LogProv: Project Report

Ruoyu Wang,[1, 2⋆]

Abstract: Data analytics is involved with data cleaning, processing pipeline or workflow, model training, and result production. In practice users need to query history, reproduce intermediate or final results, and adjust parameters in runtime for reasonable decision made from repeatedly data experiments. Although there exists provenance tools, it is difficult to efficiently create provenance and query for data and semantics from provenance in the context of big data. Moreover, users need to determine trustworthiness of underlying data and analytics, and debug analytic code in runtime. Thus how to evaluate trustworthiness has been a practical problem. In this project, we propose a solution for efficient provenance and trustworthiness evaluation. We store data and logs separately, and only necessary data are stored to reduce storage. We employ ElasticSearch to deal with logs for queries. Thus, we can easily find data from logs, vice versa. The analytic semantics can be retrieved from logs such that we do not need extra user effort and information. We name our proposed system *LogProv*, which in this project is implemented with Apache Pig in a Hadoop eco-system hosted by a cloud, and then evaluate *LogProv* used for a data analytics project. The experimental results show that our proposed solution is efficient to satisfy reasonable user requirements.

Keywords: Big Data Analytics, Provenance, Elasticsearch, Hadoop, Pig, Pipeline, Trustworthiness

⋆Corresponding author: babyfish92@163.com
[1]Nicta, Canberra. ACT, Australia,
[2]BASICS Lab, School of Software, Shanghai Jiao Tong University

ISSN: 1833-9646-8559

Contents

1 Introduction

Provenance has been studied by database, workflow, scientific computing, and distributed systems communities, but provenance for big data analytics is not well explored yet. As concluded in [1], provenance has been extremely important for verifiability and reproducibility, as well as for debugging and troubleshooting workflows. Although some provenance tools have been developed, those tools are specific to either data or workflow/transformation provenance, and are designed for specific applications.

For big data analytics, engineers have to build much more complicated pipelines/workflows to process big data featured by the so-called Vs [2] than dealing with traditional data sets. In such a context, both workflow and data provenance are inevitable. To create provenance for big data, a provenance system needs to store possibly a huge amount of data in a real-time fashion, while this is challenged by the volume and the velocity of big data applications. The complicated analysis logic for data variety and veracity may lead to technical debt, for example hidden feedback loops, pipeline jungle, and data dependencies [3]. In runtime, data and analytic logic may be frequently changed, especially in data experiments and debugging. As a result, big data provenance should provide provenance information in different granularities, with various uncertainties, and reasonable flexibilities.

To be concrete, we address three gaps between provenance tools on the shelf and big data as follows: First, for fixed or small workflows/pipelines, provenance can be provided via graph database. However, in many latest data analytics projects or applications, it is too optimistic to assume fixed workflows/pipelines. Thus, graph-related techniques are challenged by data analytics in big data era [3, 4], because analytics pipeline may need to change frequently due to the veracity of data and data experiments; Second, the amount of data to be stored can be huge in big data context, and provenance users may not be aware of when and where to store intermediate data before a pipeline is executed. Thus, big data provenance should provide the flexibility of reconfiguration in runtime; Third, although statistics can be retrieved from data provenance, there is little trustworthiness of data and pipelines yet.

In this report, we propose *LogProv*, an efficient provenance and trustworthiness system. *LogProv* provides provenance functionalities by simultaneously logging pipeline events and storing intermediate data. The logs and the data are linked through embedding data locations as keys into logs. Thus, semantics of data analytics is hidden in logs. Users can enable or disable logging events and storing intermediate data in each step of a pipeline. Whenever pipelines and data are changed, we can always retrieve information from logs and use the information to find data, vice versa. Trustworthiness for both data and pipeline processing units are evaluated by our scoring system, for which we created an ELO-like [5] technique for scoring. We employ ElasticSearch to deal with the logs to guarantee near-constant-time query. Although our proposed solution is effective on other pipeline platforms, in this paper *LogProv* is only implemented in Apache Hadoop and Pig.

Big data provenance with *LogProv* is simple and flexible for users, since users do not need to be fully aware of analytics workflows/pipelines, and can choose easily when and where to provenance. However, since event logging and data dumping at each step may result in extra overhead, it should be specified if querying information from logs will take much time and if logging will impact pipelines much. To evaluate *LogProv*, we performed a series of experiments on a Hadoop ecosystem hosted in a cloud to answer the following research questions.

1. How well does *LogProv* generate logs and dump data to meet user requirements?
2. How can a normal pipeline be impacted?
3. Can the trustworthiness distinguish pipeline paths and processing units?
4. How quickly can a query be processed to return answers to users?
5. Is the data storage efficient?

In the experiments, we analysed 380K - 2090K rows of public wifi access data collected in Geelong, Melbourne, Australia. The experimental results well answer the research questions and thus evidence the effectiveness and efficiency of *LogProv*.

Note that, because provenance is demanded by a variety of applications and systems in real world, there have been different concepts and some debates on "correct" provenance. *LogProv* is designed

exactly for the pipelines of big data analytics, e.g.[6], although it can benefit other applications and systems. Generally speaking, only if graph representation of provenance is too complicated or has to be changed too often to deal with query, storage, and management for big data, would *LogProv* be useful and beneficial.

In section 2, we briefly enumerated several contributions we made in this project. Section 3 provides some background knowledge for the software systems and tools involved in *LogProv*. Section 4 describes the entire architecture adopted in *LogProv* in detail. Section 5 teaches how to deploy *LogProv* and put it into practice use. Section 6 gives a complete list of files we wrote and shared on *GitHub*. Section 7 shows all experiments we performed to test correctness and performance of *LogProv*. Section 8 concludes the entire project and proposed some future plans in next stage.

2 Contribution

In project *LogProv*, we have made mainly three contributions in total.

1. ***Practicability***: First contribution of all is that we have proved via several prototype programs that the idea of both data and workflow provenance is practicable, so as trustworthiness evaluation.

2. ***Architecture***: Second and most important contribution is that we have designed a relatively stable and efficient architecture for *LogProv*. Several components may vary according to different requirements yet the framework remains all the same.

3. ***Hadoop Deployment***: This should be an unexpected gains. At the beginning of multi-node Hadoop cluster deployment in practice, we have tried several tutorials from various websites, not aware of how unreliable they are, failing every time. After times and times of attempts, we have found an authentic way to do that. And we include those steps in this report thus everyone else can refer to it.

3 Preliminaries

LogProv is a compound system. Not all components are developed out of nothing. For simplicity, *LogProv* is constructed above several tools already on shelf. Those tools enhance performance and expand the trust base of entire system.

3.1 Elasticsearch

Elasticsearch is a real-time distributed search and analytics engine. It is good at full-text search, structured search and analytics on searched data. It is like a combination of both SQL and No-SQL database, while it is operated via RESTful APIs. Elasticsearch has been widely used in several well-known Internet companies and organizations, such as Wikipedia, Stack Overflow and GitHub.

3.2 Apache Pig & Pig Latin

Pig is an engine for executing data flows in parallel on Hadoop. It automatically designs efficient plans for data processing tasks on Hadoop framework, according to user specification. It can be run both on Hadoop clusters and local machines. It is very useful in reaserch on raw data, interactive processing as well as *Extract Transformation Load*(ETL) data pipelines, where pipelines are the largest use cases.

For example, behind a website, tons of logs will be passed to Pig from web servers, being cleaned and aggregated. Then Pig will output the analyzed data to administrators as a server monitor. Also, Pig can be used offline to build behaviour prediction models.

Pig is an open source project belonged to Apache, which means every one is free to download and modify under the terms of Apache License.

Pig Latin is a script language used to communicate with Pig engine, expressing data flows. It describes a directed acyclic graph(DAG). However, it is not like other traditional programming languages such as C and Java, which describes control flows with branches and loops like ‘if’ and ‘while’. It looks like

some SQL database operating languages, including traditional data operations, such as GROUP, JOIN, FITER, SORT and etc.

Also, it is able to support *User Defined Functions*(UDFs) written in Java, Python, etc. UDFs give Pig Latin the ability to scale up dramatically to where it didn't attempt to. They can be written for purposes of loading, processing as well as storing. Developers usually contribute on UDFs and upload their achievements to Piggy Bank.

3.3 Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Core Hadoop includes:

- Hadoop Common: The common utilities that support the other Hadoop modules.
- Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access to application data.
- YARN: A framework for job scheduling and cluster resource management.
- MapReduce: A YARN-based system for parallel processing of large data sets.

4 Architecture

As a provenance system, *LogProv* is able to monitor pipeline structure and semantics, record processing log, grab final/intermediate result data and prepare all those stuff for later retrieve. *LogProv* consists of a *Pipeline Monitor*(PM), an *ElasticSearch Cluster*(ES) and *Data Service*(DS), establishing on the basis of Pig Latin scripts and Pig UDFs, as shown in Fig. 1. PM opens several RESTful APIs for users and Pig engine to access, providing a fine scalability.

Pipeline Monitor inspects user pipelines, collects running context information, and controls storing of intermediate data. After an analytic pipeline is finished, PM will evaluate both data and processing units on the entire involved pipeline paths according to an "Oracle", which can return a boolean value, indicating whether or not an outcome from an analytic pipeline is correct, or more imprecisely, if possible, return a real value as the quality of the final result. In most real world applications, this is possible since end users can know the correctness of data analysis, although there may be a delay. In some cases, outcomes can be verified automatically. Thus, Oracle represents either end users or machines that can make a correct evaluation on outcomes of analytics pipelines.

