year:int);
```

Referencing Map Data

- Consider a file with the following data

```
Bob     [salary#52000,age#52]
```

- And loaded with the following schema

```
details = LOAD 'data' AS (name:chararray, info:map[]);
```

- Here is the syntax for referencing data within the map and bag

```
salaries = FOREACH details GENERATE info#'salary';
```

Chapter Topics

Processing Complex Data with Pig | **Data ETL and Analysis With Pig**

- Storage Formats
- Complex/Nested Data Types
- **Grouping**
- Built-in Functions for Complex Data
- Iterating Grouped Data
- Conclusion
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Grouping Records By a Field (1)

- **Sometimes you need to group records by a given field**
  - For example, so you can calculate commissions for each employee

```
Alice       729
Bob         3999
Alice       27999
Carol       32999
Carol       4999
```

- **Use `GROUP BY` to do this in Pig Latin**
  - The new relation has one record per unique value in the specified field

```
byname = GROUP sales BY name;
```

Grouping Records By a Field (2)

- **The new relation always contains two fields**

```
grunt> byname = GROUP sales BY name;
grunt> DESCRIBE byname;
byname: {group: chararray,sales: {(name:
chararray,price: int)}}
```

- **The first field is *literally* named `group` in all cases**
  - Contains the value from the field specified in `GROUP BY`
- **The second field is named after the relation specified in `GROUP BY`**
  - It is a bag containing one tuple for each corresponding value

Grouping Records By a Field (3)

- **The example below shows the data after grouping**

```
grunt> byname = GROUP sales BY name;
grunt> DUMP byname;
(Bob,{(Bob,3999)})
(Alice,{(Alice,729),(Alice,27999)})
(Carol,{(Carol,32999),(Carol,4999)})
```

group field

sales field

Input Data (**sales**)

```
Alice    729
Bob      3999
Alice    27999
Carol    32999
Carol    4999
```

Using GROUP BY to Aggregate Data

- **Aggregate functions create one output value from multiple input values**
  - For example, to calculate total sales by employee
  - Usually applied to grouped data

```
grunt> byname = GROUP sales BY name;
grunt> DUMP byname;
(Bob,{(Bob,3999)})
(Alice,{(Alice,729),(Alice,27999)})
(Carol,{(Carol,32999),(Carol,4999)})

grunt> totals = FOREACH byname GENERATE
                                group, SUM(sales.price);
grunt> dump totals;
(Bob,3999)
(Alice,28728)
(Carol,37998)
```

Grouping Everything Into a Single Record

- We just saw that `GROUP BY` creates one record for each unique value
- `GROUP ALL` puts *all* data into one record

```
grunt> grouped = GROUP sales ALL;
grunt> DUMP grouped;
(all,{(Alice,729),(Bob,3999),(Alice,27999),
(Carol,32999),(Carol,4999)})
```

Using GROUP ALL to Aggregate Data

- **Use GROUP ALL when you need to aggregate one or more columns**
  - For example, to calculate total sales for all employees

```
grunt> grouped = GROUP sales ALL;
grunt> DUMP grouped;
(all,{(Alice,729),(Bob,3999),(Alice,27999),(Carol,32999),
(Carol,4999)})

grunt> totals = FOREACH grouped GENERATE SUM(sales.price);
grunt> dump totals;
(70725)
```

Removing Nesting in Data

- **Some operations in Pig, like grouping, produce nested data structures**

```
grunt> byname = GROUP sales BY name;
grunt> DUMP byname;
(Bob,{(Bob,3999)})
(Alice,{(Alice,729),(Alice,27999)})
(Carol,{(Carol,32999),(Carol,4999)})
```

- **Grouping can be useful to supply data to aggregate functions**
- **However, sometimes you want to work with a "flat" data structure**
  - The `FLATTEN` operator removes a level of nesting in data

An Example of FLATTEN

- The following shows the nested data and what **FLATTEN** does to it

```
grunt> byname = GROUP sales BY name;
grunt> DUMP byname;
(Bob,{(Bob,3999)})
(Alice,{(Alice,729),(Alice,27999)})
(Carol,{(Carol,32999),(Carol,4999)})

grunt> flat = FOREACH byname GENERATE group,
       FLATTEN(sales.price);
grunt> DUMP flat;
(Bob,3999)
(Alice,729)
(Alice,27999)
(Carol,32999)
(Carol,4999)
```

Chapter Topics

Processing Complex Data with Pig | **Data ETL and Analysis With Pig**

- Storage Formats
- Complex/Nested Data Types
- Grouping
- **Built-in Functions for Complex Data**
- Iterating Grouped Data
- Conclusion
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Pig's Built-In Aggregate Functions

- **Pig has built-in support for other aggregate functions besides `SUM`**
- **Examples:**
  - `AVG`: Calculates the average (mean) of all values
  - `MIN`: Returns the smallest value
  - `MAX`: Returns the largest value
- **Pig has two built-in functions for counting records**
  - `COUNT`: Returns the number of **non-null** elements in the bag
  - `COUNT_STAR`: Returns the number of **all** elements in the bag

Other Notable Built-in Functions

- **Here are a some other useful Pig functions**
  - See the Pig documentation for a complete list

Function	Description
`DIFF`	Finds tuples that appear in only one of two supplied bags
`IsEmpty`	Used with `FILTER` to match bags or maps that contain no data
`SIZE`	Returns the size of the field (definition of size varies by data type)
`TOKENIZE`	Splits a text string (chararray) into a bag of individual words

Chapter Topics

Record Iteration

- **We have seen that `FOREACH . . . GENERATE` iterates through records**
- **The goal is to transform records to produce a new relation**
  - Sometimes to select only certain columns

```
price_column_only = FOREACH sales GENERATE price;
```

  - Sometimes to create new columns

```
taxes = FOREACH sales GENERATE price * 0.07;
```

  - Sometimes to invoke a function on the data

```
totals = FOREACH grouped GENERATE SUM(sales.price);
```

Nesting the FOREACH Keyword

- **A variation on FOREACH applies a set of operations to each record**
  - This is often used to apply a series of transformations in a group
- **This is called a nested FOREACH**
  - Allows only relational operations (e.g., LIMIT, FILTER, ORDER BY)
  - GENERATE must be the last line in the block

Nested FOREACH Example (1)

- **Our input data contains a list of employee job titles and corresponding salaries**
- **Goal: identify the three highest salaries within each title**

Input Data

```
President   192000
Director    152500
Director    161000
Director    167000
Director    165000
Director    147000
Engineer    92300
Engineer    85000
Engineer    83000
Engineer    81650
Engineer    82100
Engineer    87300
Engineer    76000
Manager     87000
Manager     81000
Manager     75000
Manager     79000
Manager     67500
```

Nested FOREACH Example (2)

- **First load the data from the file**
- **Next, group employees by title**
  - Assigned to new relation `title_group`

Input Data (excerpt)

```
President   192000
Director    152500
Director    161000
...
Engineer    92300
...
Manager     67500
```

```
employees = LOAD 'data' AS (title:chararray, salary:int);
title_group = GROUP employees BY title;

top_salaries = FOREACH title_group {
   sorted = ORDER employees BY salary DESC;
   highest_paid = LIMIT sorted 3;
   GENERATE group, highest_paid;
};
```

Nested FOREACH Example (3)

- **The nested FOREACH iterates through every record in the group (i.e., each job title)**
  - It sorts each record in that group in descending order of salary
  - It then selects the top three
  - GENERATE outputs the title and salaries

Input Data (excerpt)

```
President   192000
Director    152500
Director    161000
...
Engineer    92300
...
Manager     67500
```

