# Pig

## Executing data flows in parallel on Hadoop

**Tomasz Bednarz** | Computational Research Scientist
3nd October 2012

MATHEMATICS, INFORMATICS AND STATISTICS / CSS TCP
www.csiro.au

CSIRO

# Pig Philosophy

- Pigs eat anything
  - Input data can come in any format – popular formats, such as tab-delimited are natively supported. Users can add functions to support other data formats.
  - Operates on data: relational, nested. semi-structured, or unstructured
- Pigs live anywhere
- Pigs are domestic animals
- Pigs fly
  - Pig processes data quickly.



CSIRO

# Pig Latin Example

- Query: get the list of pages visited by users whose age is between 20 and 25 years

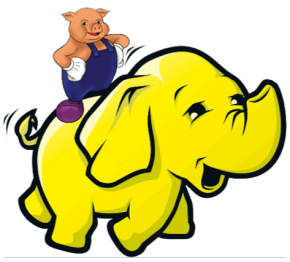
```
1  -- load users
2  users = load 'users' as (name, age);
3
4  -- all users between 20 and 25 year
5  users_20_to_25 = filter users by age > 20 and age <= 25;
6
7  -- load pages
8  page_views = load 'pages' as (user, url);
9
10 -- combine users with pages
11 page_views_u20_to_25 = join users_20_to_25 by name, page_views by user;
12
13 -- print out the result
14 dump page_views_u20_to_25;
```



# What is Pig

- Pig provides an engine for executing data flows in parallel on Hadoop
- Pig includes a language called *Pig Latin* for expressing data flows
- *Pig Latin* includes operators for many of the traditional data operations (not to be re-invented as in Hadoop): **JOIN**, **SORT**, **FILTER**, **FOREACH**, **GROUP**, **LOAD** and **STORE**.
- Pig makes use of: the Hadoop Distributed File System (HDFS) and processing system *MapReduce*

- Why?
  - Faster Development (increases productivity 10x)
    - Express data transformation tasks in just a few lines if code
    - Don't reinvent the wheel
    - 10 lines of Pig Latin = ~200 lines of Java
  - Flexible
    - Metadata is optional
    - Extensible
    - Procedural programming



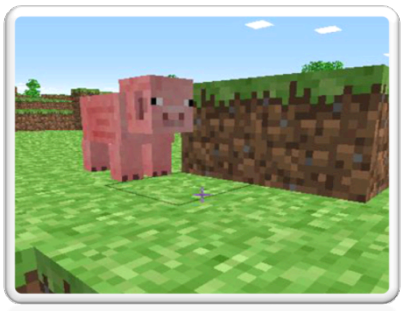
# Who uses Pig

- Yahoo! Research (90% Hadoop jobs written in Pig Latin)
- Twitter
- LinkedIn
- AOL
- Netflix
- IBM
- HartonWorks
- Autodesk
- Nokia Ovi Maps
- Mendeley



# Pig Workflow

- A **LOAD** statement reads data from the file system.
- A series of transformation statements process the data.
- A **STORE** statement writes output to the file system or, a **DUMP** statement displays output to the screen.
- Pig always at first validates the syntax and semantics of all statements and execute them only when encounters DUMP or STORE statements.



# The whole picture

**Pig Latin**

```
1  -- load the input file
2  -- call the single field in the record 'line'
3  in = load 'text-csiro.txt' as (line);
4
5  -- TOKENIZE --> splits the line into a field for each word
6  -- flatten(TOKENIZE(...)) - create collection of word(s)
7  words = foreach in generate flatten(TOKENIZE(line)) as word;
8
9  -- group records by word
10 grouped  = group words by word;
11
12 -- count words
13 out  = foreach grouped generate group, COUNT(words);
14
15 -- print out the results
16 dump out;
```

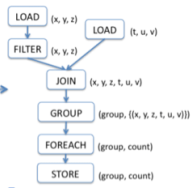
**pig.jar**

- Parses
- Checks
- Optimizes
- Plans execution
- Submits jar to Hadoop
- Monitors job progress


