ANALYTIXLABS

Hadoop-Pig

Hadoop-Pig

Recall: Hadoop Eco-System – Analytics mapping

APACHE OOZIE (WORK FLOW)

Tools→ HIVE DW SYSTEM | PIG LATIN DATA ANALYSIS | MAHOUT MACHINE LEARNING | Other YARN Frame works (MPI, Giraph) | HBASE

Process→ Map Reduce Framework

YARN (Cluster Resource Management)

Storage→ HDFS(Hadoop Distributed File System) Storage

← These Tools can be replaced by R-Hadoop/ PyHadoop

FLUME

Import & Export

SQOOP

Import Export →

Processing

Unstructured & Semi structured

Structured

✓ Zookeeper is software project, providing an open source distributed configuration service and synchronization service and naming registry for large distributed system

Pig overview

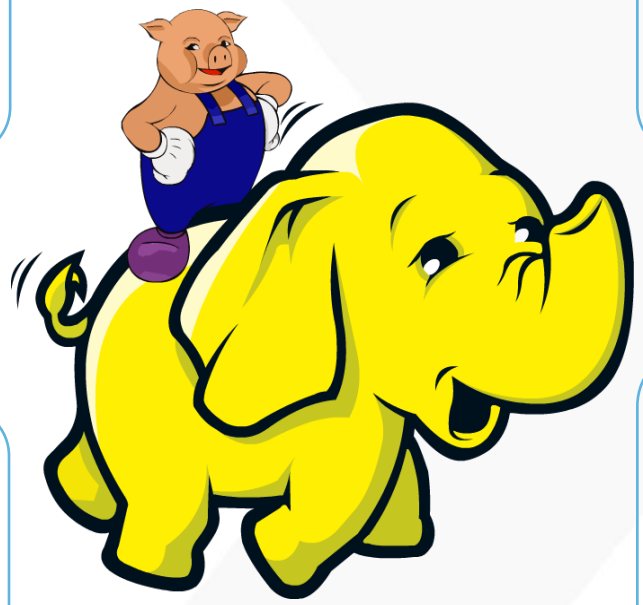
- Created in Yahoo Research Labs
- Built to avoid low level programming of Map Reduce
- Committers: Yahoo, Hortonworks, LinkedIN, Netflix, IBM, Twitter

Pig overview

Pig is one of the components of the Hadoop eco-system.

Pig is a high-level data flow scripting language.

Pig runs on the Hadoop clusters.



Pig is an Apache open-source project.

Pig uses HDFS for storing and retrieving data and Hadoop MapReduce for processing Big Data.

Pig overview

- **Apache Pig** is a platform for analysing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs.
- The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets.

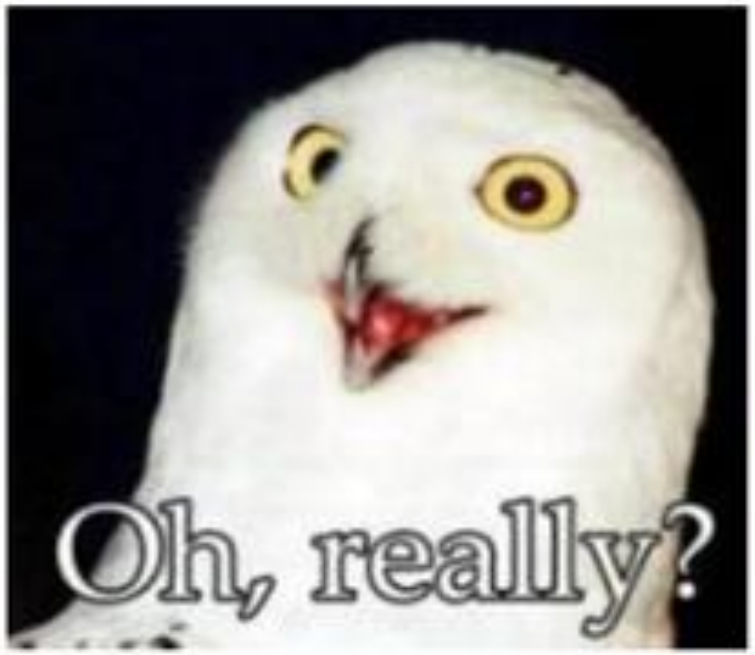
Pig overview

- Pig infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs
- Pig language layer currently consists of a textual language called Pig Latin
- Performance is close to Python or Java
- Not for general programming

Why pig?

✓ Do you know Java?

- ✓ 10 lines of PIG = 200 lines of Java



Why Pig?

- Map Reduce is very powerful but:
  - It requires a Java programmer.
  - User has to re-invent common
  - Increases productivity
  - 10 lines of Pig Latin ≈ 200 lines of Java.
  - Takes less than 25 % of time for coding as in Java
  - Opens the system to non-Java programmers.
  - Provides common operations like join, group, filter, sort.

Why Pig ?

→ Java not required

→ Can take any data
- Structured data
- Semi-Structured data
- Unstructured data

→ Easy to learn, Easy to write and Easy to read
- Similar to SQL
- Reads like a series of steps

→ Extensible by UDF (User Defined Functions)
- Java
- Python
- JavaScript
- Ruby

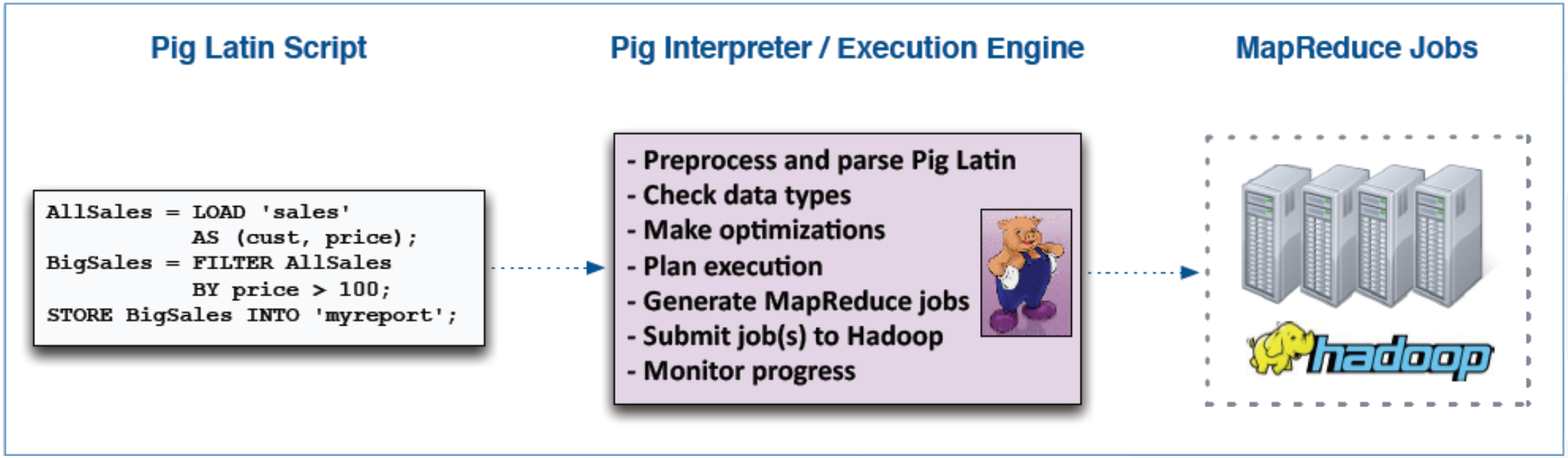
→ Provides common data operations filters, joins, ordering, etc. and nested data types tuples, bags, and maps missing from MapReduce.

