

**The Web Laboratory**

**Cornell University**

**Pig Latin Evaluation through PageRank Algorithm  
Implementation**

---

**Spring 2009**

**Advisor:**

Professor, William Arms

Aditi Vad ([aav35@cornell.edu](mailto:aav35@cornell.edu))

May 14, 2009



## Table of Contents

I.	Abstract.....	3
II.	Introduction.....	4
III.	Overview of Pig .....	6
	1. Introduction .....	6
	2. Infrastructure Model.....	7
	3. Methods to Run Pig.....	8
	4. Ways to Run Pig .....	9
	5. Debugging Options .....	9
	6. Basic Pig Latin Syntax .....	10
	7. UDFs .....	12
IV.	PageRank in Pig Latin .....	14
	1. PageRank algorithm.....	14
	2. Formulation into Pig Latin .....	15
	3. Challenges Faced .....	17
	4. Evaluation.....	18
	5. Summary .....	20
V.	Acknowledgements .....	21
VI.	References.....	22
VII.	Appendix: Source Code.....	23
	1. Script: pigFormatter.pig .....	23
	2. Embedded: pig_pagerank.java.....	25
	3. UDF: myGenSum.java.....	29



## I. Abstract

This project is a sub-project (referred to as project hereafter) of an open source suite of programs for web graph analysis on the Hadoop cluster. Two more sub-projects were conducted in parallel, namely, an open suite for Calculation of PageRank and an open source GUI for Web Graph cleaning, both on the Hadoop cluster. A separate report was written for each sub-project.

The project 'Pig Latin Evaluation through PageRank Algorithm Implementation' is an analytical type of project. The main goal of this project was to analyze and evaluate the possible use of Pig Latin, a high-level, general data flow language, for the usual pattern of programs written as a part of the Cornell Web Lab project. The report provides an introduction to Pig Latin programming, and analytical information to aid the decision of whether Pig Latin should be harnessed for the above-mentioned type of programs.



## II. Introduction

This project was a part of The Web Laboratory, a joint project of Cornell University and Internet Archive, to provide data and tools for research on Web. Among the projects done this semester, 'Pig Latin Evaluation through PageRank Algorithm Implementation' is a sub-project of the main project 'An Open Source Suite of Programs for Web Graph Analysis on the Hadoop cluster', which consists of the following sub-projects:

- Pig Latin Evaluation through PageRank Algorithm Implementation
- An open source suite for calculating the PageRank on the Hadoop cluster
- An open source GUI for Web Graph cleaning on Hadoop

A separate report was written for each sub-project. This report corresponds to the first sub-project (referred to as project hereafter), namely 'Pig Latin Evaluation through PageRank Algorithm Implementation'.

Huge data sets require cluster computing. But, programming for a cluster computing environment is difficult and time-consuming. Computations on the Hadoop cluster are usually formulated using the Map-Reduce paradigm. However, writing in the Map-Reduce paradigm is not natural, especially to people who are underexposed to Map-Reduce. Researchers and general users of large data results are interested in the final outcome of the operation and not in formulating the program, especially in a not-very-easy-to-follow form of logic like Map-Reduce, that does not come easily.

Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The highest abstraction layer in Pig is a query language interface, namely Pig Latin, using which users can express data analysis tasks as queries, in the style of SQL or Relational Algebra[1]. Pig Latin makes it easy to harness the power of cluster computing for ad-hoc data analysis, especially for users who are interested in primarily the results of the computation.

The main goal of this particular project was to gauge the usability of Pig in context of the usually analysis type of complex programs that are a part of the Web Lab project. These programs generally use very sophisticated algorithms and complex implementation, including recursion, to support the algorithms. The evaluation is done through an example implementation of PageRank algorithm in Pig Latin. The report provides an introduction to Pig Latin programming, some important details regarding implementation and analytical information to aid



the decision of whether Pig Latin should be harnessed for the above-mentioned type of programs. Thus, it analyzes and evaluates the possible use of Pig Latin for the above-mentioned pattern of programs written as a part of the Cornell Web Lab project.

There are many simple PageRank implementations available online. However these examples use simple data and a very preliminary algorithm. These programs can be implemented using in-built UDFs (such as COUNT , SUM , AVG etc.) which are provided by default in Pig Latin. On the other hand, the PageRank algorithm that this project uses[7], is very sophisticated and takes a range of cases into account. Hence implementation of this algorithm is complex and requires specialized treatment. The challenge was adapting the computations to Pig constraints, to generate correct results.

