Control Structures

Table of contents

1 Embedded Pig - Python, JavaScript and Groovy

To enable control flow, you can embed Pig Latin statements and Pig commands in the Python, JavaScript and Groovy scripting languages using a JDBC-like compile, bind, run model. For Python, make sure the Jython jar is included in your class path. For JavaScript, make sure the Rhino jar is included in your classpath. For Groovy, make sure the groovy-all jar is included in your classpath.

Note that host languages and the languages of UDFs (included as part of the embedded Pig) are completely orthogonal. For example, a Pig Latin statement that registers a Python UDF may be embedded in Python, JavaScript, or Java. The exception to this rule is "combined" scripts – here the languages must match (see the Advanced Topics for Python, Advanced Topics for JavaScript and Advanced Topics for Groovy).

1.1 Invocation Basics

Embedded Pig is supported in batch mode only, not interactive mode. You can request that embedded Pig be used by adding the `--embedded` option to the Pig command line. If this option is passed as an argument, that argument will refer to the language Pig is embedded in, either Python, JavaScript or Groovy. If no argument is specified, it is taken to refer to the reference implementation for Python.

Python

```
 $ pig myembedded.py
OR
$ java -cp <jython jars>:<pig jars>; [--embedded python] /tmp/myembedded.py
```

Pig will look for the `#!/usr/bin/python` line in the script.

```
#!/usr/bin/python

# explicitly import Pig class
from org.apache.pig.scripting import Pig

# COMPILE: compile method returns a Pig object that represents the pipeline
P = Pig.compile("a = load '$in'; store a into '$out';")

input = 'original'
output = 'output'

# BIND and RUN
result = P.bind({'in':input, 'out':output}).runSingle()

if result.isSuccessful() :
    print 'Pig job succeeded'
else :
    raise 'Pig job failed'
```

JavaScript

```
$ pig myembedded.js
OR
$ java -cp <rhino jars>:<pig jars>; [--embedded javascript] /tmp/myembedded.js
```

Pig will look for the *.js extension in the script.

```
importPackage(Packages.org.apache.pig.scripting.js)

Pig = org.apache.pig.scripting.js.JSPig

function main() {
    input = "original"
    output = "output"

    P = Pig.compile("A = load '$in'; store A into '$out';")

    result = P.bind({'in':input, 'out':output}).runSingle()

    if (result.isSuccessful()) {
        print("Pig job succeeded")
    } else {
        print("Pig job failed")
    }
}
```

Groovy

```
$ pig myembedded.groovy
OR
$ java -cp <groovy-all jar>:<pig jars>; [--embedded groovy] /tmp/myembedded.groovy
```

Pig will look for the *.groovy extension in the script.

```
import org.apache.pig.scripting.Pig;

public static void main(String[] args) {
  String input = "original"
  String output = "output"

  Pig P = Pig.compile("A = load '\$in'; store A into '\$out';")

  result = P.bind(['in':input, 'out':output]).runSingle()

  if (result.isSuccessful()) {
    print("Pig job succeeded")
  } else {
    print("Pig job failed")
  }
```

```
}
```

Invocation Process

You invoke Pig in the host scripting language through an embedded Pig object.

Compile: Compile is a static function on the Pig class and in its simplest form takes a fragment of Pig Latin that defines the pipeline as its input:

```
# COMPILE: complie method returns a Pig object that represents the pipeline
P = Pig.compile("""A = load '$in'; store A into '$out';""")
```

Compile returns an instance of Pig object. This object can have certain values undefined. For example, you may want to define a pipeline without yet specifying the location of the input to the pipeline. The parameter will be indicated by a dollar sign followed by a sequence of alpha-numeric or underscore characters. Values for these parameters must be provided later at the time bind() is called on the Pig object. To call run() on a Pig object without all parameters being bound is an error.

Bind: Resolve the parameters during the bind call.

```
input = "original"
output = "output"

# BIND: bind method binds the variables with the parameters in the pipeline and returns a
 BoundScript object
Q = P.bind({'in':input, 'out':output})
```

Please note that all parameters must be resolved during bind. Having unbound parameters while running your script is an error. Also note that even if your script is fully defined during compile, bind without parameters still must be called.

Run: Bind call returns an instance of BoundScript object that can be used to execute the pipeline. The simplest way to execute the pipeline is to call runSingle function. (However, as mentioned later, this works only if a single set of variables is bound to the parameters. Otherwise, if multiple set of variables are bound, an exception will be thrown if runSingle is called.)

```
result = Q.runSingle()
```

The function returns a PigStats object that tells you whether the run succeeded or failed. In case of success, additional run statistics are provided.