```
employees = LOAD 'data' AS (title:chararray, salary:int);
title_group = GROUP employees BY title;

top_salaries = FOREACH title_group {
   sorted = ORDER employees BY salary DESC;
   highest_paid = LIMIT sorted 3;
   GENERATE group, highest_paid;
};
```

Nested FOREACH Example (4)

Code (LOAD statement removed for brevity)

```
title_group = GROUP employees BY title;

top_salaries = FOREACH title_group {
  sorted = ORDER employees BY salary DESC;
  highest_paid = LIMIT sorted 3;
  GENERATE group, highest_paid;
};
```

Input Data (excerpt)

```
President   192000
Director    152500
Director    161000
...
Engineer    92300
...
Manager     67500
```

Output produced by **DUMP top_salaries**

```
(Director,{(Director,167000),(Director,165000),(Director,161000)})
(Engineer,{(Engineer,92300),(Engineer,87300),(Engineer,85000)})
(Manager,{(Manager,87000),(Manager,81000),(Manager,79000)})
(President,{(President,192000)})
```

Chapter Topics

Processing Complex Data with Pig | **Data ETL and Analysis With Pig**

- Storage Formats
- Complex/Nested Data Types
- Grouping
- Built-in Functions for Complex Data
- Iterating Grouped Data
- **Conclusion**
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Essential Points

- **Pig has three complex data types: `tuple`, `bag`, and `map`**
  - A map is simply a collection of key/value pairs
- **These structures can contain simple types like `int` or `chararray`**
  - But they can also contain complex data types
  - Nested data structures are common in Pig
- **Pig provides methods for grouping and ungrouping data**
  - You can remove a level of nesting using the `FLATTEN` operator
- **Pig offers several built-in aggregate functions**

Chapter Topics

Processing Complex Data with Pig | **Data ETL and Analysis With Pig**

- Storage Formats
- Complex/Nested Data Types
- Grouping
- Built-in Functions for Complex Data
- Iterating Grouped Data
- Conclusion
- **Hands-On Exercise: Analyzing Ad Campaign Data with Pig**

Hands-On Exercise: Analyzing Ad Campaign Data with Pig

- **In this Hands-On Exercise, you will analyze data from Dualcore's online ad campaign**
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®
Ask Bigger Questions

Multi-Dataset Operations with Pig

Chapter 6

Course Chapters

▪ Introduction ▪ Hadoop Fundamentals	Course Introduction
▪ Introduction to Pig ▪ Basic Data Analysis with Pig ▪ Processing Complex Data with Pig ▪ **Multi-Dataset Operations with Pig** ▪ Pig Troubleshooting and Optimization	**Data ETL and Analysis With Pig**
▪ Introduction to Impala and Hive ▪ Querying With Impala and Hive ▪ Impala and Hive Data Management ▪ Data Storage and Performance	Introduction to Impala and Hive
▪ Relational Data Analysis With Impala and Hive ▪ Working with Impala ▪ Analyzing Text and Complex Data with Hive ▪ Hive Optimization ▪ Extending Hive	Data Analysis With Impala and Hive
▪ Choosing the Best Tool for the Job ▪ Conclusion	Course Conclusion

Multi-Dataset Operations with Pig

In this chapter, you will learn

- **How we can use grouping to combine data from multiple sources**
- **What types of join operations Pig supports and how to use them**
- **How to concatenate records to produce a single data set**
- **How to split a single data set into multiple relations**

Chapter Topics

Multi-Dataset Operations with Pig | **Data ETL and Analysis With Pig**

- **Techniques for Combining Data Sets**
- Joining Data Sets in Pig
- Set Operations
- Splitting Data Sets
- Conclusion
- Hands-On Exercise: Analyzing Disparate Data Sets with Pig

Overview of Combining Data Sets

- **So far, we have concentrated on processing single data sets**
  - Valuable insight often results from combining multiple data sets
- **Pig offers several techniques for achieving this**
  - Using the `GROUP` operator with multiple relations
  - Joining the data as you would in SQL
  - Performing set operations like `CROSS` and `UNION`
- **We will cover each of these in this chapter**

Example Data Sets (1)

- **Most examples in this chapter will involve the same two data sets**
- **The first is a file containing information about Dualcore's stores**
- **There are two fields in this relation**
  1. `store_id:chararray` (unique key)
  2. `name:chararray` (name of the city in which the store is located)

Stores

```
A   Anchorage
B   Boston
C   Chicago
D   Dallas
E   Edmonton
F   Fargo
```

Example Data Sets (2)

- **Our other data set is a file containing information about Dualcore's salespeople**
- **This relation contains three fields**
  1. `person_id:int` (unique key)
  2. `name:chararray` (salesperson name)
  3. `store_id:chararray` (refers to store)

Stores

```
A    Anchorage
B    Boston
C    Chicago
D    Dallas
E    Edmonton
F    Fargo
```

Salespeople

```
1    Alice      B
2    Bob        D
3    Carlos     F
4    Dieter     A
5    Étienne    F
6    Fredo      C
7    George     D
8    Hannah     B
9    Irina      C
10   Jack
```

Grouping Multiple Relations

- **We previously learned about the GROUP operator**
  - Groups values in a relation based on the specified field(s)
- **The GROUP operator can also group *multiple* relations**
  - In this case, using the synonymous `COGROUP` operator is preferred

```
grouped = COGROUP stores BY store_id, salespeople BY store_id;
```

- **This collects values from both data sets into a new relation**
  - As before, the new relation is keyed by a field named `group`
  - This `group` field is associated with one bag for each input

```
(group, {bag of records}, {bag of records})
```

`store_id` records from `stores` records from `salespeople`

Example of COGROUP

Stores

```
A   Anchorage
B   Boston
C   Chicago
D   Dallas
E   Edmonton
F   Fargo
```

Salespeople

```
1   Alice      B
2   Bob        D
3   Carlos     F
4   Dieter     A
5   Étienne    F
6   Fredo      C
7   George     D
8   Hannah     B
9   Irina      C
10  Jack
```

```
grunt> grouped = COGROUP stores BY store_id,
salespeople BY store_id;

grunt> DUMP grouped;
(A,{(A,Anchorage)},{(4,Dieter,A)})
(B,{(B,Boston)},{(1,Alice,B),(8,Hannah,B)})
(C,{(C,Chicago)},{(6,Fredo,C),(9,Irina,C)})
(D,{(D,Dallas)},{(2,Bob,D),(7,George,D)})
(E,{(E,Edmonton)},{})
(F,{(F,Fargo)},{(3,Carlos,F),(5,Étienne,F)})
(,{},{(10,Jack,)})
```

Chapter Topics

Multi-Dataset Operations with Pig | **Data ETL and Analysis With Pig**

- Techniques for Combining Data Sets
- **Joining Data Sets in Pig**
- Set Operations
- Splitting Data Sets
- Conclusion
- Hands-On Exercise: Analyzing Disparate Data Sets with Pig

Join Overview

- **The `COGROUP` operator creates a nested data structure**
- **Pig Latin's `JOIN` operator creates a flat data structure**
  - Similar to joins in a relational database
- **A `JOIN` is similar to doing a `COGROUP` followed by a `FLATTEN`**
  - Though they handle null values differently

Key Fields

- **Like COGROUP, joins rely on a field shared by each relation**

```
joined = JOIN stores BY store_id, salespeople BY store_id;
```

- **Joins can also use multiple fields as the key**

```
joined = JOIN customers BY (name, phone_number),
              accounts BY (name, phone_number);
```

Inner Joins

- **The default `JOIN` in Pig Latin is an inner join**

```
joined = JOIN stores BY store_id, salespeople BY store_id;
```

- **An inner join outputs records only when the key is found in *all* inputs**
  - In the above example, stores that have at least one salesperson
- **You can do an inner join on multiple relations in a single statement**
  - But you must use the same key to join them

Inner Join Example

stores