→ An ad-hoc way of creating and executing map-reduce jobs on very large data sets

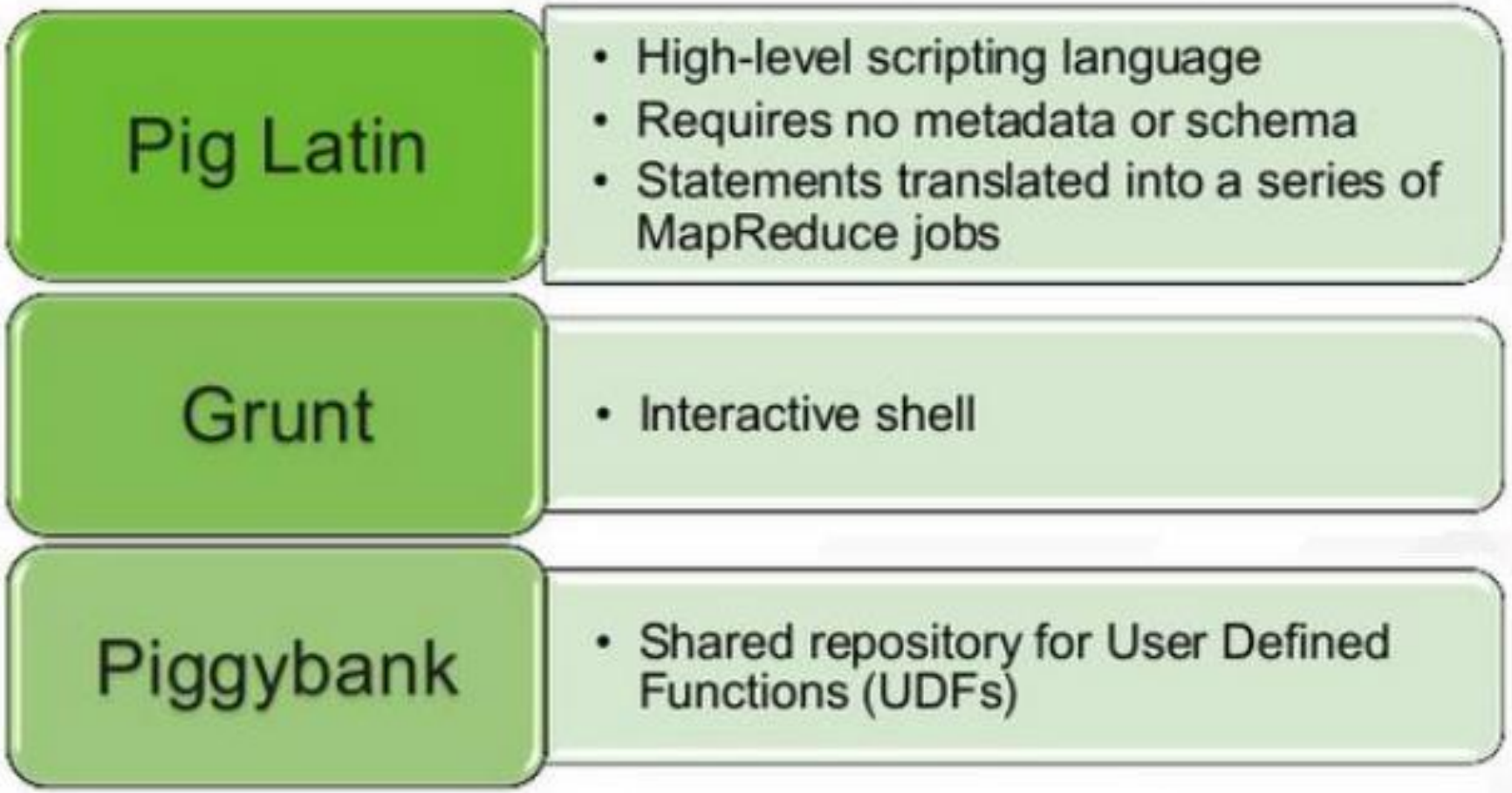
→ Open source and actively supported by a community of developers.

Main components 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



Pig


Pig Latin
• High-level scripting language
• Requires no metadata or schema
• Statements translated into a series of MapReduce jobs
Grunt
• Interactive shell
Piggybank
• Shared repository for User Defined Functions (UDFs)

Pig Strengths

- Rapid Data preparation
- Interactive
- Great for
  - Data pipelines
  - Analysis with raw data

Pig Features

- Simple Data Analysis Tool.
- High Level language in terms of Pig-Latin Scripting Language
- HDFS Manipulation
- Schema is optional.
- Data Flow
- support relational-style operations such as filter, union, group, and join
- Unix shell commands
- Positional references for fields
- Common mathematical functions
- Support for custom functions and data formats
- Complex data structures

How organizations use 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
  - Ad hoc queries across large data sets.
  - Rapid prototyping of algorithms for processing large data sets

- Pig is also frequently used for data processing
  - Reorganizing existing data set
  - Joining data from multiple sources to provide new data set
  - Used as ETL tool
  - Handling complex business transformations

Where not to use pig?

