Data Analysis with Pig

Prof. Dr. Stephan Trahasch
Offenburg University of Applied Sciences

Outline

- Pig Latin Syntax
- Loading Data
- Data Output
- Filtering and Sorting Data
- Built-in functions

Pig Latin Overview

Pig Latin is a data fow language

The fow of data is expressed as a sequence of statements

Example Pig Latin script to load, alter, 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';
```

Identifiers and Case-Sensitivity

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`

Case-Sensitivity

Whether case is significant in Pig Latin depends on context

- Pig Latin keywords 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

Common Operators in Pig Latin

Many commonly--used operators in Pig Latin are familiar to SQL users

Difference: Pig Latin uses == and != for comparison

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

List of Simple Data Types

Name	Description	Example Value
`int`	Whole numbers	`2013`
`long`	Large whole numbers	`5,365,214,142L`
`float`	Decimals	`3.14159F`
`double`	Very precise decimals	`3.14159265358979323846`
`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*

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
- In this case, Pig will assume that this value is of type double
- However, it is better to specify data types explicitly when possible
- Easiest to do this upon load using the format fieldname:type

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

Outline

- Pig Latin Syntax
- Loading Data
- Data Output
- Filtering and Sorting Data
- Built-in functions

Basic Data Loading in Pig

- Pig's default loading function is called PigStorage
- 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);
```

File patterns ("globs") are also supported

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

Using Alternate Column Delimiters

You can specify an alternate delimiter as an argument to PigStorage

This example shows how to load comma-delimited data

```
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('|');
```

HCatalog

- You have seen how to specify names, types, and paths during LOAD
- 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();
```

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;
```

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}
```

Pig Data Concepts: Atom, Tuples and Bags

Atom is a String or Number e.g. (‘apple’)

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

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's Complex Data Types: 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)}
```

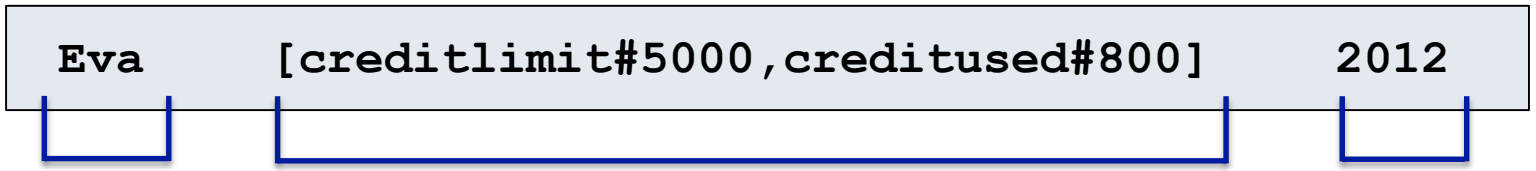
Here is the corresponding LOAD statement specifying the schema

```
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 ?le

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



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';
```

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';
```

Outline

- Pig Latin Syntax
- Loading Data
- Data Output
- Filtering and Sorting Data
- Built-in functions

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)
(Franco,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(',');
```

Outline

- Pig Latin Syntax
- Loading Data
- Data Output
- Filtering and Sorting Data
- Built-in functions

Filtering in Pig Latin

```
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

```
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

Sort Order and Limit

Use ORDER...BY to sort the records in a bag in ascending order. Add DESC to sort in descending order instead

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

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

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

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

Outline

- Pig Latin Syntax
- Loading Data
- Data Output
- Filtering and Sorting Data
- Grouping and Iterating
- Built-in functions

Grouping Records by a field

- Sometimes you need to group records by a given field
- Use GROUP BY to do this in Pig Latin
- The new relation has one record per unique value in the specified field

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

- The new relation always contains two fields
  - 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's a bag containing one tuple for each corresponding value

Example: Grouping Records By a Field

```
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,329
99), (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

```
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)
```

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

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)})
```

Outline

- Pig Latin Syntax
- Loading Data
- Data Output
- Filtering and Sorting Data
- Grouping and Iterating
- Built-in functions

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);
```

Pig has Built-in support for other aggregate functions besides SUM

- 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

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

Summary

- 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