```
A   Anchorage
B   Boston
C   Chicago
D   Dallas
E   Edmonton
F   Fargo
```

salespeople

```
1   Alice      B
2   Bob        D
3   Carlos     F
4   Dieter     A
5   Étienne    F
6   Fredo      C
7   George     D
8   Hannah     B
9   Irina      C
10  Jack
```

```
grunt> joined = JOIN stores BY store_id,
salespeople BY store_id;

grunt> DUMP joined;
(A,Anchorage,4,Dieter,A)
(B,Boston,1,Alice,B)
(B,Boston,8,Hannah,B)
(C,Chicago,6,Fredo,C)
(C,Chicago,9,Irina,C)
(D,Dallas,2,Bob,D)
(D,Dallas,7,George,D)
(F,Fargo,3,Carlos,F)
(F,Fargo,5,Étienne,F)
```

Eliminating Duplicate Fields (1)

- **As with COGROUP, the new relation still contains duplicate fields**

```
grunt> joined = JOIN stores BY store_id,
salespeople BY store_id;

grunt> DUMP joined;
(A,Anchorage,4,Dieter,A)
(B,Boston,1,Alice,B)
(B,Boston,8,Hannah,B)
(C,Chicago,6,Fredo,C)
(C,Chicago,9,Irina,C)
(D,Dallas,2,Bob,D)
(D,Dallas,7,George,D)
(F,Fargo,3,Carlos,F)
(F,Fargo,5,Étienne,F)
```

Eliminating Duplicate Fields (2)

- **We can use `FOREACH . . . GENERATE` to retain just the fields we need**
  - However, it is now slightly more complex to reference fields
  - We must fully-qualify any fields with names that are not unique

```
grunt> DESCRIBE joined;
joined: {stores::store_id: chararray,stores::name:
chararray,salespeople::person_id: int,salespeople::name:
chararray,salespeople::store_id: chararray}

grunt> cleaned = FOREACH joined GENERATE stores::store_id,
stores::name, person_id, salespeople::name;

grunt> DUMP cleaned;
(A,Anchorage,4,Dieter)
(B,Boston,1,Alice)
(B,Boston,8,Hannah)
... (additional records omitted for brevity) ...
```

Outer Joins

- **Pig Latin allows you to specify the type of join following the field name**
  - Inner joins do not specify a join type

```
joined = JOIN relation1 BY field [LEFT|RIGHT|FULL] OUTER,
relation2 BY field;
```

- **An outer join does not require the key to be found in both inputs**
- **Outer joins require Pig to know the schema for at least one relation**
  - Which relation requires schema depends on the join type
  - Full outer joins require schema for both relations

Left Outer Join Example

stores

```
A   Anchorage
B   Boston
C   Chicago
D   Dallas
E   Edmonton
F   Fargo
```

salespeople

```
1   Alice      B
2   Bob        D
3   Carlos     F
4   Dieter     A
5   Étienne    F
6   Fredo      C
7   George     D
8   Hannah     B
9   Irina      C
10  Jack
```

- **Left Outer Join: Result contains *all* records from the relation specified on the left, but only *matching* records from the one specified on the right**

```
grunt> joined = JOIN stores BY store_id
LEFT OUTER, salespeople BY store_id;

grunt> DUMP joined;
(A,Anchorage,4,Dieter,A)
(B,Boston,1,Alice,B)
(B,Boston,8,Hannah,B)
(C,Chicago,6,Fredo,C)
(C,Chicago,9,Irina,C)
(D,Dallas,2,Bob,D)
(D,Dallas,7,George,D)
(E,Edmonton,,,)
(F,Fargo,3,Carlos,F)
(F,Fargo,5,Étienne,F)
```

Right Outer Join Example

stores

```
A   Anchorage
B   Boston
C   Chicago
D   Dallas
E   Edmonton
F   Fargo
```

salespeople

```
1   Alice      B
2   Bob        D
3   Carlos     F
4   Dieter     A
5   Étienne    F
6   Fredo      C
7   George     D
8   Hannah     B
9   Irina      C
10  Jack
```

- **Right Outer Join: Result contains *all* records from the relation specified on the right, but only *matching* records from the one specified on the left**

```
grunt> joined = JOIN stores BY store_id
RIGHT OUTER, salespeople BY store_id;

grunt> DUMP joined;
(A,Anchorage,4,Dieter,A)
(B,Boston,1,Alice,B)
(B,Boston,8,Hannah,B)
(C,Chicago,6,Fredo,C)
(C,Chicago,9,Irina,C)
(D,Dallas,2,Bob,D)
(D,Dallas,7,George,D)
(F,Fargo,3,Carlos,F)
(F,Fargo,5,Étienne,F)
(,,10,Jack,)
```

Full Outer Join Example

stores

```
A   Anchorage
B   Boston
C   Chicago
D   Dallas
E   Edmonton
F   Fargo
```

salespeople

```
1   Alice      B
2   Bob        D
3   Carlos     F
4   Dieter     A
5   Étienne    F
6   Fredo      C
7   George     D
8   Hannah     B
9   Irina      C
10  Jack
```

- **Full Outer Join: Result contains *all* records where there is a match in *either* relation**

```
grunt> joined = JOIN stores BY store_id
FULL OUTER, salespeople BY store_id;

grunt> DUMP joined;
(A,Anchorage,4,Dieter,A)
(B,Boston,1,Alice,B)
(B,Boston,8,Hannah,B)
(C,Chicago,6,Fredo,C)
(C,Chicago,9,Irina,C)
(D,Dallas,2,Bob,D)
(D,Dallas,7,George,D)
(E,Edmonton,,,)
(F,Fargo,3,Carlos,F)
(F,Fargo,5,Étienne,F)
(,,10,Jack,)
```

Chapter Topics

Multi-Dataset Operations with Pig | **Data ETL and Analysis With Pig**

- Techniques for Combining Data Sets
- Joining Data Sets in Pig
- **Set Operations**
- Splitting Data Sets
- Conclusion
- Hands-On Exercise: Analyzing Disparate Data Sets with Pig

Crossing Data Sets

- **`JOIN` finds records in one relation that match records in another**
- **Pig's `CROSS` operator creates the cross product of both relations**
  - Combines all records in both tables regardless of matching
  - In other words, all possible combinations of records

```
crossed = CROSS stores, salespeople;
```

- **Careful: This can generate huge amounts of data!**

Cross Product Example

stores

```
A    Anchorage
B    Boston
D    Dallas
```

salespeople

```
1    Alice      B
2    Bob        D
8    Hannah     B
10   Jack
```

- **Generates every possible combination of records in the `stores` and `salespeople` relations**

```
grunt> crossed = CROSS stores, salespeople;

grunt> DUMP crossed;
(A,Anchorage,1,Alice,B)
(A,Anchorage,2,Bob,D)
(A,Anchorage,8,Hannah,B)
(A,Anchorage,10,Jack,)
(B,Boston,1,Alice,B)
(B,Boston,2,Bob,D)
(B,Boston,8,Hannah,B)
(B,Boston,10,Jack,)
(D,Dallas,1,Alice,B)
(D,Dallas,2,Bob,D)
(D,Dallas,8,Hannah,B)
(D,Dallas,10,Jack,)
```

Concatenating Data Sets

- **We have explored several techniques for combining data sets**
  - They have had one thing in common: they combine horizontally
- **The UNION operator combines records vertically**
  - It adds data from input relations into a new single relation
  - Pig does not require these inputs to have the same schema
  - It does not eliminate duplicate records nor preserve order
- **This is helpful for incorporating new data into your processing**

