Testing and Diagnostics

Table of contents

1. Diagnostic Operators

1.1. DESCRIBE

Returns the schema of a relation.

1.1.1. Syntax

DESCRIBE alias;

1.1.2. Terms

alias	The name of a relation.

1.1.3. Usage

Use the DESCRIBE operator to view the schema of a relation. You can view outer relations as well as relations defined in a nested FOREACH statement.

1.1.4. Example

In this example a schema is specified using the AS clause. If all data conforms to the schema, Pig will use the assigned types.

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);

B = FILTER A BY name matches 'J.+';

C = GROUP B BY name;

D = FOREACH B GENERATE COUNT(B.age);

DESCRIBE A;
A: {group, B: (name: chararray,age: int,gpa: float}

DESCRIBE B;
B: {group, B: (name: chararray,age: int,gpa: float}

DESCRIBE C;
C: {group, chararry,B: (name: chararray,age: int,gpa: float}

DESCRIBE D;
D: {long}
```

In this example no schema is specified. All fields default to type bytearray or long (see Data

Types).

```
a = LOAD 'student';

b = FILTER a BY $0 matches 'J.+';

c = GROUP b BY $0;

d = FOREACH c GENERATE COUNT(b.$1);

DESCRIBE a;
Schema for a unknown.

DESCRIBE b;
2008-12-05 01:17:15,316 [main] WARN  org.apache.pig.PigServer - bytearray
is implicitly cast to chararray under LORegexp Operator
Schema for b unknown.

DESCRIBE c;
2008-12-05 01:17:23,343 [main] WARN  org.apache.pig.PigServer - bytearray
is implicitly caste to chararray under LORegexp Operator
c: {group: bytearray,b: {null}}

DESCRIBE d;
2008-12-05 03:04:30,076 [main] WARN  org.apache.pig.PigServer - bytearray
is implicitly caste to chararray under LORegexp Operator
d: {long}
```

This example shows how to view the schema of a nested relation using the :: operator.

```
A = LOAD 'studentab10k' AS (name, age, gpa);
B = GROUP A BY name;
C = FOREACH B {
     D = DISTINCT A.age;
     GENERATE COUNT(D), group;}

DESCRIBE C::D;
D: {age: bytearray}
```

1.2. DUMP

Dumps or displays results to screen.

1.2.1. Syntax

DUMP alias;

1.2.2. Terms

alias	The name of a relation.

1.2.3. Usage

Use the DUMP operator to run (execute) Pig Latin statements and display the results to your screen. DUMP is meant for interactive mode; statements are executed immediately and the results are not saved (persisted). You can use DUMP as a debugging device to make sure that the results you are expecting are actually generated.

Note that production scripts SHOULD NOT use DUMP as it will disable multi-query optimizations and is likely to slow down execution (see Store vs. Dump).

1.2.4. Example

In this example a dump is performed after each statement.

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);

DUMP A;
(John,18,4.0F)
(Mary,19,3.7F)
(Bill,20,3.9F)
(Joe,22,3.8F)
(Jill,20,4.0F)

B = FILTER A BY name matches 'J.+';

DUMP B;
(John,18,4.0F)
(Joe,22,3.8F)
(Jill,20,4.0F)
```

1.3. EXPLAIN

Displays execution plans.

1.3.1. Syntax

EXPLAIN [–script pigscript] [–out path] [–brief] [–dot] [–param param_name = param_value] [–param_file file_name] alias;

1.3.2. Terms

–script	Use to specify a Pig script.

–out	Use to specify the output path (directory). Will generate a logical_plan[.txt|.dot], physical_plan[.text|.dot], exec_plan[.text|.dot] file in the specified path. Default (no path specified): Stdout
–brief	Does not expand nested plans (presenting a smaller graph for overview).
–dot	Text mode (default): multiple output (split) will be broken out in sections. Dot mode: outputs a format that can be passed to the dot utility for graphical display – will generate a directed-acyclic-graph (DAG) of the plans in any supported format (.gif, .jpg ...).
–param param_name = param_value	See Parameter Substitution.
–param_file file_name	See Parameter Substitution.
alias	The name of a relation.