Embedded Python Example

A complete embedded example is shown below.

```
#!/usr/bin/python

# explicitly import Pig class
from org.apache.pig.scripting import Pig

# COMPILE: compile method returns a Pig object that represents the pipeline
P = Pig.compile("""A = load '$in'; store A into '$out';""")

input = "original”
output = "output”

# BIND: bind method binds the variables with the parameters in the pipeline and returns a
 BoundScript object
Q = P.bind({'in':input, 'out':output})

# In this case, only one set of variables is bound to the pipeline, runSingle method
 returns a PigStats object.
# If multiple sets of variables are bound to the pipeline, run method instead must be
 called and it returns
# a list of PigStats objects.
result = Q.runSingle()

# check the result
if result.isSuccessful():
    print "Pig job succeeded"
else:
    raise "Pig job failed"


OR, SIMPLY DO THIS:


#!/usr/bin/python

# explicitly import Pig class
from org.apache.pig.scripting import Pig

in = "original”
out = "output”

# implicitly bind the parameters to the local variables
result= Pig.compile("""A = load '$in'; store A into '$out';""").bind().runSingle()

if result.isSuccessful():
    print "Pig job succeeded"
else:
    raise "Pig job failed"
```

1.2 Invocation Details

All three APIs (compile, bind, run) discussed in the previous section have several versions depending on what you are trying to do.

1.2.1 Compile

In its basic form, compile just takes a Pig Latin fragment that defines the pipeline as described in the previous section. Additionally, the pipeline can be given a name. This name is only used when the embedded script is invoked via the PigRunner Java API (as discussed later in this document).

```
 P = Pig.compile("P1", """A = load '$in'; store A into '$out';""")
```

In addition to providing Pig script via a string, you can store it in a file and pass the file to the compile call:

```
P = Pig.compileFromFile("myscript.pig")
```

You can also name a pipeline stored in the script:

```
P = Pig.compileFromFile("P2", "myscript.pig")
```

1.2.2 Bind

In its simplest form, bind takes no parameters. In this case, an implicit bind is performed; Pig internally constructs a map of parameters from the local variables specified by the user in the script.

```
Q = P.bind()
```

Finally, you might want to run the same pipeline in parallel with a different set of parameters, for instance for multiple dates. In this case, bind function, needs to be passed a list of maps with each element of the list containing parameters for a single invocation. In the example below, the pipeline is run for the US, the UK, and Brazil.

```
P = Pig.compile("""A = load '$in';
                   B = filter A by user is not null;
                   ...
                   store Z into '$out';
                """)

Q = P.bind([{'in':'us_raw','out':'us_processed'},
        {'in':'uk_raw','out':'uk_processed'},
        {'in':'brazil_raw','out':'brazil_processed'}])

results = Q.run() # it blocks until all pipelines are completed
```

```
for i in [0, 1, 2]:
    result = results[i]
    ... # check result for each pipeline
```

1.2.3 Run

We have already seen that the simplest way to run a script is to call runSingle without any parameters. Additionally, a Java Properties object or a file containing a list of properties can be passed to this call. The properties are passed to Pig and a treated as any other properties passed from command line.

```
# In a jython script

from java.util import Properties
... ...

props = Properties()
props.put(key1, val1)
props.put(key2, val2)
... ...

Pig.compile(...).bind(...).runSingle(props)
```

A more general version of run allows to run one or more pipelines concurrently. In this case, a list of PigStats results is returned – one for each pipeline run. The example in the previous section shows how to make use of this call.

As the case with runSingle, a set of Java Properties or a property file can be passed to the call.

1.2.4 Passing Parameters to a Script

Inside your script, you can define parameters and then pass parameters from command line to your script. There are two ways to pass parameters to your script:

1.2.4.1 1. -param

Similar to regular Pig parameter substitution, you can define parameters using -param/–param_file on Pig's command line. This variable will be treated as one of the binding variables when binding the Pig Latin script. For example, you can invoke the below Python script using: pig –param loadfile=student.txt script.py.

```
#!/usr/bin/python
from org.apache.pig.scripting import Pig

P = Pig.compile("""A = load '$loadfile' as (name, age, gpa);
store A into 'output';""")
```

```
Q = P.bind()

result = Q.runSingle()
```

1.2.4.2 2. Command line arguments

Currently this feature is only available in Python and Groovy. You can pass command line arguments (the arguments after the script file name) to Python. These will become sys.argv in Python and will be passed as main's arguments in Groovy. For example: pig script.py student.txt. The corresponding script is:

```
#!/usr/bin/python
import sys
from org.apache.pig.scripting import Pig

P = Pig.compile("A = load '" + sys.argv[1] + "' as (name, age, gpa);" +
"store A into 'output';");

Q = P.bind()

result = Q.runSingle()
```

and in Groovy, pig script.groovy student.txt:

```
import org.apache.pig.scripting.Pig;

public static void main(String[] args) {

  P = Pig.compile("A = load '" + args[1] + "' as (name, age, gpa);" +
                      "store A into 'output';");

  Q = P.bind()

  result = Q.runSingle()
}
```

1.3 PigRunner API

Starting with Pig 0.8, some applications such as Oozie workflow invoke Pig using the PigRunner Java class rather than through the command line. For these applications, the PigRunner interface has been expanded to accommodate embedded Pig. PigRunner accepts Python and JavaScript scripts as input. These scripts can potentially contain multiple Pig pipelines; therefore, we need a way to return results for all of them.

To do this and to preserve backward compatibility PigStats and related objects were expanded as shown below:

- PigStats is now an abstract class. (PigStats as it was before has become SimplePigStats.)

- SimplePigStats is a new class that extends PigStats. SimplePigStats.getAllStats() will return null.
- EmbeddedPigStats is a new class that extends PigStats. EmbeddedPigStats will return null for methods not listed in the proposal below.
- isEmbedded() is a new abstract method that accommodates embedded Pig.
- getAllStats() and List< > getAllErrorMessages() methods were added to the PigStats class. The map returned from getAllStats is keyed on the name of the pipeline provided in the compile call. If the name was not compiled an internally generated id would be used.
- The PigProgressNotificationListener interface was modified to add script id to all its methods.