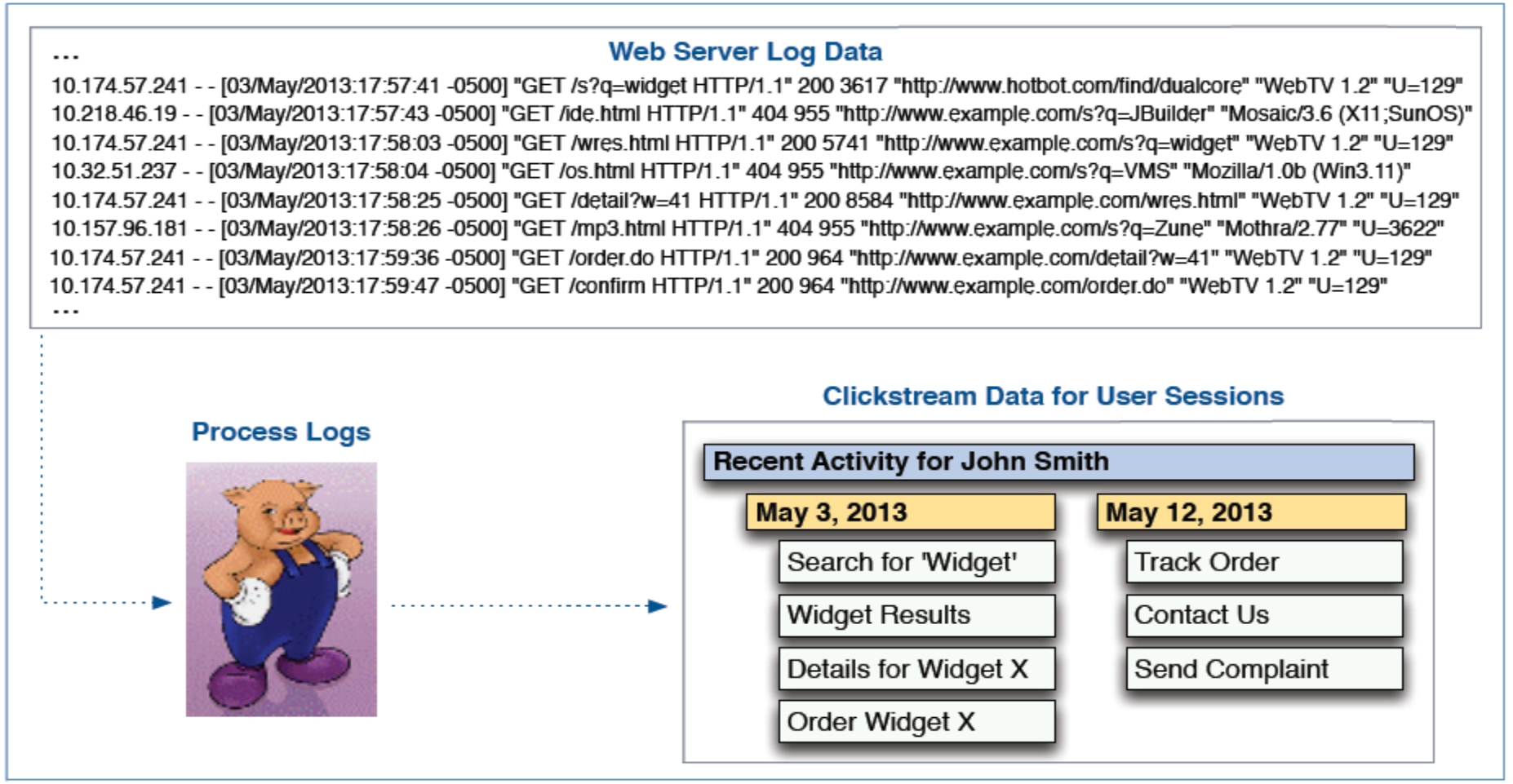
→ Really nasty data formats or completely unstructured data (video, audio, raw human-readable text).

→ Perfectly implemented MapReduce code can sometimes execute jobs slightly faster than equally well-written Pig code.

→ When you would like more power to optimize your code.