Pig Latin

```
A = LOAD 'file1' AS (x, y, z);
B = LOAD 'file2' AS (t, u, v);
C = FILTER A by y > 0;
D = JOIN C BY x, B BY u;
E = GROUP D BY z;
F = FOREACH E GENERATE
    group, COUNT(D);
STORE F INTO 'output';
```

Logical Plan




Execution Plan
Map:
Filter
Reduce:
Count



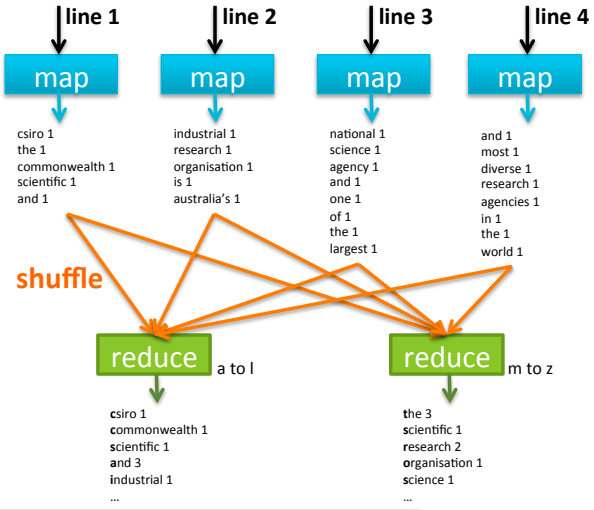
# MapReduce again

- MapReduce is a simple but powerful parallel data processing paradigm
- Every job in MapReduce consists of three main phases:
  - **Map phase**: the application operates on each record in the input separately. Many maps are started at the same time.
  - **Shuffle phase**: data is collected together by the key the user has chosen and distributed to different machines for the reduce phase. Every record for a given key will go to the same reducer.
  - **Reduce phase**: the application is presented each key, together with all of the records containing that key (done in parallel). After processing each group, the reducer can write its output.

# Words Count Example

"CSIRO the Commonwealth Scientific and Industrial Research Organisation is Australia's national science agency and one of the largest and most diverse research agencies in the world"

- **Map**: reads each line in the text one at a time, splits out each word into a separate string, and for each word output the word and a 1 to indicate it has seen the word one time.
- **Shuffle**: uses the word as the key, hashing the records to reducers.
- **Reduce**: sums up the number of times each word was seen and write that together with the word as output.






# Words Count Example

DEMO

```
1  -- load the input file
2  -- call the single field in the record 'line'
3  in = load 'text-csiro.txt' as (line);
4
5  -- TOKENIZE --> splits the line into a field for each word
6  -- flatten(TOKENIZE(...)) - create collection of word(s)
7  words = foreach in generate flatten(TOKENIZE(line)) as word;
8
9  -- group records by word
10 grouped  = group words by word;
11
12 -- count words
13 out  = foreach grouped generate group, COUNT(words);
14
15 -- print out the results
16 dump out;
```

```
(in,1)
(is,1)
(of,1)
(and,3)
(one,1)
(the,3)
(most,1)
(CSIRO,1)
(world,1)
(agency,1)
(diverse,1)
(largest,1)
(science,1)
(Research,1)
(agencies,1)
(national,1)
(research,1)
(Industrial,1)
(Scientific,1)
(Australia's,1)
(Commonwealth,1)
(Organisation,1)
```

"CSIRO the Commonwealth Scientific and Industrial Research Organisation is Australia's national science agency and one of the largest and most diverse research agencies in the world"

# Pig Latin

- Pig Latin is a dataflow language → allows users to describe how data from one or more inputs should be read, processed and stored to one or more outputs in parallel.
- Data flows can be:
  - Linear: as in the word count example
  - Complex: multiple inputs are joined and where data is split into multiple streams to be processed by different operators
- Pig Latin script describes a directed acyclic graph (DAG) where the edges are data flows and the nodes are operators that process the data
- Pig Latin has no *if* statements or *for* loops (= it focuses on data flow)
  - Traditional procedural and OO programing languages describe control flow; data flow is a side effect of the program.