1.3.3. Usage

Use the EXPLAIN operator to review the logical, physical, and map reduce execution plans that are used to compute the specified relationship.

If no script is given:

- The logical plan shows a pipeline of operators to be executed to build the relation. Type checking and backend-independent optimizations (such as applying filters early on) also apply.
- The physical plan shows how the logical operators are translated to backend-specific physical operators. Some backend optimizations also apply.
- The mapreduce plan shows how the physical operators are grouped into map reduce jobs.

If a script without an alias is specified, it will output the entire execution graph (logical, physical, or map reduce).

If a script with a alias is specified, it will output the plan for the given alias.

1.3.4. Example

In this example the EXPLAIN operator produces all three plans. (Note that only a portion of the output is shown in this example.)

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);

B = GROUP A BY name;

C = FOREACH B GENERATE COUNT(A.age);

EXPLAIN C;
-----------------------------------------------
Logical Plan:
-----------------------------------------------
Store xxx-Fri Dec 05 19:42:29 UTC 2008-23 Schema: {long} Type: Unknown
|
|---ForEach xxx-Fri Dec 05 19:42:29 UTC 2008-15 Schema: {long} Type: bag
 etc ...

-----------------------------------------------
Physical Plan:
-----------------------------------------------
Store(fakefile:org.apache.pig.builtin.PigStorage) - xxx-Fri Dec 05 19:42:29
UTC 2008-40
|
|---New For Each(false)[bag] - xxx-Fri Dec 05 19:42:29 UTC 2008-39
    |   |
    |   POUserFunc(org.apache.pig.builtin.COUNT)[long] - xxx-Fri Dec 05
 etc ...

-------------------------------------------------
| Map Reduce Plan
-------------------------------------------------
MapReduce node xxx-Fri Dec 05 19:42:29 UTC 2008-41
Map Plan
Local Rearrange[tuple]{chararray}(false) - xxx-Fri Dec 05 19:42:29 UTC
2008-34
|   |
|   Project[chararray][0] - xxx-Fri Dec 05 19:42:29 UTC 2008-35
 etc ...
```

1.4. ILLUSTRATE

Displays a step-by-step execution of a sequence of statements.

1.4.1. Syntax

ILLUSTRATE {alias | -script scriptfile};

1.4.2. Terms

alias	The name of a relation.
-script scriptfile	The script keyword followed by the name of a Pig script (for example, myscript.pig). The script file should not contain an ILLUSTRATE statement.

1.4.3. Usage

Use the ILLUSTRATE operator to review how data is transformed through a sequence of Pig Latin statements. ILLUSTRATE allows you to test your programs on small datasets and get faster turnaround times.

ILLUSTRATE is based on an example generator (see Generating Example Data for Dataflow Programs). The algorithm works by retrieving a small sample of the input data and then propagating this data through the pipeline. However, some operators, such as JOIN and FILTER, can eliminate tuples from the data - and this could result in no data following through the pipeline. To address this issue, the algorithm will automatically generate example data, in near real-time. Thus, you might see data propagating through the pipeline that was not found in the original input data, but this data changes nothing and ensures that you will be able to examine the semantics of your Pig Latin statements.

As shown in the examples below, you can use ILLUSTRATE to review a relation or an entire Pig script.

1.4.4. Example - Relation

This example demonstrates how to use ILLUSTRATE with a relation. Note that the LOAD statement must include a schema (the AS clause).

```
grunt> visits = LOAD 'visits.txt' AS (user:chararray, url:chararray,
timestamp:chararray);
grunt> DUMP visits;

(Amy,yahoo.com,19990421)
(Fred,harvard.edu,19991104)
(Amy,cnn.com,20070218)
(Frank,nba.com,20070305)
```

```
(Fred,berkeley.edu,20071204)
(Fred,stanford.edu,20071206)

grunt> recent_visits = FILTER visits BY timestamp >= '20071201';
grunt> user_visits = GROUP recent_visits BY user;
grunt> num_user_visits = FOREACH user_visits GENERATE group,
COUNT(recent_visits);
grunt> DUMP num_user_visits;