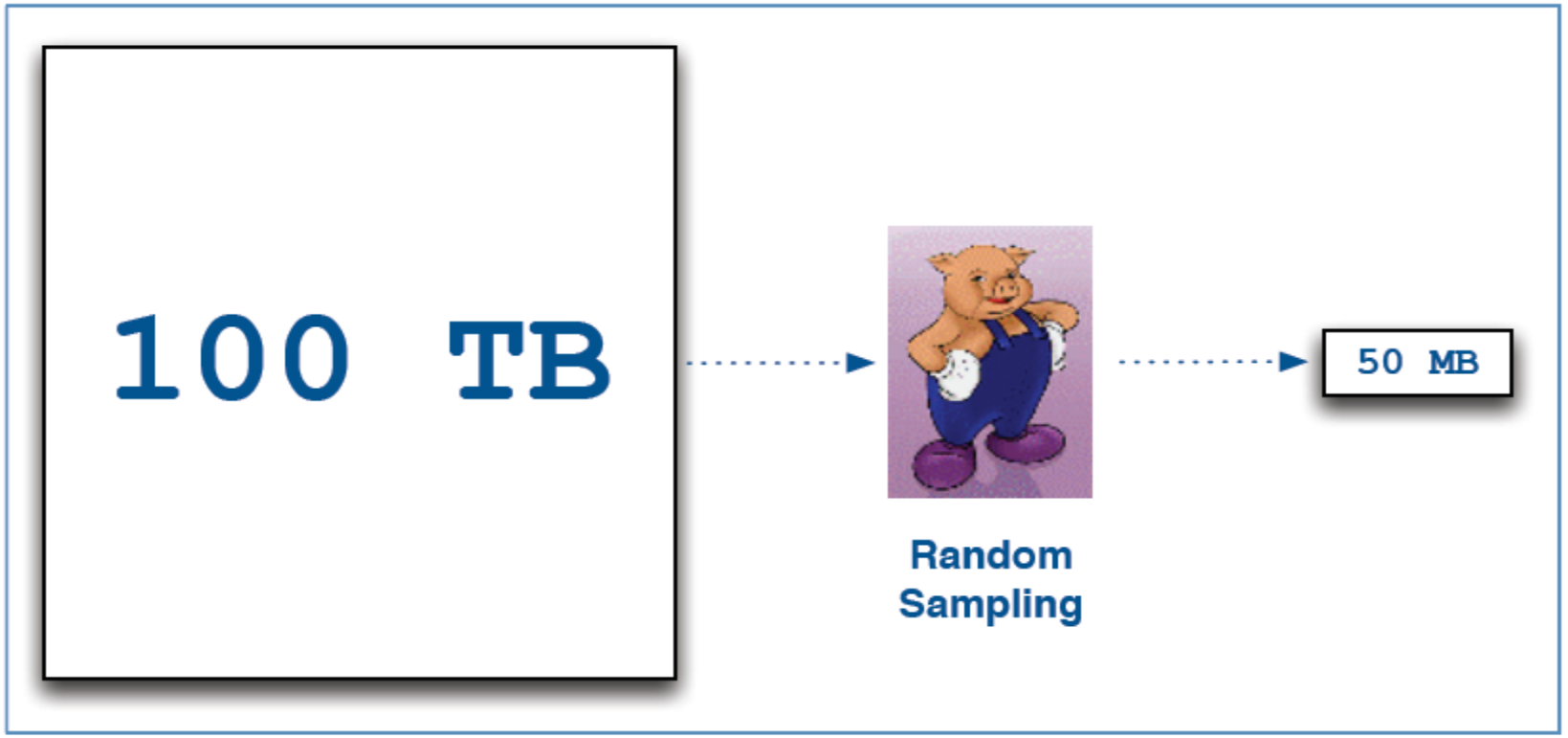
Use Case1: Web log sessionization

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



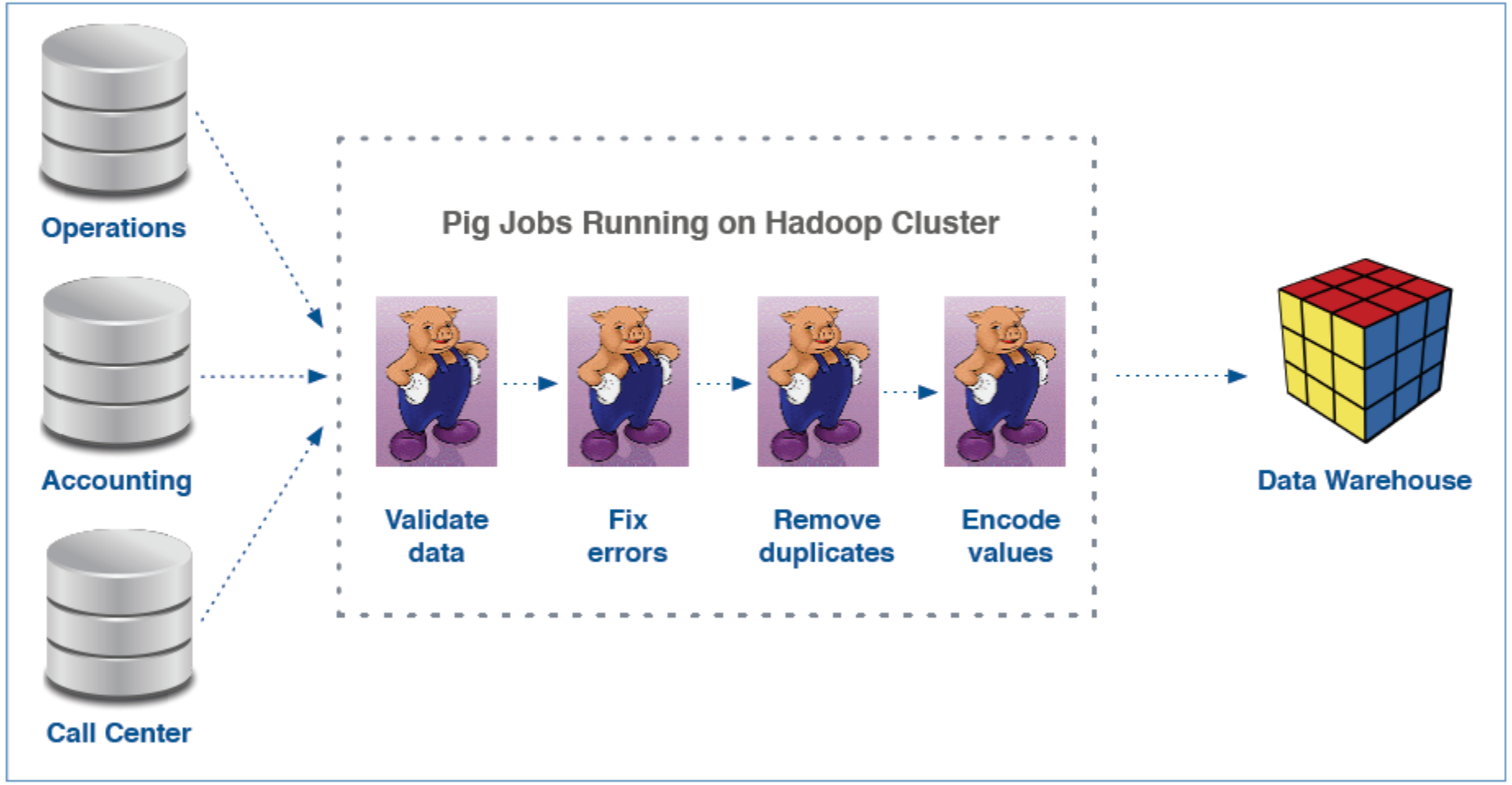
Use Case2: Data Sampling

- **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



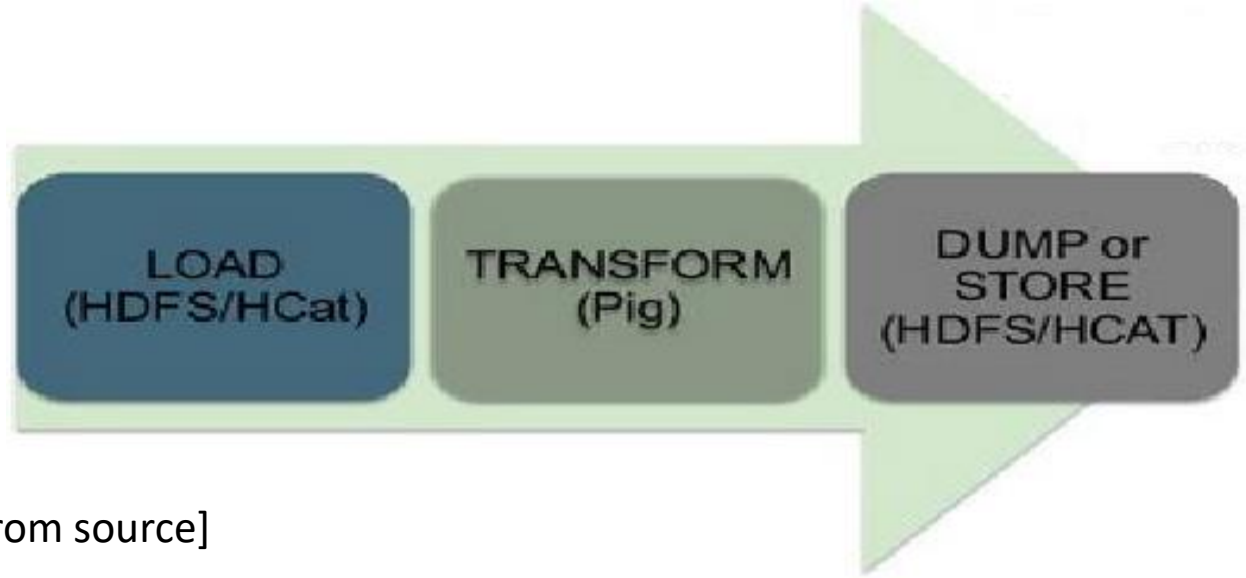
Use Case3: ETL Processing

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



Pig data flow

Read data | Manipulate data | Output data



Code :

var1 = Load [data from source]

var2 = [Transform var1]

dump var2

store var2 into [some folder/file]

Running Pig

You can run Pig (execute Pig Latin statements and Pig commands) using various modes.