```
both = UNION june_items, july_items;
```

UNION Example

june

```
Adapter   549
Battery   349
Cable     799
DVD       1999
HDTV      79999
```

july

```
Fax       17999
GPS       24999
HDTV      65999
Ink       3999
```

- **Concatenates all records from `june` and `july`**

```
grunt> both = UNION june_items, july_items;

grunt> DUMP both;
(Fax,17999)
(GPS,24999)
(HDTV,65999)
(Ink,3999)
(Adapter,549)
(Battery,349)
(Cable,799)
(DVD, 1999)
(HDTV,79999)
```

Chapter Topics

Multi-Dataset Operations with Pig | **Data ETL and Analysis With Pig**

- Techniques for Combining Data Sets
- Joining Data Sets in Pig
- Set Operations
- **Splitting Data Sets**
- Conclusion
- Hands-On Exercise: Analyzing Disparate Data Sets with Pig

Splitting Data Sets

- **You have learned several ways to combine data sets into a single relation**
- **Sometimes you need to split a data set into multiple relations**
  - Server logs by date range
  - Customer lists by region
  - Product lists by vendor
  - Etc.
- **Pig Latin supports this with the `SPLIT` operator**

```
SPLIT relation INTO relationA IF expression1,
      relationB IF expression2,
      relationC IF expression3...;
```

  - Expressions need not be mutually exclusive

SPLIT Example

- **Split customers into groups for a rewards program, based on lifetime value**

customers

```
Annette 9700
Bruce   23500
Charles 17800
Dustin  21250
Eva     8500
Felix   9300
Glynn   27800
Henry   8900
Ian     43800
Jeff    29100
Kai     34000
Laura   7800
Mirko   24200
```

```
grunt> SPLIT customers INTO
       gold_program IF ltv >= 25000,
       silver_program IF ltv >= 10000
          AND ltv < 25000;

grunt> DUMP gold_program;
(Glynn,27800)
(Ian,43800)
(Jeff,29100)
(Kai,34000)

grunt> DUMP silver_program;
(Bruce,23500)
(Charles,17800)
(Dustin,21250)
(Mirko,24200)
```

Chapter Topics

Multi-Dataset Operations with Pig | **Data ETL and Analysis With Pig**

- Techniques for Combining Data Sets
- Joining Data Sets in Pig
- Set Operations
- Splitting Data Sets
- **Conclusion**
- Hands-On Exercise: Analyzing Disparate Data Sets with Pig

Essential Points

- **You can use `COGROUP` to group multiple relations**
  - This creates a nested data structure
- **Pig supports common SQL join types**
  - Inner, left outer, right outer, and full outer
  - You may need to fully-qualify field names when using joined data
- **Pig's `CROSS` operator creates every possible combination of input data**
  - This can create huge amounts of data – use it carefully!
- **You can use a `UNION` to concatenate data sets**
- **In addition to combining data sets, Pig supports splitting them too**

Chapter Topics

Multi-Dataset Operations with Pig | **Data ETL and Analysis With Pig**

- Techniques for Combining Data Sets
- Joining Data Sets in Pig
- Set Operations
- Splitting Data Sets
- Conclusion
- **Hands-On Exercise: Analyzing Disparate Data Sets with Pig**

Hands-On Exercise: Analyzing Disparate Data Sets with Pig

- **In this Hands-On Exercise, you will analyze multiple data sets with Pig**
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®
Ask Bigger Questions

Pig Troubleshooting and Optimization

Chapter 7

Course Chapters

▪ Introduction ▪ Hadoop Fundamentals	Course Introduction
▪ Introduction to Pig ▪ Basic Data Analysis with Pig ▪ Processing Complex Data with Pig ▪ Multi-Dataset Operations with Pig ▪ **Pig Troubleshooting and Optimization**	**Data ETL and Analysis With Pig**
▪ Introduction to Impala and Hive ▪ Querying With Impala and Hive ▪ Impala and Hive Data Management ▪ Data Storage and Performance	Introduction to Impala and Hive
▪ Relational Data Analysis With Impala and Hive ▪ Working with Impala ▪ Analyzing Text and Complex Data with Hive ▪ Hive Optimization ▪ Extending Hive	Data Analysis With Impala and Hive
▪ Choosing the Best Tool for the Job ▪ Conclusion	Course Conclusion

Pig Troubleshooting and Optimization

In this chapter, you will learn

- **How to control the information that Pig and Hadoop write to log files**
- **How Hadoop's Web UI can help you troubleshoot failed jobs**
- **How to use `SAMPLE` and `ILLUSTRATE` to test and debug Pig jobs**
- **How Pig creates MapReduce jobs from your Pig Latin code**
- **How several simple changes to your Pig Latin code can make it run faster**
- **Which resources are especially helpful for troubleshooting Pig errors**

Chapter Topics

Pig Troubleshooting And Optimization

Data ETL and Analysis With Pig

- **Troubleshooting Pig**
- Logging
- Using Hadoop's Web UI
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Your Pig Jobs
- Conclusion

Troubleshooting Overview

- **We have now covered how to use Pig for data analysis**
  - Unfortunately, sometimes your code may not work as you expect
  - It is important to remember that Pig and Hadoop are intertwined
- **Here we will cover some techniques for isolating and resolving problems**
  - We will start with a few options to the Pig command

Helping Yourself

- **We will discuss some options for the `pig` command in this chapter**
  - You can view all of them by using the `-h` (help) option
  - Keep in mind that many options are advanced or rarely used
- **One useful option is `-c` (check), which validates the syntax of your code**

```
$ pig -c myscript.pig
myscript.pig syntax OK
```

- **The `-dryrun` option is very helpful if you use parameters or macros**

```
$ pig -p INPUT=demodata -dryrun myscript.pig
```

  - Creates a `myscript.pig.substituted` file in current directory

Getting Help from Others

- **Sometimes you may need help from others**
  - Mailing lists or newsgroups
  - Forums and bulletin board sites
  - Support services
- **You will probably need to provide the version of Pig and Hadoop you are using**

```
$ pig -version
Apache Pig version 0.12.0-cdh5.2.0

$ hadoop version
Hadoop 2.5.0-cdh5.2.0
```

Chapter Topics

Pig Troubleshooting And Optimization | **Data ETL and Analysis With Pig**

- Troubleshooting Pig
- **Logging**
- Using Hadoop's Web UI
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Your Pig Jobs
- Conclusion

Customizing Log Messages

- **You may wish to change how much information is logged**
  - A recent change in Hadoop can cause lots of warnings when using Pig
- **Pig and Hadoop use the Log4J library, which is easily customized**
- **Edit the `/etc/pig/conf/log4j.properties` file to include:**

```
log4j.logger.org.apache.pig=ERROR
log4j.logger.org.apache.hadoop.conf.Configuration=ERROR
```

- **Edit the `/etc/pig/conf/pig.properties` file to set this property:**

```
log4jconf=/etc/pig/conf/log4j.properties
```

Customizing Log Messages on a Per-Job Basis

- **Often you just want to *temporarily* change the log level**
  - Especially while trying to troubleshoot a problem with your script
- **You can specify a Log4J properties file to use when you invoke Pig**
  - This overrides the default Log4J configuration
- **Create a `customlog.properties` file to include:**

```
log4j.logger.org.apache.pig=DEBUG
```

- **Specify this file via the `-log4jconf` argument to Pig**

```
$ pig -log4jconf customlog.properties
```

Controlling Client-Side Log Files

- **When a job fails, Pig may produce a log file to explain why**
  - These are typically produced in your current directory
- **To use a different location, use the `-l` (log) option when starting Pig**

```
$ pig -l /tmp
```

- **Or set it permanently by editing `/etc/pig/conf/pig.properties`**
  - Specify a different directory using the `log.file` property