# Running Pig / Starting Grunt

DEMO

- Pig supports *local* mode: useful for prototyping and debugging Pig Latin scrips. Test on small data and move to large data.
- Pig also runs in *mapreduce* mode: it does parsing, checking and planning locally, but executes MapReduce jobs on Hadoop cluster (it needs to know where *NameNode* and *JobTracker* are located).
- You can execute *Pig Latin* statements:
  - Using command line / grunt shell
  - In *local* mode or *mapreduce* mode (to interact with HDFS on your cluster)
  - Either interactively or in batch
  - Embedded Pig

```
$ pig ...
grunt> my_data_a = load 'data.txt';
grunt> my_data_b = ...
grunt> ...
```

```
$ pig -x local my_script.pig
```

```
$ pig my_script.pig
```



## Grunt – controlling Pig

```
File system commands:
    fs <fs arguments> - Equivalent to Hadoop dfs command: http://hadoop.apache.org/common/docs/current/hdfs_shell.html
Diagnostic commands:
    describe <alias>[::<alias] - Show the schema for the alias. Inner aliases can be described as A::B.
    explain [-script <pigscript>] [-out <path>] [-brief] [-dot] [-param <param_name>=<param_value>]
        [-param_file <file_name>] [<alias>] - Show the execution plan to compute the alias or for entire script.
        -script - Explain the entire script.
        -out - Store the output into directory rather than print to stdout.
        -brief - Don't expand nested plans (presenting a smaller graph for overview).
        -dot - Generate the output in .dot format. Default is text format.
        -param <param_name - See parameter substitution for details.
        -param_file <file_name> - See parameter substitution for details.
        alias - Alias to explain.
    dump <alias> - Compute the alias and writes the results to stdout.
Utility Commands:
    exec [-param <param_name>=param_value] [-param_file <file_name>] <script> -
        Execute the script with access to grunt environment including aliases.
        -param <param_name - See parameter substitution for details.
        -param_file <file_name> - See parameter substitution for details.
        script - Script to be executed.
    run [-param <param_name>=param_value] [-param_file <file_name>] <script> -
        Execute the script with access to grunt environment.
        -param <param_name - See parameter substitution for details.
        -param_file <file_name> - See parameter substitution for details.
        script - Script to be executed.
    sh  <shell command> - Invoke a shell command.
    kill <job_id> - Kill the hadoop job specified by the hadoop job id.
    set <key> <value> - Provide execution parameters to Pig. Keys and values are case sensitive.
        The following keys are supported:
        default_parallel - Script-level reduce parallelism. Basic input size heuristics used by default.
        debug - Set debug on or off. Default is off.
        job.name - Single-quoted name for jobs. Default is PigLatin:<script name>
        job.priority - Priority for jobs. Values: very_low, low, normal, high, very_high. Default is normal
        stream.skippath - String that contains the path. This is used by streaming.
        any hadoop property.
    help - Display this message.
    quit - Quit the grunt shell.
grunt>
```

# Pig data types – scalar types

| Category | Type | Description | Literal Example |
|---|---|---|---|
| Numeric | int | Signed 32-bit integer | 3 |
| Numeric | long | Signed 64-bit integer | 40L |
| Numeric | float | 32-bit floating point number | 3.14F |
| Numeric | double | 64-bit floating point number | 3.14 |
| Text | chararray | Character array (string) in Unicode UTF-8 | ‘a’ |
| Binary | bytearray | Blob of binary data | |

# Pig data types – complex types

| Category | Type | Description | Literal Example |
|---|---|---|---|
| Complex | map | An associative array - a set of key-value pairs. Keys must be character arrays; values may be of any type.<br>The key is used as an index to find value. | [‘name’#’Tom’, ‘age’#15] |
| Complex | tuple | Sequence of fields of any type | (‘Tom’, 15) |
| Complex | bag | An unsorted collection of tuples, possibly with duplicates | {(‘Tom’, 15), (‘John’, 30), (‘Luke’, 12)} |