All logs will be sent to *ElasticSearch Cluster* for efficient storing and retrieving, while intermediate data will selectively be sent to *Hadoop Distributed File System* (HDFS) for storing. The selection depends on the processing time, the amount of data, and explicit user requests. PM will assign a unique index for each data instance, which will be stored on disk. The data can also be stored in memory for data-oriented queries. This functionality has been realised in ElasticSearch-Hadoop.

In run-time, users can search for pipeline parameters, logs and history data via PM, which provides several RESTful APIs for query. Also, command line interfaces and web interfaces are provided for easier accesses.

The provenance procedure can be described by Figure 2.

where each step is:

1. Start a new pipeline
   PM API: **/_start**
   Data Stream:
   line 1: **String** HDFS_Path
   line 2: **String** Remarks

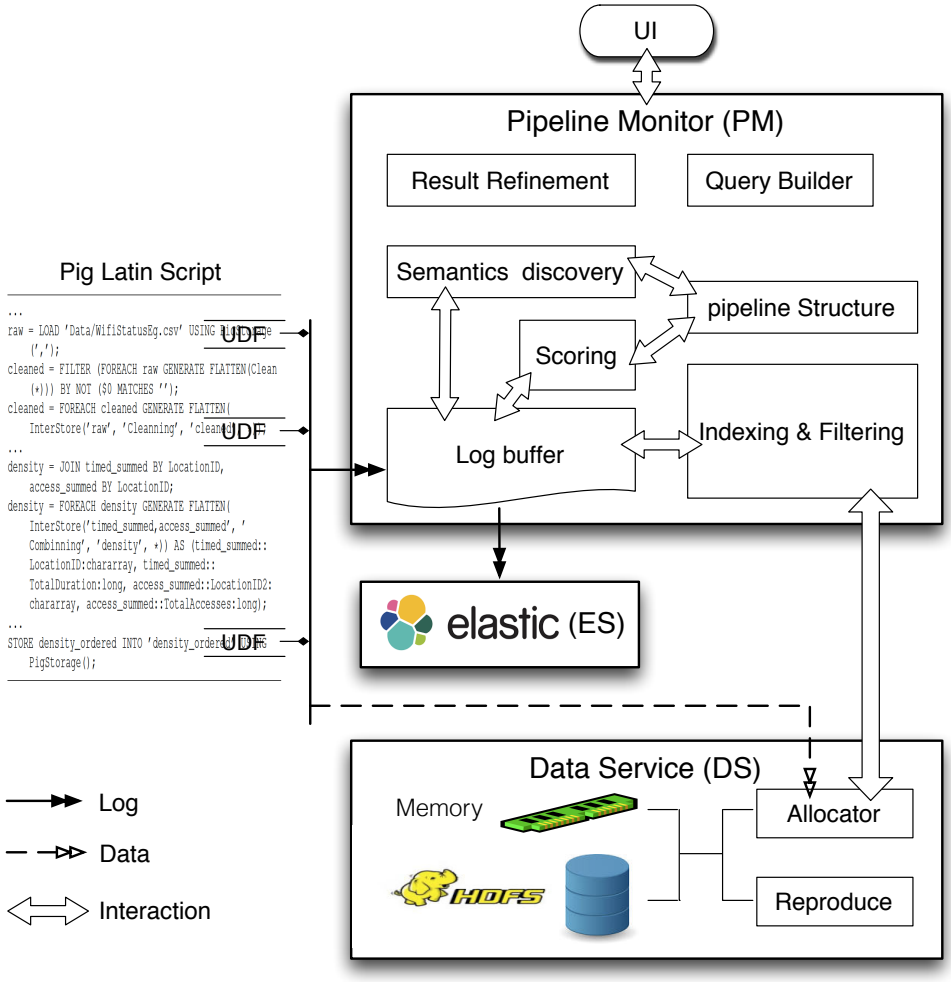

Figure 1: System Overall Architecture

2. Allocate a PID, and record a mapping of pipeline meta info: <PID, (HDFS_Path, Redundant_Info, Start_Time)>and write the record into ES(ES index/type: **logprov/pipelines**)

3. Return PID to Pig
   Data Stream: line 1: **String** PID

4. Require data storage
   PM API: **/_reqDS**
   Data Stream:
   line 1: **String** PID
   line 2: **String** Srcvar
   line 3: **String** Operation
   line 4: **String** Dstvar

5. Record mapping of writing slaves: <PID, <Varname , (Data_Idx, Allocated_Num)>>

6. Return data storage path
   Data Stream: line 1: **String** Data_Path

7. Slaves write data to HDFS

8. Complete a logline and write to ES(ES index/type: **logprov/logs**)

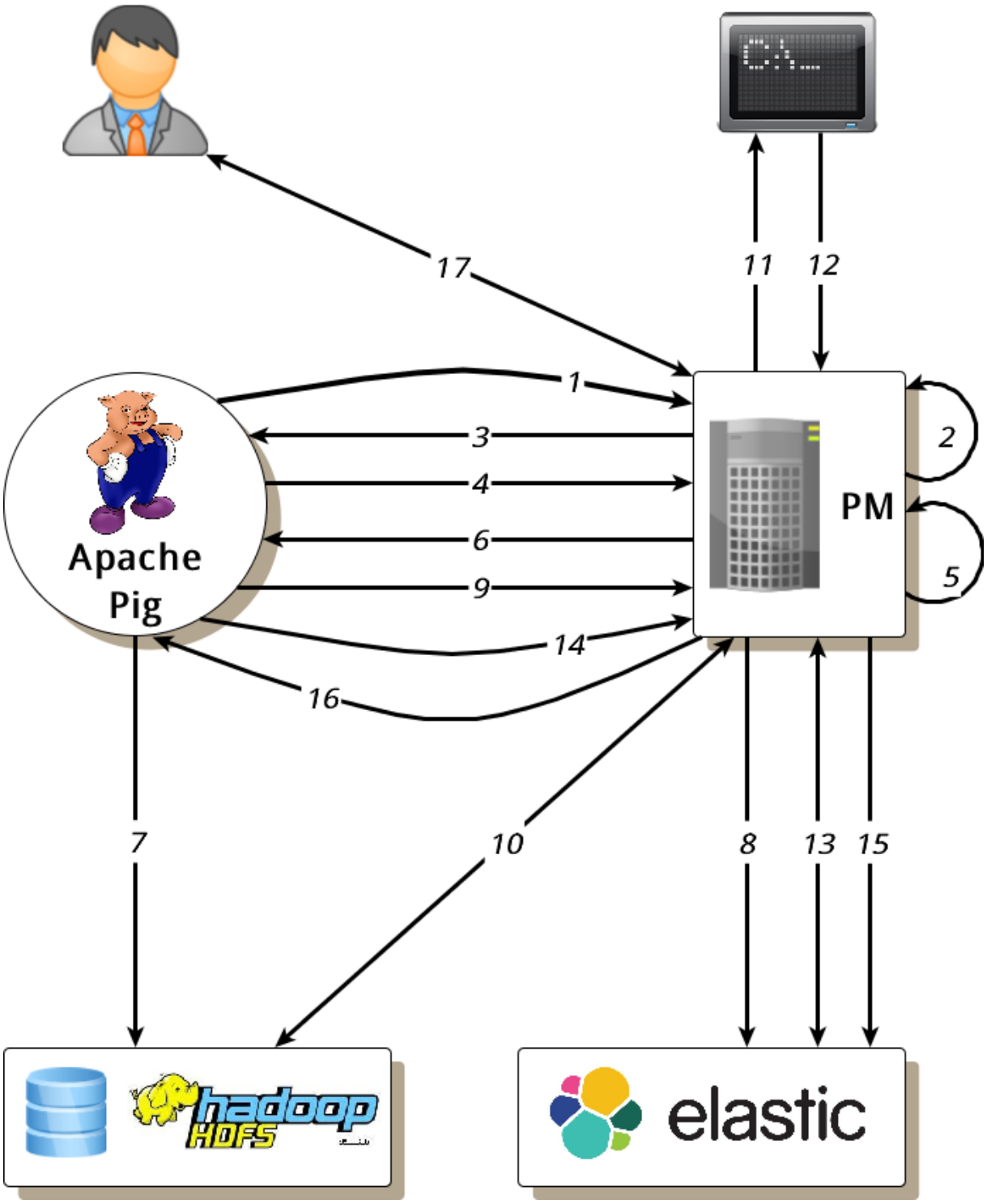

Figure 2: Procedure Flow

9. Inform PM to evaluate a result
   PM API: **/_eval**
   Data Stream:

line 1: **String** PID
line 2: **String** Varname

10. Fetch data from HDFS
11. Send data to Oracle
12. Oracle return a score
    Data Stream: **Float** Score
13. Modify score in loglines in ES
14. Inform PM a pipeline finished
    PM API: **/_terminate**
    Data Stream: line 1: **String** PID
15. Add a finish time into pipeline record in ES
16. Send an ACK to Pig
17. User interacts with PM

4.1 Pipeline Monitoring

This part consists of a pipeline monitoring server named *Pipeline Monitor*, and all components involving Pig Latin scripts, all of whom work together to implement the basic functionality of *LogProv*.

PM receives and deals with logs and intermediate/final result data. It receives half-completed log lines and uses those information to infer pipeline semantics. Normally, each log line points to some result data. After that, those log lines are sent to log database. Once a final result is received, PM will ask an oracle to evaluate the quality of the data and uses the feedback to score relative pipeline paths.

Since PM recieves logs passively, there's no impact on pipeline logic and no active interrupt on pipeline execution.

4.1.1 Pig Latin UDFs

In order to get most benefit with least labour, we leave entire Apache Pig untouched currently. Thus, Pig Latin UDFs have become our only choice to link Pig to *LogProv*, although it may experience a relatively higher overhead for provenance.