For more details, see Java Objects.

1.4 Usage Examples

1.4.1 Passing a Pig Script

This example shows you how to pass an entire Pig script to the compile call.

```
#!/usr/bin/python

from org.apache.pig.scripting import Pig

P = Pig.compileFromFile("""myscript.pig""")

input = "original"
output = "output"

result = p.bind({'in':input, 'out':output}).runSingle()
if result.isSuccessful():
    print "Pig job succeeded"
else:
    raise "Pig job failed"
```

1.4.2 Convergence

There is a class of problems that involve iterating over a data pipeline an indeterminate number of times until a certain value is reached. Examples arise in machine learning, graph traversal, and a host of numerical analysis problems which involve finding interpolations, extrapolations or regressions. The Python example below shows one way to achieve convergence using Pig scripts.

```
#!/usr/bin/python

# explicitly import Pig class
from org.apache.pig.scripting import Pig
```

```
P = Pig.compile("""A = load '$input' as (user, age, gpa);
                   B = group A all;
                   C = foreach B generate AVG(A.gpa);
                   store C into '$output';
                """)
# initial output
input = "studenttab5"
output = "output-5"
final = "final-output"

for i in range(1, 4):
    Q = P.bind({'input':input, 'output':output}) # attaches $input, $output in Pig Latin to
 input, output Python variable
    results = Q.runSingle()

    if results.isSuccessful() == "FAILED":
        raise "Pig job failed"
    iter = results.result("C").iterator()
    if iter.hasNext():
        tuple = iter.next()
        value = tuple.get(0)
        if float(str(value)) < 3:
            print "value: " + str(value)
            input = "studenttab" + str(i+5)
            output = "output-" + str(i+5)
            print "output: " + output
        else:
           Pig.fs("mv " + output + " " + final)
           break
```

1.4.3 Automated Pig Latin Generation

A number of user frameworks do automated generation of Pig Latin.

1.4.3.1 Conditional Compilation

A sub-use case of automated generation is conditional code generation. Different processing might be required based on whether this is weekday or a weekend.

```
str = "A = load 'input';"
if today.isWeekday():
    str = str + "B = filter A by weekday_filter(*);"
else:
    str = str + "B = filter A by weekend_filter(*);"
str = str + "C = group B by user;"
results = Pig.compile(str).bind().runSingle()
```

1.4.3.2 Parallel Execution

Another sub-use case of automated generation is parallel execution of identical pipelines. You may have a single pipeline that you would like to run multiple data sets through in parallel. In the example below, the pipeline is run for the US, the UK, and Brazil.

```
P = Pig.compile("""A = load '$in';
                   B = filter A by user is not null;
                   ...
                   store Z into '$out';
                """)

Q = P.bind([{'in':'us_raw','out':'us_processed'},
        {'in':'uk_raw','out':'uk_processed'},
        {'in':'brazil_raw','out':'brazil_processed'}])

results = Q.run() # it blocks until all pipelines are completed

for i in [0, 1, 2]:
    result = results[i]
    ... # check result for each pipeline
```