Interactive Mode

- You can run Pig in interactive mode using the Grunt shell. Invoke the Grunt shell using the "pig" command (as shown below) and then enter your Pig Latin statements and Pig commands interactively at the command line.

Batch Mode

- You can run Pig in batch mode using Pig scripts and the "pig" command (in local or hadoop mode).

Execution Modes: Pig has two execution modes or exectypes:

- **Local Mode** - To run Pig in local mode, you need access to a single machine; all files are installed and run using your local host and file system. Specify local mode using the -x flag (pig -x local).
- **Mapreduce Mode** - To run Pig in mapreduce mode, you need access to a Hadoop cluster and HDFS installation. Mapreduce mode is the default mode; you can, *but don't need to*, specify it using the -x flag (pig OR pig -x mapreduce).
- You can run Pig in either mode using the "pig" command (the bin/pig Perl script) or the "java" command (java -cp pig.jar ...).

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 & Unix

- **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;
```

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

```
grunt> sh date;
Fri May 10 13:05:31 PDT 2013
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 & 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 MapReduce jobs on the *local machine* instead of the cluster
  - Local mode uses the local filesystem instead ofHDFS
  - Can be helpful for testing before deploying a job to production

```
$ pig -x local                          -- interactive

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

Pig Latin Overview

- Pig Latin is a data flow language rather than procedural or declarative.
- User code and existing binaries can be included almost anywhere.
- Metadata not required, but used when available.
- Support for nested types.
- Operates on files in HDFS.

- **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 properties

- **Ease of programming:** Pig Latin can break complex tasks into simple data flow sequences, making them easy to write, understand, and maintain.
- **Optimization: Pig scripts o**ptimize their execution automatically allowing the user to focus on semantics rather than efficiency.
- E**xtensibility:** Users can create their own functions to do special-purpose processing.

Pig Latin - 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 – Identifiers (names)

- **Identifiers are names assigned to fields and other data structures**
- **Identifiers must confirm to Pig's naming rules**
- **An identifier must begin with letter and this may only followed by letters, numbers and underscores.**

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

Pig Latin – Comments

- **Pig Latin supports two types of comments**
- **Single line comments begin with –**
- **Multi line comments with begin with /* and ends 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';
```

Pig Latin – Case Sensitivity

- **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';
```

Pig Latin – Common Operators

- **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
/	>		
%	<=		
	>=		

Pig-Latin Relationship operators

Category	Operator	Description
Loading and Storing	LOAD STORE DUMP	Loads data from the file system or other storage into a relation Saves a relation to the file system or other storage. Prints a relation to the console.
Filtering	FILTER DISTINCT FOREACH…GENERATE STREAM	Removes unwanted rows from a relation. Removes duplicate rows from a relation. Adds or removes fields from a relation. Transforms a relation using an external program.
Grouping and Joining	JOIN COGROUP GROUP CROSS	Joins two or more relations. Groups the data in two or more relations. Groups the data in a single relation. Creates the cross product of two or more relations.
Sorting	ORDER LIMIT	Sorts a relation by one or more fields. Limits the size of a relation to a maximum number of tuples.
Combining and Splitting	UNION SPLIT	Combines two or more relations into one. Splits a relation into two or more relations.

Pig Commands

Command	Use	Example
Load	get the data accessible to Pig	A = load 'a.csv' using PigStorage(',') as (name:chararray, num:int);
Dump	Display the results on screen	Dump A;
Store	Saves results in a file	Store A into 'outfile.out';
Foreach	One record at a time (map)	B = foreach A generate name, num, num*10 as num2;
Filter	Select few records based upon condition	filB = filter B by num > 10;
Group	Group records based upon some key	grpB = Group B by name;
Count	Count number of records based upon some key	cntB = foreach grpB generate group, COUNT(B);
Join	Join two datasets based upon some key	joinXY = join X by name, Y by name;

Pig latin – 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

Pig latin – Complex data types

tuple	(19, 2, 1)	A constant in this form creates a tuple.
bag	{ (19, 2), (1, 2) }	A constant in this form creates a bag.
map	['name' # 'John', 'ext' # 5555]	A constant in this form creates a map.

Pig latin – Complex data types

- **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

tuple

bag

Pig latin – Complex data types

- **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

Pig latin – Representing Complex data types

- **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']`

Pig latin – Loading & using Complex types

- **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)});
```

Pig latin – Loading & using Complex types

- **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);
```

Pig latin – 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';
```

Pig latin – Specifying data types

- **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!**

Pig Latin - Hcatlog

- **You have seen how to specify names, types, and paths during LOAD**
- **A new project called HCatalog can store this information permanently**
  - So it need not be specified each time
  - Simplifies sharing of metadata between Pig, Hive, and MapReduce
- **You can load data easily after first setting it up with HCatalog**

```
allsales = LOAD 'sales'
           USING org.apache.hcatalog.pig.HCatLoader();
```