Every Pig Latin script processing flow begins with records loading and terminates by result storing (or display). Complete logic is wrapped inside. Thus, we planned to program three utility functions for loading, inter-storing and final-storing. However, pig has integrated its own loader and storer, thus we implemented an inter-storer function only. Syntax and usage for that UDF is shown below:

Syntax:

REGISTER InterStore com.LogProv.pigUDF.InterStore('pm_location', pid);
*dstvar = FOREACH dstvar GENERATE FLATTEN(InterStore('srcvar', 'processor', 'dstvar', *));*

Where, you should always initialize this function via Pig Latin macro definition, providing it with a proper address of pipeline monitor and the PID assigned for this pipeline execution; 'dstvar' is the relation that's going to be stored, we suggest using same relation name when applying this operation for the sake of performance and correctness; 'srcvar' is a string denotes the source variable, if there's more than one, all source variable names are separated by ',';'processor' stands for the user specified name for this procedure. This function shall always be used inside a FLATTEN operator, as shown in the example below:

Example:

REGISTER InterStore com.LogProv.pigUDF.InterStore('http://localhost:58888', 'abcd-efgh');
...
dstvar = JOIN srcvar1 BY $0, srcvar2 BY $0;
dstvar=FOREACH dstvar GENERATE FLATTEN(InterStore('srcvar1,srcvar2','Example','dstvar',*));

...

Users customize their own pipelines via Pig Latin, using *InterStore()* to join the script into LogProv. While the script is executing, *InterStore()* will grab final/intermediate results, generate processing logs and send them to PM. PM will then return a more specific location in HDFS to store that bulk of data. Then, in *InterStore()*, data records will be written into that location directly on each slave machine that is generating results. At mean time, PM will send logs to ES and infer pipeline semantics from them. PM stores pipeline structure locally in memory. After final results have been generated, PM will initiate the scoring mechanism to evaluate corresponding pipeline paths. User may compare and find best path among them.

4.1.2 Semantics Inference

Since each log line records one execution of certain operation, it can be linked with others according to same indices. For instance, if 'dstidx' of operation A is the same as 'srcidx' of operation B, then it means B is one successor of A. Thus, we can mining among all logs to search for that feature and link all components together to recover the semantics.

4.1.3 Scoring System

Once the Oracle has evaluated the final results, the scoring system will be triggered to evaluate the involved pipeline paths. The scoring system applies an Elo-like method to calculate scores of each processing unit (one code fragment between two ProvInterStores) along a path. Elo method was developed for calculating the relative skill levels of players in games, such as chess. In Pig Latin, we consider one pipeline path (from load to store) as one player.

Positive values from the Oracle suggests a win, negative means a lose, and zero implies a draw. PM will pass the end scores backward and accumulate along certain paths. After several iterations of input, different paths will possess different scores. Thus we can compare among those paths and know which one is the best.

As we have known from the above, we do not permanently record graphs representing semantics of analytics, a temporary graph can be accessed from memory, or a graph can be retrieved from logs. Given a directed graph $G = (V, E)$ where V is a set of vertices in the graph and E is a set of edges in the form of (v_1, v_2) which means there's a directed edge from v_1 to v_2, we now make some definitions for finally illustrating Elo-like scoring algorithm. Note that, we only discuss on a single DAG, since most analytics pipeline can be represented by DAG. Scoring is triggered by Oracle's evaluation of analytics results. Oracle's evaluation is a real number J between $J \in [-1, 1]$.

Definition 1: *Single Path*

For two vertices $v_s, v_d \in V$, if there exists a sequence $(v_s, v_0, v_1, ..., v_{n-1}, v_d)$ s.t. $\forall v_i (0 \leq v < n) \in V$, and $\forall v_i (0 < i < n-1), (v_i, v_{i+1}) \in E, (v_s, v_0) \in E, (v_{n-1}, v_d) \in E$, the sequence is called a single path from v_s to v_d, denoted by $v_s \xrightarrow{v_0, v_1, ..., v_{n-1}} v_d$. Thus, each edge $(v_1, v_2) \in E$ can be written as $v_1 \rightarrow v_2$. If there's no single path from v_s to v_d, we write $v_s \not\rightarrow v_d$.

Definition 2: *Simple Path*

For two vertices $v_s, v_d \in V$, the simple path from v_s to v_d of length $n+1$ is a set of all single path of from v_s to v_d of length $n+1$:

$$v_s \rightarrow_{n+1} v_d = \{(v_s, v_{i_0}, ..., v_{i_{n-1}}, v_d) | v_s \xrightarrow{v_{i_0}, ..., v_{i_{n-1}}} v_d\}$$

If there's no simple path from v_s to v_d of length n, then $v_s \rightarrow_n v_d = \varnothing$.

Definition 3: *Full Path*

For two vertices $v_s, v_d \in V$, the full path from v_s to v_d is a set of all single paths from v_s to v_d:

$$v_s \rightarrow^* v_d = \bigcup_{n \geq 0} v_s \rightarrow_n v_d$$

Definition 4: *Complete Path*

For two vertices $v_s, v_d \in V$, paths from v_s to v_d are complete $\iff \forall v' \in V, v' \not\rightarrow v_s, v_d \not\rightarrow v'$, written $v_s \Rightarrow v_d$. If $v_s \Rightarrow v_d$ and $v_s \rightarrow^* v_d \neq \varnothing$, then full path from v_s to v_d is written as $v_s \Rightarrow^* v_d$. Otherwise, $v_s \Rightarrow^* v_d = \varnothing$.

Definition 5: *Source Path*
For a vertex v_d, its source path is a set of all complete paths ended with it:

$$\lceil v_d \rceil = \underset{v_s \in V}{\cup} v_s \Rightarrow^* v_d$$

Definition 6: *Sink Path*
For a vertex v_s, its sink path is a set of all complete paths started with it:

$$\lfloor v_s \rfloor = \underset{v_s \in V}{\cup} v_s \Rightarrow^* v_d$$

Definition 7: *Directed Acyclic Graph(DAG)*
A graph $G = (V, E)$ is DAG $\iff \forall v \in V, v \not\to v$.
Definition 8: *Reverse Graph*
For a directed graph $G = (V, E)$, a reverse graph for G is $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(v_1, v_2) | (v_2, v_1) \in E\}$.

Algorithm 1: Scoring Algorithm

Input: A DAG $G = (V, E)$ where each node has already been assigned with a original score
Input: A node $v \in V$
Output: G
Data: $P \leftarrow$ a complete path

```
Find ⌈v⌉ in G;
if Oracle has given Evaluation J then
    calculate Δs according to J;
    for all P in ⌈v⌉ do
        for all v' in P do
            if v' is a operation node then
                Add Δs to score of v';
            end
        end
    end
end
return (G)
```

4.2 Data Service

One product generated from pipeline is result data. They will be grabbed by *InterStore()* and send to DS to handle. Data controller is currently implemented as a service on PM, together with a powerful distributed file system–HDFS.

When a data alloation request is received, PM will firstly assign a unique index to the data and then return the index as a more specific location to *InterStore()*(who sends the request). Then *InterStore()* will connect to HDFS and write to a single file under that location on behalf of the slave machine it is running on. Since all slaves write individually, the result data may be divided into several pieces under the same path.

As a living system, it is highly inadvisable to put all data on disk, ignoring utility of memory. Meanwhile, some data are just too tiny or too redundant to be worthy of storing. For instance, if an operation is to only add one on certain column of a table, it will cost much more effort to store the exact result data than merely recognize the operation itself. Thus, data controller is able to filter out highly redundant requests to save both time and space. Moreover, it will decide whether it should put the data into memory or disk according to some historical experience to provide a more efficient service.

When a get request is received, it will simply check meta info to confirm whether the target data is located in memory or on disk, or it should be recovered according to some recorded operation and earlier source data. After that, it fetches data from corresponding repository and return to client.

4.3 Logging

Table 1: Log Examples

Srcvar	Srcidx	Operator	Dstvar	Dstidx
timed	-1f0a3f51:1535d975dd0:-7fda	TGrouping	timed_grouped	-1f0a3f51:1535d975dd0:-7fd9
timed_grouped	-1f0a3f51:1535d975dd0:-7fd9	TAccumulation	timed_summed	-1f0a3f51:1535d975dd0:-7fd8
timed_summed	-1f0a3f51:1535d975dd0:-7fd8	TimedRanking	timed_ordered	-1f0a3f51:1535d975dd0:-7fd7
named	-1f0a3f51:1535d975dd0:-7fdb	AGrouping	access_grouped	-1f0a3f51:1535d975dd0:-7fd6
timed	-1f0a3f51:1535d975dd0:-7ffe	TGrouping	timed_grouped	-1f0a3f51:1535d975dd0:-7ffd
named	-1f0a3f51:1535d975dd0:-7ffb	AGrouping	access_grouped	-1f0a3f51:1535d975dd0:-7ffa
density	-1f0a3f51:1535d975dd0:-7ff7	DScoring	density_scored	-1f0a3f51:1535d975dd0:-7ff6
density_ordered	-1f0a3f51:1535d975dd0:-7ff5	density_ordered_Storer	(NULL)	(NULL)
timed_summed, access_summed	-1f0a3f51:1535d975dd0:-7fd8, -1f0a3f51:1535d975dd0:-7fd5	Combinning	density	-1f0a3f51:1535d975dd0:-7fdf
density	-1f0a3f51:1535d975dd0:-7fdf	DScoring	density_scored	-1f0a3f51:1535d975dd0:-7fde

Logs are generated by Pig Latin UDFs with predefined format and terminology, i.e. a predefined protocol. An ideal logging should be adaptive to arbitrary keyword dictionary, other than to make specific definition case by case. However, it is difficult to perfectly create a universal protocol for logging. Thus, we predefine the format and the terminology, which are extendable and revisable. For this sake, we show log examples in Table 1, which is from a real analytics pipeline in our experiments. 'Srcvar' indicates the names used in a certain operation as sources(input) in Pig Latin scripts; 'Srcidx' stores the unique ID assigned with respective input variable; 'Operator', as the terminology speaks, is a name string to denote one specific operation; 'Dstvar' indicates the names used in a certain operation as destination (output); 'Dstidx' stores the unique ID assigned with respective output variable. As we can see, in this log structure, not only the pipeline semantics but also the execution times of operations can be recorded. The execution times can be used for pipeline diagnosis. Since for each operation, the predecessor operation and the successor operation are all recorded including the stored data and the IDs, it is simple to trace the logs for pipeline semantics.