1.5 Java Objects

1.5.1 Pig Object

```
public class Pig {
    /**
     * Run a filesystem command.  Any output from this command is written to
     * stdout or stderr as appropriate.
     * @param cmd Filesystem command to run along with its arguments as one
     * string.
     * @throws IOException
     */
    public static void fs(String cmd) throws IOException {...}

    /**
     * Register a jar for use in Pig.  Once this is done this jar will be
     * registered for ALL SUBSEQUENT Pig pipelines in this script.
     * If you wish to register it for only a single Pig pipeline, use
     * register within that definition.
     * @param jarfile Path of jar to include.
     * @throws IOException if the indicated jarfile cannot be found.
     */
    public static void registerJar(String jarfile) throws IOException {...}

    /**
     * Register script UDFs for use in Pig. Once this is done all UDFs
     * defined in the file will be available for ALL SUBSEQUENT
     * Pig pipelines in this script. If you wish to register UDFS for
     * only a single Pig pipeline, use register within that definition.
     * @param udffile Path of the script UDF file
     * @param namespace namespace of the UDFs
     * @throws IOException
     */
    public static void registerUDF(String udffile, String namespace) throws IOException
 {...}

    /**
     * Define an alias for a UDF or a streaming command.  This definition
```

```
     * will then be present for ALL SUBSEQUENT Pig pipelines defined in this
     * script.  If you wish to define it for only a single Pig pipeline, use
     * define within that definition.
     * @param alias name of the defined alias
     * @param definition string this alias is defined as
     */
    public static void define(String alias, String definition) throws IOException {...}

    /**
     * Set a variable for use in Pig Latin.  This set
     * will then be present for ALL SUBSEQUENT Pig pipelines defined in this
     * script.  If you wish to set it for only a single Pig pipeline, use
     * set within that definition.
     * @param var variable to set
     * @param value to set it to
     */
    public static void set(String var, String value) throws IOException {...}

    /**
     * Define a Pig pipeline.
     * @param pl Pig Latin definition of the pipeline.
     * @return Pig object representing this pipeline.
     * @throws IOException if the Pig Latin does not compile.
     */
    public static Pig compile(String pl) throws IOException {...}

    /**
     * Define a named portion of a Pig pipeline.  This allows it
     * to be imported into another pipeline.
     * @param name Name that will be used to define this pipeline.
     * The namespace is global.
     * @param pl Pig Latin definition of the pipeline.
     * @return Pig object representing this pipeline.
     * @throws IOException if the Pig Latin does not compile.
     */
    public static Pig compile(String name, String pl) throws IOException {...}

    /**
     * Define a Pig pipeline based on Pig Latin in a separate file.
     * @param filename File to read Pig Latin from.  This must be a purely
     * Pig Latin file.  It cannot contain host language constructs in it.
     * @return Pig object representing this pipeline.
     * @throws IOException if the Pig Latin does not compile or the file
     * cannot be found.
     */
    public static Pig compileFromFile(String filename) throws IOException {...}

    /**
     * Define a named Pig pipeline based on Pig Latin in a separate file.
     * This allows it to be imported into another pipeline.
     * @param name Name that will be used to define this pipeline.
     * The namespace is global.
     * @param filename File to read Pig Latin from.  This must be a purely
     * Pig Latin file.  It cannot contain host language constructs in it.
     * @return Pig object representing this pipeline.
     * @throws IOException if the Pig Latin does not compile or the file
     * cannot be found.
     */
    public static Pig compileFromFile(String name, String filename) throws IOException
{...}
```

```
    /**
     * Bind this to a set of variables. Values must be provided
     * for all Pig Latin parameters.
     * @param vars map of variables to bind.  Keys should be parameters defined
     * in the Pig Latin.  Values should be strings that provide values for those
     * parameters.  They can be either constants or variables from the host
     * language.  Host language variables must contain strings.
     * @return a {@link BoundScript} object
     * @throws IOException if there is not a key for each
     * Pig Latin parameter or if they contain unsupported types.
     */
    public BoundScript bind(Map<String, String> vars) throws IOException {...}

    /**
     * Bind this to multiple sets of variables.  This will
     * cause the Pig Latin script to be executed in parallel over these sets of
     * variables.
     * @param vars list of maps of variables to bind.  Keys should be parameters defined
     * in the Pig Latin.  Values should be strings that provide values for those
     * variables.  They can be either constants or variables from the host
     * language.  Host language variables must be strings.
     * @return a {@link BoundScript} object
     * @throws IOException  if there is not a key for each
     * Pig Latin parameter or if they contain unsupported types.
     */
    public BoundScript bind(List<Map<String, String>> vars) throws IOException {...}

    /**
     * Bind a Pig object to variables in the host language (optional
     * operation).  This does an implicit mapping of variables in the host
     * language to parameters in Pig Latin.  For example, if the user
     * provides a Pig Latin statement
     * p = Pig.compile("A = load '$input';");
     * and then calls this function it will look for a variable called
     * input in the host language.  Scoping rules of the host
     * language will be followed in selecting which variable to bind.  The
     * variable bound must contain a string value.  This method is optional
     * because not all host languages may support searching for in scope
     * variables.
     * @throws IOException if host language variables are not found to resolve all
     * Pig Latin parameters or if they contain unsupported types.
     */
    public BoundScript bind() throws IOException {...}

}
```

1.5.2 BoundScript Object

```
public class BoundScript {

    /**
     * Run a pipeline on Hadoop.
     * If there are no stores in this pipeline then nothing will be run.
     * @return {@link PigStats}, null if there is no bound query to run.
     * @throws IOException
     */
```

```
    public PigStats runSingle() throws IOException {...}

    /**
     * Run a pipeline on Hadoop.
     * If there are no stores in this pipeline then nothing will be run.
     * @param prop Map of properties that Pig should set when running the script.
     * This is intended for use with scripting languages that do not support
     * the Properties object.
     * @return {@link PigStats}, null if there is no bound query to run.
     * @throws IOException
     */
    public PigStats runSingle(Properties prop) throws IOException {...}

    /**
     * Run a pipeline on Hadoop.
     * If there are no stores in this pipeline then nothing will be run.
     * @param propfile File with properties that Pig should set when running the script.
     * @return {@link PigStats}, null if there is no bound query to run.
     * @throws IOException
     */
    public PigStats runSingle(String propfile) throws IOException {...}

    /**
     * Run multiple instances of bound pipeline on Hadoop in parallel.
     * If there are no stores in this pipeline then nothing will be run.
     * Bind is called first with the list of maps of variables to bind.
     * @return a list of {@link PigStats}, one for each map of variables passed
     * to bind.
     * @throws IOException
     */
    public List<PigStats> run() throws IOException {...}

    /**
     * Run multiple instances of bound pipeline on Hadoop in parallel.
     * @param prop Map of properties that Pig should set when running the script.
     * This is intended for use with scripting languages that do not support
     * the Properties object.
     * @return a list of {@link PigStats}, one for each map of variables passed
     * to bind.
     * @throws IOException
     */
    public List<PigStats>  run(Properties prop) throws IOException {...}

    /**
     * Run multiple instances of bound pipeline on Hadoop in parallel.
     * @param propfile File with properties that Pig should set when running the script.
     * @return a list of PigResults, one for each map of variables passed
     * to bind.
     * @throws IOException
     */
    public List<PigStats>  run(String propfile) throws IOException {...}

    /**
     * Run illustrate for this pipeline.  Results will be printed to stdout.
     * @throws IOException if illustrate fails.
     */
    public void illustrate() throws IOException {...}

    /**
     * Explain this pipeline.  Results will be printed to stdout.
```

```
     * @throws IOException if explain fails.
     */
    public void explain() throws IOException {...}

    /**
     * Describe the schema of an alias in this pipeline.
     * Results will be printed to stdout.
     * @param alias to be described
     * @throws IOException if describe fails.
     */
    public void describe(String alias) throws IOException {...}

}
```