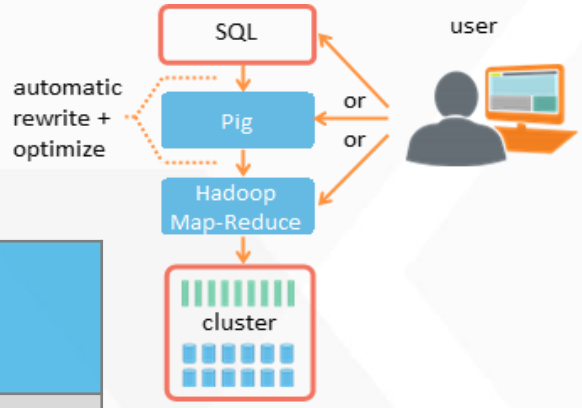
Pig Latin – Handling invalid data

- **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;
```

Pig vs. SQL

The difference between Pig and SQL are given in the table below:



Difference	Pig	SQL
Definition	Scripting language used to interact with HDFS	Query language used to interact with databases
Query Style	Step-by-step	Single block
Evaluation	Lazy Evaluation	Immediate evaluation
Pipeline Splits	Pipeline splits are supported	Requires the join to be run twice or materialized as an intermediate result

Pig vs. SQL Example

SQL	Pig

SELECT c_id , **SUM**(amount) AS CTotal
FROM customers c
JOIN sales s ON c.c_id = s.c_id
WHERE c.city = 'Texas'
GROUP BY c_id
HAVING SUM(amount) > 2000
ORDER BY CTotal **DESC**
	customer = **LOAD** '/data/customer.dat' **AS** (c_id,name,city);
	sales = **LOAD** '/data/sales.dat' **AS** (s_id,c_id,date,amount);
	salesTX = **FILTER** customer **BY** city == 'Texas';
	joined= **JOIN** customer **BY** c_id, salesTX BY c_id;
	grouped = **GROUP** joined **BY** c_id;
	summed= **FOREACH** grouped **GENERATE GROUP**,
	SUM(joined.salesTX::amount);
	spenders= **FILTER** summed **BY** $1 > 2000;
	sorted = **ORDER** spenders **BY** $1 **DESC**;
	DUMP sorted;

Pig-Latin File loaders

BinStorage – “binary” storage
PigStorage - Loads and stores data that is delimited by something
TextLoader – Loads data line by line (delimited by the newline character)
CSVLoader – Loads csv files
XMLLoader(XMLStorage) – Loads XML files
JsonLoader(JsonStorage) – Loads JSON files
HBaseLoader(HBaseStorage) – Loads data into Hbase

Converting XML into CSV Using Pig

Now we will take a sample XML data. After installing hadoop we get many configuration files in xml format and in this case we are taking **hdfs-site.xml** as our input data**.**

✓A = load '/hdfs-site.xml' using org.apache.pig.piggybank.storage.XMLLoader('property') **as** (x:chararray);

Here we will load the xml file using the default XML loader available in pig, inside the XML loader we are specifying that our root element is **property** and we are storing the whole thing with an alias name ***x*** as **chararray**.

✓B = **foreach** A generate REPLACE(x,'[\\n]','') **as** x;

Here we are bringing the contents between the ***property*** tag in one line.

✓C = **foreach** B generate REGEX_EXTRACT_ALL(x,'.*(?:<name>)([^<]*).*(?:<value>)([^<]*).*');

Now we are removing the brackets by using the above mentioned regular expression

✓D =**FOREACH** C GENERATE FLATTEN (($0));

Here by using flatten it will remove the remaining brackets. Now the Final result looks like this.

✓STORE D INTO '/pig_conversions/xml_to_csv' USING org.apache.pig.piggybank.storage.CSVExcelStorage();

This output is stored in the location **/pig_conversions/xml_to_csv with name part-m-00000** of HDFS. We can download and see the contents of the file

Converting JSON into CSV Using Pig

load the JSON data into pig using the below command

✓loadJson = LOAD '/olympic.json' USING JsonLoader('athelete:chararray,age:INT,country:chararray,year:chararray, closing:chararray,sport:chararray,gold:INT,silver:INT,bronze:INT,total:INT');

Pig provides API for loading Json format of data, Using the above command we can load the data into pig.
In this case, we are using JsonLoader() as our loader function .

Now we have successfully loaded the JSON data into pig, to convert it into CSV we just need to store the JSON data with CSV API provided by pig.

If we load JSON data using JSON loader, the data will be parsed automatically by the loader and will be visible as CSV format.

✓STORE loadJson INTO '/pig_conversions/json_to_csv' USING org.apache.pig.piggybank.storage.CSVExcelStorage();

The above command will store the output using CSV storage available with pig.
You can download the CSV file from the location /pig_conversions/json_to_csv with name part-m-00000.

Pig-Diagnostic operators

Pig Latin Diagnostic Operators

Types of Pig Latin Diagnostic Operators:

DESCRIBE - Prints a relation's schema.
EXPLAIN - Prints the logical and physical plans.
ILLUSTRATE - Shows a sample execution of the logical plan, using a generated subset of the input.

Data flow : FOREACH

```
emp = LOAD 'inputs/empfile.txt' AS
(name:chararray,age:int,state:chararray,salary:double);
f = FOREACH emp GENERATE name,age,salary;
o = ORDER f by age;
```

Examples for nested FOREACH

Objective: Find top three salaries by each title in employee table

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)})
```

Pig Latin: Group Filter

```
weblogs = LOAD 'inputs/weblog.log' AS
(eventid:int,date:chararray,type:int,level:chararray,message:chararray);
fatal = FILTER weblogs BY(level == 'fatal');
groupthem = GROUP fatal by type;
dump groupthem;
```

Pig Latin – 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

Pig Latin – Grouping multiple relations

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,)})
```

Pig Latin: JOIN

✓The COGROUP operator creates a nested data structure
✓Pig Latin's JOIN operator creates a flat data structure similar to joins in RDBMS
✓A JOIN is similar to doing a COGROUP followed by FLATTEN though they handle null values differently

- **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);
```

Pig Latin – 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)
```

Pig Latin – 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

Pig Latin – 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
```

- **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)
```

Pig Latin – Set Operations – 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

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,)
```

Pig Latin – Set Operations – 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

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)
```

Pig Latin – Splitting data set into multiple 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
- **Split customers into groups for 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)
```

Pig Latin – Making the script More flexible with parameters

- **Instead of hardcoding values, Pig allows you to use parameters**
  - These are replaced with specified values at runtime

```
allsales = LOAD '$INPUT' AS (name, price);
bigsales = FILTER allsales BY price > $MINPRICE;

bigsales_name = FILTER bigsales BY name == '$NAME';
STORE bigsales_name INTO '$NAME';
```

  - Then specify the values on the command line

```
$ pig -p INPUT=sales -p MINPRICE=999 \
  -p NAME='Jo Anne' reporter.pig
```

Pig Latin – Making the script More flexible with parameters

- **You can also specify parameter values in a text file**
  - An alternative to typing each one on the command line

```
INPUT=sales
MINPRICE=999
# comments look like this
NAME='Alice'
```

  - Use `-m filename` option to tell Pig which file contains the values

- **Parameter values can be defined with the output of a shell command**
  - For example, to set `MONTH` to the current month:

```
MONTH=`date +'%m'`    # returns 03 for March, 05 for May
```