Users can choose where to do logging in Pig Latin scripts and what to record into logs. Although in this paper we have not used ES-Hadoop to collect data inside map-reduce steps, it is possible to refine provenance grain to that level. This grants users as much flexibility as possible. We compare an original script and an enhanced script below.

Both of the two scripts use the same loader and storer. The only difference between them is that 'RankingHDFS.pig' adopts 'InterStore( )' after every single operation (except for load and store). 'RankingHDFS.pig' passes to each 'InterStore( )' several parameters where the first three are 'Srcvar', 'Operator' and 'Destvar' respectively, and the rest are all ordered data columns to be sotred. 'Srcidx' and 'Dstidx' are not provided since 'InterStore( )' handles them automatically. 'InterStore( )' receives first three parameters to build a partially completed log line and send it to PM for later process. Then InterStore stringfies the rest parameters and records them onto HDFS via DS. Since there is one more grabbing operation after every inner operation, 'RankingHDFS.pig' nearly doubles pipeline path in comparison with 'RankingHD.pig'. Despite the changes, those original lines inherited from 'RankingHD.pig' are left untouched. After all of those have been finished, 'InterStore( )' will return the inspected data such that original pipeline will not be affected.

RankingHD.pig

```
...
raw = LOAD 'Data/WifiStatusEg.csv' USING PigStorage(',');
cleaned = FILTER (FOREACH raw GENERATE FLATTEN(Clean(*))) BY NOT ($0 MATCHES '');
...
density = JOIN timed_summed BY LocationID, access_summed BY LocationID;
...
STORE density_ordered INTO 'density_ordered' USING PigStorage();
```

RankingHDFS.pig

```
...
```

```
raw = LOAD 'Data/WifiStatusEg.csv' USING PigStorage(',');
cleaned = FILTER (FOREACH raw GENERATE FLATTEN(Clean(*))) BY NOT ($0 MATCHES '');
cleaned = FOREACH cleaned GENERATE FLATTEN(InterStore('raw', 'Cleanning', 'cleaned
    ', *));
...
density = JOIN timed_summed BY LocationID, access_summed BY LocationID;
density = FOREACH density GENERATE FLATTEN(InterStore('timed_summed,access_summed
    ', 'Combinning', 'density', *)) AS (timed_summed::LocationID:chararray,
    timed_summed::TotalDuration:long, access_summed::LocationID2:chararray,
    access_summed::TotalAccesses:long);
...
STORE density_ordered INTO 'density_ordered' USING PigStorage();
```

4.4 Query

Query is the most important part a user will use to interact with LogProv. It is also implemented as an API in PM: /_search.

It provides query for intermediate/final result data, executing metadata(logs) and pipeline status. To illustrate query condition, it provides users with a simple, SQL-like query language. For example, "$term == value$", "$(term1 < value1)\&\&!(term2 == value2)$" are all valid expressions in the language. QS provides both a webpage and RESTful APIs to users.

5 Deploy & Operate

5.1 Deploy

To deploy this prototype of *LogProv*, one should follow those steps below:

5.1.1 Install Java, Hadoop and Pig

See appendix A and B.

5.1.2 Initiate Hadoop and Elasticsearch

Start a multi-node Hadoop cluster. Then, download and deploy Elasticsearch Server on master machine of the cluster. And establish an index named 'provenance', a type named 'log'. Following commands will do the job:

```
1 $ [path to where your elasticsearch is installed]/bin/elasticsearch &
```

5.1.3 Initiate Answer Server

Start the oracle to evaluate quality of final result data, using following command:

```
1 $ java -jar AnswerServer.jar
```

Or one can write his own oracle.

5.1.4 Initiate Pipeline Monitor

This is the most important step. To start pipeline server, Elasticsearch Server must already be started. Type following command into shell.

```
1 $ java -jar LogProv.jar
```

Now *LogProv* is running on master machine.

5.2 Operate

5.2.1 Run Pig Latin Scripts

To use *InterStore()* in Pig Latin scripts, **LogProvUDF.jar** has to be registered at the beginning. Then users can add inspection points with *InterStore()* wherever they want. An example script is shown in appendix C.

To run pig, type the following command:

```
1    $ pig [name of script]
```

5.2.2 Query

PM privdes an API '/_search' for users. Syntax of this API is as follows:

```
1  $ curl http://[PM address]/_search -d '[command]
2  >[query string]'
```

Where there are four types of 'command':

1. 'pipeline': query for pipeline execution information
2. 'data': query for result data
3. 'operation': query for operation meta information(i.e. logs)
4. 'semantics': query for pipeline semantics

6 Files & Code

As a backup, all files and codes developed in this project had already been uploaded to *GitHub*[1]. The repository contains project codes and test data. All files are shown in lists below. Bold names stands for directories.

6.1 /

- **Demo**

  This directory contains data files, Pig Latin scripts and *.jar* packages to run demo programs and experiments.

- **src/main/java/com/logprov**

  This directory contains all source code for pipeline monitor.

- .gitignore

  Git-ignore file.

- README.md

  Self description of project *LogProv*. It contains some meta information.

- pom.xml

  Maven framework specification.

[1] https://github.com/TramsWang/LogProv

6.2 /Demo

- **Data**
  This directory contains several test data file and some auxiliary shell scripts.
- **Mapping**
  This directory contains JSON objects used to initiate index and type mapping in Elastic.
- DemoUDF.jar
  This Java package file will be used in Pig Latin scripts to perform some UDFs for test purposes.
- RankingHD.pig
  Pig Latin script that performs the experiment pipeline without any provenance support.
- RankingHDFS.pig
  Pig Latin script that performs the experiment pipeline with provenance support.
- testHDFS.sh
  Shell script to perform test automatically.

6.3 /Demo/Data

- GeelongWifiStatsSchema.txt
  Text file that records the schema of source data set.
- WifiRankingAns.csv
  A relatively "correct" answer used by oracle.
- WifiStatusEg.csv
  Some sample lines of total data set. Used for debugging.
- WifiStatusTotal.csv
  Total source data set.
- integrate.sh
  Shell script used for create larger data set.
- split.sh
  Shell script used for create smaller data set.

6.4 /Demo/Mapping

- log.json
  JSON object used in Elastic to establish mapping for ES index/type: "logprov/logs".
- pipelines.json
  JSON object used in Elastic to establish mapping for ES index/type: "logprov/pipelines".

6.5 /src/main/java/com/logprov

- pigUDF/InterStore.java

  Source code of Pig Latin UDF 'InterStore()'.

- **pipeline**

  This directory contains codes used in pipeline server.

- Config.java

  LogProv configuration specific reader and configuration pool. It reads, stores and manages all configuration entries used in *LogProv*.

- LogProv.java

  LogProv initializer.

6.6 /src/main/java/com/logprov/pipeline

- ESSlave.java

  Source code file for an auxiliary class 'ESSlave', which helps pipeline server connect and send requests to ESS.

- LogLine.java

  Log class.

- PipelineInfo.java

  Pipeline information class.

- PipelineMonitor.java

  Main source code of pipeline server.

7 Test & Experiment

In this section, a series of experiments and their results are presented. The purpose is to answer some research questions towards two basic functionalities, provenance and trustworthiness evaluation. *LogProv* can be used for many purposes, which should be decided by users themselves, for example, which data should be stored and what the query results will be used for. In this project, we do not test all of them, since almost all of them are based on the two basic functionalities that we tested. Note that, we do not pursue sophisticated analytics of the data, while we are interested in *LogProv* itself.

7.1 Setting and Data

To test functionalities and performance, we have implemented *LogProv* on Nectar Cloud[7], on which a basic Hadoop eco-system is running and ElasticSearch is also deployed. The summary of the experiment environment is listed below.

Table 2: Experiment Environment

Instances	1 master and 2 - 32 slaves
OS	Ubuntu 15.04 (kernel 3.19.0-28-generic)
VCPUs	1 each
Memory	4GB each
Disk	30GB on master machine, 40GB on every salve machines
Hadoop	2.7.2
Pig	0.15.0 (r1682971)
Elasticsearch	1.7.3

The data set is from Australia Government[2] and consists of 380K lines of public Wifi utility data collected from Geelong, Melbourne, Australia, including public wifi location and user connection details. A sample is below.

```
1  1893824
2  718
3  1
4  23:27.7
5  56:36.5
6  38
7  1893824
8  Instagram 3.1.1 (iPod touch; iPhone OS 6.0; en_AU) AppleWebKit/420+
9  http://www.weatherchannel.com.au/your-weather-widget.aspx?style=
10 City Moorabool3 Nth
11 Johnstone Park
12 10.224.0.14
13 InfoNet Moorabool Nth / Bus
14 3003 - Geelong
15 VIC
16 3220
17 144.360431
18 -38.147744
```

By the experiments, we are going to answer the research questions asked below:

1. How well does *LogProv* can generate logs and dump data to meet user requirements?
2. How can a normal pipeline be impacted?
3. Can the trustworthiness evaluation distinguish pipeline paths and processing units?
4. How quickly can a query be processed to return answers to users?
5. Is the data storage efficient?

