The Hadoop Stack, Part 1
Introduction to Pig Latin

CSE 40822 – Cloud Computing – Fall 2018

Prof. Douglas Thain
University of Notre Dame

Three Case Studies

- Workflow: Pig Latin
    - A dataflow language and execution system that provides an SQL-like way of composing workflows of multiple Map-Reduce jobs.
- Storage: HBase
    - A NoSQl storage system that brings a higher degree of structure to the flat-file nature of HDFS.
- Execution: Spark
    - An in-memory data analysis system that can use Hadoop as a persistence layer, enabling algorithms that are not easily expressed in Map-Reduce.

References

- Christopher Olston, et al, Pig Latin: A Not-so-Foreign Language for Data Processing, SIGMOD 2008.
  - http://dl.acm.org/citation.cfm?id=1376726
- Alan Gates et al. Building a high-level dataflow system on top of Map-Reduce: the Pig experience, VLDB 2009.
  - http://dl.acm.org/citation.cfm?id=1687568
- Apache Pig Documentation
  - http://pig.apache.org

Pig Latin Overview

- Map-Reduce programs can be challenging to write, and are limited to performing only one analysis step at a time.
- Pig Latin: Chain together multiple Map-Reduce runs, each one expressed by a compact statement similar to SQL.
- By using a high level query language, the system can optimize the order of operations in order to produce a better plan.
- Pig Pen: Test out queries on sampled data.

Dataflow Programming

- In a dataflow programming system (an old idea), a program is a directed graph of the steps by which multiple data items are transformed from input to output. The programmer is **not** responsible for choosing the order in which items are processed.

Procedural Programming

```
int A[] = { 10, 20, 30, … };
for( i=0; i<1000; i++ )
     B[i] = F( A[i] );
for( i=0; i<1000; i++ )
     C[i] = G( A[i] );
```

Dataflow Programming

```
set A = { 10, 20, 30, … }
set B = apply F(x) to A
set C = apply G(x) to A
```

Sample Objective

- Find each category that contains more than 10^6 entries with a pagerank of 0.2 or higher.

category	url	pagerank
Academic	www.nd.edu	3.0
Commercial	www.Studebaker.com	1.5
Non-Profit	www.redcross.org	5.3
. . .	(billions of rows)	. . .

If we did it in Map-Reduce:

Round One – Compute Averages by Category

```
map( data ) {
    split line into category, url, pagerank
    if(pagerank>0.2) emit(category,pagerank)
}

reduce( key, list( values ) ) {
     for each item in list {
             count++;
             total+=value;
     }
     emit( count, category, total/count );
}
```

Round Two – Collect Results into One File

```
map( data ) {
     split line into count, category, average
     if(count>10^6) emit(1,(category));
}

reduce( key, list( values ) ) {
      for each value in list {
              emit(value);
      }
}
```

If we did it in SQL…

```
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
```

category	AVG(pagerank)
Academic	1.75
Commercial	0.56
Non-Profit	3.23
. . .	. . .

Same thing in Pig Latin…

```
urls = LOAD “urls.txt” AS (url, category, pagerank)
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)> 10^6;
output = FOREACH big_group GENERATE category,
                                       AVG(good_urls.pagerank);
```

Basic Idea

- Express queries in Pig Latin.
- Pig translates statements into DAG.
- DAG is translated into Map-Reduce Jobs.
- Map-Reduce jobs run in Hadoop.

- Encourage Unstructured, Nested Data
- User Defined Functions:
  - A small bit of Java that can be imported into the query language.
  - Could potentially be a large/expensive piece of code.

Normalized Database (SQL)

ProductID	Name
1	Bicycle
2	Tricycle
3	Unicycle

ProductID	PartID	Quantity
1	1	2
1	3	1
1	2	1
2	1	3
2	3	1
2	4	1

PartID	Name	Price
1	Wheel	10
2	Chain	15
3	Handlebars	3
4	Seat	12

Codd's definitions of normal forms:
1NF - The domain of each attribute contains only atomic values, and the value of each attribute contains only a single value from that domain.
2NF - No non-prime attribute in the table is functionally dependent on a proper subset of any candidate key.
3NF - Every non-prime attribute is non-transitively dependent on every candidate key in the table. The attributes that do not contribute to the description of the primary key are removed from the table. In other words, no transitive dependency is allowed.