# Schemas in Pig

- Pig has a very lax attitude when it come to schemas (= Pig eats everything)
- If schema for data is available, Pig will use it
- If schema for data is not available, Pig will process the data and will make the best guesses (on how script treats data)

```
1  -- load rainfall data downloaded from BOM
2  A = load 'data.cvs' using PigStorage(',') as (code:chararray,
3      station:chararray, year:int, month:int, day:int,
4      rainfall:int, period:int, quality:chararray);
5  -- histogram of rainfalls
6  X = group A by rainfall;
7  Y = foreach X generate group, COUNT(A);
8  dump Y;
```

**Loaders**
PigStorage
BinStorage
HBaseStorage
CassandraStorage
AvroStorage
JsonLoader/JsonStorage

# Few hints

DEMO

- Use **LIMIT** to specify how many tuples (rows) to return back

```
grunt> lmt = LIMIT log 4;
grunt> DUMP lmt;
(2A9EABFB35F5B954,970916105432L,+md foods +proteins)
(BED75271605EBD0C,970916001949L,yahoo chat)
(BED75271605EBD0C,970916001954L,yahoo chat)
(BED75271605EBD0C,970916003523L,yahoo chat)
```

- Use **DESCRIBE** to check the schema

```
grunt> DESCRIBE log;
log: {user: chararray,time: long,query: chararray}
```

- Use **ILLUSTRATE** to see how Pig process data step-by-step
- Use **EXPLAIN** to show logical and physical plan in detail



# Schemas in Pig

| Data type | Syntax | Example |
|---|---|---|
| int / long | int / long | as (a:int) / as (a:long) |
| float / double | float / double | as (a:float) / as (a:double) |
| chararray | chararray | as (a:charrray) |
| bytearray | bytearray | as (a:bytearray) |
| map | map[] or map[*type*] – this declares all values in the map of this valid *type* | as (a:map[], b:map[int]) |
| tuple | tuple() or tuple (*list_of_fields*), where *list_of_fields* is a comma separated list of field declarations | as (a:tuple(), b:tuple(x:int, y:int)) |
| bag | bag() or bag{t:(*list_of_fields*)} where *list_of_fields* is comma separated list of field declaration. The tuple inside the bag must have a name, here specified as t, even though not accessible directly. | (a:bag{}, b:bag(t:(x:int, y:int)}) |

| Pig Command | What it does |
|---|---|
| **load** | Read data from file system |
| **store** | Write data to file system |
| **foreach** | Apply expression to each record and output one or more records |
| **filter** | Apply predicate and remove records that do not return true |
| **group / cogroup** | Collect records with the same key from one or more inputs |
| **join** | Join two or more inputs based on a key |
| **order** | Sort records based on a key |
| **distinct** | Remove duplicate records |
| **union** | Merge two data sets |
| **split** | Split data into 2 or more sets, based on filter conditions |
| **stream** | Send all records through a user provided binary |
| **dump** | Write output to stdout |
| **limit** | Limit the number of records |




# Parameter(s) substitution

- Pig supports parameter substitution, so user can specify them at runtime
- Denote such parameters by the $ prefix within the script

```
log = LOAD '$input' AS (user, time, query);
lmt = LIMIT log $size; DUMP lmt;
```

- To run script:

```
pig -param input=my_data.log -param size=4 my_script.pig
```

- Possible to substitute Unix commands by using back ticks (`):

```
pig -param input=web-`date +%y-%m-%d`.log -param size=4 my_script.pig
```

- When you have many parameters, possible to use file **my_params.txt** to specify them:

```
# comment
input=my_data.log
size=4
```

```
pig –param_file my_params.txt my_script.pig
```



- **UNION** combines multiple relations together
- **SPLIT** partitions a relation into multiple ones

```
grunt> a = load 'A' using PigStorage(',') as (a1:int, a2:int, a3:int);
grunt> b = load 'B' using PigStorage(',') as (b1:int, b2:int, b3:int);
grunt> DUMP a;
(0,1,2)
(1,3,4)
grunt> DUMP b;
(0,5,2)
(1,7,8)
grunt> c = UNION a, b;
grunt> DUMP c;
(0,1,2)
(0,5,2)
(1,3,4)
(1,7,8)
grunt> SPLIT c INTO d IF $0 == 0, e IF $0 == 1;
grunt> DUMP d;
(0,1,2)
(0,5,2)
grunt> DUMP e;
(1,3,4)
(1,7,8)
```