Pig Latin – Macros – Need for macros

- **Parameters simplify repetitive code by allowing you to pass in values**
  - But sometimes you would like to reuse the actual code too

```
allsales = LOAD 'sales' AS (name, price);
byperson = FILTER allsales BY name == 'Alice';

SPLIT byperson INTO low IF price < 1000,
      high IF price >= 1000;

amt1 = FOREACH low GENERATE name, price * 0.07 AS amount;
amt2 = FOREACH high GENERATE name, price * 0.12 AS amount;

commissions = UNION amt1, amt2;
grpd = GROUP commissions BY name;

out = FOREACH grpd GENERATE SUM(commissions.amount) AS total;
```

Pig Latin – Macros – Defining Macro

- **Macros allow you to define a block of code to reuse easily**
  - Similar (but not identical) to a function in a programming language

```
define calc_commission (NAME, SPLIT_AMT, LOW_PCT, HIGH_PCT)
returns result {
  allsales = LOAD 'sales' AS (name, price);
  byperson = FILTER allsales BY name == '$NAME';

  SPLIT byperson INTO low if price < $SPLIT_AMT,
        high IF price >= $SPLIT_AMT;

  amt1 = FOREACH low GENERATE name, price * $LOW_PCT AS amount;
  amt2 = FOREACH high GENERATE name, price * $HIGH_PCT AS amount;

  commissions = UNION amt1, amt2;
  grouped = GROUP commissions BY name;

  $result = FOREACH grouped GENERATE SUM(commissions.amount);
};
```

Pig Latin – Macros – invoking Macro

- **To invoke a macro, call it by name and supply values in the correct order**

```
define calc_commission (NAME, SPLIT_AMT, LOW_PCT, HIGH_PCT)
returns result {
  allsales = LOAD 'sales' AS (name, price);

  ... (other code removed for brevity) ...

  $result = FOREACH grouped GENERATE SUM(commissions.amount);
};
```

```
alice_comm = calc_commission('Alice', 1000, 0.07, 0.12);
carlos_comm = calc_commission('Carlos', 2000, 0.08, 0.14);
```

Pig Latin – Macros – Reusing code with imports

- **After defining a macro, you may wish to use it in multiple scripts**
- **You can include one script within another, starting with Pig 0.9**
  - This is done with the `import` keyword and path to file being imported

```
-- We saved the macro to a file named commission_calc.pig

import 'commission_calc.pig';

alice_comm = calc_commission('Alice', 1000, 0.07, 0.12);
```

Pig Latin - 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()`		`0.48161326652569`
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);
```

Pig Latin - Built-in 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

Pig Latin – 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

User Defined Function (UDF's)

- **We have covered many of Pig's built-in functions already**
- **It is also possible to define your own functions**
  - Pig allows writing UDFs in several languages

Language	Supported in Pig Versions
Java	`All`
Python	`0.8 and later`
JavaScript (*experimental*)	`0.9 and later`
Ruby (*experimental*)	`0.10 and later`
Groovy (*experimental*)	`0.11 and later`

- **In the next few slides, you will see how to use UDFs in Java, and how to write and use UDFs in Python**

What are UDF's in Pig?

A **user-defined function** (**UDF**) is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment.

There are four type of UDF's in Pig

- ✓Eval functions
- ✓Aggregate functions
- ✓Filter functions
- ✓Load functions

Using UDF's written in JAVA

- **UDFs are packaged into Java Archive (JAR) files**
- **There are only two required steps for using them**
  - Register the JAR file(s) containing the UDF and its dependencies
  - Invoke the UDF using the fully-qualified classname

```
REGISTER '/path/to/myudf.jar';
...
data = FOREACH allsales GENERATE com.example.MYFUNC(name);
```

- **You can optionally define an alias for the function**

```
REGISTER '/path/to/myudf.jar';
DEFINE FOO com.example.MYFUNC;
...
data = FOREACH allsales GENERATE FOO(name);
```

Example for user defined function in JAVA

✓ A Program to create UDF:

```
public class IsOfAge extends FilterFunc {
        @Override
        public Boolean exec(Tuple tuple) throws IOException {
                if (tuple == null || tuple.size() == 0) {
                return false;
                }

                try {
                        Object object = tuple.get(0);
                        if (object == null) {
                                return false;
                        }
                        int i = (Integer) object;
                        if (i == 18 || i == 19 || i == 21 || i == 23 || i == 27) {
                                return true;
                        } else {
                                return false;
                        }
                } catch (ExecException e) {
                        throw new IOException(e);
                }
        }
}
```

How to call a UDF?

register myudf.jar;

X = filter A by **IsOfAge**(age);

Example for user defined function in Python

- **Now we will see how to write a UDF in Python**
- **The data we want to process has inconsistent phone number formats**

```
Alice     (314) 555-1212
Bob       212.555.9753
Carlos    405-555-3912
David     (202) 555.8471
```

- **We will write a Python UDF that can consistently extract the area code**

Example for user defined function in Python

- **Our Python code is straightforward**
- **The only unusual thing is the optional `@outputSchema` decorator**
  - This tells Pig what data type we are returning
  - If not specified, Pig will assume `bytearray`

```
@outputSchema("areacode:chararray")
def get_area_code(phone):
    areacode = "???" # return this for unknown formats

    if len(phone) == 12:
        # XXX-YYY-ZZZZ or XXX.YYY.ZZZZ format
        areacode = phone[0:3]
    elif len(phone) == 14:
        # (XXX) YYY-ZZZZ or (XXX) YYY.ZZZZ format
        areacode = phone[1:4]

    return areacode
```

Using UDF's written in python

- **Using this UDF from our Pig Latin is also easy**
  - We saved our Python code as `phonenumber.py`
  - This Python file is in our current directory