(Fred,2)

grunt> ILLUSTRATE num_user_visits;
-------------------------------------------------------------------------
| visits     | user: chararray | url: chararray | timestamp: chararray |
-------------------------------------------------------------------------
|            | Fred            | berkeley.edu   | 20071204             |
|            | Fred            | stanford.edu   | 20071206             |
|            | Frank           | nba.com        | 20070305             |
-------------------------------------------------------------------------
-------------------------------------------------------------------------------
| recent_visits     | user: chararray | url: chararray | timestamp:
chararray |
-------------------------------------------------------------------------------
|                   | Fred            | berkeley.edu   | 20071204
|
|                   | Fred            | stanford.edu   | 20071206
|
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
| user_visits     | group: chararray | recent_visits: bag({user:
chararray,url: chararray,timestamp: chararray}) |
-------------------------------------------------------------------------------------------
|                 | Fred             | {(Fred, berkeley.edu, 20071204),
(Fred, stanford.edu, 20071206)}          |
-------------------------------------------------------------------------------------------
-------------------------------------------------
| num_user_visits     | group: chararray | long  |
-------------------------------------------------
|                     | Fred             | 2     |
-------------------------------------------------
```

1.4.5. Example - Script

This example demonstrates how to use ILLUSTRATE with a Pig script. Note that the script itself should not contain an ILLUSTRATE statement.

```
grunt> cat visits.txt
Amy     yahoo.com       19990421
Fred    harvard.edu     19991104
Amy     cnn.com 20070218
Frank   nba.com 20070305
Fred    berkeley.edu    20071204
Fred    stanford.edu    20071206
```

```
grunt> cat visits.pig
visits = LOAD 'visits.txt' AS (user, url, timestamp);
recent_visits = FILTER visits BY timestamp >= '20071201';
historical_visits = FILTER visits BY timestamp <= '20000101';
DUMP recent_visits;
DUMP historical_visits;
STORE recent_visits INTO 'recent';
STORE historical_visits INTO 'historical';

grunt> exec visits.pig

(Fred,berkeley.edu,20071204)
(Fred,stanford.edu,20071206)

(Amy,yahoo.com,19990421)
(Fred,harvard.edu,19991104)


grunt> illustrate -script visits.pig

-------------------------------------------------------------------------
| visits     | user: bytearray | url: bytearray | timestamp: bytearray |
-------------------------------------------------------------------------
|            | Amy             | yahoo.com      | 19990421             |
|            | Fred            | stanford.edu   | 20071206             |
-------------------------------------------------------------------------
-------------------------------------------------------------------------------
| recent_visits     | user: bytearray | url: bytearray | timestamp:
bytearray |
-------------------------------------------------------------------------------
|                   | Fred            | stanford.edu   | 20071206
|
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| Store : recent_visits     | user: bytearray | url: bytearray | timestamp:
bytearray |
-------------------------------------------------------------------------------
|                           | Fred            | stanford.edu   | 20071206
|
-------------------------------------------------------------------------------
-------------------------------------------------------------------------
| historical_visits     | user: bytearray | url: bytearray | timestamp:
bytearray |
-------------------------------------------------------------------------
|                       | Amy             | yahoo.com      | 19990421
|
-------------------------------------------------------------------------
-------------------------------------------------------------------------------
| Store : historical_visits     | user: bytearray | url: bytearray |
timestamp: bytearray |
-------------------------------------------------------------------------------
|                               | Amy             | yahoo.com      |
19990421             |
```

2. Pig Scripts and MapReduce Job IDs

Complex Pig scripts often generate many MapReduce jobs. To help you debug a script, Pig prints a summary of the execution that shows which relations (aliases) are mapped to each MapReduce job.