This report provides first, an introduction to Pig, its infrastructure model, some standard syntax and constructs, implementation methods, debugging options and examples of usage. Since the scope of information available is very extensive, I have mostly limited the lists to that which I have used in this project or have actively considered using. Then second, the use of Pig Latin in the particular scenario of PageRank implementation and the resulting analysis is described. The report is concluded with an outline regarding the type of programs for which harnessing Pig Latin would be optimal. The resultant embedded Pig Latin code which has been successfully tested forms part of this report. The results of the same are appended for ready reference.



### III. Overview of Pig

#### 1. Introduction

Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with the infrastructure for evaluating these programs. It is widely used in Yahoo! for Hadoop jobs[2].

Pig's infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs, for which large-scale parallel implementations already exist (e.g., the Hadoop sub-project). Pig's language layer consists of a textual language called **Pig Latin**, which provides the following advantages:

- Ease of programming  
It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.
- Optimization opportunities  
The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.
- Extensibility  
Users can create their own functions to do special-purpose processing.
- Interoperability  
Data may be read/written in a format accepted by other applications such as text editors or graph generators, versus the need for storage in a database if SQL is used for procedural processing.

Thus Pig Latin provides a high-level procedural relational style programming facility for cluster applications. Hence, it has high adoption by programmers who find procedural programming more natural than declarative programming.



## ***2. Infrastructure Model***

The system can, conceptually, be divided into three levels.

- User Interface

This could be the grunt shell, the command line interface or usage through Java APIs. Grunt is an interactive shell that allows users to submit Pig commands. The command line offers a mechanism for batch mode execution via scripts. The Java APIs provide a programmatic mechanism of accessing Pig using embedded Pig Latin.

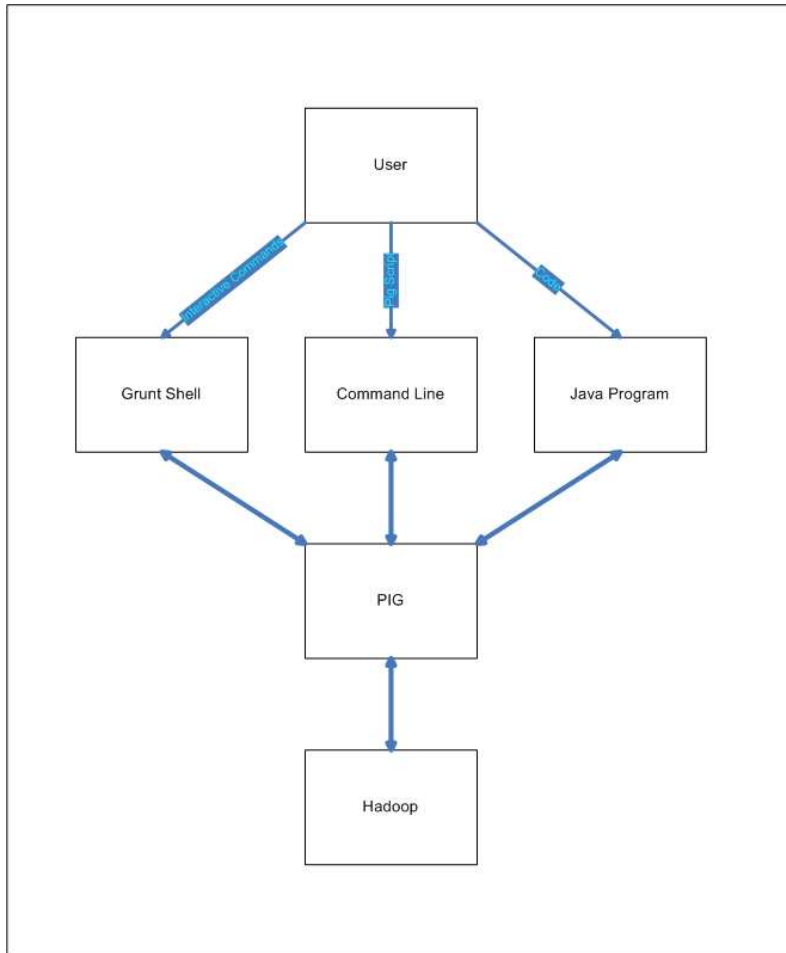
- Pig

Irrespective of the mechanism, the control and data flow through Pig. Pig formulates the Pig Latin commands, received from the user, into Map-Reduce tasks that are run on the back-end execution framework, namely Hadoop.

- Hadoop

Currently, Hadoop is the back-end execution framework for Pig.

A schematic view of the system is illustrated through the following diagram[2]:



### 3. Methods to Run Pig

Pig has two modes of execution[5]:

- **Local Mode**

To run Pig in local mode, access to a single machine is needed.

