User Defined Functions

Table of contents

1 Introduction..... 2
2 Writing Java UDFs..... 2
3 Writing Jython UDFs..... 35
4 Writing JavaScript UDFs..... 38
5 Writing Ruby UDFs..... 40
6 Writing Groovy UDFs..... 43
7 Writing Python UDFs..... 47
8 Piggy Bank..... 48

Copyright © 2007 The Apache Software Foundation. All rights reserved.

1 Introduction

Pig provides extensive support for user defined functions (UDFs) as a way to specify custom processing. Pig UDFs can currently be implemented in six languages: Java, Jython, Python, JavaScript, Ruby and Groovy.

The most extensive support is provided for Java functions. You can customize all parts of the processing including data load/store, column transformation, and aggregation. Java functions are also more efficient because they are implemented in the same language as Pig and because additional interfaces are supported such as the Algebraic Interface and the Accumulator Interface.

Limited support is provided for Jython, Python, JavaScript, Ruby and Groovy functions. These functions are new, still evolving, additions to the system. Currently only the basic interface is supported; load/store functions are not supported. Furthermore, JavaScript, Ruby and Groovy are provided as experimental features because they did not go through the same amount of testing as Java or Jython. At runtime note that Pig will automatically detect the usage of a scripting UDF in the Pig script and will automatically ship the corresponding scripting jar, either Jython, Rhino, JRuby or Groovy-all, to the backend. Python does not require any runtime engine since it invoke python command line and stream data in and out of it.

Pig also provides support for Piggy Bank, a repository for JAVA UDFs. Through Piggy Bank you can access Java UDFs written by other users and contribute Java UDFs that you have written.

2 Writing Java UDFs

2.1 Eval Functions

2.1.1 How to Use a Simple Eval Function

Eval is the most common type of function. It can be used in `FOREACH` statements as shown in this script:

```
-- myscript.pig
REGISTER myudfs.jar;
A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

The command below can be used to run the script. Note that all examples in this document run in local mode for simplicity but the examples can also run in Tez local/Mapreduce/ Tez mode. For more information on how to run Pig, please see the PigTutorial.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
pig -x local myscript.pig
```

The first line of the script provides the location of the `jar file` that contains the UDF. (Note that there are no quotes around the jar file. Having quotes would result in a syntax error.) To locate the jar file, Pig first checks the `classpath`. If the jar file can't be found in the classpath, Pig assumes that the location is either an absolute path or a path relative to the location from which Pig was invoked. If the jar file can't be found, an error will be printed: `java.io.IOException: Can't read jar file: myudfs.jar`.

Multiple `register` commands can be used in the same script. If the same fully-qualified function is present in multiple jars, the first occurrence will be used consistently with Java semantics.

The name of the UDF has to be fully qualified with the package name or an error will be reported: `java.io.IOException: Cannot instantiate:UPPER`. Also, the function name is case sensitive (UPPER and upper are not the same). A UDF can take one or more parameters. The exact signature of the function should be clear from its documentation.

The function provided in this example takes an ASCII string and produces its uppercase version. If you are familiar with column transformation functions in SQL, you will recognize that UPPER fits this concept. However, as we will see later in the document, eval functions in Pig go beyond column transformation functions and include aggregate and filter functions.

If you are just a user of UDFs, this is most of what you need to know about UDFs to use them in your code.

2.1.2 How to Write a Simple Eval Function

Let's now look at the implementation of the `UPPER` UDF.

```
1  package myudfs;
2  import java.io.IOException;
3  import org.apache.pig.EvalFunc;
4  import org.apache.pig.data.Tuple;
5
6  public class UPPER extends EvalFunc<String>
7  {
8    public String exec(Tuple input) throws IOException {
9        if (input == null || input.size() == 0 || input.get(0) == null)
10           return null;
11       try{
12           String str = (String)input.get(0);
13          return str.toUpperCase();
14       }catch(Exception e){
15           throw new IOException("Caught exception processing input row ", e);
16       }
17   }
18 }
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

Line 1 indicates that the function is part of the `myudfs` package. The UDF class extends the `EvalFunc` class which is the base class for all eval functions. It is parameterized with the return type of the UDF which is a Java `String` in this case. We will look into the `EvalFunc` class in more detail later, but for now all we need to do is to implement the `exec` function. This function is invoked on every input tuple. The input into the function is a tuple with input parameters in the order they are passed to the function in the Pig script. In our example, it will contain a single string field corresponding to the student name.

The first thing to decide is what to do with invalid data. This depends on the format of the data. If the data is of type `bytearray` it means that it has not yet been converted to its proper type. In this case, if the format of the data does not match the expected type, a NULL value should be returned. If, on the other hand, the input data is of another type, this means that the conversion has already happened and the data should be in the correct format. This is the case with our example and that's why it throws an error (line 15.)

Also, note that lines 9-10 check if the input data is null or empty and if so returns null.

The actual function implementation is on lines 12-13 and is self-explanatory.

Now that we have the function implemented, it needs to be compiled and included in a jar. You will need to build `pig.jar` to compile your UDF. You can use the following set of commands to checkout the code from SVN repository and create pig.jar:

```
svn co http://svn.apache.org/repos/asf/pig/trunk
cd trunk
ant
```

You should see `pig.jar` in your current working directory. The set of commands below first compiles the function and then creates a jar file that contains it.

```
cd myudfs
javac -cp pig.jar UPPER.java
cd ..
jar -cf myudfs.jar myudfs
```

You should now see `myudfs.jar` in your current working directory. You can use this jar with the script described in the previous section.

2.1.3 Aggregate Functions

Aggregate functions are another common type of eval function. Aggregate functions are usually applied to grouped data, as shown in this script:

```
-- myscript2.pig
A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
B = GROUP A BY name;
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
C = FOREACH B GENERATE group, COUNT(A);
DUMP C;
```

The script above uses the `COUNT` function to count the number of students with the same name. There are a couple of things to note about this script. First, even though we are using a function, there is no `register` command. Second, the function is not qualified with the package name. The reason for both is that `COUNT` is a `builtin` function meaning that it comes with the Pig distribution. These are the only two differences between builtins and UDFs. Builtins are discussed in more detail later in this document.

2.1.4 Algebraic Interface

An aggregate function is an eval function that takes a bag and returns a scalar value. One interesting and useful property of many aggregate functions is that they can be computed incrementally in a distributed fashion. We call these functions `algebraic`. `COUNT` is an example of an algebraic function because we can count the number of elements in a subset of the data and then sum the counts to produce a final output. In the Hadoop world, this means that the partial computations can be done by the map and combiner, and the final result can be computed by the reducer.

It is very important for performance to make sure that aggregate functions that are algebraic are implemented as such. Let's look at the implementation of the COUNT function to see what this means. (Error handling and some other code is omitted to save space. The full code can be accessed here.)

```
public class COUNT extends EvalFunc<Long> implements Algebraic{
    public Long exec(Tuple input) throws IOException {return count(input);}
    public String getInitial() {return Initial.class.getName();}
    public String getIntermed() {return Intermed.class.getName();}
    public String getFinal() {return Final.class.getName();}
    static public class Initial extends EvalFunc<Tuple> {
        public Tuple exec(Tuple input) throws IOException {return
 TupleFactory.getInstance().newTuple(count(input));}
    }
    static public class Intermed extends EvalFunc<Tuple> {
        public Tuple exec(Tuple input) throws IOException {return
 TupleFactory.getInstance().newTuple(sum(input));}
    }
    static public class Final extends EvalFunc<Long> {
        public Tuple exec(Tuple input) throws IOException {return sum(input);}
    }
    static protected Long count(Tuple input) throws ExecException {
        Object values = input.get(0);
        if (values instanceof DataBag) return ((DataBag)values).size();
        else if (values instanceof Map) return new Long(((Map)values).size());
    }
    static protected Long sum(Tuple input) throws ExecException, NumberFormatException {
        DataBag values = (DataBag)input.get(0);
        long sum = 0;
        for (Iterator (Tuple) it = values.iterator(); it.hasNext();) {
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
            Tuple t = it.next();
            sum += (Long)t.get(0);
        }
        return sum;
    }
}
```

COUNT implements Algebraic interface which looks like this:

```
public interface Algebraic{
    public String getInitial();
    public String getIntermed();
    public String getFinal();
}
```

For a function to be algebraic, it needs to implement Algebraic interface that consist of definition of three classes derived from EvalFunc. The contract is that the exec function of the Initial class is called once and is passed the original input tuple. Its output is a tuple that contains partial results. The exec function of the Intermed class can be called zero or more times and takes as its input a tuple that contains partial results produced by the Initial class or by prior invocations of the Intermed class and produces a tuple with another partial result. Finally, the exec function of the Final class is called and produces the final result as a scalar type.

Here's the way to think about this in the Hadoop world. The exec function of the Initial class is invoked once for each input tuple by the map process and produces partial results. The exec function of the Intermed class is invoked once by each combiner invocation (which can happen zero or more times) and also produces partial results. The exec function of the Final class is invoked once by the reducer and produces the final result.

Take a look at the COUNT implementation to see how this is done. Note that the exec function of the Initial and Intermed classes is parameterized with Tuple and the exec of the Final class is parameterized with the real type of the function, which in the case of the COUNT is Long. Also, note that the fully-qualified name of the class needs to be returned from getInitial, getIntermed, and getFinal methods.

2.1.5 Accumulator Interface

In Pig, problems with memory usage can occur when data, which results from a group or cogroup operation, needs to be placed in a bag and passed in its entirety to a UDF.

This problem is partially addressed by Algebraic UDFs that use the combiner and can deal with data being passed to them incrementally during different processing phases (map, combiner, and reduce). However, there are a number of UDFs that are not Algebraic, don't use the combiner, but still don’t need to be given all data at once.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

The new Accumulator interface is designed to decrease memory usage by targeting such UDFs. For the functions that implement this interface, Pig guarantees that the data for the same key is passed continuously but in small increments. To work with incremental data, here is the interface a UDF needs to implement:

```
public interface Accumulator <T> {
   /**
    * Process tuples. Each DataBag may contain 0 to many tuples for current key
    */
    public void accumulate(Tuple b) throws IOException;
    /**
     * Called when all tuples from current key have been passed to the accumulator.
     * @return the value for the UDF for this key.
     */
    public T getValue();
    /**
     * Called after getValue() to prepare processing for next key.
     */
    public void cleanup();
}
```

There are several things to note here:

1. Each UDF must extend the EvalFunc class and implement all necessary functions there.
2. If a function is algebraic but can be used in a FOREACH statement with accumulator functions, it needs to implement the Accumulator interface in addition to the Algebraic interface.
3. The interface is parameterized with the return type of the function.
4. The accumulate function is guaranteed to be called one or more times, passing one or more tuples in a bag, to the UDF. (Note that the tuple that is passed to the accumulator has the same content as the one passed to exec – all the parameters passed to the UDF – one of which should be a bag.)
5. The getValue function is called after all the tuples for a particular key have been processed to retrieve the final value.
6. The cleanup function is called after getValue but before the next value is processed.

Here us a code snippet of the integer version of the MAX function that implements the interface:

```
public class IntMax extends EvalFunc<Integer> implements Algebraic, Accumulator<Integer> {
    …….
    /* Accumulator interface */

    private Integer intermediateMax = null;

    @Override
    public void accumulate(Tuple b) throws IOException {
        try {
            Integer curMax = max(b);
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
            if (curMax == null) {
                return;
            }
            /* if bag is not null, initialize intermediateMax to negative infinity */
            if (intermediateMax == null) {
                intermediateMax = Integer.MIN_VALUE;
            }
            intermediateMax = java.lang.Math.max(intermediateMax, curMax);
        } catch (ExecException ee) {
            throw ee;
        } catch (Exception e) {
            int errCode = 2106;
            String msg = "Error while computing max in " + this.getClass().getSimpleName();
            throw new ExecException(msg, errCode, PigException.BUG, e);
        }
    }

    @Override
    public void cleanup() {
        intermediateMax = null;
    }

    @Override
    public Integer getValue() {
        return intermediateMax;
    }
}
```

2.1.6 Filter Functions

Filter functions are eval functions that return a `boolean` value. Filter functions can be used anywhere a Boolean expression is appropriate, including the `FILTER` operator or `bincond` expression.

The example below uses the `IsEmpy` builtin filter function to implement joins.

```
-- inner join
A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
B = LOAD 'voter_data' AS (name: chararray, age: int, registration: chararay, contributions:
 float);
C = COGROUP A BY name, B BY name;
D = FILTER C BY not IsEmpty(A);
E = FILTER D BY not IsEmpty(B);
F = FOREACH E GENERATE flatten(A), flatten(B);
DUMP F;
```

Note that, even if filtering is omitted, the same results will be produced because the `foreach` results is a cross product and cross products get rid of empty bags. However, doing up-front filtering is more efficient since it reduces the input of the cross product.

```
-- full outer join
A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
B = LOAD 'voter_data' AS (name: chararray, age: int, registration: chararay, contributions:
 float);
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
C = COGROUP A BY name, B BY name;
D = FOREACH C GENERATE group, flatten((IsEmpty(A) ? null : A)), flatten((IsEmpty(B) ?
 null : B));
dump D;
```

The implementation of the `IsEmpty` function looks like this:

```
import java.io.IOException;
import java.util.Map;

import org.apache.pig.FilterFunc;
import org.apache.pig.PigException;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.DataType;

/**
 * Determine whether a bag or map is empty.
 */
public class IsEmpty extends FilterFunc {

    @Override
    public Boolean exec(Tuple input) throws IOException {
        try {
            Object values = input.get(0);
            if (values instanceof DataBag)
                return ((DataBag)values).size() == 0;
            else if (values instanceof Map)
                return ((Map)values).size() == 0;
            else {
                int errCode = 2102;
                String msg = "Cannot test a " +
                DataType.findTypeName(values) + " for emptiness.";
                throw new ExecException(msg, errCode, PigException.BUG);
            }
        } catch (ExecException ee) {
            throw ee;
        }
    }
}
```

2.1.7 Implement UDF by Simulation

When implementing more advanced types of EvalFuncs, the simpler implementations can be automatically provided by Pig. Thus if your UDF implements Algebraic then you will get the Accumulator interface and basic the basic EvalFunc exec method for free. Similarly, if your UDF implements Accumulator Interface you will get the basic EvalFunc exec method for free. You will not get the Algebraic implemenation. Note that these free implementations are based on simulation, which might not be the most efficient. If you wish to ensure the efficiency of your Accumulator of EvalFunc exec method, you may still implement them yourself and your implementations will be used.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

2.1.8 Pig Types and Native Java Types

The main thing to know about Pig's type system is that Pig uses native Java types for almost all of its types, as shown in this table.

Pig Type	Java Class
bytearray	DataByteArray
chararray	String
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
datetime	DateTime
bigdecimal	BigDecimal
biginteger	BigInteger
tuple	Tuple
bag	DataBag
map	Map<Object, Object>

All Pig-specific classes are available here.

`Tuple` and `DataBag` are different in that they are not concrete classes but rather interfaces. This enables users to extend Pig with their own versions of tuples and bags. As a result, UDFs cannot directly instantiate bags or tuples; they need to go through factory classes: `TupleFactory` and `BagFactory`.

The builtin `TOKENIZE` function shows how bags and tuples are created. A function takes a text string as input and returns a bag of words from the text. (Note that currently Pig bags always contain tuples.)

```
package org.apache.pig.builtin;

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.BagFactory;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.Tuple;
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
import org.apache.pig.data.TupleFactory;

public class TOKENIZE extends EvalFunc<DataBag> {
    TupleFactory mTupleFactory = TupleFactory.getInstance();
    BagFactory mBagFactory = BagFactory.getInstance();

    public DataBag exec(Tuple input) throws IOException
        try {
            DataBag output = mBagFactory.newDefaultBag();
            Object o = input.get(0);
            if (!(o instanceof String)) {
                throw new IOException("Expected input to be chararray, but  got " +
 o.getClass().getName());
            }
            StringTokenizer tok = new StringTokenizer((String)o, " \",()*", false);
            while (tok.hasMoreTokens())
 output.add(mTupleFactory.newTuple(tok.nextToken()));
            return output;
        } catch (ExecException ee) {
            // error handling goes here
        }
    }
}
```

2.1.9 Schemas and Java UDFs

Pig uses type information for validation and performance. It is important for UDFs to participate in type propagation. Our UDFs generally make no effort to communicate their output schema to Pig. This is because Pig can usually figure out this information by using Java's Reflection. If your UDF returns a scalar or a map, no work is required. However, if your UDF returns a tuple or a bag (of tuples), it needs to help Pig figure out the structure of the tuple.

If a UDF returns a tuple or a bag and schema information is not provided, Pig assumes that the tuple contains a single field of type bytearray. If this is not the case, then not specifying the schema can cause failures. We look at this next.

Let's assume that we have UDF `Swap` that, given a tuple with two fields, swaps their order. Let's assume that the UDF does not specify a schema and look at the scripts below:

```
register myudfs.jar;
A = load 'student_data' as (name: chararray, age: int, gpa: float);
B = foreach A generate flatten(myudfs.Swap(name, age)), gpa;
C = foreach B generate $2;
D = limit B 20;
dump D;
```

This script will result in the following error cause by line 4 (`C = foreach B generate $2;`).

```
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
java.io.IOException: Out of bound access. Trying to access non-existent column: 2. Schema
 {bytearray,gpa: float} has 2 column(s).
```

This is because Pig is only aware of two columns in B while line 4 is requesting the third column of the tuple. (Column indexing in Pig starts with 0.)

The function, including the schema, looks like this:

```
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.TupleFactory;
import org.apache.pig.impl.logicalLayer.schema.Schema;
import org.apache.pig.data.DataType;

public class Swap extends EvalFunc<Tuple> {
    public Tuple exec(Tuple input) throws IOException {
        if (input == null || input.size()   2
            return null;
        try{
            Tuple output = TupleFactory.getInstance().newTuple(2);
            output.set(0, input.get(1));
            output.set(1, input.get(0));
            return output;
        } catch(Exception e){
            System.err.println("Failed to process input; error - " + e.getMessage());
            return null;
        }
    }
    public Schema outputSchema(Schema input) {
        try{
            Schema tupleSchema = new Schema();
            tupleSchema.add(input.getField(1));
            tupleSchema.add(input.getField(0));
            return new Schema(new
 Schema.FieldSchema(getSchemaName(this.getClass().getName().toLowerCase(),
 input),tupleSchema, DataType.TUPLE));
        }catch (Exception e){
                return null;
        }
    }
}
```

The function creates a schema with a single field (of type `FieldSchema) of type tuple`. The name of the field is constructed using the `getSchemaName` function of the `EvalFunc` class. The name consists of the name of the UDF function, the first parameter passed to it, and a sequence number to guarantee uniqueness. In the previous script, if you replace `dump D;` with `describe B;` , you will see the following output:

```
B: {myudfs.swap_age_3::age: int,myudfs.swap_age_3::name: chararray,gpa: float}
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

The second parameter to the `FieldSchema` constructor is the schema representing this field, which in this case is a tuple with two fields. The third parameter represents the type of the schema, which in this case is a `TUPLE`. All supported schema types are defined in the `org.apache.pig.data.DataType` class.

```
public class DataType {
    public static final byte UNKNOWN   =   0;
    public static final byte NULL      =   1;
    public static final byte BOOLEAN   =   5; // internal use only
    public static final byte BYTE      =   6; // internal use only
    public static final byte INTEGER   =  10;
    public static final byte LONG      =  15;
    public static final byte FLOAT     =  20;
    public static final byte DOUBLE    =  25;
    public static final byte DATETIME  =  30;
    public static final byte BYTEARRAY =  50;
    public static final byte CHARARRAY =  55;
    public static final byte BIGINTEGER =  65;
    public static final byte BIGDECIMAL =  70;
    public static final byte MAP       = 100;
    public static final byte TUPLE     = 110;
    public static final byte BAG       = 120;
    public static final byte ERROR     =  -1;
    // more code here
}
```

You need to import the `org.apache.pig.data.DataType` class into your code to define schemas. You also need to import the schema class `org.apache.pig.impl.logicalLayer.schema.Schema`.

The example above shows how to create an output schema for a tuple. Doing this for a bag is very similar. Let's extend the `TOKENIZE` function to do that.