```
JobId Maps Reduces MaxMapTime MinMapTIme AvgMapTime MaxReduceTime
    MinReduceTime AvgReduceTime Alias Feature Outputs
job_201004271216_12712 1 1 3 3 3 12 12 12 B,C GROUP_BY,COMBINER
job_201004271216_12713 1 1 3 3 3 12 12 12 D SAMPLER
job_201004271216_12714 1 1 3 3 3 12 12 12 D ORDER_BY,COMBINER
    hdfs://mymachine.com:9020/tmp/temp743703298/tmp-2019944040,
```

3. Pig Statistics

Pig Statistics is a framework for collecting and storing script-level statistics for Pig Latin. Characteristics of Pig Latin scripts and the resulting MapReduce jobs are collected while the script is executed. These statistics are then available for Pig users and tools using Pig (such as Oozie) to retrieve after the job is done.

The new Pig statistics and the existing Hadoop statistics can also be accessed via the Hadoop job history file (and job xml file). Piggybank has a HadoopJobHistoryLoader which acts as an example of using Pig itself to query these statistics (the loader can be used as a reference implementation but is NOT supported for production use).

3.1. Java API

Several new public classes make it easier for external tools such as Oozie to integrate with Pig statistics.

The Pig statistics are available here: http://pig.apache.org/docs/r0.9.0/api/

The stats classes are in the package: org.apache.pig.tools.pigstats

- PigStats
- JobStats
- OutputStats
- InputStats

The PigRunner class mimics the behavior of the Main class but gives users a statistics object back. Optionally, you can call the API with an implementation of progress listener which will be invoked by Pig runtime during the execution.

```
package org.apache.pig;

public abstract class PigRunner {
    public static PigStats run(String[] args,
PigProgressNotificationListener listener)
}

public interface PigProgressNotificationListener extends
java.util.EventListener {
    // just before the launch of MR jobs for the script
    public void LaunchStartedNotification(int numJobsToLaunch);
    // number of jobs submitted in a batch
    public void jobsSubmittedNotification(int numJobsSubmitted);
    // a job is started
    public void jobStartedNotification(String assignedJobId);
    // a job is completed successfully
    public void jobFinishedNotification(JobStats jobStats);
    // a job is failed
    public void jobFailedNotification(JobStats jobStats);
    // a user output is completed successfully
    public void outputCompletedNotification(OutputStats outputStats);
    // updates the progress as percentage
    public void progressUpdatedNotification(int progress);
    // the script execution is done
    public void launchCompletedNotification(int numJobsSucceeded);
}
```

3.2. Job XML

The following entries are included in job conf:

Pig Statistic	**Description**
pig.script.id	The UUID for the script. All jobs spawned by the script have the same script ID.
pig.script	The base64 encoded script text.
pig.command.line	The command line used to invoke the script.
pig.hadoop.version	The Hadoop version installed.
pig.version	The Pig version used.
pig.input.dirs	A comma-separated list of input directories for the job.

pig.map.output.dirs	A comma-separated list of output directories in the map phase of the job.
pig.reduce.output.dirs	A comma-separated list of output directories in the reduce phase of the job.
pig.parent.jobid	A comma-separated list of parent job ids.
pig.script.features	A list of Pig features used in the script.
pig.job.feature	A list of Pig features used in the job.
pig.alias	The alias associated with the job.

3.3. Hadoop Job History Loader

The HadoopJobHistoryLoader in Piggybank loads Hadoop job history files and job xml files from file system. For each MapReduce job, the loader produces a tuple with schema (j:map[], m:map[], r:map[]). The first map in the schema contains job-related entries. Here are some of important key names in the map:

PIG_SCRIPT_ID	USER	SUBMIT_TIME
CLUSTER	HADOOP_VERSION	LAUNCH_TIME
QUEUE_NAME	PIG_VERSION	FINISH_TIME
JOBID	PIG_JOB_FEATURE	TOTAL_MAPS
JOBNAME	PIG_JOB_ALIAS	TOTAL_REDUCES
STATUS	PIG_JOB_PARENTS	

Examples that use the loader to query Pig statistics are shown below.

3.4. Examples

Find scripts that generate more then three MapReduce jobs:

```
a = load '/mapred/history/done' using HadoopJobHistoryLoader() as (j:map[],
m:map[], r:map[]);
b = group a by (j#'PIG_SCRIPT_ID', j#'USER', j#'JOBNAME');
c = foreach b generate group.$1, group.$2, COUNT(a);
```

```
d = filter c by $2 > 3;
dump d;
```

Find the running time of each script (in seconds):