```
log.file=/tmp
```

Chapter Topics

Pig Troubleshooting And Optimization | **Data ETL and Analysis With Pig**

- Troubleshooting Pig
- Logging
- **Using Hadoop's Web UI**
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Your Pig Jobs
- Conclusion

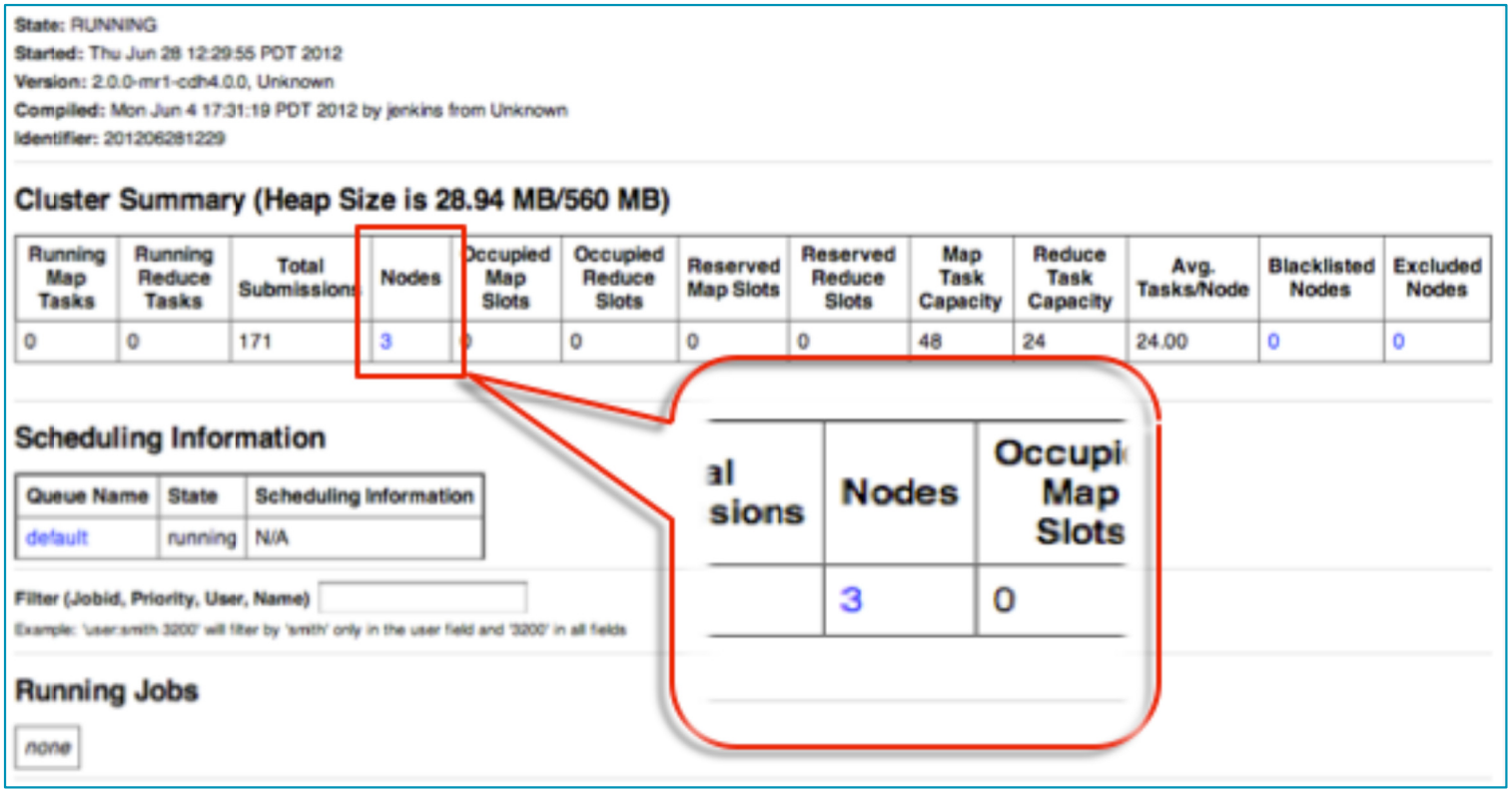
The Hadoop Web UI

- **Each Hadoop daemon has a corresponding Web application**
  - This allows us to easily see cluster and job status with a browser
  - In pseudo-distributed mode, the hostname is `localhost`

	Daemon Name	Address
HDFS	NameNode	`http://hostname:50070/`
	DataNode	`http://hostname:50075/`
MR1	**JobTracker**	**`http://hostname:50030/`**
	TaskTracker	`http://hostname:50060/`
MR2	**ResourceManager**	**`http://hostname:8088/`**
	NodeManager	`http://hostname:8042/`

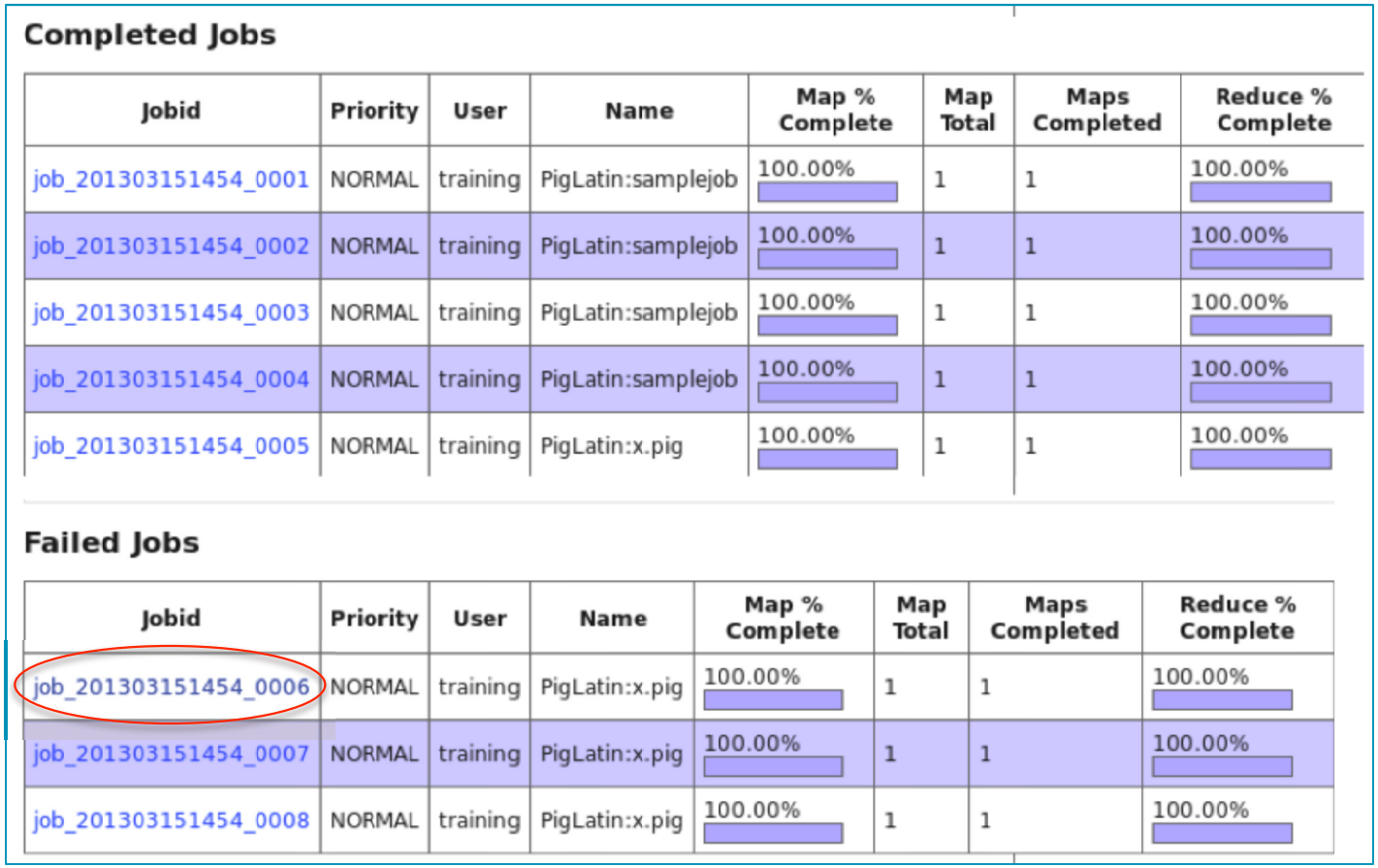
The JobTracker Web UI (1)

- **The JobTracker offers the most useful of Hadoop's Web UIs**
  - It displays MapReduce status information for the Hadoop cluster



The JobTracker Web UI (2)

- **The JobTracker Web UI also shows historical information**
  - You can click one of the links to see details for a particular job



Completed Jobs

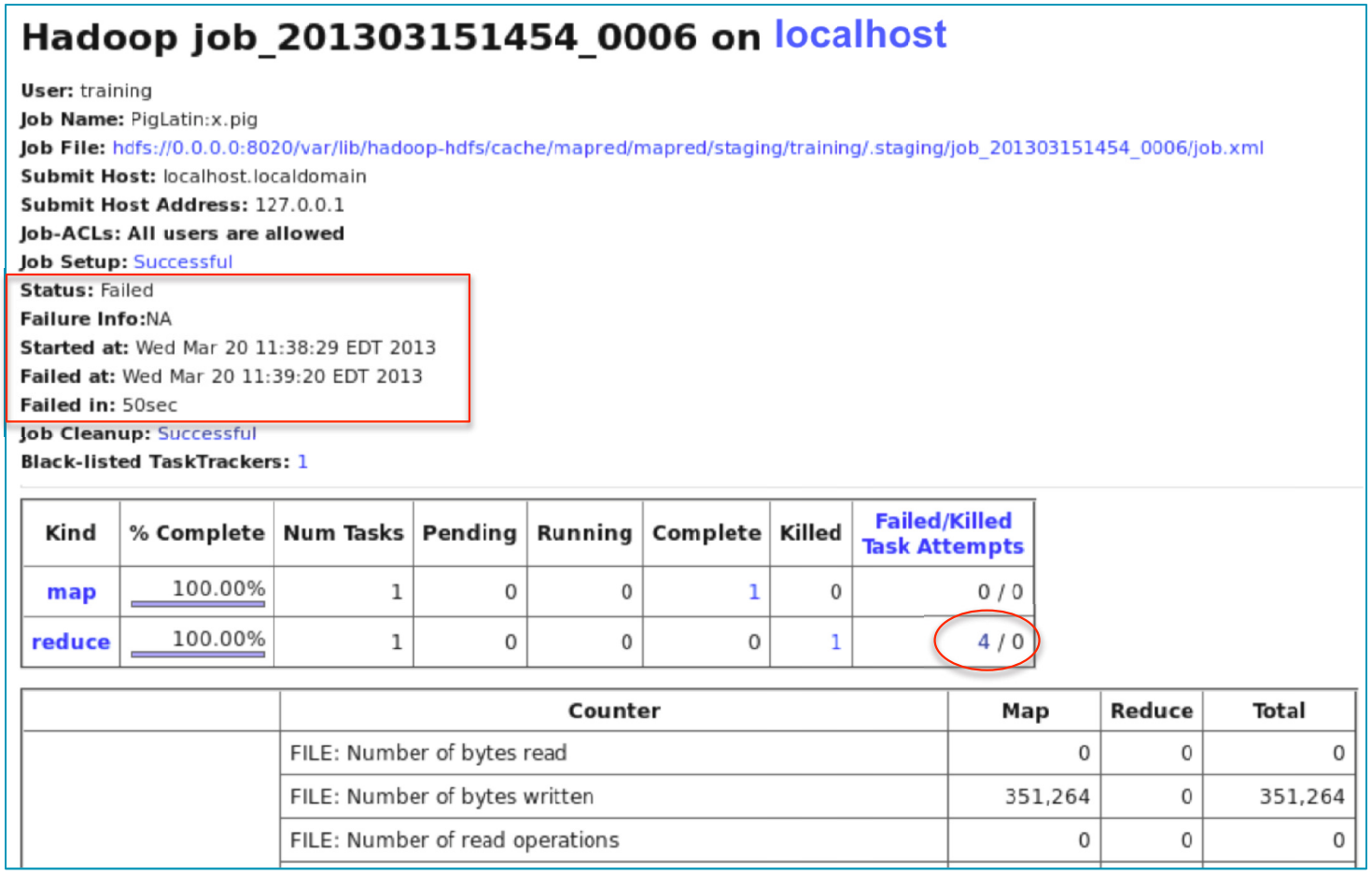
Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
job_201303151454_0001	NORMAL	training	PigLatin:samplejob	100.00%	1	1	100.00%
job_201303151454_0002	NORMAL	training	PigLatin:samplejob	100.00%	1	1	100.00%
job_201303151454_0003	NORMAL	training	PigLatin:samplejob	100.00%	1	1	100.00%
job_201303151454_0004	NORMAL	training	PigLatin:samplejob	100.00%	1	1	100.00%
job_201303151454_0005	NORMAL	training	PigLatin:x.pig	100.00%	1	1	100.00%

Failed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
job_201303151454_0006	NORMAL	training	PigLatin:x.pig	100.00%	1	1	100.00%
job_201303151454_0007	NORMAL	training	PigLatin:x.pig	100.00%	1	1	100.00%
job_201303151454_0008	NORMAL	training	PigLatin:x.pig	100.00%	1	1	100.00%