To quantify the answers to the research questions, we use the following metrics:

1. Measure the time gaps between a processing unit being fired and a log line and corresponding data, if existing, being ready for query;
2. Compare the time used by data analytics pipeline without provenance and with provenance;
3. Compare the trustworthiness scores after different numbers of runs have been finished;
4. Measure the time users need to get the query answered;
5. Compare the amount of data generated in pipeline life and the amount we store for provenance.

7.2 Pipeline and Methodology

We analyse and rank the "temperature" of each WIFI hotspot. During the analysis, we adopt three versions of ranking approaches, i.e. three paths. Therefore, each input will generate three output candidates. Thus, scoring system is used to evaluate the different pipeline paths. Performance overhead due to provenance is measured by comparing to a version of Pig Latin script that contains no provenance operations.

For the baseline experiment, we extract first 100K records (31.4 MB) in the source data and divid the records evenly into 10 samples, each containing 10K lines (approximately 3.1 MB each) of records. In this way, we can run a pipeline 10 times. We analyse entire 100K records to get a "correct answer" of

[2]http://data.gov.au

wifi temperature ranking, and we program an Oracle to evaluate the results from samples according to that "correct answer". Then, we initiat *LogProv* and run the pipeline with all samples one by one. Each of the 10 results of the samples is provided to the Oracle for evaluation against the "correct answer." We also use the entire 100K records to run the pipeline with and without provenance respectively, and then compared their execution times upon different numbers of slaves.

When we need larger data sets, we scale up from the baseline setting by factors, and the maximum size of data set is 822.46MB with each sample of approximately 82MB. We do not employ even larger data sets for three reasons: First, the time of experiments is not too long and the stored data is not too large, so that we can reduce our experiment cost; Second, since time delay caused by generating logs is not impacted by data size, and is nearly constant time, the overhead introduced by *LogProv* for processing smaller data sets is more obvious than for processing much big data sets. Third, the time delay caused by storing intermediate data indeed depends on the amount of data: If storing data can block the progress of a pipeline, any provenance systems suffer from the longer delay caused by larger amount of data, and this fact does not worth more experiments on larger date sets to observe; If the delay can be avoided by making provenance non-blocking, it is not necessary to take this overhead into account.

A simplified pipeline diagram is shown in Fig. 3. At the beginning, all input data should be cleaned and named since there may be several 'broken' lines in the data that cannot be parsed correctly into given scheme. Next, they will be passed to three different branches to perform the ranking work flow, where each of them produces its own results. The left most path evaluates according to connected time accumulated among all users. The right most path evaluates according to total access amount. And the path in the middle combines results from the former two paths to calculate an access density for each WIFI hot spot. Since the first two paths use only part of the information in raw data, while the last one processes against a relatively more comprehensive source, the 'density' path will get a better result than the other two.

7.3 Results

The experimental results are shown and used to answer the research questions in the following.

1) How well does *LogProv* generate logs and dump data to meet user requirements?

To answer this question, we tested the time span from the moment logs are generated to the time they can be finally reached in ESC. It turns out to be a constant value: approximately 900ms(Table 3).

Table 3: Logging Performance

Component	A	B	C	D	E	F
Time(ms)	870.31	478.78	939.23	316.52	943.20	873.72

2) How can a normal pipeline be impacted?

As described above, a pipeline with provenance is usually longer than without provenance, since more steps are needed. Thus, *LogProv* indeed introduce extra overhead. The experiments in Table 4 record all data after every operation and block the pipeline during data transfer. The results show a low performance degradation around 10% $\sim$ 20% as in Table 4. We can know from this result that the overhead is nearly constant and not impacted by the number of salve machines.

Table 4: Overhead With Different Number of Slaves

Slaves	2	4	8	16	32
Without LogProv (ms)	367821	298552	239897	202730	202749
With LogProv (ms)	415923	350787	272703	243125	231920
Overhead(%)	13.08	17.50	13.68	19.93	14.08

In Table 5, we test larger data sets. Since we have known that the overhead is not impacted by the number of slaves, we use two slaves with higher performance here. The overhead with blocking for larger data sets increases, since more data need more time for storing and transferring. The second row in Table 5 indicates the total execution times of Pig engine without LogProv (RankingHD.pig). The third row records the total execution times with LogProv (RankingHDFS.pig). The "Intermediate Data" row

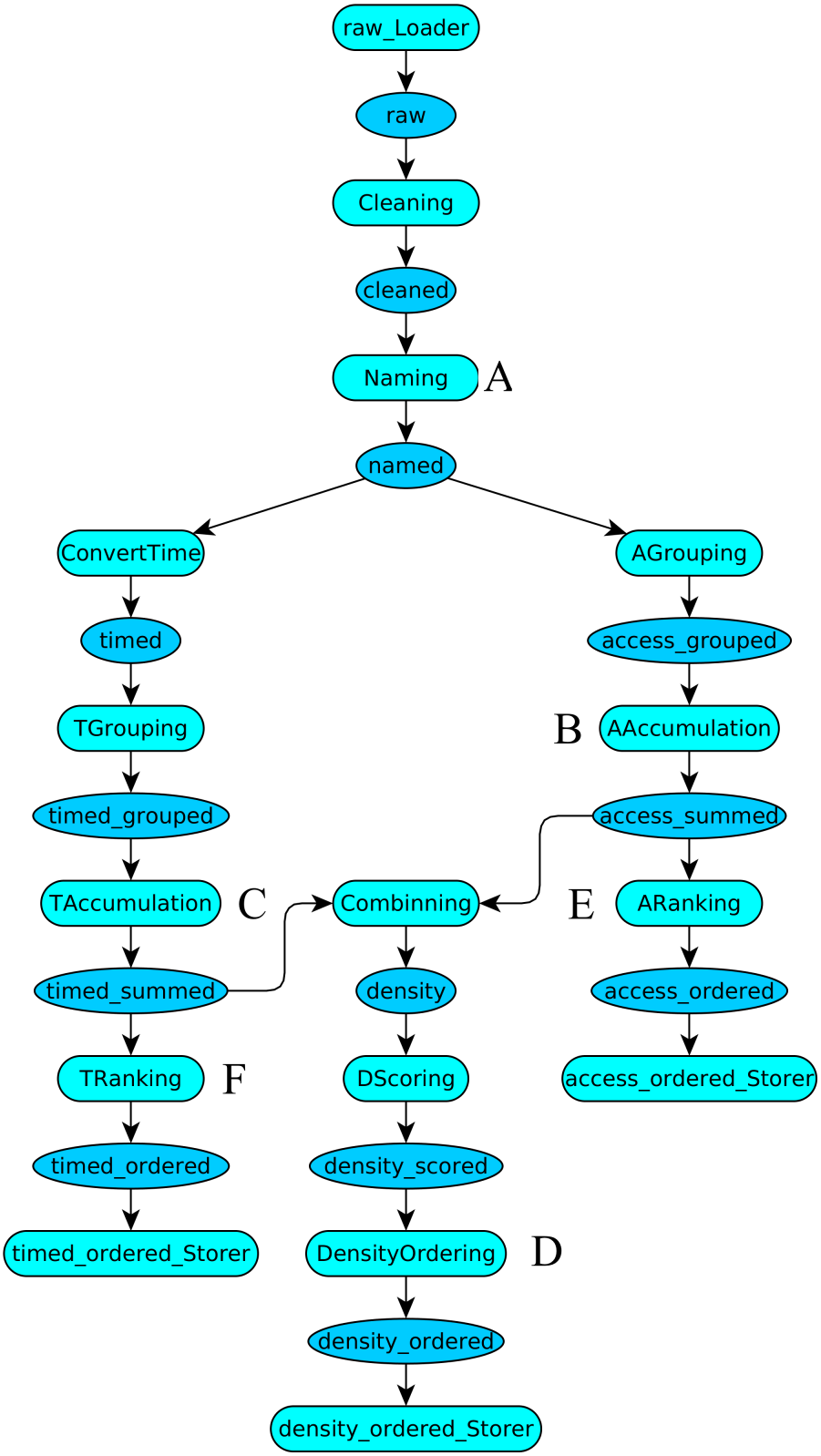

Figure 3: An analytics pipeline. The ovals are intermediate data, while the others are operations.

records the amounts of data to be written into HDFS, and the writing times are taken down below. It is not difficult to find that the overhead increment is mainly due to the increasing amount of data being written into. By excluding the time of writing into HDFS, i.e. making *LogProv* non-blocking, in the last row, it can be concluded that the overhead caused by non-blocking *LogProv* is nearly a constant no more than 10%, although much more memory is required. The non-blocking is possible for our experiments here, because the amount of data is not too huge, compared to the hardware capacity shown in Table 2.

The above two groups of experiments show possibly the highest overhead and the lowest overhead respectively, given the scales of system and data. In practice, duplicating only a part of data, selecting data for storage, supplying more physical memory, increasing HDFS I/O speed and so on can all be used to reduce the extra overhead introduced by *LogProv* to the lowest, i.e. the non-blocking case.

Table 5: Overhead With Different Size of Data Set

Data Size (MB)	164.49	328.98	493.48	657.97	822.46
Without LogProv (ms)	149693	150601	151476	156493	159853
With LogProv (ms)	162005	181706	200067	223578	230065
Intermediate Data (MB)	125.07	239.64	359.25	483.12	581.47
Writing to HDFS(ms)	13851	24943	31926	49104	65598
Overhead Blocking (%)	8.22	20.65	31.84	42.87	43.92
Overhead Non-blocking(%)	-0.94	3.51	8.93	8.75	2.05

3) Can the trustworthiness evaluation distinguish pipeline paths and processing units?