```
REGISTER 'phonenumber.py' USING jython AS phoneudf;

names = LOAD 'names' AS (name:chararray, phone:chararray);

areacodes = FOREACH names GENERATE
            phoneudf.get_area_code(phone) AS ac;
```

Contributed UDF's - PiggyBank

- **Piggy Bank ships with Pig**
  - You will need to register the `piggybank.jar` file
  - The location may vary depending on source and version
  - In CDH on our VMs, it is at `/usr/lib/pig/piggybank.jar`
- **Some UDFs in Piggy Bank include (package names omitted for brevity)**

Class Name	Description
`ISOToUnix`	Converts an ISO 8601 date/time format to UNIX format
`UnixToISO`	Converts a UNIX date/time format to ISO 8601 format
`LENGTH`	Returns the number of characters in the supplied string
`HostExtractor`	Returns the host name from a URL
`DiffDate`	Returns number of days between two dates

Contributed UDF's - DataFu

- **DataFu does not ship with Pig, but is part of CDH 4.1.0 and later**
  - You will need to register the DataFu JAR file
  - In VM, at `/usr/lib/pig/datafu-0.0.4-cdh4.2.0.jar`
- **Some UDFs in DataFu include (package names omitted for brevity)**

Class Name	Description
`Quantile`	Calculates quantiles for a data set
`Median`	Calculates the median for a data set
`Sessionize`	Groups data into sessions based on a specified time window
`HaversineDistInMiles`	Calculates distance in miles between two points, given latitude and longitude

Contributed UDF's – How to use?

- **Here is an example of using a UDF from DataFu to calculate distance**

Input data

```
37.789336    -122.401385    40.707555    -74.011679
```

Pig Latin

```
REGISTER '/usr/lib/pig/datafu-*.jar';
DEFINE DIST datafu.pig.geo.HaversineDistInMiles;

places = LOAD 'data' AS (lat1:double, lon1:double,
         lat2:double, lon2:double);

dist = FOREACH places GENERATE DIST(lat1, lon1, lat2, lon2);
DUMP dist;
```

Output data

```
(2564.207116295711)
```

Pig- UDF statements

Pig Latin UDF Statements

Types of Pig Latin UDF Statements:

REGISTER - Registers a JAR file with the Pig runtime.
DEFINE - Creates an alias for a UDF, streaming script, or a command specification.

Appendix

Pig- Processing data with an external script

- **While Pig Latin is powerful, some tasks are easier in another language**
- **Pig allows you to stream data through another language for processing**
  - This is done using the `STREAM` keyword
- **Similar concept to Hadoop Streaming**
  - Data is supplied to the script on standard input as tab-delimited fields
  - Script writes results to standard output as tab-delimited fields

Pig- Processing data with an external script

- **While Pig Latin is powerful, some tasks are easier in another language**
- **Pig allows you to stream data through another language for processing**
  - This is done using the `STREAM` keyword
- **Similar concept to Hadoop Streaming**
  - Data is supplied to the script on standard input as tab-delimited fields
  - Script writes results to standard output as tab-delimited fields

Pig- Stream example in Python

- **Our example will calculate a user's age given that user's birthdate**
  - This calculation is done in a Python script named `agecalc.py`
- **Here is the corresponding Pig Latin code**
  - Backticks used to quote script name following the alias
  - Single quotes used for quoting script name within `SHIP`
  - The schema for the data produced by the script follows the `AS` keyword

```
DEFINE MYSCRIPT `agecalc.py` SHIP('agecalc.py');
users = LOAD 'data' AS (name:chararray, birthdate:chararray);

out = STREAM users THROUGH MYSCRIPT AS (name:chararray, age:int);

DUMP out;
```

Pig- Stream example in Python

- **Python code for `agecalc.py`**

```
#!/usr/bin/env python

import sys
from datetime import datetime

for line in sys.stdin:
   line = line.strip()
   (name, birthdate) = line.split("\t")

   d1 = datetime.strptime(birthdate, '%Y-%m-%d')
   d2 = datetime.now()

   age = int((d2 - d1).days / 365)

   print "%s\t%i" % (name, age)
```

Pig- Stream example in Python

- **The Pig script again, and the data it reads and writes**

```
DEFINE MYSCRIPT `agecalc.py` SHIP('agecalc.py');
users = LOAD 'data' AS (name:chararray, birthdate:chararray);

out = STREAM users THROUGH MYSCRIPT AS (name:chararray,
age:int);

DUMP out;
```

Input data

```
andy     1963-11-15
betty    1985-12-30
chuck    1979-02-23
debbie   1982-09-19
```

Output data

```
(andy,49)
(betty,27)
(chuck,34)
(debbie,30)
```

Trouble shooting & Optimizing performance

Pig- Trouble shooting- Helping your self

- **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

Pig- 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
```

Pig- 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
```

Pig- Naming your job

- **Hadoop clusters are typically shared resources**
  - There might be dozens or hundreds of others using it
  - 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

Pig- 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
```

Pig- Data Sampling – Create sample using SAMPLE

- **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';
```

Pig- Data Sampling – Create sample using 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

Pig- Understand Data flow using ILLUSTRATE

- **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         |
-------------------------------------------------
```

Pig- General Debugging strategies

- **Use `DUMP`, `DESCRIBE`, and `ILLUSTRATE` often**
  - The data might not be what you think it is
- **Look at a sample of the data**
  - Verify that it matches the fields in your `LOAD` specification
- **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

Pig- Trouble shooting- 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 0.10.0-cdh4.2.0

$ hadoop version
Hadoop 2.0.0-cdh4.2.0
```

Pig- Tips – Runtime optimization

- 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;
```

Pig- Tips – Pig's optimizer

- **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';
```

Pig- Tips – Don’t produce output you don’t 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';
```

Pig- Tips – 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';
```

Pig- Tips – 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';
```

Pig- Tips – 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';
```

Pig- Tips – Specify the smaller data set first in 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';
```

Pig- Tips – 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) ...
```

Pig- Tips

- **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;
```

Hive and Pig

- Most business use cases can be handled by Hive or Pig
- Hive is a good choice
  - When you want to query the data
  - When you need answer to specific questions
  - If you are familiar with SQL

- Pig is a good choice
  - for ETL scripts (Extract -> Transform -> Load)
  - for pre-processing data for easier analysis
  - when there are many business transforms are steps need to be applied on the data

Comparison

Feature	Map Reduce	Pig	Hive
Record format	Key Value pairs	Tuple	Record
Data Model	User defined	int,float,string,bytes,maps,tuples,bags	int,float,string,maps,structs, lists,char,varchar,decimal
Schema	Encoded in app	Declared in script or read from loader	Read from metadata
Data location	Encoded in app	Declared in script	Read from metadata
Data format	Encoded in app	Declared in script	Read from metadata

Contact Us

Visit us on: http://www.analytixlabs.in/

For more information, please contact us: http://www.analytixlabs.co.in/contact-us/

Or email: info@analytixlabs.co.in

Call us we would love to speak with you: (+91) 9910509849

Join us on:

Twitter - http://twitter.com/#!/AnalytixLabs

Facebook - http://www.facebook.com/analytixlabs

LinkedIn - http://www.linkedin.com/in/analytixlabs

Blog - http://www.analytixlabs.co.in/category/blog/