As you can see in the example below, this is very similar to the output schema definition in the `Swap` function. One difference is that instead of reusing input schema, we create a brand new field schema to represent the tokens stored in the bag. The other difference is that the type of the schema created is BAG (not TUPLE).

```
package org.apache.pig.builtin;

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.BagFactory;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.TupleFactory;
import org.apache.pig.impl.logicalLayer.schema.Schema;
import org.apache.pig.data.DataType;

public class TOKENIZE extends EvalFunc<DataBag> {
    TupleFactory mTupleFactory = TupleFactory.getInstance();
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
    BagFactory mBagFactory = BagFactory.getInstance();
    public DataBag exec(Tuple input) throws IOException {
        try {
            DataBag output = mBagFactory.newDefaultBag();
            Object o = input.get(0);
            if ((o instanceof String)) {
                throw new IOException("Expected input to be chararray, but  got " +
 o.getClass().getName());
            }
            StringTokenizer tok = new StringTokenizer((String)o, " \",()*", false);
            while (tok.hasMoreTokens())
 output.add(mTupleFactory.newTuple(tok.nextToken()));
            return output;
        } catch (ExecException ee) {
            // error handling goes here
        }
    }
    public Schema outputSchema(Schema input) {
         try{
             Schema bagSchema = new Schema();
             bagSchema.add(new Schema.FieldSchema("token", DataType.CHARARRAY));

             return new Schema(new
 Schema.FieldSchema(getSchemaName(this.getClass().getName().toLowerCase(), input),
                                                     bagSchema, DataType.BAG));
         }catch (Exception e){
            return null;
         }
    }
}
```

One more note about schemas and UDFs. Users have requested the ability to examine the input schema of the data before processing the data via a UDF. For example, they would like to know how to convert an input tuple to a map such that the keys in the map are the names of the input columns. Currently there is no way to do this. This is a feature we would like to support in the future.

2.1.10 Error Handling

There are several types of errors that can occur in a UDF:

1. An error that affects a particular row but is not likely to impact other rows. An example of such an error would be a malformed input value or divide by zero problem. A reasonable handling of this situation would be to emit a warning and return a null value. ABS function in the next section demonstrates this approach. The current approach is to write the warning to `stderr`. Eventually we would like to pass a logger to the UDFs. Note that returning a NULL value only makes sense if the malformed value is of type `bytearray`. Otherwise the proper type has been already created and should have an appropriate value. If this is not the case, it is an internal error and should cause the system to fail. Both cases can be seen in the implementation of the ABS function in the next section.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

2. An error that affects the entire processing but can succeed on retry. An example of such a failure is the inability to open a lookup file because the file could not be found. This could be a temporary environmental issue that can go away on retry. A UDF can signal this to Pig by throwing an `IOException` as with the case of the ABS function below.
3. An error that affects the entire processing and is not likely to succeed on retry. An example of such a failure is the inability to open a lookup file because of file permission problems. Pig currently does not have a way to handle this case. Hadoop does not have a way to handle this case either. It will be handled the same way as 2 above.

2.1.11 Function Overloading

Before the type system was available in Pig, all values for the purpose of arithmetic calculations were assumed to be doubles as the safest choice. However, this is not very efficient if the data is actually of type integer or long. (We saw about a 2x slowdown of a query when using double where integer could be used.) Now that Pig supports types we can take advantage of the type information and choose the function that is most efficient for the provided operands.

UDF writers are encouraged to provide type-specific versions of a function if this can result in better performance. On the other hand, we don't want the users of the functions to worry about different functions - the right thing should just happen. Pig allows for this via a function table mechanism as shown in the next example.

This example shows the implementation of the ABS function that returns the absolute value of a numeric value passed to it as input.

```
import java.io.IOException;
import java.util.List;
import java.util.ArrayList;
import org.apache.pig.EvalFunc;
import org.apache.pig.FuncSpec;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.logicalLayer.FrontendException;
import org.apache.pig.impl.logicalLayer.schema.Schema;
import org.apache.pig.data.DataType;

public class ABS extends EvalFunc<Double> {
    public Double exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        Double d;
        try{
            d = DataType.toDouble(input.get(0));
        } catch (NumberFormatException nfe){
            System.err.println("Failed to process input; error - " + nfe.getMessage());
            return null;
        } catch (Exception e){
            throw new IOException("Caught exception processing input row ", e);
        }
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
        return Math.abs(d);
    }
    public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
        List<FuncSpec> funcList = new ArrayList<FuncSpec>();
        funcList.add(new FuncSpec(this.getClass().getName(), new Schema(new
 Schema.FieldSchema(null, DataType.BYTEARRAY))));
        funcList.add(new FuncSpec(DoubleAbs.class.getName(),  new Schema(new
 Schema.FieldSchema(null, DataType.DOUBLE))));
        funcList.add(new FuncSpec(FloatAbs.class.getName(),   new Schema(new
 Schema.FieldSchema(null, DataType.FLOAT))));
        funcList.add(new FuncSpec(IntAbs.class.getName(),  new Schema(new
 Schema.FieldSchema(null, DataType.INTEGER))));
        funcList.add(new FuncSpec(LongAbs.class.getName(),  new Schema(new
 Schema.FieldSchema(null, DataType.LONG))));
        return funcList;
    }
}
```

The main thing to notice in this example is the `getArgToFuncMapping()` method. This method returns a list that contains a mapping from the input schema to the class that should be used to handle it. In this example the main class handles the `bytearray` input and outsources the rest of the work to other classes implemented in separate files in the same package. The example of one such class is below. This class handles integer input values.

```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class IntAbs extends EvalFunc<Integer> {
    public Integer exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        Integer d;
        try{
            d = (Integer)input.get(0);
        } catch (Exception e){
            throw new IOException("Caught exception processing input row ", e);
        }
        return Math.abs(d);
    }
}
```

A note on error handling. The `ABS` class covers the case of the `bytearray` which means the data has not been converted yet to its actual type. This is why a null value is returned when `NumberFormatException` is encountered. However, the `IntAbs` function is only called if the data is already of type `Integer` which means it has already been converted to the real type and bad format has been dealt with. This is why an exception is thrown if the input can't be cast to `Integer`.

The example above covers a reasonably simple case where the UDF only takes one parameter and there is a separate function for each parameter type. However, this will not always be the case. If Pig can't find an `exact match` it tries to do a `best match`.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

The rule for the best match is to find the most efficient function that can be used safely. This means that Pig must find the function that, for each input parameter, provides the smallest type that is equal to or greater than the input type. The type progression rules are: `int>long>float>double`.

For instance, let's consider function `MAX` which is part of the `piggybank` described later in this document. Given two values, the function returns the larger value. The function table for `MAX` looks like this:

```
public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
    List<FuncSpec> funcList = new ArrayList<FuncSpec>();
    Util.addToFunctionList(funcList, IntMax.class.getName(), DataType.INTEGER);
    Util.addToFunctionList(funcList, DoubleMax.class.getName(), DataType.DOUBLE);
    Util.addToFunctionList(funcList, FloatMax.class.getName(), DataType.FLOAT);
    Util.addToFunctionList(funcList, LongMax.class.getName(), DataType.LONG);

    return funcList;
}
```

The `Util.addToFunctionList` function is a helper function that adds an entry to the list as the first argument, with the key of the class name passed as the second argument, and the schema containing two fields of the same type as the third argument.

Let's now see how this function can be used in a Pig script:

```
REGISTER piggybank.jar
A = LOAD 'student_data' AS (name: chararray, gpa1: float, gpa2: double);
B = FOREACH A GENERATE name, org.apache.pig.piggybank.evaluation.math.MAX(gpa1, gpa2);
DUMP B;
```

In this example, the function gets one parameter of type `float` and another of type `double`. The best fit will be the function that takes two double values. Pig makes this choice on the user's behalf by inserting implicit casts for the parameters. Running the script above is equivalent to running the script below:

```
A = LOAD 'student_data' AS (name: chararray, gpa1: float, gpa2: double);
B = FOREACH A GENERATE name, org.apache.pig.piggybank.evaluation.math.MAX((double)gpa1,
 gpa2);
DUMP B;
```

A special case of the `best fit` approach is handling data without a schema specified. The type for this data is interpreted as `bytearray`. Since the type of the data is not known, there is no way to choose a best fit version. The only time a cast is performed is when the function table contains only a single entry. This works well to maintain backward compatibility.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

Let's revisit the UPPER function from our first example. As it is written now, it would only work if the data passed to it is of type `chararray`. To make it work with data whose type is not explicitly set, a function table with a single entry needs to be added:

```
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try{
            String str = (String)input.get(0);
            return str.toUpperCase();
        }catch(Exception e){
            System.err.println("WARN: UPPER: failed to process input; error - " +
 e.getMessage());
            return null;
        }
    }
    public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
        List<FuncSpec> funcList = new ArrayList<FuncSpec>();
        funcList.add(new FuncSpec(this.getClass().getName(), new Schema(new
 Schema.FieldSchema(null, DataType.CHARARRAY))));
        return funcList;
    }
}
```

Now the following script will ran:

```
-- this is myscript.pig
REGISTER myudfs.jar;
A = LOAD 'student_data' AS (name, age, gpa);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Variable-length arguments:

The last input schema field in `getArgToFuncMapping()` can be marked as vararg, which enables UDF writers to create UDFs that take variable length arguments. This is done by overriding the `getSchemaType()` method:

```
@Override
public SchemaType getSchemaType() {
    return SchemaType.VARARG;
}
```

For an example see CONCAT.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

2.1.12 Using Counters

Hadoop counters are easily accessible within EvalFunc by using PigStatusReporter object. Here is one example:

```
public class UPPER extends EvalFunc<String>
{
        public String exec(Tuple input) throws IOException {
                if (input == null || input.size() == 0) {
                    PigStatusReporter reporter = PigStatusReporter.getInstance();
                    if (reporter != null) {
                       reporter.incrCounter(PigWarning.UDF_WARNING_1, 1);
                    }
                    return null;
                }
                try{
                        String str = (String)input.get(0);
                        return str.toUpperCase();
                }catch(Exception e){
                    throw new IOException("Caught exception processing input row ", e);
                }
        }
}
```

2.1.13 Access input schema inside EvalFunc

Not only inside outputSchema at compile time, input schema is also accessible in exec at runtime. For example:

```
public class AddSchema extends EvalFunc<String>
{
        public String exec(Tuple input) throws IOException {
                if (input == null || input.size() == 0)
                    return null;
                String result = "";
                for (int i=0;i<input.size();i++) {
                    result += getInputSchema().getFields().get(i).alias;
                    result += ":";
                    result += input.get(i);
                }
                return result;
        }
}
```

2.1.14 Reporting Progress

A challenge of running a large shared system is to make sure system resources are used efficiently. One aspect of this challenge is detecting runaway processes that are no longer making progress. Pig uses a heartbeat mechanism for this purpose. If any of the tasks stops sending a heartbeat, the system assumes that it is dead and kills it.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

Most of the time, single-tuple processing within a UDF is very short and does not require a UDF to heartbeat. The same is true for aggregate functions that operate on large bags because bag iteration code takes care of it. However, if you have a function that performs a complex computation that can take an order of minutes to execute, you should add a progress indicator to your code. This is very easy to accomplish. The `EvalFunc` class provides a `progress` function that you need to call in your `exec` method.

For instance, the `UPPER` function would now look as follows:

```
public class UPPER extends EvalFunc<String>
{
        public String exec(Tuple input) throws IOException {
                if (input == null || input.size() == 0)
                return null;
                try{
                        progress();
                        String str = (String)input.get(0);
                        return str.toUpperCase();
                }catch(Exception e){
                    throw new IOException("Caught exception processing input row ", e);
                }
        }
}
```

2.1.15 Using Distributed Cache

Use getCacheFiles or getShipFiles to return a list of HDFS files or local files that need to be shipped to distributed cache. Inside exec method, you can assume that these files already exist in distributed cache. For example:

```
public class Udfcachetest extends EvalFunc<String> {

    public String exec(Tuple input) throws IOException {
        String concatResult = "";
        FileReader fr = new FileReader("./smallfile1");
        BufferedReader d = new BufferedReader(fr);
        concatResult +=d.readLine();
        fr = new FileReader("./smallfile2");
        d = new BufferedReader(fr);
        concatResult +=d.readLine();
        return concatResult;
    }

    public List<String> getCacheFiles() {
        List<String> list = new ArrayList<String>(1);
        list.add("/user/pig/tests/data/small#smallfile1");  // This is hdfs file
        return list;
    }

    public List<String> getShipFiles() {
        List<String> list = new ArrayList<String>(1);
        list.add("/home/hadoop/pig/smallfile2");  // This local file
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
        return list;
    }
}

a = load '1.txt';
b = foreach a generate Udfcachetest(*);
dump b;
```

2.1.16 Compile time evaluation

If the parameters of the EvalFunc are all constants, Pig could evaluate the result at compile time. The benefit of evaluating at compile time is performance optimization, and enable certain other optimizations at front end (such as partition pruning, which only allow constant not UDF in filter condition). By default, compile time evaluation is disabled in EvalFunc to prevent potential side effect. To enable it, override allowCompileTimeCalculation. For example:

```
public class CurrentTime extends EvalFunc<DateTime> {
    public String exec(Tuple input) throws IOException {
        return new DateTime();
    }
    @Override
    public boolean allowCompileTimeCalculation() {
        return true;
    }
}
```

2.1.17 Typecasting from bytearrays

Just like Load Function and Streaming, Java UDF has a getLoadCaster() method that returns LoadCaster to convert byte arrays to specific types. A UDf implementation should implement this if casts (implicit or explicit) from DataByteArray fields to other types need to be supported. Default implementation returns null and Pig will determine if all parameters passed to the UDF have identical loadcaster and use it when true.

2.1.18 Clean up static variable in Tez

In Tez, jvm could reuse for other tasks. It is important to cleanup static variable to make sure there is no side effect. Here is one example:

```
public class UPPER extends EvalFunc<String>
{
        static boolean initialized = false;
        static {
            JVMReuseManager.getInstance().registerForStaticDataCleanup(UPPER.class);
        }
        public String exec(Tuple input) throws IOException {
            if (!initialized) {
                init();
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
                initialized = true;
            }
            ......
        }
        @StaticDataCleanup
        public static void staticDataCleanup() {
            initialized = false;
        }
}
```

2.2 Load/Store Functions

The load/store UDFs control how data goes into Pig and comes out of Pig. Often, the same function handles both input and output but that does not have to be the case.

The Pig load/store API is aligned with Hadoop's InputFormat and OutputFormat classes. This enables you to create new LoadFunc and StoreFunc implementations based on existing Hadoop InputFormat and OutputFormat classes with minimal code. The complexity of reading the data and creating a record lies in the InputFormat while the complexity of writing the data lies in the OutputFormat. This enables Pig to easily read/write data in new storage formats as and when an Hadoop InputFormat and OutputFormat is available for them.

Note: Both the LoadFunc and StoreFunc implementations should use the Hadoop 20 API based classes (InputFormat/OutputFormat and related classes) under the **new** org.apache.hadoop.mapreduce package instead of the old org.apache.hadoop.mapred package.

2.2.1 Load Functions

LoadFunc abstract class has three main methods for loading data and for most use cases it would suffice to extend it. There are three other optional interfaces which can be implemented to achieve extended functionality:

- LoadMetadata has methods to deal with metadata - most implementation of loaders don't need to implement this unless they interact with some metadata system. The getSchema() method in this interface provides a way for loader implementations to communicate the schema of the data back to pig. If a loader implementation returns data comprised of fields of real types (rather than DataByteArray fields), it should provide the schema describing the data returned through the getSchema() method. The other methods are concerned with other types of metadata like partition keys and statistics. Implementations can return null return values for these methods if they are not applicable for that implementation.
- LoadPushDown has methods to push operations from Pig runtime into loader implementations. Currently only the pushProjection() method is called by Pig to communicate to the loader the exact fields that are required in the Pig script. The loader implementation can choose to honor the request (return only those fields required by Pig

Copyright © 2007 The Apache Software Foundation. All rights reserved.

script) or not honor the request (return all fields in the data). If the loader implementation can efficiently honor the request, it should implement LoadPushDown to improve query performance. (Irrespective of whether the implementation can or cannot honor the request, if the implementation also implements getSchema(), the schema returned in getSchema() should describe the entire tuple of data.)

- pushProjection(): This method tells LoadFunc which fields are required in the Pig script, thus enabling LoadFunc to optimize performance by loading only those fields that are needed. pushProjection() takes a requiredFieldList. requiredFieldList is read only and cannot be changed by LoadFunc. requiredFieldList includes a list of requiredField: each requiredField indicates a field required by the Pig script; each requiredField includes index, alias, type (which is reserved for future use), and subFields. Pig will use the column index requiredField.index to communicate with the LoadFunc about the fields required by the Pig script. If the required field is a map, Pig will optionally pass requiredField.subFields which contains a list of keys that the Pig script needs for the map. For example, if the Pig script needs two keys for the map, "key1" and "key2", the subFields for that map will contain two requiredField; the alias field for the first RequiredField will be "key1" and the alias for the second requiredField will be "key2". LoadFunc will use requiredFieldResponse.requiredFieldRequestHonored to indicate whether the pushProjection() request is honored.

- LoadCaster has methods to convert byte arrays to specific types. A loader implementation should implement this if casts (implicit or explicit) from DataByteArray fields to other types need to be supported.
- LoadPredicatePushdown has the methods to push predicates to the loader. It is different than LoadMetadata.setPartitionFilter in that loader may load records which does not satisfy the predicates. In other words, predicates is only a hint. Note this interface is still in development and might change in next version. Currently only OrcStorage implements this interface.
- NonFSLoadFunc is a marker interface to indicate that a LoadFunc implementation is not a filesystem loader. This is useful for LoadFunc classes that for example supply queries instead of filesystem pathes to the LOAD operator.

The LoadFunc abstract class is the main class to extend for implementing a loader. The methods which need to be overridden are explained below:

- getInputFormat(): This method is called by Pig to get the InputFormat used by the loader. The methods in the InputFormat (and underlying RecordReader) are called by Pig in the same manner (and in the same context) as by Hadoop in a MapReduce java program. If the InputFormat is a Hadoop packaged one, the implementation should use the new API based one under org.apache.hadoop.mapreduce. If it is a custom InputFormat, it should be implemented using the new API in org.apache.hadoop.mapreduce.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

If a custom loader using a text-based InputFormat or a file-based InputFormat would like to read files in all subdirectories under a given input directory recursively, then it should use the PigTextInputFormat and PigFileInputFormat classes provided in org.apache.pig.backend.hadoop.executionengine.mapReduceLayer. The Pig InputFormat classes work around a current limitation in the Hadoop TextInputFormat and FileInputFormat classes which only read one level down from the provided input directory. For example, if the input in the load statement is 'dir1' and there are subdirs 'dir2' and 'dir2/dir3' beneath dir1, the Hadoop TextInputFormat and FileInputFormat classes read the files under 'dir1' only. Using PigTextInputFormat or PigFileInputFormat (or by extending them), the files in all the directories can be read.

- setLocation(): This method is called by Pig to communicate the load location to the loader. The loader should use this method to communicate the same information to the underlying InputFormat. This method is called multiple times by pig - implementations should bear this in mind and should ensure there are no inconsistent side effects due to the multiple calls.
- prepareToRead(): Through this method the RecordReader associated with the InputFormat provided by the LoadFunc is passed to the LoadFunc. The RecordReader can then be used by the implementation in getNext() to return a tuple representing a record of data back to pig.
- getNext(): The meaning of getNext() has not changed and is called by Pig runtime to get the next tuple in the data - in this method the implementation should use the underlying RecordReader and construct the tuple to return.

The following methods have default implementations in LoadFunc and should be overridden only if needed:

- setUdfContextSignature(): This method will be called by Pig both in the front end and back end to pass a unique signature to the Loader. The signature can be used to store into the UDFContext any information which the Loader needs to store between various method invocations in the front end and back end. A use case is to store RequiredFieldList passed to it in LoadPushDown.pushProjection(RequiredFieldList) for use in the back end before returning tuples in getNext(). The default implementation in LoadFunc has an empty body. This method will be called before other methods.
- relativeToAbsolutePath(): Pig runtime will call this method to allow the Loader to convert a relative load location to an absolute location. The default implementation provided in LoadFunc handles this for FileSystem locations. If the load source is something else, loader implementation may choose to override this.
- getCacheFiles(): Return a list of hdfs files to ship to distributed cache.
- getShipFiles(): Return a list of local files to ship to distributed cache.

Example Implementation

Copyright © 2007 The Apache Software Foundation. All rights reserved.

The loader implementation in the example is a loader for text data with line delimiter as '\n' and '\t' as default field delimiter (which can be overridden by passing a different field delimiter in the constructor) - this is similar to current PigStorage loader in Pig. The implementation uses an existing Hadoop supported Inputformat - TextInputFormat - as the underlying InputFormat.