In Fig. 3, six key operations are marked. The trustworthiness scores of these operations are recorded in Fig. 4. We can easily see the difference after a number of data have been processed. The tendency shown in this figure testifies that the gaps between different operations will become larger and larger as more data are processed.

If one simply uses the scores to choose a reasonable analytics pipeline, the path should start from **A** with the highest score. Then **B** and **C** do not differ from each other too much, but both of them should be selected if **D** is chosen. After selecting **D** and filling the missing operations, an optimal pipeline can be found.

4) How quickly can a query be processed to return answers to users?

We list some query examples in Table 6. The response time of simple queries is shorter than that of complicated ones, but the times are all in the scale of milliseconds for the current pipeline. For ElasticSearch, the response time can be impacted by what to query, the amount of logs, and memory size. In most cases, if all logs can be fit into memory, the response time is as short as seconds [8].

Table 6: Query Examples

Query String	Time(ms)
(empty)	2
srcvar == raw	3
dstvar == density || operator == Naming	4
srcvar == named && dstvar == timed	6
srcvar == named && (dstvar == timed || operator == Combinning)	8

5) Is the data storage efficient?

We do not expand our test to compression and data selection in this paper, since those techniques can be add-ons to *LogProv*. We show some possible storage policies and corresponding amount of raw data stored in Table 7. Users can set different policies to *LogProv*. As shown in the table, for our pipeline, if all intermediate data after each operations should be stored, the actual amount of data is not too huge; If only data before branching should be stored, the amount of data is much smaller than all data; If only data at the beginning and the end should be stored, the efficiency seems much worse than storing before branching, provided that data reproducing is allowed.

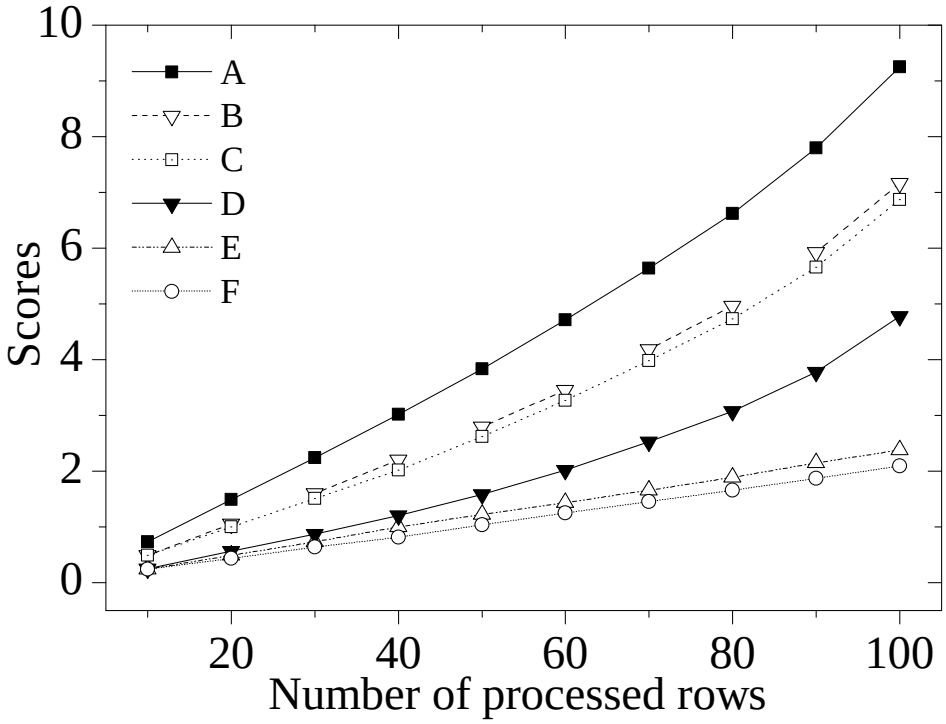

Figure 4: Scoring Test

8 Conclusion & Future Work

In this report, we designed and implemented a provenance system called *LogProv* which can be used to help data scientists and analysts for big data analytics, including history query, intermediate/final results storing and reproduction, data experiments, analysis logic redesigning and trustworthiness evaluating. *LogProv* does not adopt graphs and graph databases for data and workflow provenance, and thus it avoids the problems caused by frequently changed pipelines and data in big data context, while *LogProv* grants users the flexibility of provenance by logging at where it is necessary. Then *LogProv* takes the advantages of ElasticSearch to quickly query logs. A concrete implementation has been done with Apache Pig and Hadoop, and we created Pig UDFs to connect analytics pipelines governed by Pig with *LogProv*. We performed several experiments to test its functionality and performance. It turns out that *LogProv* is able to accomplish what it was expected and possesses a reasonable performance overhead.

In the current implementation of *LogProv*, users still need to change their original Pig Latin scripts. We are going to modify Pig straightly to integrate into Pig engine what UDFs are programmed to do. It will not only make new Pig Latin scripts more intuitive and illustrative, but also saves many redundant operations in Hadoop cluster to enhance performance. It is also possible to generate add-ons or plug-ins to other pipeline management software for provenance.

A Deploy Multi-node Hadoop(2.7.2) Cluster On Ubuntu(15.10)

There have already been kinds of different tutorials on the Internet about Hadoop cluster deployment, including single-node cluster, pseudo-multi-node cluster and real multi-node cluster. It's easy to follow those instructions to build a well performed cluster of the first two types. However, I struggled a lot when moving to deploy a real-functioning multi-node Hadoop cluster. I've tried lots of tutorials and instructions. But none of them really works. Luckily, I never give up trying and experiment. And finally discovered an effective approach for it. Thus, I wrote this as a guide and hint for remembrance as well as giving a hand to those who also struggles in the same problem. I cannot guarantee this is the simplest

Table 7: Data Summary (KB)

Data	All	Before branch	Head & Tail
cleaned	32622.4	-	32622.4
named	2227.4	2227.4	-
access_grouped	2427.7	-	-
access_summed	0.5	0.5	-
access_ordered	0.5	-	0.5
timed	657.4	-	-
timed_grouped	857.7	-	-
timed_summed	0.6	0.6	-
timed_ordered	0.6	-	0.6
density	1.1	-	-
density_scored	1.3	-	-
density_ordered	1.3	-	1.3
total	38798.5	2228.5	32624.5

guide, but I'm sure it works just fine.

In the following instructions, I will illustrate an example of $(n+1)$-node cluster: *ONE* as master and other n machines as slaves. For reasons of resources saving, I used only one machine as dedicated master, which performs *Name Node*, *Secondary Name Node*, *Resource Manager* and *Job History Server* altogether. However, we should know that they can be deployed separately on different machines. And according to comments on Apache Hadoop homepage, they suggest we to do so.

Ingredients

- **Bare system of Ubuntu 15.10 on all $n+1$ machines.** I use bare system in order to illustrate every necessary step. If there's already some tools and programs installed on the machine, one can only modify those configurations if necessary. Moreover, suppose the IP addresses are known as below:

$$master : ip_m$$
$$slave_1, slave_2, ..., slave_n : ip_{s1}, ip_{s2}, ..., ip_{sn}$$

- **Reliable network.** Make sure that you can log on your machines via *ssh* or *remote desktop*. And it is very important to make sure that all your machines can communicate with each other without any obstacles. [3] Otherwise, you may discover not a single slave will follow master's commands. Also, open some important ports on master machine to public to view cluster status on web browsers.

- **Skillful hands and sharp eyes.** When writing configuration files, human hands are highly vulnerable of mistakes. Moreover, Hadoop doesn't provide a strict configuration check of initialization, which requires your eyes can discover mistakes at first time it happends.[4]

Steps

By default, every instruction below should be done on ALL machines. If there's something special, there would be a hint aside sub-section title. Attention again, these steps are for Hadoop 2.7.2 and Ubuntu 15.10. Some details may vary with different Hadoop/Ubuntu version and Linux distributions.

[3] When I performed experiments on *NeCTAR Cloud* virtual machines. I had met up with a problem of network security. Their 'default' security group only opens ports between machines that apply that rule. I have to modify it to open all ports to all IPs. The way to do that is to adopt new 'Ingress' rules for both IPv4 and IPv6 that open all ports to any IP on all protocols: *TCP*, *UDP* and *ICMP*, instead of old ones.

[4] It once took me an entire day to debug, only discovering one spelling mistake.

A.1 Install Java

Java is the base of Hadoop. Type into terminal the following command to install Java:

```
$ sudo apt-add-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
#verify java installation:
$ java -version
```

A.2 Download Hadoop 2.7.2

Download package file for Hadoop and unpack it.

```
$ wget [URL for '.tar.gz' package]
$ tar -xzf [local path for Hadoop package]
```

A.3 Modify Hosts

It would be much clearer and illustrative to use names instead of IPs. So we modify **/etc/hosts** file to add all IP mappings:

/etc/hosts

```
[ip_m]  master
[ip_s1]  slave1
...
[ip_sn]  slaven
```

You should know that both master and slaves are supposed to know all IP mappings, since master and slaves ought to know each other to communicate on tasks and resources information and slaves should know each other to share HDFS file shards.