Nested Data (Pig)

JSON-Like Syntax:

Atom: ‘hello’ or 25
Ordered Tuple: (a,b,c)
Unordered Bag: {a,b,c}
Map: {a->x, b->y, c->z}

```
{
        Name-> ‘Bicycle’,
        Price -> 105,
        Parts -> {
                (1, ‘Wheel’, 2, 10.00 ),
                (2, ‘Chain’, 1, 15.00 ),
                (3, ‘Handlebars’, 1, 3.00),
                …
        };
        Name -> ‘Tricycle’,
        Price -> $55,
        Parts -> {
                (1, ‘Wheel’, 3),
                (3, ‘Handlebars’,1),
                …
        }
}
```

$$\texttt{t} = \left(\texttt{'alice'}, \left\{ \begin{array}{c} \texttt{('lakers', 1)} \\ \texttt{('iPod', 2)} \end{array} \right\}, [\texttt{'age'} \rightarrow 20] \right)$$

Let fields of tuple `t` be called `f1`, `f2`, `f3`

Expression Type	**Example**	**Value for `t`**
Constant	`'bob'`	Independent of `t`
Field by position	`$0`	`'alice'`
Field by name	`f3`	$[\texttt{'age'} \rightarrow 20]$
Projection	`f2.$0`	$\left\{ \begin{array}{c} \texttt{('lakers')} \\ \texttt{('iPod')} \end{array} \right\}$
Map Lookup	`f3#'age'`	20
Function Evaluation	`SUM(f2.$1)`	`1 + 2 = 3`
Conditional Expression	`f3#'age'>18? 'adult':'minor'`	`'adult'`

Flattening	`FLATTEN(f2)`	`'lakers', 1`
		`'iPod', 2`

Table 1: Expressions in Pig Latin.

Operations

- bag = LOAD “file” USING format AS (field-list)
- bag = FILTER bag BY function
- bag = FOREACH bag GENERATE expression-list
- bag = DISTINCT bag
- bag = ORDER bag BY expression-list
- bag = GROUP bag BY (field-list)
- bag = COGROUP bag-list BY (field-list)
- STORE bag INTO file USING format

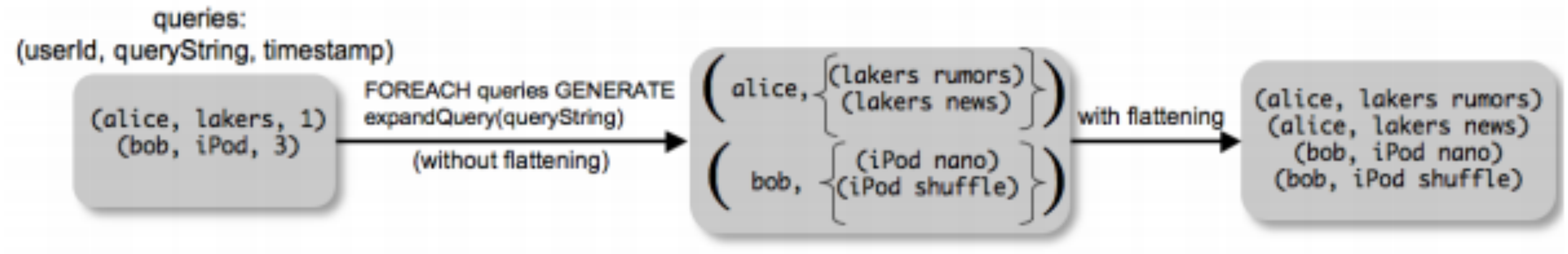


Figure 1: Example of *flattening* in `FOREACH`.