```
public class SimpleTextLoader extends LoadFunc {
    protected RecordReader in = null;
    private byte fieldDel = '\t';
    private ArrayList<Object> mProtoTuple = null;
    private TupleFactory mTupleFactory = TupleFactory.getInstance();
    private static final int BUFFER_SIZE = 1024;

    public SimpleTextLoader() {
    }

    /**
     * Constructs a Pig loader that uses specified character as a field delimiter.
     *
     * @param delimiter
     *            the single byte character that is used to separate fields.
     *            ("\t" is the default.)
     */
    public SimpleTextLoader(String delimiter) {
        this();
        if (delimiter.length() == 1) {
            this.fieldDel = (byte)delimiter.charAt(0);
        } else if (delimiter.length() >  1 & & delimiter.charAt(0) == '\\') {
            switch (delimiter.charAt(1)) {
            case 't':
                this.fieldDel = (byte)'\t';
                break;

            case 'x':
               fieldDel =
                    Integer.valueOf(delimiter.substring(2), 16).byteValue();
               break;

            case 'u':
                this.fieldDel =
                    Integer.valueOf(delimiter.substring(2)).byteValue();
                break;

            default:
                throw new RuntimeException("Unknown delimiter " + delimiter);
            }
        } else {
            throw new RuntimeException("PigStorage delimeter must be a single character");
        }
    }

    @Override
    public Tuple getNext() throws IOException {
        try {
            boolean notDone = in.nextKeyValue();
            if (notDone) {
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
                return null;
            }
            Text value = (Text) in.getCurrentValue();
            byte[] buf = value.getBytes();
            int len = value.getLength();
            int start = 0;

            for (int i = 0; i < len; i++) {
                if (buf[i] == fieldDel) {
                    readField(buf, start, i);
                    start = i + 1;
                }
            }
            // pick up the last field
            readField(buf, start, len);

            Tuple t =  mTupleFactory.newTupleNoCopy(mProtoTuple);
            mProtoTuple = null;
            return t;
        } catch (InterruptedException e) {
            int errCode = 6018;
            String errMsg = "Error while reading input";
            throw new ExecException(errMsg, errCode,
                    PigException.REMOTE_ENVIRONMENT, e);
        }

    }

    private void readField(byte[] buf, int start, int end) {
        if (mProtoTuple == null) {
            mProtoTuple = new ArrayList<Object>();
        }

        if (start == end) {
            // NULL value
            mProtoTuple.add(null);
        } else {
            mProtoTuple.add(new DataByteArray(buf, start, end));
        }
    }

    @Override
    public InputFormat getInputFormat() {
        return new TextInputFormat();
    }

    @Override
    public void prepareToRead(RecordReader reader, PigSplit split) {
        in = reader;
    }

    @Override
    public void setLocation(String location, Job job)
            throws IOException {
        FileInputFormat.setInputPaths(job, location);
    }
}
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

2.2.2 Store Functions

StoreFunc abstract class has the main methods for storing data and for most use cases it should suffice to extend it. There is an optional interface which can be implemented to achieve extended functionality:

- StoreMetadata: This interface has methods to interact with metadata systems to store schema and store statistics. This interface is optional and should only be implemented if metadata needs to stored.
- StoreResources: This interface has methods to put hdfs files or local files to distributed cache.
- ErrorHandling: This interface allow you to skip bad records in the storer so the storer will not throw exception and terminate the job. You can implement your own error handler by overriding ErrorHandler interface, or use predefined error handler: CounterBasedErrorHandler. ErrorHandling can be turned on by setting the property pig.error-handling.enabled to true in pig.properties. Default is false. CounterBasedErrorHandler uses two settings - pig.error-handling.min.error.records (the minimum number of errors to trigger error handling) and pig.error-handling.error.threshold (percentage of the number of records as a fraction exceeding which error is thrown).

The methods which need to be overridden in StoreFunc are explained below:

- getOutputFormat(): This method will be called by Pig to get the OutputFormat used by the storer. The methods in the OutputFormat (and underlying RecordWriter and OutputCommitter) will be called by pig in the same manner (and in the same context) as by Hadoop in a map-reduce java program. If the OutputFormat is a hadoop packaged one, the implementation should use the new API based one under org.apache.hadoop.mapreduce. If it is a custom OutputFormat, it should be implemented using the new API under org.apache.hadoop.mapreduce. The checkOutputSpecs() method of the OutputFormat will be called by pig to check the output location up-front. This method will also be called as part of the Hadoop call sequence when the job is launched. So implementations should ensure that this method can be called multiple times without inconsistent side effects.
- setStoreLocation(): This method is called by Pig to communicate the store location to the storer. The storer should use this method to communicate the same information to the underlying OutputFormat. This method is called multiple times by pig - implementations should bear in mind that this method is called multiple times and should ensure there are no inconsistent side effects due to the multiple calls.
- prepareToWrite(): Writing of the data is through the OutputFormat provided by the StoreFunc. In prepareToWrite() the RecordWriter associated with the OutputFormat provided by the StoreFunc is passed to the StoreFunc. The RecordWriter can then be

Copyright © 2007 The Apache Software Foundation. All rights reserved.

used by the implementation in putNext() to write a tuple representing a record of data in a manner expected by the RecordWriter.
- putNext(): This method is called by Pig runtime to write the next tuple of data - this is the method wherein the implementation will use the underlying RecordWriter to write the Tuple out.

The following methods have default implementations in StoreFunc and should be overridden only if necessary:

- setStoreFuncUDFContextSignature(): This method will be called by Pig both in the front end and back end to pass a unique signature to the Storer. The signature can be used to store into the UDFContext any information which the Storer needs to store between various method invocations in the front end and back end. The default implementation in StoreFunc has an empty body. This method will be called before other methods.
- relToAbsPathForStoreLocation(): Pig runtime will call this method to allow the Storer to convert a relative store location to an absolute location. An implementation is provided in StoreFunc which handles this for FileSystem based locations.
- checkSchema(): A Store function should implement this function to check that a given schema describing the data to be written is acceptable to it. The default implementation in StoreFunc has an empty body. This method will be called before any calls to setStoreLocation().

Example Implementation

The storer implementation in the example is a storer for text data with line delimiter as '\n' and '\t' as default field delimiter (which can be overridden by passing a different field delimiter in the constructor) - this is similar to current PigStorage storer in Pig. The implementation uses an existing Hadoop supported OutputFormat - TextOutputFormat as the underlying OutputFormat.

```
public class SimpleTextStorer extends StoreFunc {
    protected RecordWriter writer = null;

    private byte fieldDel = '\t';
    private static final int BUFFER_SIZE = 1024;
    private static final String UTF8 = "UTF-8";
    public PigStorage() {
    }

    public PigStorage(String delimiter) {
        this();
        if (delimiter.length() == 1) {
            this.fieldDel = (byte)delimiter.charAt(0);
        } else if (delimiter.length() > 1delimiter.charAt(0) == '\\') {
            switch (delimiter.charAt(1)) {
            case 't':
                this.fieldDel = (byte)'\t';
                break;
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
            case 'x':
               fieldDel =
                    Integer.valueOf(delimiter.substring(2), 16).byteValue();
               break;
            case 'u':
                this.fieldDel =
                    Integer.valueOf(delimiter.substring(2)).byteValue();
                break;

            default:
                throw new RuntimeException("Unknown delimiter " + delimiter);
            }
        } else {
            throw new RuntimeException("PigStorage delimeter must be a single character");
        }
    }

    ByteArrayOutputStream mOut = new ByteArrayOutputStream(BUFFER_SIZE);

    @Override
    public void putNext(Tuple f) throws IOException {
        int sz = f.size();
        for (int i = 0; i < sz; i++) {
            Object field;
            try {
                field = f.get(i);
            } catch (ExecException ee) {
                throw ee;
            }

            putField(field);

            if (i != sz - 1) {
                mOut.write(fieldDel);
            }
        }
        Text text = new Text(mOut.toByteArray());
        try {
            writer.write(null, text);
            mOut.reset();
        } catch (InterruptedException e) {
            throw new IOException(e);
        }
    }

    @SuppressWarnings("unchecked")
    private void putField(Object field) throws IOException {
        //string constants for each delimiter
        String tupleBeginDelim = "(";
        String tupleEndDelim = ")";
        String bagBeginDelim = "{";
        String bagEndDelim = "}";
        String mapBeginDelim = "[";
        String mapEndDelim = "]";
        String fieldDelim = ",";
        String mapKeyValueDelim = "#";

        switch (DataType.findType(field)) {
        case DataType.NULL:
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
            break; // just leave it empty

        case DataType.BOOLEAN:
            mOut.write(((Boolean)field).toString().getBytes());
            break;

        case DataType.INTEGER:
            mOut.write(((Integer)field).toString().getBytes());
            break;

        case DataType.LONG:
            mOut.write(((Long)field).toString().getBytes());
            break;

        case DataType.FLOAT:
            mOut.write(((Float)field).toString().getBytes());
            break;

        case DataType.DOUBLE:
            mOut.write(((Double)field).toString().getBytes());
            break;

        case DataType.BYTEARRAY: {
            byte[] b = ((DataByteArray)field).get();
            mOut.write(b, 0, b.length);
            break;
                                }

        case DataType.CHARARRAY:
            // oddly enough, writeBytes writes a string
            mOut.write(((String)field).getBytes(UTF8));
            break;

        case DataType.MAP:
            boolean mapHasNext = false;
            Map<String, Object> m = (Map<String, Object>)field;
            mOut.write(mapBeginDelim.getBytes(UTF8));
            for(Map.Entry<String, Object> e: m.entrySet()) {
                if(mapHasNext) {
                    mOut.write(fieldDelim.getBytes(UTF8));
                } else {
                    mapHasNext = true;
                }
                putField(e.getKey());
                mOut.write(mapKeyValueDelim.getBytes(UTF8));
                putField(e.getValue());
            }
            mOut.write(mapEndDelim.getBytes(UTF8));
            break;

        case DataType.TUPLE:
            boolean tupleHasNext = false;
            Tuple t = (Tuple)field;
            mOut.write(tupleBeginDelim.getBytes(UTF8));
            for(int i = 0; i < t.size(); ++i) {
                if(tupleHasNext) {
                    mOut.write(fieldDelim.getBytes(UTF8));
                } else {
                    tupleHasNext = true;
                }
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
                try {
                    putField(t.get(i));
                } catch (ExecException ee) {
                    throw ee;
                }
            }
            mOut.write(tupleEndDelim.getBytes(UTF8));
            break;

        case DataType.BAG:
            boolean bagHasNext = false;
            mOut.write(bagBeginDelim.getBytes(UTF8));
            Iterator<Tuple> tupleIter = ((DataBag)field).iterator();
            while(tupleIter.hasNext()) {
                if(bagHasNext) {
                    mOut.write(fieldDelim.getBytes(UTF8));
                } else {
                    bagHasNext = true;
                }
                putField((Object)tupleIter.next());
            }
            mOut.write(bagEndDelim.getBytes(UTF8));
            break;

        default: {
            int errCode = 2108;
            String msg = "Could not determine data type of field: " + field;
            throw new ExecException(msg, errCode, PigException.BUG);
        }

        }
    }

    @Override
    public OutputFormat getOutputFormat() {
        return new TextOutputFormat<WritableComparable, Text>();
    }

    @Override
    public void prepareToWrite(RecordWriter writer) {
        this.writer = writer;
    }

    @Override
    public void setStoreLocation(String location, Job job) throws IOException {
        job.getConfiguration().set("mapred.textoutputformat.separator", "");
        FileOutputFormat.setOutputPath(job, new Path(location));
        if (location.endsWith(".bz2")) {
            FileOutputFormat.setCompressOutput(job, true);
            FileOutputFormat.setOutputCompressorClass(job,  BZip2Codec.class);
        }  else if (location.endsWith(".gz")) {
            FileOutputFormat.setCompressOutput(job, true);
            FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
        }
    }
}
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

2.3 Using Short Names

There are two ways to call a Java UDF using a short name. One way is specifying the package to an import list via Java property, and the other is defining an alias of the UDF by DEFINE statement.

2.3.1 Import Lists

An import list allows you to specify the package to which a UDF or a group of UDFs belong, eliminating the need to qualify the UDF on every call. An import list can be specified via the udf.import.list Java property on the Pig command line:

```
pig -Dudf.import.list=com.yahoo.yst.sds.ULT
```

You can supply multiple locations as well:

```
pig -Dudf.import.list=com.yahoo.yst.sds.ULT:org.apache.pig.piggybank.evaluation
```

To make use of import scripts, do the following:

```
myscript.pig:
A = load '/data/SDS/data/searcg_US/20090820' using ULTLoader as (s, m, l);
....

command:
pig -cp sds.jar -Dudf.import.list=com.yahoo.yst.sds.ULT myscript.pig
```

2.3.2 Defining Aliases

You can define an alias for a function using DEFINE statement:

```
REGISTER piggybank.jar
DEFINE MAXNUM org.apache.pig.piggybank.evaluation.math.MAX;
A = LOAD 'student_data' AS (name: chararray, gpa1: float, gpa2: double);
B = FOREACH A GENERATE name, MAXNUM(gpa1, gpa2);
DUMP B;
```

The first parameter of DEFINE statement is an alias of the function. The second parameter is the fully-qualified name of the function. After the statement, you can call the function using the alias instead of the fually-qualified name.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

2.4 Advanced Topics

2.4.1 UDF Interfaces

Java UDFs can be invoked multiple ways. The simplest UDF can just extend EvalFunc, which requires only the exec function to be implemented (see How to Write a Simple Eval Function). Every eval UDF must implement this. Additionally, if a function is algebraic, it can implement `Algebraic` interface to significantly improve query performance in the cases when combiner can be used (see Algebraic Interface). Finally, a function that can process tuples in an incremental fashion can also implement the Accumulator interface to improve query memory consumption (see Accumulator Interface).

The optimizer selects the exact method by which a UDF is invoked based on the UDF type and the query. Note that only a single interface is used at any given time. The optimizer tries to find the most efficient way to execute the function. If a combiner is used and the function implements the Algebraic interface then this interface will be used to invoke the function. If the combiner is not invoked but the accumulator can be used and the function implements Accumulator interface then that interface is used. If neither of the conditions is satisfied then the exec function is used to invoke the UDF.

2.4.2 Function Instantiation

One problem that users run into is when they make assumption about how many times a constructor for their UDF is called. For instance, they might be creating side files in the store function and doing it in the constructor seems like a good idea. The problem with this approach is that in most cases Pig instantiates functions on the client side to, for instance, examine the schema of the data.

Users should not make assumptions about how many times a function is instantiated; instead, they should make their code resilient to multiple instantiations. For instance, they could check if the files exist before creating them.

2.4.3 Passing Configurations to UDFs

The singleton UDFContext class provides two features to UDF writers. First, on the backend, it allows UDFs to get access to the JobConf object, by calling getJobConf. This is only available on the backend (at run time) as the JobConf has not yet been constructed on the front end (during planning time).

Second, it allows UDFs to pass configuration information between instantiations of the UDF on the front and backends. UDFs can store information in a configuration object when they are constructed on the front end, or during other front end calls such as checkSchema. They can then read that information on the backend when exec (for EvalFunc) or getNext (for

Copyright © 2007 The Apache Software Foundation. All rights reserved.

LoadFunc) is called. Note that information will not be passed between instantiations of the function on the backend. The communication channel only works from front end to back end.

To store information, the UDF calls getUDFProperties. This returns a Properties object which the UDF can record the information in or read the information from. To avoid name space conflicts UDFs are required to provide a signature when obtaining a Properties object. This can be done in two ways. The UDF can provide its Class object (via this.getClass()). In this case, every instantiation of the UDF will be given the same Properties object. The UDF can also provide its Class plus an array of Strings. The UDF can pass its constructor arguments, or some other identifying strings. This allows each instantiation of the UDF to have a different properties object thus avoiding name space collisions between instantiations of the UDF.

2.4.4 Monitoring Long-Running UDFs

Sometimes one may discover that a UDF that executes very quickly in the vast majority of cases turns out to run exceedingly slowly on occasion. This can happen, for example, if a UDF uses complex regular expressions to parse free-form strings, or if a UDF uses some external service to communicate with. As of version 0.8, Pig provides a facility for monitoring the length of time a UDF is executing for every invocation, and terminating its execution if it runs too long. This facility can be turned on using a simple Java annotation:

```
import org.apache.pig.builtin.MonitoredUDF;

@MonitoredUDF
public class MyUDF extends EvalFunc<Integer> {
  /* implementation goes here */
}
```

Simply annotating your UDF in this way will cause Pig to terminate the UDF's exec() method if it runs for more than 10 seconds, and return the default value of null. The duration of the timeout and the default value can be specified in the annotation, if desired:

```
import org.apache.pig.builtin.MonitoredUDF;

@MonitoredUDF(timeUnit = TimeUnit.MILLISECONDS, duration = 100, intDefault = 10)
public class MyUDF extends EvalFunc<Integer> {
  /* implementation goes here */
}
```

intDefault, longDefault, doubleDefault, floatDefault, and stringDefault can be specified in the annotation; the correct default will be chosen based on the return type of the UDF. Custom defaults for tuples and bags are not supported at this time.

If desired, custom logic can also be implemented for error handling by creating a subclass of MonitoredUDFExecutor.ErrorCallback, and overriding its handleError and/or handleTimeout

Copyright © 2007 The Apache Software Foundation. All rights reserved.

methods. Both of those methods are static, and are passed in the instance of the EvalFunc that produced an exception, as well as an exception, so you may use any state you have in the UDF to process the errors as desired. The default behavior is to increment Hadoop counters every time an error is encountered. Once you have an implementation of the ErrorCallback that performs your custom logic, you can provide it in the annotation:

```
import org.apache.pig.builtin.MonitoredUDF;

@MonitoredUDF(errorCallback=MySpecialErrorCallback.class)
public class MyUDF extends EvalFunc<Integer> {
  /* implementation goes here */
}
```

Currently the MonitoredUDF annotation works with regular and Algebraic UDFs, but has no effect on UDFs that run in the Accumulator mode.

3 Writing Jython UDFs

3.1 Registering the UDF

You can register a Jython script as shown here. This example uses org.apache.pig.scripting.jython.JythonScriptEngine to interpret the Jython script. You can develop and use custom script engines to support multiple programming languages and ways to interpret them. Currently, Pig identifies jython as a keyword and ships the required scriptengine (jython) to interpret it.

```
Register 'test.py' using jython as myfuncs;
```

The following syntax is also supported, where myfuncs is the namespace created for all the functions inside test.py.