Next file to modify is **/etc/hostname**. It would be better to change a different hostname, otherwise, you will have to comment all content involved with original hostname for IPv6 in **/etc/hosts** every time after machine reboot. As for hostname, every machine should have its own name, such as master, slave1, slave2 and so on. New hostname should be the same as what you choose in **/etc/hosts**. When this is finished, reboot machines to make sure it actually takes effect.[5]

A.4 Generate Passphrase-less ssh Key Pair[Master Only]

It is very important to make all machines communicate without any obstacle. First, the network should be open and reliable. Second, there will be better if no password is required to log on to virtual machine. Third, we need a passphrase-less ssh connection so that all machines can automatically start conversations.

```
$ ssh-keygen -t rsa -P ""
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
#Change file permissions to avoid ssh permission error
$ chmod 600 ~/.ssh/authorized_keys
#Copy public key to all slaves
$ ssh-copy-id -i ~/.ssh/id_rsa.pub slave1
...
$ ssh-copy-id -i ~/.ssh/id_rsa.pub slaven
```

When that is finished. Don't forget to check whether those ssh connections can be actually established.

[5] To inspect hostname, observe the name in terminal prompt string which is following '@' notation.

```
1  $ ssh master
2  $ ssh slave1
3  ...
4  $ ssh slaven
```

A.5 Disable IPv6

Hadoop is only tested in IPv4 network, it may have problems using IPv6. Thus it is necessary to disable IPv6 functionality in system. To do this, add those lines in **/etc/sysctl.conf**:

/etc/sysctl.conf

```
1  net.ipv6.conf.all.disable_ipv6 = 1
2  net.ipv6.conf.default.disable_ipv6 = 1
3  net.ipv6.conf.lo.disable_ipv6 = 1
```

And then make that configuration take effect:

```
1  $ sudo sysctl -p
2  #Check if it really takes effect
3  $ cat /proc/sys/net/ipv6/conf/all/disable_ipv6
```

If the last command returns 1, then it means IPv6 has been successfully disabled.

A.6 Modify System Environment

Next step is to set up several system environment for Hadoop to use. Modify **~/.bashrc** to add those lines in.

~/.bashrc

```
1  export JAVA_HOME=[path to where your java is installed]
2  export HADOOP_PREFIX=[path to where your Hadoop is installed]
3  export HADOOP_HOME=$HADOOP_PREFIX
4  export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
5  export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

To put those modification into use, type the following command:

```
1  $ source ~/.bashrc
2  #Check whether those configuration take effect:
3  $ echo $PATH
4  $ echo $JAVA_HOME
5  $ echo $HADOOP_PREFIX
6  $ echo $HADOOP_HOME
7  $ echo $HADOOP_CONF_DIR
```

A.7 Modify Hadoop Environment

Hadoop uses its own script to setup some environment. So you need to modify one property in **$HADOOP_CONF_DIR/hadoop-env.sh**, telling it where your JAVA is installed:

$HADOOP_CONF_DIR/hadoop-env.sh

```
1  # The java implementation to use.
2  export JAVA_HOME=[where your java is installed, should be the same as
       $JAVA_HOME]
```

A.8 Configure Hadoop Configuration Files

This is the most important step among all. Be careful to make no mistake in configuration files. In this step we shall modify four configuration files in directory '$HADOOP_CONF_DIR': **core-site.xml**, **yarn-site.xml**, **mapred-site.xml** and **hdfs-site.xml**. All configurations should be in the following form:

```
<property>
  <name>[property name]</name>
  <value>[property value</value>
</property>
```

Here's one example of **hdfs-site.xml**:

$HADOOP_CONF_DIR/hdfs-site.xml

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>

  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/home/ubuntu/hadoop-2.7.2/namenode</value>
  </property>
</configuration>
```

All properties should be set are given in the tables below. Hostname 'master' involved in those properties should be substituted by your own hostname defined in file **/etc/hostname** on your master machine. All ports assigned in those properties are free to change, what I wrote there are merely conventional choices.

$HADOOP_CONF_DIR/core-site.xml

```
<configuration>
  <property>
    <!-- Set where HDFS namenode locates -->
    <name>fs.defaultFS</name>
    <value>hdfs://master:8020</value>
  </property>
</configuration>
```

$HADOOP_CONF_DIR/yarn-site.xml

```
<configuration>
  <property>
    <!-- Set where ResourceManager locates. It will assign tasks to
        slaves. -->
    <name>yarn.resourcemanager.address</name>
    <value>master:8050</value>
  </property>

  <property>
    <!-- ResourceManager location for Application Masters to talk to
        Scheduler to obtain resources. -->
    <name>yarn.resourcemanager.scheduler.address</name>
```

```
11      <value>master:8030</value>
12    </property>
13
14    <property>
15      <!-- ResourceManager location for NodeManagers to communicate -->
16      <name>yarn.resourcemanager.resource-tracker.address</name>
17      <value>master:8025</value>
18    </property>
19
20    <property>
21      <!-- ResourceManager location for administrative commands -->
22      <name>yarn.resourcemanager.admin.address</name>
23      <value>master:8087</value>
24    </property>
25
26    <property>
27      <!-- ResourceManager location for users to inspect cluster status via
            web browser -->
28      <name>yarn.resourcemanager.webapp.address</name>
29      <value>master:8088</value>
30    </property>
31
32    <property>
33      <!-- Shuffle service that needs to be set for Map Reduce application
          -->
34      <name>yarn.nodemanager.aux-services</name>
35      <value>mapreduce_shuffle</value>
36    </property>
37
38    <property>
39      <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
40      <value>org.apache.hadoop.mapred.ShuffleHandler</value>
41    </property>
42  </configuration>
```

In **$HADOOP_CONF_DIR**, there may not exist **mapred-site.xml**, instead there's one file called **mapred-site.xml.template**. It is okay to just rename and modify it.

$HADOOP_CONF_DIR/mapred-site.xml

```
1   <configuration>
2     <property>
3       <!-- We choose YARN as the mapreduce resource dispatcher -->
4       <name>mapreduce.framework.name</name>
5       <value>yarn</value>
6     </property>
7
8     <property>
9       <!-- Server that log job history. If not set, there may be errors
            where Map Reduce jobs cannot finish properly -->
10      <name>mapreduce.jobhistory.address</name>
11      <value>master:10020</value>
12    </property>
13
14    <property>
15      <!-- Job history server location for users to inspect via web browser
```

```
        -->
16      <name>mapreduce.jobhistory.webapp.address</name>
17      <value>master:19888</value>
18    </property>
19  </configuration>
```

Attention: There's a slight difference in **$HADOOP_CONF_DIR/hdfs-site.xml** among master and slaves.

$HADOOP_CONF_DIR/hdfs-site.xml[Master]

```
1  <configuration>
2    <property>
3      <!-- HDFS file replication number -->
4      <name>dfs.replication</name>
5      <value>[Number of slaves]</value>
6    </property>
7
8    <property>
9      <!-- Set local position to store HDFS naming information. If the path
           does not exist, you should create it first. -->
10     <name>dfs.namenode.name.dir</name>
11     <value>[Path to your local directory where namenode information will be
         stored]</value>
12   </property>
13 </configuration>
```

$HADOOP_CONF_DIR/hdfs-site.xml[Slave]

```
1  <configuration>
2    <property>
3      <!-- HDFS file replication number -->
4      <name>dfs.replication</name>
5      <value>[Number of slaves]</value>
6    </property>
7
8    <property>
9      <!-- Set local position to store HDFS data content. If the path does
           not exist, you should create it first. -->
10     <name>dfs.datanode.data.dir</name>
11     <value>[Path to your local directory where datanode data will be stored]</value>
12   </property>
13 </configuration>
```

One convenient way to configure all files on all machines is to configure those files on master first and then use *scp* command to copy those files to one slave machine and modify **hdfs-site.xml** than use *scp* again to copy to other slaves.

A.9 Configure Slave List[Master Only]

Last file to configure is **$HADOOP_CONF_DIR/slaves** on master machine. This tells Hadoop master which machines are available. Initially, there's one line written 'localhost' in it. Remove this line and add all slaves' hostnames in it. For example:

$HADOOP_CONF_DIR/slaves

```
slave1
slave2
...
slaven
```

We want master machine to be a master only, rather than do slave's work as well. Thus we only write slaves' hostnames and remove 'localhost'. Every time you change the number of slaves or change machine environment of slaves(such as use another machines, modify Hadoop installation directory and so on), you should reformat HDFS. Approach for formating and reformating HDFS is written in the next sub-section.

A.10 Initialize HDFS

Before starting Hadoop Cluster, HDFS should be firstly formatted. And before that, please make sure those paths for 'dfs.namenode.name.dir' and 'dfs.datanode.data.dir' are valid. Then use the following command(on master machine only) to format it:

```
$ hdfs namenode -format
```

And then you will get a lot of output. Don't worry, if you see 'successfully formatted' and 'exit status 0', that means you've done it well and sound.

A.11 Start Hadoop Cluster

Finally, it comes to the most exiting moment. Make sure all steps above have been done well. Then, we're going to start the cluster up. Use the following commands(on master only) to initiate all services on both master and slave machines.

```
#Start HDFS
$ $HADOOP_HOME/sbin/start-dfs.sh
#Start proxyserver
$ $HADOOP_HOME/sbin/yarn-daemon.sh --config $HADOOP_CONF_DIR start
    proxyserver
#Start YARN services
$ $HADOOP_HOME/sbin/start-yarn.sh
#Start job history server
$ $HADOOP_HOME/sbin/mr-jobhistory-daemon.sh --config $HADOOP_CONF_DIR
    start historyserver
```

After this step, your cluster should be running on every machine. However, it's not time to relax yet. No error report doesn't mean the cluster is healthy and well functioning. We have to perform some further tests which are shown in the next sub-section.