- Use **DISTINCT** operator to remove duplicates



- **FILTER** operation trims a relation down to only tuples that pass a certain test

```
grunt> DUMP c;
(0,1,2)
(0,5,2)
(1,3,4)
(1,7,8)
grunt> f = FILTER c BY $1 > 3;
grunt> DUMP f;
(0,5,2)
(1,7,8)
```

- **GROUP** operation (output has always two fields – 1st group key, 2nd bag)

```
grunt> g = GROUP c BY $2;
grunt> DUMP g;
(2,{(0,1,2),(0,5,2)})
(4,{(1,3,4)})
(8,{(1,7,8)})
grunt> DESCRIBE c;
c: {a1: int,a2: int,a3: int}
grunt> DESCRIBE g;
g: {group: int,c: {a1: int,a2: int,a3: int}}
grunt> h = GROUP g ALL;
(all,{(0,1,2),(0.5,2),(1,3,4),(1,7,8)})
grunt> i = FOREACH h GENERATE COUNT($1); dump i;        ← counts number of tuples
(4L)
```



- **COGROUP** groups together tuples from multiple relations

```
grunt> DUMP a;
(0,1,2)
(1,3,4)
grunt> DUMP b;
(0,5,2)
(1,7,8)

grunt> j = COGROUP a BY $2, b BY $2;
grunt> DUMP j;
(2,{(0,1,2)},{(0,5,2)})
(4,{(1,3,4)},{})
(8,{},{(1,7,8)})
grunt> DESCRIBE j;
j: {group: int,a: {a1: int,a2: int,a3: int},b: {b1: int,b2: int,b3: int}}
```

- **GROUP** always generates two fields in its output, **COGROUP** always generates three (more if co-grouping more than two relations). The first field is the group key, the second and third fields are bags. These bags hold tuples from the co-grouping relations that match the grouping key.
- If the grouping key matches only tuples from one relation but not the other, then the field corresponding to the nonmatching relation will have an empty bag.




- **INNER** in **COGROUP** ignores group keys that don't exist for a relation

```
grunt> DUMP a;
(0,1,2)
(1,3,4)
grunt> DUMP b;
(0,5,2)
(1,7,8)

grunt> j = COGROUP a BY $2, b BY $2 INNER;
grunt> dump j;
(2,{(0,1,2)},{(0,5,2)})
(8,{},{(1,7,8)})
grunt> j = COGROUP a BY $2 INNER, b BY $2 INNER;
grunt> dump j;
(2,{(0,1,2)},{(0,5,2)})
```

- **JOIN** creates a flat set of output records, as indicated by looking at the schema:

```
grunt> j = JOIN a BY $2, b BY $2;
grunt> dump j;
(0,1,2,0,5,2)
grunt> DESCRIBE j;
j: {a::a1: int,a::a2: int,a::a3: int,b::b1: int,b::b2: int,b::b3: int}
```



- **FOREACH** goes through all tuples in a relation and generates new tuples as output

```
grunt> DUMP c;
(0,1,2)
(0,5,2)
(1,3,4)
(1,7,8)
grunt> k = FOREACH c GENERATE a2, a2 * a3;
grunt> DUMP k;
(1,2)
(5,10)
(3,12)
(7,56)
grunt> k = FOREACH g GENERATE group, COUNT(c);
grunt> DUMP k;
(2,2L)
(4,1L)
(8,1L)
grunt> k = FOREACH g GENERATE group, FLATTEN(c);
grunt> DUMP k;
(2,0,1,2)
(2,0,5,2)
(4,1,3,4)
(8,1,7,8)
grunt> DESCRIBE k;
k: {group: int,c::a1: int,c::a2: int,c::a3: int}
```



