# CMU MSP 36602 Apache Pig, Part 1

**H. Seltman, Mar 27 2019**

1) **Apache Pig** is a system to perform MapReduce analyses using a high-level language called **Pig Latin**. It is intended to be easier to use that Java or Python, automatically optimized, and extensible (using Java, Python, or C++).

   a) According to https://pig.apache.org/philosophy.html, the ***name "***Pig" is due to these facts:

      i) Pigs Eat Anything: Pig can work with many different kinds of data.

      ii) Pigs Live Anywhere: Pig is a language for parallel data processing that is not tied to one particular parallel framework.

      iii) Pigs Are Domestic Animals: Pig is designed to be easily controlled, modified, and extended by its users.

      iv) Pigs Fly: Pig processes data quickly. Implementation of features in ways that weigh Pig down so it can't fly are avoided.

**2) Testing**

   a) Check that the pig bin folder is in your path.

   b)
```
One time setup for our examples
mkdir ~/pigex
cd ~/pigex
pig -help
```

   c) Test MapReduce mode (first check `jps` to assure Hadoop is running):
```
pig
help
fs -ls
quit
```

   d) Test Local mode:
```
pig -x local
pwd = LOAD '/etc/passwd' USING PigStorage(':');
pwd = FOREACH pwd GENERATE $0 AS id;
DUMP pwd;
quit
```

   e) Test Batch mode:
   Load and view 602/code/id0.pig.
   Run the lines at the Linux prompt:
```
pig -x local id.pig
ls –alrt out
cat out/part-m-00000
hexdump out/.part-m-00000.crc  # For curiosity sake
```

   f) Batch mode with parameters:
   Load and view 602/code/id.pig.
   Run this line at the Linux prompt:
```
pig -x local -param data=/etc/passwd -param out=newout id.pig
```
   (Can also use -param_file=filename, with parameters in the file.)

3) Learning **Pig Latin**

a) Pig Latin describes how you want to carry out a (big) data analysis. Like any analysis, it has input, processing, and output. The input and output are files. The statements are generally ***organized*** as follows:

One or more LOAD statements to read data from the file system.
A series of transformation statements (**relation operators**) to process the data (**relations**)
A DUMP statement to view results for debugging or one or more STORE statements to save the results to the file system.

b) Pig Latin ***general characteristics***:

i) **case sensitive** for relation names, field names, and function names, but case insensitive for keywords

ii) requires a "`;`" **line terminator**

iii) uses either `/* */` **comments** or trailing double dash comments, e.g.:

```
myStatement; -- this is my comment
```

iv) uses **identifiers** in the form of a letter followed by any number of letters, digits, and underscores

v) uses standard **operators**

(1) Arithmetic: `+`, `-`, `/`, and `*` plus `%` for modulo and `?:` for 'ifelse' (`a==1 ? 'x' : 'y'`)

(2) Boolean: `and, or, not`

(3) Comparison: `==, !=, <, <=, >, >=, is null, is not null`

vi) Generally, you need to **use single quotes** rather than double quotes.

vii) Use `(newtype)myVar` to **cast** data to a different type.

c) ***Definitions***:

i) The smallest useable amount of data is a **field**. A field can be of any of the primary data types, which are int, long, float, double, chararray (utf-8), bytearray, boolean (true or false), datetime, biginteger or bigdecimal, or it can be a **tuple** or **bag** or **map** (see below).

ii) A **tuple** (think data table row) is an ordered set of fields, using the syntax (`field1, …`).

iii) A **bag** is a collection of tuples using the syntax `{tuple1, …}`.

iv) A **relation** (a.k.a. outer bag) is a bag that is not inside any other bag (think data table).

v) A **map** is a set of key/value pairs of the form [key#value, …] where key is a chararray (string) and value is of any type. Because lookup is only by constant, not variable, these are seldom used.

vi) ***Missing data*** are coded as SQL-style **null**, checked with `is null` or `is not null`.

d) The **LOAD** statement reads data from a file into a relation

   i) `A = LOAD 'myfile.txt';` by ***default*** reads in tab separated data from the file and stores it as unnamed fields of type "bytearray" (ascii string). This is equivalent to `A = LOAD 'myfile.txt' USING PigStorage('\t');`, and you can change the delimiter from tab to anything else.