The JobTracker Web UI (3)

- **The job detail page can help you troubleshoot a problem**



Hadoop job_201303151454_0006 on localhost

User: training
Job Name: PigLatin:x.pig
Job File: hdfs://0.0.0.0:8020/var/lib/hadoop-hdfs/cache/mapred/mapred/staging/training/.staging/job_201303151454_0006/job.xml
Submit Host: localhost.localdomain
Submit Host Address: 127.0.0.1
Job-ACLs: All users are allowed
Job Setup: Successful
Status: Failed
Failure Info:NA
Started at: Wed Mar 20 11:38:29 EDT 2013
Failed at: Wed Mar 20 11:39:20 EDT 2013
Failed in: 50sec
Job Cleanup: Successful
Black-listed TaskTrackers: 1

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	1	0	0	1	0	0 / 0
reduce	100.00%	1	0	0	0	1	4 / 0

	Counter	Map	Reduce	Total
	FILE: Number of bytes read	0	0	0
	FILE: Number of bytes written	351,264	0	351,264
	FILE: Number of read operations	0	0	0

Naming Your Job

- **Hadoop clusters are typically shared resources**
  - There might be dozens or hundreds of others using the cluster
  - As a result, sometimes it is hard to find *your* job in the Web UI
- **We recommend setting a name in your scripts to help identify your jobs**
  - Set the `job.name` property, either in Grunt or your script

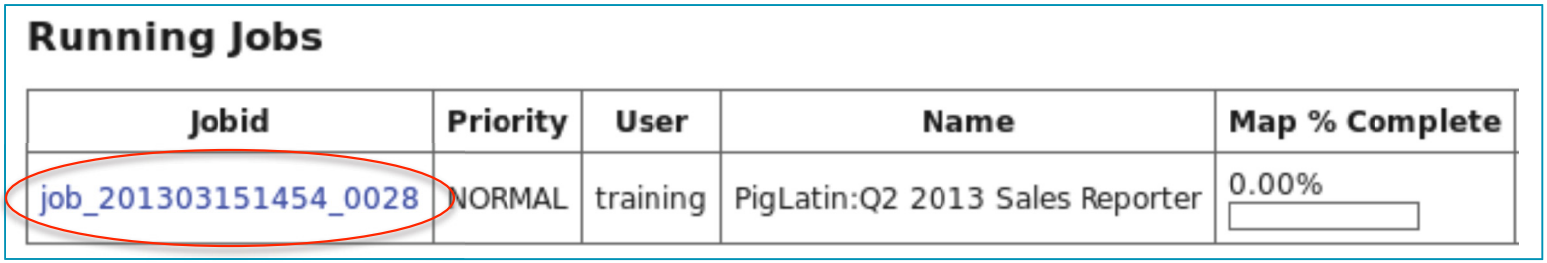
```
grunt> set job.name 'Q2 2013 Sales Reporter';
```

Completed Jobs

Jobid	Priority	User	Name
job_201303151454_0024	NORMAL	training	PigLatin:Q2 2013 Sales Reporter

Killing a Job

- **A job that processes a lot of data can take hours to complete**
  - Sometimes you spot an error in your code just after submitting a job
  - Rather than wait for the job to complete, you can kill it
- **First, find the Job ID on the front page of the JobTracker Web UI**



Running Jobs

Jobid	Priority	User	Name	Map % Complete
job_201303151454_0028	NORMAL	training	PigLatin:Q2 2013 Sales Reporter	0.00%

- **Then, use the `kill` command in Pig along with that Job ID**