- **Hadoop (also known as 'mapreduce') Mode**

To run Pig in Hadoop mode, access to a Hadoop cluster and HDFS installation is needed. The required degree of parallelism can be achieved by using the PARALLEL operator.



#### ***4. Ways to Run Pig***

Pig can be run in three ways:

- **Grunt shell**

Pig commands are entered manually using Pig's interactive shell, Grunt

- **Script File**

Pig commands in a script file and the script is run. A normal script file in Pig Latin and an embedded Pig Latin program relate just as jsp and servlets do.

As an example of Pig Latin script, the `pigFormatter.pig` is provided in the Appendix section.

- **Embedded Program**

Pig commands are embedded in a host language, and the program is run. Embedded programs can be used to achieve iterative execution of a certain generative and related batch of statements that may not be supported in Pig Latin.

As an example of embedded Pig Latin, the `pig_pagerank.java` is provided in the Appendix.

#### ***5. Debugging Options***

PigPen is an eclipse plug-in that helps users create pig-scripts, test them using the example generator and then submit them to a Hadoop cluster as well. It provides syntax highlighting, graphical script construction, example result generation and schema descriptions. The scripts are tested on sandbox data, to verify correctness through the data output generated at each step[6].

In addition to PigPen, 'dump', 'describe', 'illustrate' statements can also be used to check the correctness of the output from time-to-time, on real data.



## 6. Basic Pig Latin Syntax

### Simple Data Types

• int	• bytearray
• long	• chararray
• float	
• double	

### Complex Data Types

Type	Description	Example
Tuple	A 'tuple' is an ordered set of field Dereference operator is '.'	(18, 4, 5)
Bag	There are 2 types: - Outer Bag: A relation is also considered a 'bag', specifically, an 'outer bag' - Inner Bag: A 'bag' is a collection of tuples. When a 'bag' is referred, it usually means an 'inner bag' Dereference operator is '.'	{(3, 1), (18, 5)}
Map	A 'map' is a set of key value pairs, where key values within a relation must be unique. Dereference operator is '#'	['name#'Harry', 'ext'#3546]

Fields within a complex data type are referred using the Dereference operators. So applying a dereference operator to a tuple will retrieve a field, while a column will be retrieved for a bag[4].

**Operators**

<ul style="list-style-type: none"> <li><b>Boolean Operators</b> AND OR NOT</li> </ul>	<ul style="list-style-type: none"> <li><b>Comparison Operators</b> ==, !=  &gt;, &lt;  &gt;=, &lt;=</li> </ul>
<ul style="list-style-type: none"> <li><b>Arithmetic Operators</b>  +, -, *, /  % modulo  ?: bincond</li> </ul>	<ul style="list-style-type: none"> <li><b>Inbuilt Eval Functions</b> AVG, MIN, MAX, SUM COUNT CONCAT SIZE TOKENIZE</li> </ul>
<ul style="list-style-type: none"> <li><b>Dereference Operators</b> '.' for tuple, bag '#' for map</li> </ul>	<ul style="list-style-type: none"> <li><b>Null Operators</b> is null is not null</li> </ul>

**Relational Operators**

COGROUP	Groups the data in 2 or more relations based on the common field values. It creates a nested set of output tuples
CROSS	Computes the cross(Cartesian) product of 2 or more relations
DISTINCT	Removes duplicate tuples in a relation
DUMP	Displays the contents of a relation on the standard output. This can be used as a simple debugger
FILTER	Selects tuples or rows from a relation based on some condition
FOREACH ... GENERATE	Generates data transformations based on columns of data
GROUP	Groups together tuples, from a single relation, by key. The relation generated contains the key and grouped tuples from the original relation
JOIN	Joins 2 or more relations based on common field values. It creates a flat set of output tuples
LIMIT	Limits the number of output tuples



LOAD	Loads the data from the file system. The data can be loaded specifying a schema, or even without doing so
ORDER	Sorts a relation based on the conditions specified
SPLIT	Partitions the contents of a relation into 2 or more relations based on the conditions stated
UNION	Computes the union of the contents of 2 or more relations. This operator, by default, does not eliminate duplicate tuples. It also does not ensure, like in a database, that the schemas of the relations being combined matches
DESCRIBE	Returns the schema of the relation
ILLUSTRATE	Displays a step-by-step execution of a sequence of statements
REGISTER	Registers a jar file so that the udf s in the file can be used
STORE	Stores data into the file system in the specified file
TOKENIZE	Splits a string and outputs a bag of words
FLATTEN	This flattens a bag of fields(words or tuples) into individual fields, each part of a new tuple