```
a = load '/mapred/history/done' using HadoopJobHistoryLoader() as (j:map[],
m:map[], r:map[]);
b = foreach a generate j#'PIG_SCRIPT_ID' as id, j#'USER' as user,
j#'JOBNAME' as script_name,
        (Long) j#'SUBMIT_TIME' as start, (Long) j#'FINISH_TIME' as end;
c = group b by (id, user, script_name)
d = foreach c generate group.user, group.script_name, (MAX(b.end) -
MIN(b.start)/1000;
dump d;
```

Find the number of scripts run by user and queue on a cluster:

```
a = load '/mapred/history/done' using HadoopJobHistoryLoader() as (j:map[],
m:map[], r:map[]);
b = foreach a generate j#'PIG_SCRIPT_ID' as id, j#'USER' as user,
j#'QUEUE_NAME' as queue;
c = group b by (id, user, queue) parallel 10;
d = foreach c generate group.user, group.queue, COUNT(b);
dump d;
```

Find scripts that have failed jobs:

```
a = load '/mapred/history/done' using HadoopJobHistoryLoader() as (j:map[],
m:map[], r:map[]);
b = foreach a generate (Chararray) j#'STATUS' as status, j#'PIG_SCRIPT_ID'
as id, j#'USER' as user, j#'JOBNAME' as script_name, j#'JOBID' as job;
c = filter b by status != 'SUCCESS';
dump c;
```

Find scripts that use only the default parallelism:

```
a = load '/mapred/history/done' using HadoopJobHistoryLoader() as (j:map[],
m:map[], r:map[]);
b = foreach a generate j#'PIG_SCRIPT_ID' as id, j#'USER' as user,
j#'JOBNAME' as script_name, (Long) r#'NUMBER_REDUCES' as reduces;
c = group b by (id, user, script_name) parallel 10;
d = foreach c generate group.user, group.script_name, MAX(b.reduces) as
max_reduces;
e = filter d by max_reduces == 1;
dump e;
```

4. PigUnit

PigUnit is a simple xUnit framework that enables you to easily test your Pig scripts. With PigUnit you can perform unit testing, regression testing, and rapid prototyping. No cluster set up is required if you run Pig in local mode.

4.1. Build PigUnit

To compile PigUnit run the command shown below from the Pig trunk. The compile will create the pigunit.jar file.

```
$pig_trunk ant pigunit-jar
```

4.2. Run PigUnit

You can run PigUnit using Pig's local mode or mapreduce mode.

4.2.1. Local Mode

PigUnit runs in Pig's local mode by default. Local mode is fast and enables you to use your local file system as the HDFS cluster. Local mode does not require a real cluster but a new local one is created each time.

4.2.2. Mapreduce Mode

PigUnit also runs in Pig's mapreduce mode. Mapreduce mode requires you to use a Hadoop cluster and HDFS installation. It is enabled when the Java system property pigunit.exectype.cluster is set to any value: e.g. -Dpigunit.exectype.cluster=true or System.getProperties().setProperty("pigunit.exectype.cluster", "true"). The cluster you select must be specified in the CLASSPATH (similar to the HADOOP_CONF_DIR variable).

4.3. PigUnit Example

Many PigUnit examples are available in the PigUnit tests.

The example included here computes the top N of the most common queries. The Pig script, top_queries.pig, is similar to the Query Phrase Popularity in the Pig tutorial. It expects an input a file of queries and a parameter n (n is 2 in our case in order to do a top 2).

Setting up a test for this script is easy because the argument and the input data are specified by two text arrays. It is the same for the expected output of the script that will be compared to the actual result of the execution of the Pig script.

4.3.1. Java Test

```
@Test
public void testTop2Queries() {
  String[] args = {
```

```
        "n=2",
        };

    PigTest test = new PigTest("top_queries.pig", args);

    String[] input = {
        "yahoo",
        "yahoo",
        "yahoo",
        "twitter",
        "facebook",
        "facebook",
        "linkedin",
    };

    String[] output = {
        "(yahoo,3)",
        "(facebook,2)",
    };

    test.assertOutput("data", input, "queries_limit", output);
  }
```

4.3.2. top_queries.pig