```
grunt> kill job_201303151454_0028
```

Chapter Topics

Pig Troubleshooting And Optimization | **Data ETL and Analysis With Pig**

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- **Optional Demo: Troubleshooting a Failed Job with the Web UI**
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Your Pig Jobs
- Conclusion

Optional Demo: Overview

- **Time permitting, your instructor will now demonstrate how to use the JobTracker's Web UI to isolate a bug in our code that causes a job to fail**
- **The code and instructions are already on the VM**

```
$ cd ~/training_materials/analyst/webuidemo
$ cat README.txt
```

Chapter Topics

Pig Troubleshooting And Optimization | **Data ETL and Analysis With Pig**

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- **Data Sampling and Debugging**
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Your Pig Jobs
- Conclusion

Using `SAMPLE` to Create a Smaller Data Set

- **Your code might process terabytes of data in production**
  - However, it is convenient to test with smaller amounts during development
- **Use `SAMPLE` to choose a random set of records from a data set**
- **This example selects about 5% of records from `bigdata`**
  - Stores them in a new directory called `mysample`

```
everything = LOAD 'bigdata';
subset = SAMPLE everything 0.05;
STORE subset INTO 'mysample';
```

Intelligent Sampling with `ILLUSTRATE`

- **Sometimes a random sample may lack data needed for testing**
  - For example, matching records in two data sets for a `JOIN` operation
- **Pig's `ILLUSTRATE` keyword can do more intelligent sampling**
  - Pig will examine the code to determine what data is needed
  - It picks a few records that properly exercise the code
- **You should specify a schema when using `ILLUSTRATE`**
  - Pig will generate records when yours don't suffice

Using ILLUSTRATE Helps You to Understand Data Flow

- **Like DUMP and DESCRIBE, ILLUSTRATE aids in debugging**
  - The syntax is the same for all three

```
grunt> allsales = LOAD 'sales' AS (name:chararray, price:int);
grunt> bigsales = FILTER allsales BY price > 999;
grunt> ILLUSTRATE bigsales;
(Bob,3625)
-------------------------------------------------
| allsales     | name:chararray   | price:int    |
-------------------------------------------------
|              | Bob              | 3625         |
|              | Bob              | 998          |
-------------------------------------------------
-------------------------------------------------
| bigsales     | name:chararray   | price:int    |
-------------------------------------------------
|              | Bob              | 3625         |
-------------------------------------------------
```

General Debugging Strategies

- **Use `DUMP`, `DESCRIBE`, and `ILLUSTRATE` often**
- **Look at a sample of the data**
  - Verify that it matches the fields in your `LOAD` specification
  - The data might not be what you think it is
- **Other helpful steps for tracking down a problem**
  - Use `-dryrun` to see the script after parameters and macros are processed
  - Test external scripts (`STREAM`) by passing some data from a local file
  - Look at the logs, especially task logs available from the Web UI

Chapter Topics

Pig Troubleshooting And Optimization

Data ETL and Analysis With Pig

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- Data Sampling and Debugging
- **Performance Overview**
- Understanding the Execution Plan
- Tips for Improving the Performance of Your Pig Jobs
- Conclusion

Performance Overview

- **We have discussed several techniques for finding errors in Pig Latin code**
  - Once you get your code working, you'll often want it to work *faster*
- **Performance tuning is a broad and complex subject**
  - Requires a deep understanding of Pig, Hadoop, Java, and Linux
  - Typically the domain of engineers and system administrators
- **Most of these topics are beyond the scope of this course**
  - We will cover the basics here, and offer several performance improvement tips
  - See *Programming Pig* (chapters 7 and 8) for detailed coverage

Chapter Topics

Pig Troubleshooting And Optimization | **Data ETL and Analysis With Pig**

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- Data Sampling and Debugging
- Performance Overview
- **Understanding the Execution Plan**
- Tips for Improving the Performance of your Pig jobs
- Conclusion

How Pig Latin Becomes a MapReduce Job

- **Pig Latin code ultimately runs as MapReduce jobs on the Hadoop cluster**
- **However, Pig does not translate your code into Java MapReduce**
  - Much like relational databases don't translate SQL to C language code
  - Like a database, Pig interprets the Pig Latin to develop execution plans
  - Pig's execution engine uses these to submit MapReduce jobs to Hadoop
- **The `EXPLAIN` keyword details Pig's three execution plans**
  - Logical
  - Physical
  - MapReduce
- **Seeing an example job will help us better understand `EXPLAIN`'s output**

Description of Our Example Code and Data

- **Our goal is to produce a list of per-store sales**

stores

```
A   Anchorage
B   Boston
C   Chicago
D   Dallas
E   Edmonton
F   Fargo
```

sales

```
A   1999
D   2399
A   4579
B   6139
A   2489
B   3699
E   2479
D   5799
```

```
grunt> stores = LOAD 'stores'
       AS (store_id:chararray, name:chararray);
grunt> sales  = LOAD 'sales'
       AS (store_id:chararray, price:int);
grunt> groups = GROUP sales BY store_id;
grunt> totals = FOREACH groups GENERATE group,
       SUM(sales.price) AS amount;
grunt> joined = JOIN totals BY group,
       stores BY store_id;
grunt> result = FOREACH joined
       GENERATE name, amount;
grunt> DUMP result;
(Anchorage,9067)
(Boston,9838)
(Dallas,8198)
(Edmonton,2479)
```

Using the EXPLAIN Keyword

- **Using EXPLAIN rather than DUMP will show the execution plans**

```
grunt> DUMP result;
(Anchorage,9067)
(Boston,9838)
(Dallas,8198)
(Edmonton,2479)

grunt> EXPLAIN result;
#-----------------------------------------------
# New Logical Plan:
#-----------------------------------------------
result: (Name: LOStore Schema:
stores::name#49:chararray,totals::amount#70:long)
|
|---result: (Name: LOForEach Schema:
stores::name#49:chararray,totals::amount#70:long)
```

(other lines, including physical and MapReduce plans, would follow)

Chapter Topics

Pig Troubleshooting And Optimization

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- **Tips for Improving the Performance of your Pig Jobs**
- Conclusion

Pig's Runtime Optimizations

- **Pig does not necessarily run your statements exactly as you wrote them**
- **It may remove operations for efficiency**