## 7. UDFs

Pig provides extensive support for user-defined functions (UDFs) as a way to specify custom processing. Functions can be a part of almost every operator in Pig. Usually these functions can be classified as belonging to one of the following classes[7] and then the basic building syntax for that particular class is followed:

- Comparison: for custom comparators used by ORDER operator
- Evaluation: for evaluation functions like aggregates and column transformations
- Filtering: for custom conditions used in FILTER operator
- Grouping: for grouping functions



- Storage: for load/store functions

Pig also includes some built-in functions like `TOKENIZE`, `FLATTEN` and more. However these functions need not be registered or qualified. They can be directly used. On the other hand, to use UDFs created, the user needs to register the jar file that contains the UDFs, so that they can be used in that Pig Latin program. Else, if the user has access to the Pig Latin source code directory, and has the permissions to build Pig, then the UDF can directly be placed in the 'builtin' directory within 'trunk'; Pig is then built from trunk using the 'ant' command. These functions are then naturally available to all users of Pig.



## IV. PageRank in Pig Latin

The PageRank algorithm from the ‘PageRank Calculation using Map Reduce, Fall 2008’ report[7] was adhered to during formulation of the PageRank in Pig Latin. The main challenge in this project was to adapt the previous logic and implementation details to Pig Latin properties- strengths and constraints.

### 1. PageRank algorithm

The PageRank algorithm is based on the following probabilistic model:

A user starts at a random page. He will select one of the hyperlinks on the document to redirect to the corresponding page with a probability  $d$ . Otherwise, he will select any random page and jump to it with a probability of  $(1 - d)$ . PageRank of a document is the probability that the user will arrive at any particular page. If we iterate this process for a large number of times, the values will converge to give the PageRank of the document.

Consider the following example.

Assume a collection of four documents A, B, C & D. Since PageRank is the probability of arriving at a document, the initial probability for each of the document is equal at 0.25. If B & C each have a link only to A, they increase the importance of A by contributing to its PageRank. Thus, PageRank of A will be sum of contribution from all incoming links.

$$PR(A) = PR(B) + PR(C) = 0.50$$

Now, suppose C has another link to D. This will split the contribution of PageRank from C amongst A & D. Thus  $PR(A) = PR(B) + PR(C)/2 = 0.375$ .

Thus, PageRank can be represented in a general way as

$$PR(u) = \sum_{v \in \langle v, u \rangle} \frac{PR(v)}{L(v)}$$

where  $\langle v, u \rangle$  is the set of all links to u.

Also, PageRank assumes that the user may also stop browsing or may jump to a random page with a probability of  $(1 - d)$ . The probability that the user will continue clicking on the links,  $d$ , is called the damping factor. Thus all the contribution from each of the incoming links will only be valid if the user decides to continue. Hence, the total probability of reaching link u, from link v is



$$PR(u) = \frac{1-d}{n} + d \sum_{v \in \langle v, u \rangle} \frac{PR(v)}{L(v)}$$

where  $\langle v, u \rangle$  is the set of all links to  $u$  &  $d$  is the damping factor. In terms of matrices, PageRank values are entries of eigenvector  $\mathbf{p}$ , i.e.  $\mathbf{p}$ , where  $u_1, u_2 \dots u_n$  are the pages in the collection and  $\mathbf{p}$  is the solution of equation

$$\mathbf{p} = \begin{bmatrix} (1-d)/n \\ (1-d)/n \\ : \\ : \\ (1-d)/n \end{bmatrix} + d \begin{bmatrix} l(u_1, u_1) & l(u_1, u_2) & \dots & \dots & l(u_1, u_n) \\ l(u_2, u_1) & l(u_2, u_2) & \dots & \dots & l(u_2, u_n) \\ : & \dots & \dots & \dots & : \\ : & \dots & \dots & \dots & : \\ l(u_n, u_1) & l(u_n, u_2) & \dots & \dots & l(u_n, u_n) \end{bmatrix} \mathbf{p},$$

where  $l(u_i, u_j)$  indicates the presence of a link from page  $u_i$  to  $u_j$ . Its value is 0, if there's no link present; else it is normalized so that for each  $j$ , the sum of  $l(u_i, u_j) = 1$ . The computation starts with an initial PageRank for each document. The PageRank of each document depends on the PageRank of other documents. Thus, there arises the need of iterating through the calculation multiple times so that the PageRank is calculated with the updated values and it in turn makes the calculation of other documents even more accurate.

## 2. Formulation into Pig Latin

The PageRank algorithm was formulated into Pig Latin. Output obtained from the formulation was checked against results obtained from the original PageRank Java implementation for different iterations and verified.