```
data =
    LOAD 'input'
    AS (query:CHARARRAY);

queries_group =
    GROUP data
    BY query;

queries_count =
    FOREACH queries_group
    GENERATE
        group AS query,
        COUNT(data) AS total;

queries_ordered =
    ORDER queries_count
    BY total DESC, query;

queries_limit =
    LIMIT queries_ordered $n;

STORE queries_limit INTO 'output';
```

4.3.3. Run

The test can be executed by JUnit (or any other Java testing framework). It requires:

1. pig.jar

2. pigunit.jar

The test takes about 25s to run and should pass. In case of error (for example change the parameter n to n=3), the diff of output is displayed:

```
junit.framework.ComparisonFailure: null expected:<...ahoo,3)
(facebook,2)[]> but was:<...ahoo,3)
(facebook,2)[
(linkedin,1)]>
        at junit.framework.Assert.assertEquals(Assert.java:81)
        at junit.framework.Assert.assertEquals(Assert.java:87)
        at org.apache.pig.pigunit.PigTest.assertEquals(PigTest.java:272)
```

4.4. Troubleshooting Tips

Common problems you may encounter are discussed below.

4.4.1. Classpath in Mapreduce Mode

When using PigUnit in mapreduce mode, be sure to include the $HADOOP_CONF_DIR of the cluster in your CLASSPATH.

The default value is ~/pigtest/conf.

```
org.apache.pig.backend.executionengine.ExecException:
ERROR 4010: Cannot find hadoop configurations in classpath
(neither hadoop-site.xml nor core-site.xml was found in the classpath).
If you plan to use local mode, please put -x local option in command line
```

4.4.2. UDF jars Not Found

This error means that you are missing some jars in your test environment.

```
WARN util.JarManager: Couldn't find the jar for
org.apache.pig.piggybank.evaluation.string.LOWER, skip it
```

4.4.3. Storing Data

Pig currently drops all STORE and DUMP commands. You can tell PigUnit to keep the commands and execute the script:

```
test = new PigTest(PIG_SCRIPT, args);
test.unoverride("STORE");
test.runScript();
```

4.4.4. Cache Archive

For cache archive to work, your test environment needs to have the cache archive options specified by Java properties or in an additional XML configuration in its CLASSPATH.

If you use a local cluster, you need to set the required environment variables before starting it:

```
export LD_LIBRARY_PATH=/home/path/to/lib
```

4.5. Future Enhancements

Improvements and other components based on PigUnit that could be built later.

For example, we could build a PigTestCase and PigTestSuite on top of PigTest to:

1. Add the notion of workspaces for each test.
2. Remove the boiler plate code appearing when there is more than one test methods.
3. Add a standalone utility that reads test configurations and generates a test report.

5. Penny

Note: *Penny is an experimental feature.*

Penny is a framework for creating Pig monitoring and debugging tools. Penny comes with a library of tools (see Penny Tool Library). However, the real power of Penny is in creating your own custom monitoring and debugging tools using Penny's simple API.

5.1. How it Works

Before you can create a tool, you need to understand how Penny instruments Pig scripts (called "dataflow programs" in the following diagram).

As shown in the diagram, Penny inserts one or more monitor agents (called "Penny agent" in the diagram) between steps of the Pig script, which observe data flowing between the Pig script steps. Monitor agents run arbitrary Java code as needed for your tool, which has access to some primitives for tagging records and communicating with other agents and with a central coordinator process (called "Penny coordinator" in the diagram). The coordinator also runs arbitrary code defined by your tool.

The whole thing is kicked off by the tool's Main program (called "application" in the diagram), which receives instructions from the user (e.g. "please figure out why this Pig script keeps crashing"), launches one or more runs of the Pig script instrumented with monitor agents, and reports the outcome back to the user (e.g. "the crash appears to be caused by one of these records: ...").

5.2. API

You need to write three Java classes: a Main class, a Coordinator class, and a MonitorAgent class (for certain, fancy tools, you may need multiple MonitorAgent classes). You can find many examples of Main/Coordinator/MonitorAgent classes that define Penny tools in the Penny source code (/pig/trunk/contrib/penny/java/src/main/java/) under org.apache.pig.penny.apps. All of the tools described in Penny Tool Library are written using this API, so you've got plenty of examples to work with. We'll paste a few code fragments below to get you going -- in fact the entire code for the "data samples" tool (all 97 lines of Java) is included below.

5.2.1. Main Class

Your Main class is the "shell" of your application. It receives instructions from the user, and configures and launches one or more Penny-instrumented runs of the user's Pig script.

You talk to Penny via the PennyServer class. You can do two things: (1) parse a user's Pig script and (2) launch an Penny-instrumented run of the Pig script. Here is the Main class for the data samples tool, described in Penny Tool Library:

```
import java.util.HashMap;
import java.util.Map;
import org.apache.pig.penny.ClassWithArgs;
import org.apache.pig.penny.ParsedPigScript;
import org.apache.pig.penny.PennyServer;

/**
 * Data samples app.
 */
public class Main {
    public static void main(String[] args) throws Exception {
        PennyServer pennyServer = new PennyServer();
        String pigScriptFilename = args[0];
        ParsedPigScript parsedPigScript =
pennyServer.parse(pigScriptFilename);
        Map<String, ClassWithArgs> monitorClasses = new HashMap<String,
ClassWithArgs>();
        for (String alias : parsedPigScript.aliases()) {
            monitorClasses.put(alias, new
ClassWithArgs(DSMonitorAgent.class));
        }
        parsedPigScript.trace(DSCoordinator.class, monitorClasses);
    }
}
```

The "monitorClasses" map dictates which monitor agent (if any) to place after each dataflow step (steps are identified by Pig script aliases). You can also pass arguments to each monitor

agent, and/or to the coordinator, as shown in this example for the "data histograms" tool:

```
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
import org.apache.pig.penny.ClassWithArgs;
import org.apache.pig.penny.ParsedPigScript;
import org.apache.pig.penny.PennyServer;

/**
 * Data summaries app. that computes a histogram of one of the fields of
one of the intermediate data sets.
 */

public class Main {
    public static void main(String[] args) throws Exception {
        PennyServer pennyServer = new PennyServer();
        String pigScriptFilename = args[0];
        ParsedPigScript parsedPigScript =
pennyServer.parse(pigScriptFilename);
        String alias = args[1]; // which alias to create histogram for
        int fieldNo = Integer.parseInt(args[2]); // which field to create
histogram for
        int min = Integer.parseInt(args[3]); // min field value
        int max = Integer.parseInt(args[4]); // max field value
        int bucketSize = Integer.parseInt(args[5]); // histogram bucket
size
        if (!parsedPigScript.aliases().contains(alias)) throw new
IllegalArgumentException("No such alias.");
        Map<String, ClassWithArgs> monitorClasses = new HashMap<String,
ClassWithArgs>();
        monitorClasses.put(alias, new ClassWithArgs(DHMonitorAgent.class,
fieldNo, min, max, bucketSize));
        TreeMap<Integer, Integer> histogram = (TreeMap<Integer, Integer>)
parsedPigScript.trace(DHCoordinator.class, monitorClasses);
        System.out.println("Histogram: " + histogram);
    }
}
```

5.2.2. MonitorAgent Class

Monitor agents implement the following API:

```
 /**
  * Furnish set of fields to monitor. (Null means monitor all fields
('*').)
  * /
 public abstract Set<Integer> furnishFieldsToMonitor(); /**
  * Initialize, using any arguments passed from higher layer.
  * /
 public abstract void init(Serializable[] args);
 /**
```

```
  * Process a tuple that passes through the monitoring point.
  *
  * @param t   the tuple
  * @param tag t's tags
  * @return FILTER_OUT to remove the tuple from the data stream;
  *    NO_TAGS to let it pass through and not give it any tags;
  *    a set of tags to let it pass through and assign those tags
  */
 public abstract Set<String> observeTuple(Tuple t, Set<String> tags) throws
ExecException;
 /**
  * Process an incoming (synchronous or asynchronous) message.
  */
 public abstract void receiveMessage(Location source, Tuple message);
 /**
  * No more tuples are going to pass through the monitoring point. Finish
any ongoing processing.
  */
 public abstract void finish();
```

Here's an example from the "data samples" tool:

```
import java.io.Serializable; import java.util.Set;

import org.apache.pig.backend.executionengine.ExecException; import
org.apache.pig.data.Tuple;

import org.apache.pig.penny.Location; import
org.apache.pig.penny.MonitorAgent;

public class DSMonitorAgent extends MonitorAgent {

    private final static int NUM_SAMPLES = 5;
    private int tupleCount = 0;
    public void finish() { }
    public Set<Integer> furnishFieldsToMonitor() {
        return null;
    }
    public void init(Serializable[] args) { }
    public Set<String> observeTuple(Tuple t, Set<String> tags) throws
ExecException {
        if (tupleCount++ < NUM_SAMPLES) {
            communicator().sendToCoordinator(t);
        }
        return tags;
    }
    public void receiveMessage(Location source, Tuple message) { }
}
```

Monitor agents have access to a "communicator" object, which is the gateway for sending messages to other agents or to the coordinator. The communicator API is:

```
  /**
   * Find out my (physical) location.
   * /
 public abstract Location myLocation();
 /**
   * Send an message to the coordinator, asynchronously.
   * /
 public abstract void sendToCoordinator(Tuple message);
 /**
   * Send a message to immediate downstream neighbor(s), synchronously.
   * If downstream neighbor(s) span a task boundary, all instances will
receive it; otherwise only same-task instances will receive it.
   * If there is no downstream neighbor, an exception will be thrown.
   * /
 public abstract void sendDownstream(Tuple message) throws
NoSuchLocationException;
 /**
   * Send a message to immediate upstream neighbor(s), synchronously.
   * If upstream neighbor(s) are non-existent or span a task boundary, an
exception will be thrown.
   * /
 public abstract void sendUpstream(Tuple message) throws
NoSuchLocationException;
 /**
   * Send a message to current/future instances of a given logical location.
   * Instances that have already terminated will not receive the message
(obviously).
   * Instances that are currently executing will receive it asynchronously
(or perhaps not at all, if they terminate before the message arrives).
   * Instances that have not yet started will receive the message prior to
beginning processing of tuples.
   * /
 public abstract void sendToAgents(LogicalLocation destination, Tuple
message) throws NoSuchLocationException;
 // The following methods mirror the ones above, but take care of packaging
a list of objects into a tuple (you're welcome!) ...
 public void sendToCoordinator(Object ... message) {
   . sendToCoordinator(makeTuple(message));
 }
 public void sendDownstream(Object ... message) throws
NoSuchLocationException {
   . sendDownstream(makeTuple(message));
 }
 public void sendUpstream(Object ... message) throws
NoSuchLocationException {
   . sendUpstream(makeTuple(message));
 }
 public void sendToAgents(LogicalLocation destination, Object ... message)
throws NoSuchLocationException {
   . sendToAgents(destination, makeTuple(message));
 }
```

5.2.3. Coordinator Class

Your tool's coordinator implements the following API:

```
 /**
  * Initialize, using any arguments passed from higher layer.
  * /
 public abstract void init(Serializable[] args);
 /**
  * Process an incoming (synchronous or asynchronous) message.
  * /
 public abstract void receiveMessage(Location source, Tuple message); /**
  * The data flow has completed and all messages have been delivered.
Finish processing.
  *   * @return             final output to pass back to application
  * /
 public abstract Object finish();
```

The coordinator for the "data samples" tool is:

```
import java.io.Serializable;
import org.apache.pig.data.Tuple;
import org.apache.pig.penny.Coordinator;
import org.apache.pig.penny.Location;
public class DSCoordinator extends Coordinator {
    public void init(Serializable[] args) { }
    public Object finish() {
        return null;
    }
    public void receiveMessage(Location source, Tuple message) {
        System.out.println("*** SAMPLE RECORD AT ALIAS " + source.logId() +
": " + truncate(message));
    }
    private String truncate(Tuple t) {
        String s = t.toString();
        return s.substring(0, Math.min(s.length(), 100));
    }
}
```