1.5.3 PigStats Object

```
public abstract class PigStats {
    public abstract boolean isEmbedded();

    /**
     * An embedded script contains one or more pipelines.
     * For a named pipeline in the script, the key in the returning map is the name of the
 pipeline.
     * Otherwise, the key in the returning map is the script id of the pipeline.
     */
    public abstract Map<String, List<PigStats>> getAllStats();

    public abstract List<String> getAllErrorMessages();
}
```

1.5.4 PigProgressNotificationListener Object

```
public interface PigProgressNotificationListener extends java.util.EventListener {

    /**
     * Invoked just before launching MR jobs spawned by the script.
     * @param scriptId id of the script
     * @param numJobsToLaunch the total number of MR jobs spawned by the script
     */
    public void launchStartedNotification(String scriptId, int numJobsToLaunch);

    /**
     * Invoked just before submitting a batch of MR jobs.
     * @param scriptId id of the script
     * @param numJobsSubmitted the number of MR jobs in the batch
     */
    public void jobsSubmittedNotification(String scriptId, int numJobsSubmitted);

    /**
     * Invoked after a MR job is started.
     * @param scriptId id of the script
     * @param assignedJobId the MR job id
     */
    public void jobStartedNotification(String scriptId, String assignedJobId);
```

```
    /**
     * Invoked just after a MR job is completed successfully.
     * @param scriptId id of the script
     * @param jobStats the {@link JobStats} object associated with the MR job
     */
    public void jobFinishedNotification(String scriptId, JobStats jobStats);

    /**
     * Invoked when a MR job fails.
     * @param scriptId id of the script
     * @param jobStats the {@link JobStats} object associated with the MR job
     */
    public void jobFailedNotification(String scriptId, JobStats jobStats);

    /**
     * Invoked just after an output is successfully written.
     * @param scriptId id of the script
     * @param outputStats the {@link OutputStats} object associated with the output
     */
    public void outputCompletedNotification(String scriptId, OutputStats outputStats);

    /**
     * Invoked to update the execution progress.
     * @param scriptId id of the script
     * @param progress the percentage of the execution progress
     */
    public void progressUpdatedNotification(String scriptId, int progress);

    /**
     * Invoked just after all MR jobs spawned by the script are completed.
     * @param scriptId id of the script
     * @param numJobsSucceeded the total number of MR jobs succeeded
     */
    public void launchCompletedNotification(String scriptId, int numJobsSucceeded);
}
```

2 Embedded Pig - Java

To enable control flow, you can embed Pig Latin statements and Pig commands in the Java programming language.

Note that host languages and the languages of UDFs (included as part of the embedded Pig) are completely orthogonal. For example, a Pig Latin statement that registers a Java UDF may be embedded in Python, JavaScript, Groovy, or Java. The exception to this rule is "combined" scripts – here the languages must match (see the Advanced Topics for Python, Advanced Topics for JavaScript and Advanced Topics for Groovy).

2.1 PigServer Interface

Currently, PigServer is the main interface for embedding Pig in Java. PigServer can now be instantiated from multiple threads. (In the past, PigServer contained references to static data that prevented multiple instances of the object to be created from different threads within

your application.) Please note that PigServer is NOT thread safe; the same object can't be shared across multiple threads.

2.2 Usage Examples

Local Mode

From your current working directory, compile the program. (Note that idlocal.class is written to your current working directory. Include "." in the class path when you run the program.)

```
$ javac -cp pig.jar idlocal.java
```

From your current working directory, run the program. To view the results, check the output file, id.out.

```
Unix:   $ java -cp pig.jar:. idlocal
Cygwin: $ java –cp '.;pig.jar' idlocal
```

idlocal.java - The sample code is based on Pig Latin statements that extract all user IDs from the /etc/passwd file. Copy the /etc/passwd file to your local working directory.

```
import java.io.IOException;
import org.apache.pig.PigServer;
public class idlocal{
    public static void main(String[] args) {
        try {
            PigServer pigServer = new PigServer("local");
            runIdQuery(pigServer, "passwd");
        }
        catch(Exception e) {
        }
    }
    public static void runIdQuery(PigServer pigServer, String inputFile) throws IOException
 {
        pigServer.registerQuery("A = load '" + inputFile + "' using PigStorage(':');");
        pigServer.registerQuery("B = foreach A generate $0 as id;");
        pigServer.store("B", "id.out");
    }
}
```

Mapreduce Mode

Point $HADOOPDIR to the directory that contains the hadoop-site.xml file. Example:

```
$ export HADOOPDIR=/yourHADOOPsite/conf
```