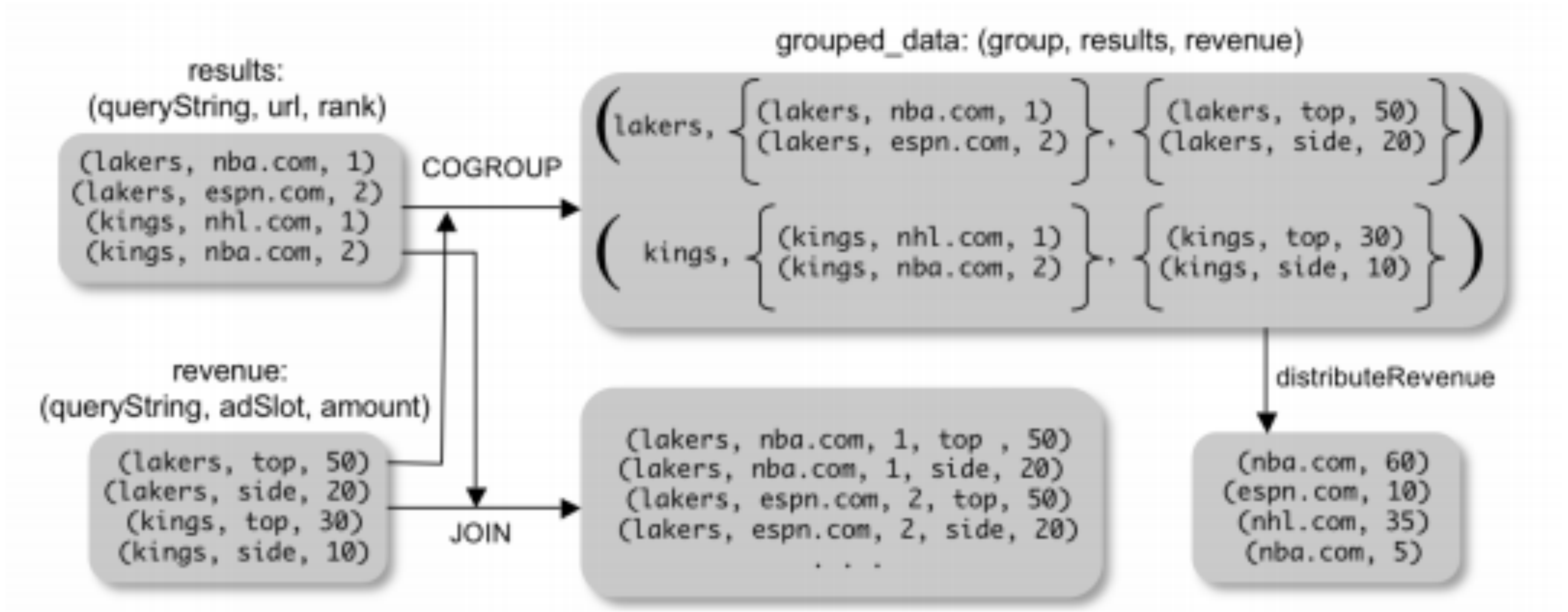

results:
(queryString, url, rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)
COGROUP
grouped_data: (group, results, revenue)
lakers, (lakers, nba.com, 1) (lakers, espn.com, 2), (lakers, top, 50) (lakers, side, 20)
kings, (kings, nhl.com, 1) (kings, nba.com, 2), (kings, top, 30) (kings, side, 10)
distributeRevenue
revenue:
(queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)
JOIN
(lakers, nba.com, 1, top , 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)
. . .
(nba.com, 60)
(espn.com, 10)
(nhl.com, 35)
(nba.com, 5)