```
register 'test.py' using org.apache.pig.scripting.jython.JythonScriptEngine as myfuncs;
```

A typical test.py looks like this:

```
@outputSchema("word:chararray")
def helloworld():
  return 'Hello, World'

@outputSchema("word:chararray,num:long")
def complex(word):
  return str(word),len(word)

@outputSchemaFunction("squareSchema")
def square(num):
  return ((num)*(num))
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
@schemaFunction("squareSchema")
def squareSchema(input):
  return input

# No decorator - bytearray
def concat(str):
  return str+str
```

The register statement above registers the Jython functions defined in test.py in Pig's runtime within the defined namespace (myfuncs here). They can then be referred later on in the pig script as myfuncs.helloworld(), myfuncs.complex(), and myfuncs.square(). An example usage is:

```
b = foreach a generate myfuncs.helloworld(), myfuncs.square(3);
```

3.2 Decorators and Schemas

To annotate a Jython script so that Pig can identify return types, use Jython decorators to define output schema for the script UDF.

- outputSchema - Defines schema for a script UDF in a format that Pig understands and is able to parse.
- outputFunctionSchema - Defines a script delegate function that defines schema for this function depending upon the input type. This is needed for functions that can accept generic types and perform generic operations on these types. A simple example is square which can accept multiple types. SchemaFunction for this type is a simple identity function (same schema as input).
- schemaFunction - Defines delegate function and is not registered to Pig.

When no decorator is specified, Pig assumes the output datatype as bytearray and converts the output generated by script function to bytearray. This is consistent with Pig's behavior in case of Java UDFs.

Sample Schema String - y:{t:(word:chararray,num:long)}, variable names inside a schema string are not used anywhere, they just make the syntax identifiable to the parser.

3.3 Example Scripts

Simple tasks like string manipulation, mathematical computations, and reorganizing data types can be easily done using Jython scripts without having to develop long and complex UDFs in Java. The overall overhead of using scripting language is much less and development cost is almost negligible. The following UDFs, developed in Jython, can be used with Pig.

```
mySampleLib.py
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
---------------------
#/usr/bin/python

##################
# Math functions #
##################
#square - Square of a number of any data type
@outputSchemaFunction("squareSchema")
def square(num):
  return ((num)*(num))
@schemaFunction("squareSchema")
def squareSchema(input):
  return input

#Percent- Percentage
@outputSchema("percent:double")
def percent(num, total):
  return num * 100 / total

####################
# String Functions #
####################
#commaFormat- format a number with commas, 12345-> 12,345
@outputSchema("numformat:chararray")
def commaFormat(num):
  return '{:,}'.format(num)

#concatMultiple- concat multiple words
@outputSchema("onestring:chararray")
def concatMult4(word1, word2, word3, word4):
  return word1 word2 word3 word4

########################
# Data Type Functions #
########################
#collectBag- collect elements of a bag into other bag
#This is useful UDF after group operation
@outputSchema("y:bag{t:tuple(len:int,word:chararray)}")
def collectBag(bag):
  outBag = []
  for word in bag:
    tup=(len(bag), word[1])
    outBag.append(tup)
  return outBag

# Few comments-
# Pig mandates that a bag should be a bag of tuples, Jython UDFs should follow this
pattern.
# Tuples in Jython are immutable, appending to a tuple is not possible.
```

3.4 Advanced Topics

3.4.1 Importing Modules

You can import Jython modules in your Jython script. Pig resolves Jython dependencies recursively, which means Pig will automatically ship all dependent Jython modules to the

Copyright © 2007 The Apache Software Foundation. All rights reserved.

backend. Jython modules should be found in the jython search path: JYTHON_HOME, JYTHONPATH, or current directory.

3.4.2 Combined Scripts

UDFs and Pig scripts are generally stored in separate files. For testing purposes you can combine the code in a single file - a "combined" script. Note, however, if you then decide to embed this "combined" script in a host language, the language of the UDF must match the host language.

This example combines Jython and Pig. This "combined" script can only be embedded in Jython.

With Jython you MUST use the `if __name__ == '__main__':` construct to separate UDFs and control flow. Otherwise the script will result in an error.

```
 #!/usr/bin/jython
from org.apache.pig.scripting import *

@outputSchema("word:chararray")
def helloworld():
   return 'Hello, World'

if __name__ == '__main__':
       P = Pig.compile("""a = load '1.txt' as (a0, a1);
                          b = foreach a generate helloworld();
                          store b into 'myoutput'; """)

result = P.bind().runSingle();
```

4 Writing JavaScript UDFs

Note: *JavaScript UDFs are an experimental feature.*

4.1 Registering the UDF

You can register JavaScript as shown here. This example uses org.apache.pig.scripting.js.JsScriptEngine to interpret JavaScript. You can develop and use custom script engines to support multiple programming languages and ways to interpret them. Currently, Pig identifies js as a keyword and ships the required scriptengine (Rhino) to interpret it.

```
 register 'test.js' using javascript as myfuncs;
```

The following syntax is also supported, where myfuncs is the namespace created for all the functions inside test.js.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
register 'test.js' using org.apache.pig.scripting.js.JsScriptEngine as myfuncs;
```

The register statement above registers the js functions defined in test.js in Pig's runtime within the defined namespace (myfuncs here). They can then be referred later on in the pig script as myfuncs.helloworld(), myfuncs.complex(), and myfuncs.square(). An example usage is:

```
b = foreach a generate myfuncs.helloworld(), myfuncs.complex($0);
```

4.2 Return Types and Schemas

Since JavaScript functions are first class objects, you can annotate them by adding attributes. Add an outputSchema attribute to your function so that Pig can identify return types for the script UDF.

- outputSchema - Defines schema for a script udf in a format that Pig understands and is able to parse.
- Sample Schema String - y:{t:(word:chararray,num:long)}
  Variable names inside a schema string are used for type conversion between Pig and JavaScript. Tuples are converted to Objects using the names and vice versa

4.3 Example Scripts

A simple JavaScript UDF (udf.js) is shown here.

```
helloworld.outputSchema = "word:chararray";
function helloworld() {
    return 'Hello, World';
}

complex.outputSchema = "(word:chararray,num:long)";
function complex(word){
    return {word:word, num:word.length};
}
```

This Pig script registers the JavaScript UDF (udf.js).

```
register 'udf.js' using javascript as myfuncs;
A = load 'data' as (a0:chararray, a1:int);
B = foreach A generate myfuncs.helloworld(), myfuncs.complex(a0);
... ...
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

4.4 Advanced Topics

UDFs and Pig scripts are generally stored in separate files. For testing purposes you can combine the code in a single file - a "combined" script. Note, however, if you then decide to embed this "combined" script in a host language, the language of the UDF and the host language must match.

This example combines JavaScript and Pig. This "combined" script can only be embedded in JavaScript.

With JavaScript, the control flow MUST be defined in the main function. Otherwise the script will result in an error.

```
importPackage(Packages.org.apache.pig.scripting.js)
pig = org.apache.pig.scripting.js.JSPig;

helloworld.outputSchema = "word:chararray"
function helloworld() {
    return 'Hello, World';
}

function main() {
  var P = pig.compile(" a = load '1.txt' as (a0, a1);”+
       “b = foreach a generate helloworld();”+
           “store b into 'myoutput';");

  var result = P.bind().runSingle();
}
```

5 Writing Ruby UDFs

Note: *Ruby UDFs are an experimental feature.*

5.1 Writing a Ruby UDF

You must extend PigUdf and define your Ruby UDFs in the class.

```
require 'pigudf'
class Myudfs < PigUdf
    def square num
        return nil if num.nil?
        num**2
    end
end
```

5.2 Return Types and Schemas

You have two ways to define the return schema:

Copyright © 2007 The Apache Software Foundation. All rights reserved.

outputSchema - Defines the schema for a UDF in a format that Pig understands.

```
outputSchema "word:chararray"



outputSchema "t:(m:[], t:(name:chararray, age:int, gpa:double), b:{t:(name:chararray,
age:int, gpa:double)})"
```

Schema function

```
outputSchemaFunction :squareSchema
def squareSchema input
    input
end
```

You need to put outputSchema/outputSchemaFunction statement right before your UDF. The schema function itself can be defined anywhere inside the class.

5.3 Registering the UDF

You can register a Ruby UDF as shown here.

```
register 'test.rb' using jruby as myfuncs;
```

This is a shortcut to the complete syntax:

```
register 'test.rb' using org.apache.pig.scripting.jruby.JrubyScriptEngine as myfuncs;
```

The `register` statement above registers the Ruby functions defined in test.rb in Pig's runtime within the defined namespace (myfuncs in this example). They can then be referred later on in the Pig Latin script as `myfuncs.square()`. An example usage is:

```
b = foreach a generate myfuncs.concat($0, $1);
```

5.4 Example Scripts

Here are two complete Ruby UDF samples.

```
require 'pigudf'
class Myudfs < PigUdf
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
outputSchema "word:chararray"
    def concat *input
        input.inject(:+)
    end
end



require 'pigudf'
class Myudfs < PigUdf
outputSchemaFunction :squareSchema
    def square num
        return nil if num.nil?
        num**2
    end
    def squareSchema input
        input
    end
end
```

5.5 Advanced Topics

You can also write Algebraic and Accumulator UDFs using Ruby. You need to extend your class from `AlgebraicPigUdf` and `AccumulatorPigUdf` respectively. For an Algebraic UDF, define `initial`, `intermed`, and `final` methods in the class. For an Accumulator UDF, define `exec` and `get` methods in the class. Below are example for each type of UDF:

```


class Count < AlgebraicPigUdf
    output_schema Schema.long
    def initial t
         t.nil? ? 0 : 1
    end
    def intermed t
         return 0 if t.nil?
         t.flatten.inject(:+)
    end
    def final t
        intermed(t)
    end
end



class Sum < AccumulatorPigUdf
    output_schema { |i| i.in.in[0] }
    def exec items
        @sum ||= 0
        @sum += items.flatten.inject(:+)
    end
    def get
        @sum
    end
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
 end
```

6 Writing Groovy UDFs

Note: *Groovy UDFs are an experimental feature.*

6.1 Registering the UDF

You can register a Groovy script as shown here. This example uses org.apache.pig.scripting.groovy.GroovyScriptEngine to interpret the Groovy script. You can develop and use custom script engines to support multiple programming languages and ways to interpret them. Currently, Pig identifies groovy as a keyword and ships the required scriptengine (groovy-all) to interpret it.

```
Register 'test.groovy' using groovy as myfuncs;
```

The following syntax is also supported, where myfuncs is the namespace created for all the functions inside test.groovy.

```
register 'test.groovy' using org.apache.pig.scripting.groovy.GroovyScriptEngine as myfuncs;
```

A registered script can contain multiple UDFs. UDFs can be static or instance methods, an instance of the enclosing class will be created as needed. Only methods for which a return schema is defined will be available in Pig.

6.2 Return Types and Schemas

You have two ways to define the return schema, both use annotations:

@OutputSchema annotation - Defines the schema for a UDF in a format that Pig understands.