From your current working directory, compile the program. (Note that idmapreduce.class is written to your current working directory. Include "." in the class path when you run the program.)

```
$ javac -cp pig.jar idmapreduce.java
```

From your current working directory, run the program. To view the results, check the idout directory on your Hadoop system.

```
Unix:   $ java -cp pig.jar:.:$HADOOPDIR idmapreduce
Cygwin: $ java –cp '.;pig.jar;$HADOOPDIR' idmapreduce
```

idmapreduce.java - The sample code is based on Pig Latin statements that extract all user IDs from the /etc/passwd file. Copy the /etc/passwd file to your home directory on the HDFS.

```
import java.io.IOException;
import org.apache.pig.PigServer;
public class idmapreduce{
    public static void main(String[] args) {
        try {
            PigServer pigServer = new PigServer("mapreduce");
            runIdQuery(pigServer, "passwd");
        }
        catch(Exception e) {
        }
    }
    public static void runIdQuery(PigServer pigServer, String inputFile) throws IOException
 {
        pigServer.registerQuery("A = load '" + inputFile + "' using PigStorage(':');")
        pigServer.registerQuery("B = foreach A generate $0 as id;");
        pigServer.store("B", "idout");
    }
}
```

3 Pig Macros

Pig Latin supports the definition, expansion, and import of macros.

3.1 DEFINE (macros)

Defines a Pig macro.

3.1.1 Syntax

Define Macro

```
DEFINE macro_name (param [, param ...]) RETURNS {void | alias [, alias ...]} { pig_latin_fragment };
```

Expand Macro

```
alias [, alias ...] = macro_name (param [, param ...]) ;
```

3.1.2 Terms

macro_name	The name of the macro. Macro names are global.
param	(optional) A comma-separated list of one or more parameters, including IN aliases (Pig relations), enclosed in parentheses, that are referenced in the Pig Latin fragment. Unlike user defined functions (UDFs), which only allow quoted strings as its parameters, Pig macros support four types of parameters: • alias (IDENTIFIER) • integer • float • string literal (quoted string) Note that type is NOT part of parameter definition. It is your responsibility to document the types of the parameters in a macro.
void	If the macro has no return alias, then void must be specified.
alias	(optional) A comma-separated list of one or more return aliases (Pig relations) that are referenced in the Pig Latin fragment. The alias must exist in the macro in the form $<alias>. If the macro has no return alias, then void must be specified.
pig_latin_fragment	One or more Pig Latin statements, enclosed in curly brackets.

3.1.3 Usage

Macro Definition

A macro definition can appear anywhere in a Pig script as long as it appears prior to the first use. A macro definition can include references to other macros as long as the referenced macros are defined prior to the macro definition. Recursive references are not allowed.

Note the following restrictions:

- Macros are not allowed inside a FOREACH nested block.

- Macros cannot contain Grunt shell commands.
- Macros cannot include a user-defined schema that has a name collision with an alias in the macro.

In this example the macro is named my_macro. Note that only aliases A and C are visible from the outside; alias B is not visible from the outside.

```
 DEFINE my_macro(A, sortkey) RETURNS C {
    B = FILTER $A BY my_filter(*);
    $C = ORDER B BY $sortkey;
}
```

Macro Expansion

A macro can be expanded inline using the macro expansion syntax. Note the following:

- Any alias in the macro which isn't visible from the outside will be prefixed with a macro name and suffixed with an instance id to avoid namespace collision.
- Macro expansion is not a complete replacement for function calls. Recursive expansions are not supported.

In this example my_macro (defined above) is expanded. Because alias B is not visible from the outside it is renamed macro_my_macro_B_0.

```
/* These statements ... */

X = LOAD 'users' AS (user, address, phone);
Y = my_macro(X, user);
STORE Y into 'bar';

/* Are expanded into these statements ... */

X = LOAD 'users' AS (user, address, phone);
macro_my_macro_B_0 = FILTER X BY my_filter(*);
Y = ORDER macro_my_macro_B_0  BY user;
STORE Y INTO 'output';
```

Macro Import

A macro can be imported from another Pig script (see IMPORT (macros)). Splitting your macros from your main Pig script is useful for making reusable code.

3.1.4 Examples

In this example no parameters are passed to the macro.

```
DEFINE my_macro() returns B {
   D = LOAD 'data' AS (a0:int, a1:int, a2:int);
   $B = FILTER D BY ($1 == 8) OR (NOT ($0+$2 > $1));
```

```
};

X = my_macro();
STORE X INTO 'output';
```

In this example parameters are passed and returned.

```
DEFINE group_and_count (A, group_key, reducers) RETURNS B {
   D = GROUP $A BY $group_key PARALLEL $reducers;
   $B = FOREACH D GENERATE group, COUNT($A);
};

X = LOAD 'users' AS (user, age, zip);
Y = group_and_count (X, user, 20);
Z = group_and_count (X, age, 30);
STORE Y into 'byuser';
STORE Z into 'byage';
```

In this example the macro does not have a return alias; thus, void must be specified.

```
DEFINE my_macro(A, sortkey) RETURNS void {
      B = FILTER $A BY my_filter(*);
      C = ORDER B BY $sortkey;
      STORE C INTO 'my_output';
};

/* To expand this macro, use the following */

my_macro(alpha, 'user');
```