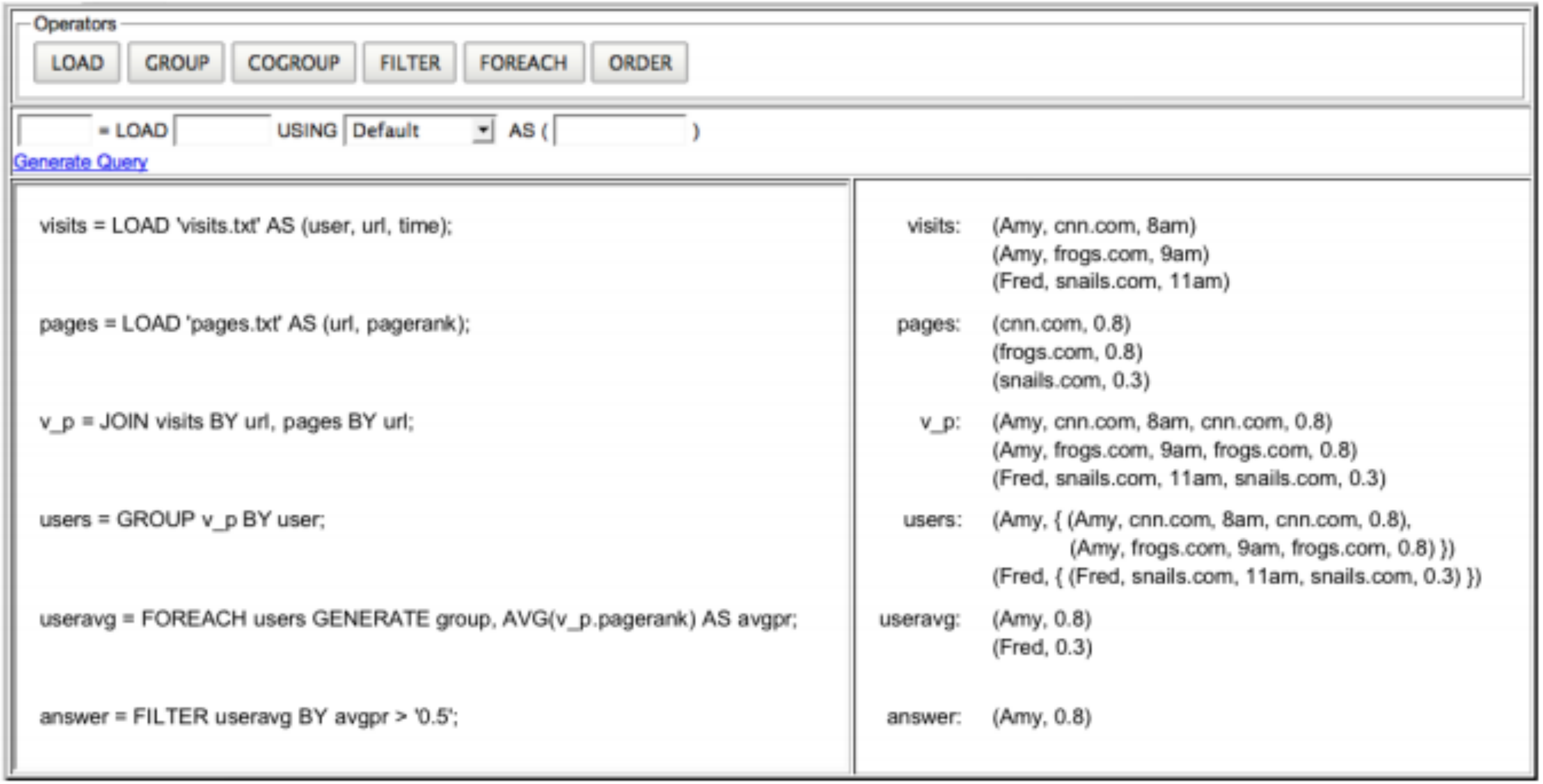
Figure 2: `COGROUP` **versus** `JOIN`**.**

From Pig to Map-Reduce

```
urls = LOAD “urls.txt” AS (url, category, pagerank)
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)> 10^6;
output = FOREACH big_group GENERATE category,
                                    AVG(good_urls.pagerank);
```

Example on Board:

Code -> DAG -> Map-Reduce Jobs



```
visits = LOAD 'visits.txt' AS (user, url, time);          visits:   (Amy, cnn.com, 8am)
                                                                    (Amy, frogs.com, 9am)
                                                                    (Fred, snails.com, 11am)

pages = LOAD 'pages.txt' AS (url, pagerank);               pages:    (cnn.com, 0.8)
                                                                    (frogs.com, 0.8)
                                                                    (snails.com, 0.3)

v_p = JOIN visits BY url, pages BY url;                    v_p:      (Amy, cnn.com, 8am, cnn.com, 0.8)
                                                                    (Amy, frogs.com, 9am, frogs.com, 0.8)
                                                                    (Fred, snails.com, 11am, snails.com, 0.3)

users = GROUP v_p BY user;                                 users:    (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8),
                                                                             (Amy, frogs.com, 9am, frogs.com, 0.8) })
                                                                    (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) })

useravg = FOREACH users GENERATE group, AVG(v_p.pagerank) AS avgpr;   useravg:  (Amy, 0.8)
                                                                    (Fred, 0.3)

answer = FILTER useravg BY avgpr > '0.5';                  answer:   (Amy, 0.8)
```

Figure 4: Pig Pen screenshot; displayed program finds users who tend to visit high-pagerank pages.

Exercise

- Given a whole lot of email stored in Hadoop, write a Pig program will yield the list of spammers who have sent 1000 or more pieces of spam email in the last year, and also generate the list of victims who have received more than 1000 items in the last year. Mail from *@nd.edu should never be considered spam.
- The mail is stored in this form:
  - (sender, receiver, contents, date)
- You have a function called IsSpam(contents) which returns true if the contents contain spam. (Expensive function.)

Points for Discussion

- Which operations may be rearranged in the DAG?
- Is Pig Latin different from SQL in a fundamental way? Or could this have simply been done with slight changes to SQL?
- Big data systems often recommend data management practices that are **completely opposite** of those developed by relational database systems. Why?