The formulation of PageRank into Pig Latin required usage of UDFs and embedded Pig Latin programming. Also, it was observed that the language is convenient for performing the desired operations on certain data formats, in the situation under consideration. Some observations, limitations and work-arounds are described below:

- **Requirement**

The input be accepted in the format `<#FromURL, FromURL, outdegree,`



List\_of\_#ToURLs\_tab\_separated>, where the number of #ToURLs in the list were variable. For each url in the list, a new tuple would then be generated.

### **Situation**

Since the number of fields(here, URLs) in the list to be accepted was variable, a simple schema could not be specified while accepting the input.

### **Solution**

- One possible solution was to write a LOAD UDF that would accept the list.
- The other solution, which was simple(and used) needed the input format to be modified. This was easily possible using Web Lab's Graph Clean project. The input was modified to provide fields of the list in space separated format. This could be accepted and processed as needed using Pig's basic syntax, thus rendering the UDF unnecessary.

### **Inference**

Pig Latin may not provide easy support for certain input formats. It is necessary to keep a lookout for such situations, where a simple change in format may render complex solutions needless.

- **Requirement**

The PageRank algorithm needed iterative processing. That is, the Map-reduce tasks in PageRank.java[7] were run as many times as the number of iterations requested.

### **Situation**

Executing a chunk of Pig Latin code iteratively in Pig may not be supported, especially if the statements do not fall within the allowed cases of the 'foreach ... generate' construct.

### **Solution**

- For this, embedded Pig Latin was harnessed. This provided the iterative computations that would not be possible within basic Pig Latin constructs.

### **Inference**

The facility of embedding Pig Latin is made available as a way to overcome Pig syntax constraints, using constructs that Java easily provides. This form should be treated as an integral part of Pig and used when necessary.



- **Requirement**

Highly conditional as well as iterative calculation tasks were required in the PageRank algorithm (For example, in Reduce of PageRank.java[7]).

- **Situation**

The tasks considered generated more than one output per grouped input row, with iterative conditions, that eliminated the aspect of ‘filtering’ the input and then processing parts of it. Limiting conditions on the ‘foreach ... generate’ construct of Pig Latin did not allow formulation of such a step.

- **Solution**

- UDFs were formulated in Java and used

- **Inference**

UDFs, though java functions, form an integral part of Pig Latin. UDFs allow Pig Latin to outsource computational complexity, where it is not possible within limitations of constructs.

### ***3. Challenges Faced***

Some of the challenges faced in the project are enumerated below:

- **Input Data Format**

Pig Latin constructs show restrictions due to which certain formats of input data may be difficult to process. Discovering this and finding a suitable format was interesting. In the process, some complex solutions were considered. Though the same were implemented successfully, a solution, namely, a format easy to handle in Pig Latin was designed and accepted being a ‘clean’ and easy-to-maintain’ solution.

- **Incompatible Configurations**

There were some incompatibilities discovered on running the Pig program on the cluster such as the Load syntax, which runs successfully on a single node, but does not run properly on the cluster. The problem is being addressed with the kind help of Lucy. This does not provide assurance that all Pig



Latin standard syntax will be acceptable easily. Certain configuration or version changes may be necessary. Also, work-arounds, like explicit typecasting, may have to be used for cluster compatibility.

- Insufficient Support for Recursive Processing as required for this problem

The PageRank program traverses the web graph starting at any random point. Hence, the algorithm implemented is intrinsically recursive in nature. To obtain a similar effect in Pig Latin, appropriate UDFs were generated and used.

- Pig Latin Natively Not Iterative

Also, the PageRank algorithm used, follows an iterative model which iteratively converges to accurate results. Thus iterations in this algorithm are mandatory, but are not natively/intuatively supported in Pig Latin Hence, embedded Pig Latin programming was used to achieve this feature.

#### ***4. Evaluation***

In Pig, the creation, modification and reuse of analysis logic has significant importance. This section aims to give a conclusive evaluation of usage of Pig Latin, and situations in which the use would be most profitable.

The situations listed under ‘Pro’ describe scenarios where use of Pig would prove optimum, while those under ‘Anti’ outline cases in which using Pig may not be a good idea. The listings are in an intertwined order, as they simultaneously attempt to follow a logical flow of arguments.

Pro:

Adjusting the process and data, and incorporation of UDFs and embedded Pig Latin programming into formulation of Pig programs can help create robust Pig programs manageably. Thus, once the basic flow of UDF classes is understood, formulating UDFs is, usually, a manageable task. Moreover, in most cases related to Web Lab computations, the UDFs required will be a part of the ‘eval’ category, which are relatively straight-forward to understand and write. After building Pig with the



UDFs in the appropriate directory, or including the jar file containing the UDFs in the Pig Latin script, these UDFs can be used directly in Pig Latin programs.