In this example a name collision will occur. Here letter B is used as alias name and as name in user-defined schema. Pig will throw an exception when name collision is detected.

```
DEFINE my_macro(A, sortkey) RETURNS E {
      B = FILTER $A BY my_filter(*);
      C = ORDER B BY $sortkey;
      D = LOAD 'in' as (B:bag{});
   $E = FOREACH D GENERATE COUNT(B);
   };
```

This example demonstrates the importance of knowing parameter types before using them in a macro script. Notice that when pass parameter $outfile to my_macro1 inside my_macro2, it must be quoted.

```
-- A: an alias
-- outfile: output file path (quoted string)
DEFINE my_macro1(A, outfile) RETURNS void {
       STORE $A INTO '$outfile';
   };

-- A: an alias
-- sortkey: column name (quoted string)
```

```
-- outfile: output file path (quoted string)
DEFINE my_macro2(A, sortkey, outfile) RETURNS void {
      B = FILTER $A BY my_filter(*);
      C = ORDER B BY $sortkey;
      my_macro1(C, '$outfile');
   };

   alpha = Load 'input' as (user, age, gpa);
   my_macro2(alpha, 'age', 'order_by_age.txt');
```

In this example a macro (group_with_parallel) refers to another macro (foreach_count).

```
DEFINE foreach_count(A, C) RETURNS B {
   $B = FOREACH $A GENERATE group, COUNT($C);
};

DEFINE group_with_parallel (A, group_key, reducers) RETURNS B {
   C = GROUP $A BY $group_key PARALLEL $reducers;
   $B = foreach_count(C, $A);
};

/* These statements ... */

X = LOAD 'users' AS (user, age, zip);
Y = group_with_parallel (X, user, 23);
STORE Y INTO 'byuser';

/* Are expanded into these statements ... */

X = LOAD 'users' AS (user, age, zip);
macro_group_with_parallel_C_0 = GROUP X by (user) PARALLEL 23;
Y = FOREACH macro_group_with_parallel_C_0 GENERATE group, COUNT(X);
STORE Y INTO 'byuser';
```

3.2 IMPORT (macros)

Import macros defined in a separate file.

3.2.1 Syntax

```
IMPORT 'file-with-macro';
```

3.2.2 Terms

file-with-macro	The name of a file (enclosed in single quotes) that contains one or more macro definitions; for example, 'my_macro.pig' or 'mypath/my_macro.pig'. Macro names are global and all macros share the same name space. While the file can contain more than one macro definition, having two macros with

	the same name in your execution context will result in an error. Files are imported based on either (1) the given file path or (2) the import path specified via the Pig property pig.import.search.path. If a file path is given, whether absolute or relative to the current directory (starting with . or ..), the import path will be ignored.

3.2.3 Usage

Use the IMPORT command to import a macro defined in a separate file into your Pig script.

IMPORT adds the macro definitions to the Pig Latin namespace; these macros can then be invoked as if they were defined in the same file.

Macros can only contain Pig Latin statements; Grunt shell commands are not supported. REGISTER statements and parameter definitions with %default or %declare are both valid however. Your macro file also IMPORT other macro files, so long as these imports are not recursive.

See also: DEFINE (macros)

3.2.4 Example

In this example, because a path is not given, Pig will use the import path specified in `pig.import.search.path`.

```
/* myscript.pig */
...
...
IMPORT 'my_macro.pig';
...
...
```

4 Parameter Substitution

4.1 Description

Substitute values for parameters at run time.

4.1.1 Syntax: Specifying Parameters Using the Pig Command Line

```
pig {-param param_name = param_value | -param_file file_name} [-debug | -dryrun] script
```

4.1.2 Syntax: Specifying Parameters Using Preprocessor Statements in a Pig Script

```
{%declare | %default} param_name param_value
```

4.1.3 Terms

pig	Keyword Note: exec, run, and explain also support parameter substitution.
-param	Flag. Use this option when the parameter is included in the command line. Multiple parameters can be specified. If the same parameter is specified multiple times, the last value will be used and a warning will be generated. Command line parameters and parameter files can be combined with command line parameters taking precedence.
param_name	The name of the parameter. The parameter name has the structure of a standard language identifier: it must start with a letter or underscore followed by any number of letters, digits, and underscores. Parameter names are case insensitive. If you pass a parameter to a script that the script does not use, this parameter is silently ignored. If the script has a parameter and no value is supplied or substituted, an error will result.
param_value	The value of the parameter. A parameter value can take two forms: • A sequence of characters enclosed in single or double quotes. In this case the unquoted version of the value is used during substitution. Quotes within the value can be escaped with the backslash character (\ ). Single word values that don't use special characters such as % or = don't have to be quoted. • A command enclosed in back ticks. The value of a parameter, in either form, can be expressed in terms of other parameters as long as the values of the dependent parameters are already defined.