A.12 Inspect Hadoop Cluster Status

There are many ways to diagnose if your cluster is healthy and well performed. What I suggest is that you should pass all tests below to conclude that your cluster is just fine.

- **Services Test.** First and easiest, you can use command *jps* to see whether services are correctly running on your machines. On master machine, output of this command should contain: *NameNode*, *SecondaryNameNode*, *ResourceManager* and *JobHisotryServer*. On slave machines, output should contain: *DataNode* and *NodeManager*.
- **Web App Test.** Second test is to browse specific website established by Hadoop master, as listed below:

- **master:8088** Cluster status site. You can check whether all nodes are correctly connected to your master and whether applications are acutally accepted and running.
- **master:50070** Namenode status site. You can check whether namenode maintains correct status and information of HDFS as well as data nodes.
- **master:19888** Job history server site. You can view job histories here and check whether job history server is correctly set up.

Attention that the specific address should be corresponding to your own configuration.

- **Application Test.** Last test is to actually run an example application on your cluster so see whether it reacts normally and finish jobs as it should do. To accomplish this task, we can use example programs download together with Hadoop. Using the following command:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-example-2.7.2.jar pi 10 100
```

This should output an estimation of π. First parameter defines the number of Map Reduce mapping. Usually, I assign two mapping tasks per VCPU. After initiation of the applicaiton, go to **master:8088/cluster/nodes** and refresh the page frequently to see whether all slaves are active and well functioning.

If there's any service not running, please check above and find which step goes wrong. Also, there are some useful commands to test whether network goes wrong.

- **netstat -ant** Check your local net services to see whether some of local ports are correctly listenning. Usually used on master.
- **nc -v -w 5 [master IP] [port]** Check whether a remote port is open and available to connect. Usually used on slaves to check it can connect to master on certain ports.

A.13 Shutdown Hadoop Cluster

At last! It's time for a cup of coffee. You now own a healthy multi-node Hadoop cluster. When your job finished and want to terminate all services. Do the following commands:

```
#Stop HDFS services
$ $HADOOP_HOME/sbin/stop-dfs.sh
#Stop YARN
$ $HADOOP_HOME/sbin/stop-yarn.sh
#Stop proxy server
$ $HADOOP_HOME/sbin/yarn-daemon.sh --config $HADOOOP_CONF_DIR stop proxyserver
#Stop job history server
$ $HADOOP_HOME/sbin/mr-jobhistory-daemon.sh --config $HADOOP_CONF_DIR stop historyserver
```

B Integrate Hadoop with Pig

Once you have a healthy Hadoop cluster, you can try to integrate it with Pig engine. Here are the specific steps:

B.1 Download Pig

Download package file for Pig and unpack it.

```
$ wget [URL for '.tar.gz' package]
$ tar -xzf [local path for Pig package]
```

B.2 Modify System Environment

Modify system environment so that you can access to Pig program easily and Pig will locate where your Hadoop is and get cluster configuration specific. Modify ∼/**.bashrc** and add those lines in.

∼/.bashrc

```
1  export PIG_HOME=[path to where your Pig is installed]
2  # Tell Pig where to extract Hadoop cluster configurations
3  export PIG_CLASSPATH=$HADOOP_CONF_DIR
4  export PATH=PATH:$PIG_HOME/bin
```

B.3 Inspect Pig Integration

Easiest way to check whether your Pig really integrates with your Hadoop cluster is to run a Pig Latin script under *Hadoop Mode*. For example, you write a script named **script.pig**(better with some map and reduce operations such as *GROUP*, *ORDER* and *JOIN*), then you just type the following command:

```
1  $ pig -x mapreduce script.pig
```

Or simply:

```
1  $ pig script.pig
```

If Pig finishes the task successfully then the integration is correct. If you want to test Pig integration under *Local Mode*, you can check the output of pig when it finishes one task. At nearly the end of its out put, you can find following two lines:

```
1  HadoopVersion  PigVersion  UserId  StartAt  FinishedAt  Features
2  2.7.2          0.15.0      ...     ...      ...         ...
```

If the version is the same as your download version for Hadoop, then it means Pig is successfully running with your Hadoop distribution. However, you should know that there's by default a Hadoop distribution of version 1.x comes together with any Pig archives. Pig will by default use this if no Hadoop is appointed or Hadoop is incorrectly integrated.

C Pig Latin Script Example

Codes bellow is the complete exmaple of our test script: **RankingHDFS.pig**

```
1  REGISTER DemoUDF.jar;
2  REGISTER LogProvUDF.jar;
3
4  DEFINE InterStore      com.logprov.pigUDF.InterStore('http://master
       :58888','d9e490a0-004a-4d40-9b95-7a94b3054de0');
5  DEFINE Clean           test.CleanByRep('19');
6  DEFINE ConvertTime     test.ConvertTime();
7  DEFINE CalDensity      test.CalculateDensity();
8
9  raw = LOAD 'Data/WifiStatusLarge_25.csv' USING PigStorage(',');
10
11 cleaned = FILTER (FOREACH raw GENERATE FLATTEN(Clean(*))) BY NOT ($0
       MATCHES '');
12 cleaned = FOREACH cleaned GENERATE FLATTEN(InterStore('raw', 'Cleanning',
       'cleaned', *));
13
```

```
14 named = FOREACH cleaned GENERATE (chararray)$1 AS LocationID:chararray,
15 (chararray)$3 AS FirstAccess:chararray,
16 (chararray)$4 AS LastAccess:chararray,
17 (int)$5 AS AccessCount:int
18 ;
19 named = FOREACH named GENERATE FLATTEN(InterStore('cleaned', 'Naming', '
       named', *))
20 AS (LocationID:chararray, FirstAccess:chararray, LastAccess:chararray,
       AccessCount:int);
21
22 timed = FOREACH named GENERATE LocationID, ConvertTime(*) AS Duration;
23 timed = FOREACH timed GENERATE FLATTEN(InterStore('named', 'ConvertTime',
        'timed', *))
24 AS (LocationID:chararray, Duration:long);
25
26 timed_grouped = GROUP timed BY LocationID;
27 timed_grouped = FOREACH timed_grouped GENERATE FLATTEN(InterStore('timed
       ', 'TGrouping', 'timed_grouped', *))
28 AS (group:chararray, timed:{(LocationID:chararray, Duration:long)});
29
30 timed_summed = FOREACH timed_grouped GENERATE group AS LocationID, SUM(
       timed.Duration) AS TotalDuration;
31 timed_summed = FOREACH timed_summed GENERATE FLATTEN(InterStore('
       timed_grouped', 'TAccumulation', 'timed_summed', *))
32 AS (LocationID:chararray, TotalDuration:long);
33
34 timed_ordered = ORDER timed_summed BY TotalDuration DESC;
35 STORE timed_ordered INTO 'timed_ordered' USING PigStorage();
36
37 access_grouped = GROUP named BY LocationID;
38 access_grouped = FOREACH access_grouped GENERATE FLATTEN(InterStore('
       named', 'AGrouping', 'access_grouped', *))
39 AS (group:chararray, named:{(LocationID:chararray, FirstAccess:chararray,
        LastAccess:chararray, AccessCount:int)});
40
41 access_summed = FOREACH access_grouped GENERATE group AS LocationID, SUM(
       named.AccessCount) AS TotalAccesses;
42 access_summed = FOREACH access_summed GENERATE FLATTEN(InterStore('
       access_grouped', 'AAccummulation', 'access_summed', *))
43 AS (LocationID:chararray, TotalAccesses:long);
44
45 access_ordered = ORDER access_summed BY TotalAccesses DESC;
46 STORE access_ordered INTO 'access_ordered' USING PigStorage();
47
48 density = JOIN timed_summed BY LocationID, access_summed BY LocationID;
49 density = FOREACH density GENERATE FLATTEN(InterStore('timed_summed,
       access_summed', 'Combinning', 'density', *))
50 AS (timed_summed::LocationID:chararray, timed_summed::TotalDuration:long,
51 access_summed::LocationID2:chararray, access_summed::TotalAccesses:long);
52
53 density_scored = FOREACH density GENERATE timed_summed::LocationID AS
       LocationID,
54 CalDensity(access_summed::TotalAccesses, timed_summed::TotalDuration) AS
       Density:double;
```

```
55  density_scored = FOREACH density_scored GENERATE FLATTEN(InterStore('
        density', 'DScoring', 'density_scored', *))
56  AS (Location:chararray, Density:double);
57
58  density_ordered = ORDER density_scored BY Density DESC;
59  STORE density_ordered INTO 'density_ordered' USING PigStorage();
```

References

[1] S. B. Davidson and J. Freire, "Provenance and scientific workflows: Challenges and opportunities," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1345–1350. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376772

[2] [Online]. Available: http://enterprisearchitects.com/the-5v-s-of-big-data/

[3] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high interest credit card of technical debt," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[4] B. Glavic, "Big data provenance: Challenges and implications for benchmarking," in *2nd Workshop on Big Data Benchmarking (WBDB)*, 2012, pp. 72–80. [Online]. Available: http://cs.iit.edu/%7edbgroup/pdfpubls/G13.pdf

[5] A. Elo, *The rating of chessplayers, past and present.* Arco Pub., 1978. [Online]. Available: https://books.google.com.au/books?id=8pMnAQAAMAAJ

[6] D. Wu, L. Zhu, X. Xu, S. Sakr, D. Sun, and Q. Lu, "Building pipelines for heterogeneous execution environments for big data processing," *IEEE Software*, vol. 33, no. 2, pp. 60–67, Mar 2016.

[7] [Online]. Available: https://nectar.org.au/

[8] [Online]. Available: http://www.logsearch.io/blog/2015/05/performance-testing-elasticsearch.html