Anti:

However, if Pig Latin is used for adhoc requests, it will be usually done to facilitate easy and quick formulation of request. But when a particular UDF is not already present, and when the situation necessitates it; it may be difficult for users not familiar with Pig internals to formulate it quickly. Thus the problem faced by such a user in this situation will be just as he would face with Map-Reduce. Perhaps, the user may face even more difficulty with Pig, as atleast there are no internal programs and pre-set structures that vary with the UDF class type in Map-Reduce.

Pro:

However, if the necessary UDFs are already provided, users can harness them via Pig Latin, as they are required. The basic Pig Latin constructs are simple and procedural enough for the users to quickly conjure up scripts as and when needed. Thus, in such a case, Pig Latin can provide a solution for routine, as well as need-of-the-hour requirements.

Anti:

Moreover, once UDFs are ready, users can use them as they wish, making it possible to obtain output at intermediate stages too very easily; this is so since making Pig Latin scripts is quite intuitive for supported tasks. For a user to be able to obtain intermediate results in Map-Reduce paradigm, the user needs to have a fair idea of the paradigm to gauge what to modify.

In addition to the above arguments, the following scenarios are also considered:

Pro:

At the risk of stating the very obvious, it can be said that, Pig can be safely and profitably used in scenarios that can be formulated completely on basic Pig Latin constructs and need no UDFs at all.

Anti:



Pig should not be used for the purpose of retrieving or modifying individual records, or small ranges of records, from a very large data set (e.g., lookup a certain person's credit profile).

Anti:

Also, it cannot not be harnessed when there are real-time data serving requirements (e.g., assemble a web page for a customer in under 100ms).

## 5. *Summary*

Thus, it can be assumed that, generally, usage of Pig scripts (written for routine analysis, and those written on the spur of the moment for a certain task) is feasible and easy-to-use (if the program does not need any UDFs at all, or all the UDFs needed are already present). For the same reason, whether to use an embedded Pig script or not depends upon the complexity of the problem, and how clearly the user understands the instructions for embedding Pig Latin into the given basic outline.



## V. Acknowledgements

I would like to express my sincere gratitude to Prof. William Arms for giving me the opportunity to work on this project, and thank him for his guidance, support and suggestions given as and when needed during the course of the project. I would also like to thank Ms Lucia Walle of Cornell University Center for Advanced Computing for her help and support regarding cluster issues. Finally, I would like to thank Tuan Cao for his guidance as required.

The Cornell Web Lab is funded in part by National Science Foundation grants CNS-0403340, SES-0537606, IIS-0634677, IIS-0705774 and IIS 0917666.



## VI. References

1. Yahoo Research- Project Pig: <http://research.yahoo.com/node/90>
2. Wiki- Hadoop Pig: <http://wiki.apache.org/pig/>
3. <http://wiki.apache.org/pig/PigTalksPapers>
4. Pig Latin Reference Manual: <http://hadoop.apache.org/pig/docs/r0.2.0/piglatin.html>
5. Pig Latin Quick Start Reference: <http://hadoop.apache.org/pig/docs/r0.2.0/quickstart.html>
6. PigPen: <http://wiki.apache.org/pig/PigPen>
7. Nayan Busa, Unmesh Jagtap, and Utkarsh Prateek, PageRank Calculation using Map Reduce. December 2008: <http://weblab.infosci.cornell.edu/publications.html>
8. Razen Alharbi, An Open Source GUI for Web Graph Cleaning on Hadoop. May 2009: <http://weblab.infosci.cornell.edu/publications.html>



## VII. Appendix: Source Code

This comprises of three sections:

1. A Pig Latin Script representing the Formatter module (as example of the type)
2. Embedded Pig Latin Program
3. UDF myGenSum (as example of the type)

### 1. Script: *pigFormatter.pig*

```
/*
Input: output of Graph Clean for Pig_PageRank in the format:
'#FromURL<tab>FromURL<tab>Outdegree<tab>List_of_#ToURL_in_space_separated_format'

Output: Input for the pigPagerankCalc in the format:
'#FromURL<tab>FromURL<tab>Type<tab>Qrerp<tab>Erp'

NOTE: All tuples are in tab-separated format unless specified otherwise
@author: Aditi Vad
*/

--Load Input file using specified schema
A = load 'pig_input_to_formatter' using PigStorage() as (okey:long, url:chararray, outdegree:double,
vlist:chararray);

--Take tuples with outdegree>0
B = filter A by outdegree>0.0;

--For each tuple in B create tuple '#Fromurl, fromurl name, outdegree, qrerp=1, erp=0.85/outdegree, #tourl'
C = foreach B generate okey as okey, url as url, FLATTEN(TOKENIZE(vlist)) as type, 1.0 as qrerp,
0.85/outdegree as erp;

--For each tuple in A create tuple as given below
D = foreach A generate okey as okey, url as url, 0 as type, 1.0 as qrerp, 0.0 as erp;

--Combine the tuples in C, D
E = union C, D;

--Store the tuples in relation E into file
store E into 'in_out/input_for_mid_module';

/*
Optional Debugger Statements
```



```
--describe A;  
--describe B;  
--describe C;  
--describe D;  
--describe E;
```

```
--dump A;  
--dump B;  
--dump C;  
--dump D;  
--dump E;  
*/
```