	There are no hard limits on the size except that parameters need to fit into memory.
-param_file	Flag. Use this option when the parameter is included in a file. Multiple files can be specified. If the same parameter is present multiple times in the file, the last value will be used and a warning will be generated. If a parameter present in multiple files, the value from the last file will be used and a warning will be generated. Command line parameters and parameter files can be combined with command line parameters taking precedence.
file_name	The name of a file containing one or more parameters. A parameter file will contain one line per parameter. Empty lines are allowed. Perl-style (#) comment lines are also allowed. Comments must take a full line and # must be the first character on the line. Each parameter line will be of the form: param_name = param_value. White spaces around = are allowed but are optional.
-debug	Flag. With this option, the script is run and a fully substituted Pig script is produced in the current working directory named original_script_name.substituted
-dryrun	Flag. With this option, the script is not run and a fully substituted Pig script is produced in the current working directory named original_script_name.substituted
script	A pig script. The pig script must be the last element in the Pig command line. • If parameters are specified in the Pig command line or in a parameter file, the script should include a $param_name for each para_name included in the command line or parameter file. • If parameters are specified using the preprocessor statements, the script should include either %declare or %default. • In the script, parameter names can be escaped with the backslash character (\ ) in which case substitution does not take place.
%declare	Preprocessor statement included in a Pig script.

	Use to describe one parameter in terms of other parameters. The declare statement is processed prior to running the Pig script. The scope of a parameter value defined using declare is all the lines following the declare statement until the next declare statement that defines the same parameter is encountered.
%default	Preprocessor statement included in a Pig script. Use to provide a default value for a parameter. The default value has the lowest priority and is used if a parameter value has not been defined by other means. The default statement is processed prior to running the Pig script. The scope is the same as for %declare.

4.2 Usage

Parameter substitution enables you to write Pig scripts that include parameters and to supply values for these parameters at run time. For instance, suppose you have a job that needs to run every day using the current day's data. You can create a Pig script that includes a parameter for the date. Then, when you run this script you can specify or supply a value for the date parameter using one of the supported methods.

4.2.1 Specifying Parameters

You can specify parameter names and parameter values as follows:

- As part of a command line.
- In parameter file, as part of a command line.
- With the declare statement, as part of Pig script.
- With default statement, as part of a Pig script.

Parameter substitution may be used inside of macros, but it is the responsibility of the user to ensure that there are no conflicts between names of parameters defined at the top level and names of arguments or return values for a macro. A simple way to ensure this is to use ALL_CAPS for top-level parameters and lower_case for macro-level parameters. See DEFINE (macros).

4.2.2 Precedence

Precedence for parameters is as follows, from highest to lowest:

1. Parameters defined using the declare statement

2. Parameters defined in the command line using -param
3. Parameters defined in parameter files specified by -param_file
4. Parameters defined using the default statement

4.2.3 Processing Order and Precedence

Parameters are processed as follows:

- Command line parameters are scanned in the order they are specified on the command line.
- Parameter files are scanned in the order they are specified on the command line. Within each file, the parameters are processed in the order they are listed.
- Declare and default preprocessors statements are processed in the order they appear in the Pig script.

4.3 Examples

4.3.1 Specifying Parameters in the Command Line

Suppose we have a data file called 'mydata' and a pig script called 'myscript.pig'.

mydata

```
1       2       3
4       2       1
8       3       4
```

myscript.pig

```
A = LOAD '$data' USING PigStorage() AS (f1:int, f2:int, f3:int);
DUMP A;
```

In this example the parameter (data) and the parameter value (mydata) are specified in the command line. If the parameter name in the command line (data) and the parameter name in the script ($data) do not match, the script will not run. If the value for the parameter (mydata) is not found, an error is generated.

```
$ pig -param data=mydata myscript.pig

(1,2,3)
(4,2,1)
(8,3,4)
```

4.3.2 Specifying parameters Using a Parameter File

Suppose we have a parameter file called 'myparams.'

```
# my parameters
data1 = mydata1
cmd = `generate_name`
```

In this example the parameters and values are passed to the script using the parameter file.

```
$ pig -param_file myparams script2.pig
```

4.3.3 Specifying Parameters Using the Declare Statement

In this example the command is executed and its stdout is used as the parameter value.

```
%declare CMD `generate_date`;
A = LOAD '/data/mydata/$CMD';
B = FILTER A BY $0>'5';

etc ...
```

4.3.4 Specifying Parameters Using the Default Statement

In this example the parameter (DATE) and value ('20090101') are specified in the Pig script using the default statement. If a value for DATE is not specified elsewhere, the default value 20090101 is used.

```
%default DATE '20090101';
A = load '/data/mydata/$DATE';

etc ...
```

4.3.5 Specifying Parameter Values as a sequence of Characters

In this example the characters (in this case, Joe's URL) can be enclosed in single or double quotes, and quotes within the sequence of characters can be escaped.

```
%declare DES 'Joe\'s URL';
A = LOAD 'data' AS (name, description, url);
B = FILTER A BY description == '$DES';

etc ...
```

In this example single word values that don't use special characters (in this case, mydata) don't have to be enclosed in quotes.

```
$ pig -param data=mydata myscript.pig
```

4.3.6 Specifying Parameter Values as a Command

In this example the command is enclosed in back ticks. First, the parameters mycmd and date are substituted when the declare statement is encountered. Then the resulting command is executed and its stdout is placed in the path before the load statement is run.

```
%declare CMD `$mycmd $date`;
A = LOAD '/data/mydata/$CMD';
B = FILTER A BY $0>'5';

etc ...
```