# UDF in Pig

- Benefits
  - Use legacy code
  - Use library in scripting language
  - Leverage Hadoop for non-Java programmers
- Currently supported languages
  - Python
  - JS
  - Ruby
- Extensible Interface
  - Minimum effort to support another language

# Writing UDF


DEMO

- Use Jython to call functions (Jython is implementation of the Python written in Java)

```
register 'test.py' using jython as myfuncs;
numbers = load 'test.txt' as (a:int);
b = foreach numbers generate myfuncs.helloworld(), myfuncs.square(a);
dump b;
```

- Python code

```
1  #!/usr/bin/python
2
3  @outputSchema("word:chararray")
4  def helloworld():
5      return 'Hello, World'
6
7  @outputSchemaFunction("squareSchema")
8  def square(num):
9      return ((num)*(num))
```



# Facts, links, resources

- Open source, Apache 2.0 license Official subproject of Apache Hadoop
- Version 0.10 released in 25 April 2012
- Get Pig http://pig.apache.org
- Easy install http://www.cloudera.com
- PiggyBank UDFs https://cwiki.apache.org/confluence/display/PIG/PiggyBank
- DataFu https://github.com/linkedin/datafu

# DataFu https://github.com/linkedin/datafu

- DataFu is a collection of user-defined functions for working with large-scale data in Hadoop and Pig.
- This library was born out of the need for a stable, well-tested library of UDFs for data mining and statistics.
- Used at LinkedIn in many of our off-line workflows for data derived products like "People You May Know" and "Skills". It contains functions for:
  - PageRank
  - Quantiles (median), variance, etc.
  - Convenience bag function (e.g., set operations, enumerating bags, etc.)
  - Convenience utility function (e.g., assertations, etc.)
  - And more…




# DataFu

- Compute the median of sequence of sorted bags:

```
-- input: 3,5,4,1,2
input = LOAD 'input' AS (val:int);
grouped = GROUP input ALL;
-- produces median of 3
medians = FOREACH grouped {
  sorted = ORDER input BY val;
  GENERATE Median(sorted.val);
}
```

- Treat sorted bags as sets and compute their intersection

```
define SetIntersect datafu.pig.bags.sets.SetIntersect();

-- ({(3),(4),(1),(2),(7),(5),(6)},{(0),(5),(10),(1),(4)})
input = LOAD 'input' AS (B1:bag{T:tuple(val:int)},B2:bag{T:tuple(val:int)});

-- ({(1),(4),(5)})
intersected = FOREACH input {
  sorted_b1 = ORDER B1 by val;
  sorted_b2 = ORDER B2 by val;
  GENERATE SetIntersect(sorted_b1,sorted_b2);
}
```



# Problem

DEMO

- We have list of local ABC Radio stations in Australia
- We have list of all Public Toilets across Australia
- We want to find closest toilet to a Radio Station

- See **example_abc_toilets**

```
-- ABC local stations
-- Source http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv
-- (a) state,
-- (b) website-URL,
-- (c) station,
-- (d) town
-- (e) latitude,
-- (f) longitude,
-- (g) talkback-number,
-- (h) Enquiries-number,
-- (i) Fax-number
-- (j) Sms-number
-- (k) Street-number
-- (l) Street-suburb
-- (m) Street-postcode
-- (n) PO-box
-- (o) PO-suburb
-- (p) PO-postcode
-- (r) Twitter
-- (s) Facebook

-- National Public Toilet Map
-- Source http://data.gov.au/dataset/national-public-toilet-map/
-- (a) toilet name;
-- (b) address;
-- (c) latitude and longitude;
-- (d) general toilet features;
-- (e) location;
-- (f) accessibility;
-- (g) opening hours;
-- (h) additional features (e.g. showers, baby change facilities etc);
-- (i) notes (e.g. coin operated showers etc).
```




# Thank you

**CMIS / CSS TCP**
Tomasz Bednarz
**t** +61 2 9325 3213
**e** tomasz.bednarz@csiro.au
**w** www.csiro.au/cmis