## 2. *Embedded: pig\_pagerank.java*

```
/*
Program generates Pagerank for the given input urls
NOTE: All tuples are in tab-separated format unless specified otherwise

Input: output of Graph Clean for Pig_PageRank in the format:
'#FromURL<tab>FromURL<tab>Outdegree<tab>List_of_#ToURL_in_space_separated_format'
Output: Output file PRpairs containing tuples in the format: '#FromURL<tab>FromURL<tab>Pagerank'

Compile and Run using commands, from the current working directory:
javac -cp /filepath/pig.jar pig_pagerank.java
java -cp /filepath/pig.jar:. pagerank

@author: Aditi Vad
*/

import java.io.IOException;
import org.apache.pig.PigServer;

public class pig_pagerank
{
public static void main(String[] args)
{
try
{
PigServer pigServer = new PigServer("local");
int iterations = 10;
String formatterInput = "pig_input_to_formatter";
String midModuleInput = "pagerank10/iter";
String finalOutput = "pagerank10/PRpairs";

runFormatter(pigServer, formatterInput, midModuleInput);
runPagerank(pigServer, iterations, midModuleInput);
runGetPR(pigServer, iterations, midModuleInput, finalOutput);
} //try end

catch(Exception e)
{
System.out.println("Caught Exception in Main : "+e);
} //catch end

} //main end
```



```
public static void runFormatter(PigServer pigServer, String inputFile, String outputFile) throws IOException
{
//-----formatter module-----

    //Load file using specified schema
    pigServer.registerQuery("A = load " + inputFile + " using PigStorage() as (okey:long, url:chararray,
outdegree:double, vlist:chararray);");

    //Take tuples with outdegree>0
    pigServer.registerQuery("B = filter A by outdegree>0.0;");

    //For each tuple in B create tuple '#Fromurl, fromurl name, outdegree, qrrp=1, erp=0.85/outdegree,
#tourl'
    pigServer.registerQuery("C = foreach B generate okey as okey, url as url, FLATTEN(TOKENIZE(vlist)) as type,
1.0 as qrrp, 0.85/outdegree as erp;");

    //For each tuple in A create tuple as given below
    pigServer.registerQuery("D = foreach A generate okey as okey, url as url, 0 as type, 1.0 as qrrp, 0.0 as
erp;");

    //Combine the tuples in C, D
    pigServer.registerQuery("E = union C, D;");

    //store E to be accessed by next module
    pigServer.store("E", outputFile+"1");

}

public static void runPagerank(PigServer pigServer, int iterations, String inputFile) throws IOException
{
//-----pagerank main module-----

for(int i = 1; i<=iterations; ++i)
{
//=====

        //Load the file that contains the output of previous module
        pigServer.registerQuery("F = load " + inputFile + i + " using PigStorage() as (okey:long, url:chararray,
type:long, qrrp:double, erp:double);");

        //For out>0 :take only tuples given by (type!=(long)0 and type!=(long)-1)
        pigServer.registerQuery("G = filter F by (type!=(long)0 and type!=(long)-1);");
    }
}
```



```
//For each tuple in G create tuples of 2 types
pigServer.registerQuery("H = foreach G generate okey, url, type, qrrp, erp; ");
pigServer.registerQuery("I = foreach G generate type as okey, 'xx' as url, (long)-1 as type, erp as qrrp, 0.0
as erp;");

//For out<=0: take only tuples given by (F-G)
pigServer.registerQuery("J = filter F by (not(type!=(long)0 and type!=(long)-1));");

//For each tuple in J create tuple as below
pigServer.registerQuery("K = foreach J generate okey, url, type, qrrp, 0.0 as erp;");

//take all tuples from H, K, I together and group by key
pigServer.registerQuery("L = union H, K, I;");
pigServer.registerQuery("M = group L by okey;");

//=====

//For each tuple in M create tuple qt using udfs GetQTurl, GetQTqrrp for conditional processing
pigServer.registerQuery("N = foreach M generate group as okey, myGetQTurl(L.url, L.type) as url, (long)0 as
type, myGetQTqrrp(L.type, L.qrrp) as qrrp, 0.0 as erp;");

//For each tuple in a row in M create an intermediate tuple using udf GenSum for iterative conditional
processing
pigServer.registerQuery("P = foreach M generate flatten($1) as (okey:long, url:chararray, type:long,
qrrp:double, erp:double), myGenSum(L.type, L.qrrp, L.erp) as sum;");

//Take only tuples from P that meet condition (type!=(long)0 and type!=(long)-1)
pigServer.registerQuery("Q = filter P by (type!=(long)0 and type!=(long)-1);");

//the tvalues tuples
//For each tuple in Q create output tuple with updated erp value
pigServer.registerQuery("R = foreach Q generate okey, url, type, sum as qrrp, (erp*sum/qrrp) as erp;");

//Take all tuples from N, R together
pigServer.registerQuery("S = union N, R;");

//store tuples from S into file to be accessed by next module or next iteration
pigServer.store("S", inputFile+(i+1));

}

} //runPagerank end
```



```
public static void runGetPR(PigServer pigServer, int lastIndex, String inputFile, String outputFile) throws
IOException
{
//-----pagerank retriever module-----

    ++lastIndex;

    //Load the file that contains the output of previous computational module
    pigServer.registerQuery("T = load '' + inputFile +lastIndex+' using PigStorage() as (okey: long,url:
chararray,type: long,qrerp: double,erp: double);");

    //Take only tuples of T that have type=0, that is, they are nodes
    pigServer.registerQuery("U = filter T by type==(long)0;");

    //For each tuple from U generate tuple '#FromURL, FromURL, pagerank value'
    pigServer.registerQuery("V = foreach U generate okey, url, qrerp;");

    //store tuples of V into outputfile
    pigServer.store("V", outputFile);

} //runGetPR end

} //end
```