```
import org.apache.pig.builtin.OutputSchema;

class GroovyUDFs {
  @OutputSchema('x:long')
  long square(long x) {
    return x*x;
  }
}


outputSchema "t:(m:[], t:(name:chararray, age:int, gpa:double), b:{t:(name:chararray,
 age:int, gpa:double)})"
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

@OutputSchemaFunction annotation - Defines the name of a function which will return the schema at runtime according to the input schema.

```
import org.apache.pig.scripting.groovy.OutputSchemaFunction;

class GroovyUDFs {
  @OutputSchemaFunction('squareSchema')
  public static square(x) {
    return x * x;
  }

  public static squareSchema(input) {
    return input;
  }
}
```

Only methods annotated with either @OutputSchema or @OutputSchemaFunction will be exposed to Pig as UDFs. In the example above, squareSchema will not be available in Pig as a UDF.

6.3 Type Conversions

The data passed back and forth between Pig and Groovy goes through a conversion process. The following conversions rules are applied:

Pig to Groovy

- Tuple: groovy.lang.Tuple
- DataBag: groovy.lang.Tuple containing the bag's size and an iterator on its content
- org.joda.time.DateTime: org.joda.time.DateTime
- Map: java.util.Map
- int/long/float/double: as is
- chararray: String
- bytearray: byte[] (content is copied)
- boolean: boolean
- biginteger: BigInteger
- bigdecimal: BigDecimal
- null: null

Anything else raises an exception

Groovy to Pig

- Object[]: Tuple
- groovy.lang.Tuple: Tuple
- org.apache.pig.data.Tuple: Tuple
- org.apache.pig.data.DataBag: DataBag

Copyright © 2007 The Apache Software Foundation. All rights reserved.

- org.joda.time.DateTime: org.joda.time.DateTime
- java.util.Map: Map
- java.util.List: DataBag
- Byte/Short/Integer: int
- Long: long
- Float: float
- Double: double
- String: chararray
- byte[]: DataByteArray (content is copied)
- Boolean: boolean
- BigInteger: biginteger
- BigDecimal: bigdecimal
- null: null

Anything else raises an exception

6.4 Advanced Topics

You can also write Algebraic and Accumulator UDFs using Groovy. Both types of UDFs are declared using annotations, a single Groovy file can therefore enclose several Algebraic/ Accumulator UDFs, all mixed with regular UDFs.

Algebraic UDFs are declared using three annotations, @AlgebraicInitial, @AlgebraicIntermed and @AlgebraicFinal which are to annotate methods that correspond to the initial, intermed and final steps of an Algebraic UDF. Those annotations have a single parameter which is the name of the Algebraic UDF that will be available in Pig. The methods annotated with @AlgebraicInitial and @AlgebraicIntermed accept a Tuple as parameter and return a Tuple. The return type of the method annotated with @AlgebraicFinal will determine the return type of the Algebraic UDF. Here is an example of an Algebraic UDF named 'sum' defined in Groovy:

```
import org.apache.pig.scripting.groovy.AlgebraicInitial;
import org.apache.pig.scripting.groovy.AlgebraicIntermed;
import org.apache.pig.scripting.groovy.AlgebraicFinal;

class GroovyUDFs {
  @AlgebraicFinal('sum')
  public static long algFinal(Tuple t) {
    long x = 0;
    for (Object o: t[1]) {
      x = x + o;
    }
    return x;
  }
  @AlgebraicInitial('sum')
  public static Tuple algInitial(Tuple t) {
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
    long x = 0;
    for (Object o: t[1]) {
      x = x + o[0];
    }
    return [x];
  }
  @AlgebraicIntermed('sum')
  public static Tuple algIntermed(Tuple t) {
    long x = 0;
    for (Object o: t[1]) {
      x = x + o;
    }
    return [x];
  }
}
```

Similarly, Accumulator UDFs are declared using the three annotations @AccumulatorAccumulate, @AccumulatorGetValue and @AccumulatorCleanup which are to annotate methods that correspond to the methods accumulate, getValue and cleanup of a Java Accumulator UDF. Those annotations have a single parameter which is the name of the Accumulator UDF that will be available in Pig. The methods annotated with @AccumulatorAccumulate and @AccumulatorCleanup return void. The methods annotated with @AccumulatorGetValue and @AccumulatorCleanup take no parameters. The method annotated with @AccumulatorAccumulate takes a Tuple as parameter. The return schema of the Accumulator UDF is determined by the @OutputSchema or @OutputSchemaFunction annotation used on the method annotated with @AccumulatorGetValue. Note that even though a method annotated with @AccumulatorGetValue has an @OutputSchema or @OutputSchemaFunction annotation, it will not be available in Pig, only the Accumulator UDF to which it belongs will.

Since Accumulator UDFs maintain state, the methods annotated with the @AccumulatorXXX annotations cannot be static. A single instance of the enclosing class will be used when invoking them, thus enabling them to access a single state.

The following example defines an Accumulator UDF named 'sumacc':

```
import org.apache.pig.builtin.OutputSchema;
import org.apache.pig.scripting.groovy.AccumulatorAccumulate;
import org.apache.pig.scripting.groovy.AccumulatorGetValue;
import org.apache.pig.scripting.groovy.AccumulatorCleanup;

class GroovyUDFs {
  private int sum = 0;
  @AccumulatorAccumulate('sumacc')
  public void accuAccumulate(Tuple t) {
    for (Object o: t[1]) {
      sum += o[0]
    }
  }
  @AccumulatorGetValue('sumacc')
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
  @OutputSchema('sum: long')
  public long accuGetValue() {
    return this.sum;
  }
  @AccumulatorCleanup('sumacc')
  public void accuCleanup() {
    this.sum = 0L;
  }
}
```

7 Writing Python UDFs

Here Python UDFs means C Python UDFs. It uses python command line to run the Python UDFs. It is different than Jython, which relies on Jython library. Instead, it streams the data in and out of the python process. The implementation mechanism is completely different than Jython.

7.1 Registering the UDF

You can register a Python script as shown here.

```
Register 'test.py' using streaming_python as myfuncs;
```

The following syntax is also supported, where myfuncs is the namespace created for all the functions inside test.py.

```
register 'test.py' using org.apache.pig.scripting.streaming.python.PythonScriptEngine as
 myfuncs;
```

A typical test.py looks like this:

```
from pig_util import outputSchema

@outputSchema("as:int")
def square(num):
    if num == None:
        return None
    return ((num) * (num))

@outputSchema("word:chararray")
def concat(word):
    return word + word
```

The register statement above registers the Python functions defined in test.py in Pig's runtime within the defined namespace (myfuncs here). They can then be referred later on in the pig script as myfuncs.square(), myfuncs.concat(). An example usage is:

```
```

Copyright © 2007 The Apache Software Foundation. All rights reserved.

```
b = foreach a generate myfuncs.concat('hello', 'world'), myfuncs.square(3);
```

7.2 Decorators and Schemas

To annotate a Python script so that Pig can identify return types, use Python decorators to define output schema for the script UDF.

- outputSchema - Defines schema for a script UDF in a format that Pig understands and is able to parse.

When no decorator is specified, Pig assumes the output datatype as bytearray and converts the output generated by script function to bytearray. This is consistent with Pig's behavior in case of Java UDFs.

Sample Schema String - words:{(word:chararray)}, variable names inside a schema string are not used anywhere, they just make the syntax identifiable to the parser.

8 Piggy Bank

Piggy Bank is a place for Pig users to share the Java UDFs they have written for use with Pig. The functions are contributed "as-is." If you find a bug in a function, take the time to fix it and contribute the fix to Piggy Bank. If you don't find the UDF you need, take the time to write and contribute the function to Piggy Bank.

Note: Piggy Bank currently supports Java UDFs. Support for Jython and JavaScript UDFs will be added at a later date.

8.1 Accessing Functions

The Piggy Bank functions are currently distributed in source form. Users are required to checkout the code and build the package themselves. No binary distributions or nightly builds are available at this time.

To build a jar file that contains all available UDFs, follow these steps:

- Checkout UDF code: `svn co http://svn.apache.org/repos/asf/pig/trunk/contrib/piggybank`
- Add pig.jar to your ClassPath: `export CLASSPATH=$CLASSPATH:/path/to/pig.jar`
- Build the jar file: from directory`trunk/contrib/piggybank/java` run `ant`. This will generate `piggybank.jar` in the same directory.

To obtain `javadoc` description of the functions run `ant javadoc` from directory `trunk/contrib/piggybank/java`. The documentation is generate in directory `trunk/contrib/piggybank/java/build/javadoc`.

Copyright © 2007 The Apache Software Foundation. All rights reserved.

To use a function, you need to determine which package it belongs to. The top level packages correspond to the function type and currently are:

- org.apache.pig.piggybank.comparison - for custom comparator used by ORDER operator
- org.apache.pig.piggybank.evaluation - for eval functions like aggregates and column transformations
- org.apache.pig.piggybank.filtering - for functions used in FILTER operator
- org.apache.pig.piggybank.grouping - for grouping functions
- org.apache.pig.piggybank.storage - for load/store functions

(The exact package of the function can be seen in the javadocs or by navigating the source tree.)

For example, to use the UPPER function:

```
REGISTER /public/share/pig/contrib/piggybank/java/piggybank.jar ;
TweetsInaug = FILTER Tweets BY org.apache.pig.piggybank.evaluation.string.UPPER(text)
    MATCHES '.*(INAUG|OBAMA|BIDEN|CHENEY|BUSH).*' ;
STORE TweetsInaug INTO 'meta/inaug/tweets_inaug' ;
```

8.2 Contributing Functions

To contribute a Java function that you have written, do the following:

1. Check the existing javadoc to make sure that the function does not already exist as described in Accessing Functions.
2. Checkout the UDF code as described in Accessing Functions.
3. Place your java code in the directory that makes sense for your function. The directory structure currently has two levels: (1) function type, as described in Accessing Functions, and (2) function subtype, for some of the types (like math or string for eval functions). If you think your function requires a new subtype, feel free to add one.
4. Make sure that your function is well documented and uses the javadoc style of documentation.
5. Make sure that your code follows Pig coding conventions described in How to Contribute to Pig.
6. Make sure that for each function, you add a corresponding test class in the test part of the tree.
7. Submit your patch following the process described in How to Contribute to Pig.

Copyright © 2007 The Apache Software Foundation. All rights reserved.