   ii) If a ***directory*** is used in place of a file name, all files in that directory are read (sequentially).

   iii) Compressed files are allowed and are automatically unzipped first. A gzip file with a ".gz" extension or a bzip file with a ".bz" extension are supported.

   iv) There are several ways to remove the header line of a csv file. One ugly way is to use tail –n +2 a.csv > b.csv in UNIX. Another, when a column name is known is to use B = FILTER A BY myColName != 'myColName'; in Pig. Note that the header line may be removed automatically if it cannot be converted according to the schema.

e) A **schema** is similar to a SAS program-data-vector description, and consists of the field names and data types. If you load data without a schema, a default schema is used, in which all data types are bytearray and there are no field names.

   i) **Defining a schema:** To ***specify variable names and or types***, add `AS (mySchema)` to the LOAD command. The form of "mySchema" is a comma-separated list of field names. Optionally data types can be specified for any field using `:myDatatype`, where the data types are listed above.

   E.g.:
```
Data = load 'student.csv' using PigStorage(',') as (id:bytearray,
        age:double, gender:bytearray, graduated:boolean);
```

   The schema only needs to be entered up to the last column you need.

   Unnamed fields are referenced using **positional notation**, starting at 0 and preceded with a "$". E.g., there is a FOREACH / GENERATE statement which extracts fields from a relation and produces a new relation. So `Data = FOREACH Data GENERATE id, $1;` will keep "id" and "age", and drop the "gender" and "graduated fields".

   ***Specification of schemas with AS should be used whenever possible!***

   ii) **Getting schema information**: If the relation has a schema, then you can used the command `DESCRIBE myRelation;` to see the field names and data types, and you can use `ILLUSTRATE myRelation;` to show data from the first "row" along with field names and data type for the specified relation and all others relations it depends on.

   iii) There are alternatives to PigStorage() in the **USING clause**. BinStorage() is based on Pig's internal binary storage format, and you should not need to use it. JsonLoader() handles Json data. TextLoader() loads unstructured UTF-8 data, similar to R's readLines() function. Other specific loaders exist, and you can write your own, too.

f) The **DUMP myRelationName;** statement sends output to the terminal showing what is in the relation, for debugging purposes.

g) The **STORE myRelationName into myTextFile;** statement stores a relation to a data file. Adding **USING PigStorage(',')**, e.g., changes the delimiter from tab to comma.

h) The **FILTER relational operator** eliminates tuples from a relation based on specified conditions E.g.,

```
school = LOAD 'school.csv' USING PigStorage(',') AS (schoolNum:int,
  classNum:int, classSize:int, rx:int, pupilNum:int, ses:double,
  budget:double, teachExp:double, teachEnth:double, male:int,
  minority:int, scPre:int, scMid:int, scPost:int);
hiSesSchools = FILTER school BY ses > 0;
```

i) The **ORDER BY** relational operator sorts the data. E.g., using the same "school" data we can assure that the data are sorted first by rx with treated=2 before control=1, then by increasing classNum as follows:

```
school = ORDER school BY rx DESC, classNum ASC;
```

j) The **GROUP** relational operator combines related tuples in a relation.

The data in "simple.tab" is:

```
Howard      computers   3
Howard      phones      1
Howard      cars        0
Joel  computers   2
Joel  cars        1
```

The code in "simpleGroup.pig" is:

```
simple = LOAD 'simple.tab' AS (name, item, count:int);
grp = GROUP simple by name;
DUMP grp;
```

The result of running `pig -x local simpleGroup.pig` is:

```
(Joel,{(Joel,cars,1),(Joel,computers,2)})
(Howard,{(Howard,cars,0),(Howard,phones,1),(Howard,computers,3)})
```

The new relation "grp" has 2 tuples, each of which is made of a string (chararray) called "group" and an "inner" bag called "simple" (because "simple" is the first argument of GROUP) containing one or more tuples with names from the original tuples.