### 3. UDF: myGenSum.java

```
/*
Program generates a variable 'sum' for the given input url details
NOTE: UDF of class Eval

Input: 3 columns namely L.type, L.qrerp, L.erp
Output: sum of type Double

@author: Aditi Vad
*/

package org.apache.pig.builtin;

import java.io.IOException;
import java.util.Scanner;
import java.util.Iterator;
import java.lang.Long;

import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

import org.apache.pig.PigException;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DataType;
import org.apache.pig.data.TupleFactory;
import org.apache.pig.impl.logicalLayer.schema.Schema;

public class myGenSum extends EvalFunc<Double>
{
    private static TupleFactory mTupleFactory = TupleFactory.getInstance();

    public Double exec(Tuple input) throws IOException
    {
        //Check for null input
        if (input == null || input.size() == 0)
            return null;
    }
}
```



```
try
  {/try1

  //Take each column in a Databag
  DataBag values_type = (DataBag)input.get(0);
  DataBag values_qr = (DataBag)input.get(1);
  DataBag values_er = (DataBag)input.get(2);

  //If handed an empty bag, return NULL
  //This is in compliance with SQL standard
  if(values_type.size() == 0 || values_qr.size() == 0)
  {
    return null;
  }

  //-----

  //Default values
  Double q = 1.0;
  Double sum = 0.0;

  Iterator<Tuple> it0 = values_type.iterator();
  Iterator<Tuple> it1 = values_qr.iterator();
  Iterator<Tuple> it2 = values_er.iterator();

  //Iterate through each column, going one step at a time for each column: Iterative Processing
  for (; it0.hasNext() && it1.hasNext() && it2.hasNext();)
  {
    Tuple t0 = it0.next();
    Tuple t1 = it1.next();
    Tuple t2 = it2.next();

    try
    {/try2

      //Parse to suitable type
      Long type = Long.parseLong(t0.get(0).toString());
      Double qrep = Double.parseDouble(t1.get(0).toString());
      Double erp = Double.parseDouble(t2.get(0).toString());

      //Conditional processing
      if(type!=0 && type!=-1) //edge
      {
        //nothing, done just to filter
      }
    }
  }
}
```



```
        else if(type==0) //node
            q = qrerp;
        else
            sum = sum + qrerp; //type is -1
    } //end try2

    catch(RuntimeException exp)
    {
        int errCode = 2103;
        String msg = "Problem while calculating sum";
        throw new ExecException(msg, errCode, PigException.BUG, exp);
    }

} //end for

//-----

if (sum==0.0)
    return 0.001;
else
    return sum;

} //end try1

catch(Exception e)
{
    throw WrappedIOException.wrap("Caught exception processing input row ", e);
}

} //end exec

} //end class
```