```
sales = LOAD 'sales' AS (store_id:chararray, price:int);
unused = FILTER sales BY price > 789;
DUMP sales;
```

- **It may also rearrange operations for efficiency**

```
grouped = GROUP sales BY store_id;
totals = FOREACH grouped GENERATE group, SUM(sales.price);
joined = JOIN totals BY group, stores BY store_id;
only_a = FILTER joined BY store_id == 'A';
DUMP only_a;
```

Optimizations You Can Make in Your Pig Latin Code

- **Pig's optimizer does what it can to improve performance**
  - But you know your own code and data better than it does
  - A few small changes in your code can allow additional optimizations
- **On the next few slides, we will rewrite this Pig code for performance**

```
stores = LOAD 'stores' AS (store_id, name, postcode, phone);
sales  = LOAD 'sales'  AS (store_id, price);
joined = JOIN sales BY store_id, stores BY store_id;
DUMP joined;
groups = GROUP joined BY sales::store_id;
totals = FOREACH groups GENERATE
            FLATTEN(joined.stores::name) AS name,
            SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
region = FILTER unique BY name == 'Anchorage' OR name == 'Edmonton';
sorted = ORDER region BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Don't Produce Output You Don't Really Need

- **In this case, we forgot to remove the `DUMP` statement**
  - Sometimes happens when moving from development to production
  - And it might go unnoticed if you're not watching the terminal

```
stores = LOAD 'stores' AS (store_id, name, postcode, phone);
sales  = LOAD 'sales'  AS (store_id, price);
joined = JOIN sales BY store_id, stores BY store_id;
DUMP joined;
groups = GROUP joined BY sales::store_id;
totals = FOREACH groups GENERATE
            FLATTEN(joined.stores::name) AS name,
            SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
region = FILTER unique BY name == 'Anchorage' OR name == 'Edmonton';
sorted = ORDER region BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Specify Schema Whenever Possible

- **Specifying schema when loading data eliminates the need for Pig to guess**
  - It may choose a 'bigger' type than you need (e.g., `long` instead of `int`)
- **The postcode and phone fields in the stores data set were also never used**
  - Eliminating them in our schema ensures they'll be omitted in `joined`

```
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales  = LOAD 'sales'  AS (store_id:chararray, price:int);
joined = JOIN sales BY store_id, stores BY store_id;
groups = GROUP joined BY sales::store_id;
totals = FOREACH groups GENERATE
            FLATTEN(joined.stores::name) AS name,
            SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
region = FILTER unique BY name == 'Anchorage' OR name == 'Edmonton';
sorted = ORDER region BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Filter Unwanted Data As Early As Possible

- **We previously did our `JOIN` before our `FILTER`**
  - This produced lots of data we ultimately discarded
  - Moving the `FILTER` operation up makes our script more efficient
  - Caveat: We now have to filter by store ID rather than store name

```
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales  = LOAD 'sales'  AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN regsales BY store_id, stores BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
            FLATTEN(joined.stores::name) AS name,
            SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
sorted = ORDER unique BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Consider Adjusting the Parallelization

- **Hadoop clusters scale by processing data in parallel**
  - Newer Pig releases choose the number of reducers based on input size
  - However, it is often beneficial to set a value explicitly in your script
  - Your system administrator can help you determine the best value

```
set default_parallel 5;
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales  = LOAD 'sales'  AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN regsales BY store_id, stores BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
           FLATTEN(joined.stores::name) AS name,
           SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
sorted = ORDER unique BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Specify the Smaller Data Set First in a Join

- **We can optimize joins by specifying the larger data set last**
  - Pig will 'stream' the larger data set instead of reading it into memory
  - In our case, we have far more records in `sales` than in `stores`
  - Changing the order in the `JOIN` statement can boost performance

```
set default_parallel 5;
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales  = LOAD 'sales'  AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN stores BY store_id, regsales BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
           FLATTEN(joined.stores::name) AS name,
           SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
sorted = ORDER unique BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Try Using Compression on Intermediate Data

- **Pig scripts often yield jobs with both a Map and a Reduce phase**
  - Remember that Mapper output becomes Reducer input
  - Compressing this intermediate data is easy and can boost performance
  - Your system administrator may need to install a compression library

```
set mapred.compress.map.output true;
set mapred.map.output.compression.codec
      org.apache.hadoop.io.compress.SnappyCodec;
set default_parallel 5;
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales  = LOAD 'sales'  AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN stores BY store_id, regsales BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
           FLATTEN(joined.stores::name) AS name,
           SUM(joined.sales::price) AS amount;
... (other lines unchanged, but removed for brevity) ...
```

A Few More Tips for Improving Performance

- **Main theme: Eliminate unnecessary data as early as possible**
  - Use `FOREACH ... GENERATE` to select just those fields you need
  - Use `ORDER BY` and `LIMIT` when you only need a few records
  - Use `DISTINCT` when you don't need duplicate records
- **Dropping records with `NULL` keys before a join can boost performance**
  - These records will be eliminated in the final output anyway
  - But Pig doesn't discard them until after the join
  - Use `FILTER` to remove records with null keys before the join

```
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales  = LOAD 'sales'  AS (store_id:chararray, price:int);

nonnull_stores = FILTER stores BY store_id IS NOT NULL;
nonnull_sales = FILTER sales BY store_id IS NOT NULL;

joined = JOIN nonnull_stores BY store_id, nonnull_sales BY store_id;
```

Chapter Topics

Pig Troubleshooting And Optimization

Data ETL and Analysis With Pig

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Optional Demo: Troubleshooting a Failed Job with the Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Your Pig Jobs
- **Conclusion**

Essential Points

- **You can boost performance by eliminating unneeded data during processing**
- **Pig's error messages don't always clearly identify the source of a problem**
  - We recommend testing your scripts with a small data sample
  - Looking at the Web UI, and especially the log messages, can be helpful
- **The resources listed on the upcoming bibliography slide may further assist you in solving problems**

Bibliography

The following offer more information on topics discussed in this chapter

- **Hadoop Log Location and Retention**
  - `http://tiny.cloudera.com/logblog`
- **Pig Testing and Diagnostics**
  - `http://tiny.cloudera.com/pigtesting`
- **Mailing List for Pig Users**
  - `http://tiny.cloudera.com/piglist`
- **Questions Tagged with "Pig" on StackOverflow**
  - `http://tiny.cloudera.com/dac08d`
- **Questions Tagged with "PigLatin" on StackOverflow**
  - `http://tiny.cloudera.com/dac08e`

cloudera®
Ask Bigger Questions

Introduction to Impala and Hive

Chapter 8

Course Chapters

▪ Introduction ▪ Hadoop Fundamentals	Course Introduction
▪ Introduction to Pig ▪ Basic Data Analysis with Pig ▪ Processing Complex Data with Pig ▪ Multi-Dataset Operations with Pig ▪ Pig Troubleshooting and Optimization	Data ETL and Analysis With Pig
▪ **Introduction to Impala and Hive** ▪ Querying With Impala and Hive ▪ Impala and Hive Data Management ▪ Data Storage and Performance	**Introduction to Impala and Hive**
▪ Relational Data Analysis With Impala and Hive ▪ Working with Impala ▪ Analyzing Text and Complex Data with Hive ▪ Hive Optimization ▪ Extending Hive	Data Analysis With Impala and Hive
▪ Choosing the Best Tool for the Job ▪ Conclusion	Course Conclusion