More complex groupings result from using the COGROUP relational operator, which combines data from several relations into a single relation.

k) The **FOREACH** relational operator creates a new relation with different fields than the original relation by dropping un-needed fields or by adding new fields which are constructed from one or more old fields and available functions. There are two forms, one for "blocks" and one for "nested blocks."

The block syntax works on outer bags in the form:

```
relationOut = FOREACH relationIn GENERATE field1, …,
                expression1 AS newField1:myDataType, …;
```

Fields that are not mentioned are dropped and expressions, e.g., (a+b)/2 are added using the "AS" name. "*" can be used to represent all fields in the input relation to add without dropping.

The block syntax can work with the result of a GROUP operation. Continuing the example from above:

```
personInfo = FOREACH grp GENERATE group, simple.item, simple.count;
```

Here is a DUMP of personInfo:

```
(Joel,{(cars),(computers)},{(1),(2)})
(Howard,{(cars),(phones),(computers)},{(0),(1),(3)})
```

And here is the DESCRIBE result:

```
personInfo: {group: bytearray,{(item: bytearray)},{(count: int)}}
```

Here is the result of using the sum function and the $ field-by-position operator:

```
sums = FOREACH personInfo GENERATE group, SUM($2.count);
DUMP sums;
```

```
(Joel,3)
(Howard,4)
```

Various functions (https://pig.apache.org/docs/latest/func.html) are available, e.g.:

You can use `R2 = FOREACH R1 GENERATE ..., REPLACE(myField, '"', '') AS myField, ...;` to ***remove unneeded quotes*** or make other changes to text.

In a GENERATE, the **FLATTEN function** can be used to do two things. First, `FLATTEN(myTuple)` will pull the elements of the tuple out to the next higher level, so that, e.g., if `$1` is (3, 7, 2), then `FLATTEN($1)` is the same as `$1.$0, $1.$1, $1.$2`. Second, `FLATTEN(myBag)` will generate a separate "unpacked" tuple for each tuple in the bag (i.e., increase the number of "rows" in the relation). E.g., if a relation has a row (tuple) equal to `('a', {(1,5), (2,4)})`, then `GENERATE $0, FLATTEN($1)` results in two rows (tuples): `('a', 1, 5)` and `('b', 2, 4)`.

l) The **nested form of FOREACH** is more complex. You can carry out several intermediate operations to achieve your final goal:

```
compPhone = FOREACH grp {
    notCars = filter simple by item != 'cars';
    cnts = notCars.count;
    GENERATE group, SUM(cnts);
};
```

The result of `DUMP compPhone;` is

(Joel,2)
(Howard,4)

m) The **`DISTINCT`** relational operator, as in `uStudent = DISTINCT student;` removes any duplicate tuples.

n) The **`UNION`** relational operator, as in `students = UNION undergrads, grads;` works like R's rbind(), but does not care if the fields are matching in number, name, or type.

o) The **`SPLIT`** relational operator, as in `SPLIT students INTO undergrad IF type=='U ', grad IF type=='G';` is similar to split() in R when applied to data.frames.

p) The **`CROSS`** relational operator, is in `allCombos = CROSS pizzaVeggies, pizzaMeats;` makes a new relation with all combinations from both original relations.

q) The JOIN operator joins on common fields, e.g., `buy = JOIN purchases BY id, custInfo BY cid;` would give one tuple per purchase, but including the corresponding customer information for each purchase. Other forms of join, including outer joins, are possible.

```
prices.tab has:
computers   1000
phones      700
cars  20000
```

```
pr = LOAD 'prices.tab' AS (item, price:int);
j = JOIN simple BY item, pr by item;
dump j;
```

```
(Joel,cars,1,cars,20000)
(Howard,cars,0,cars,20000)
(Howard,phones,1,phones,700)
(Joel,computers,2,computers,1000)
(Howard,computers,3,computers,1000)
```

4) In-class exercise: Analyze data/SalesJan2009.csv, which has columns for Transaction_date, Product, Price, Payment_Type, Name, City, State, Country, Account_Created, Last_Login, Latitude, and Longitude. As a simple start, find the number of transactions per country.