Pig UDF Manual

Table of contents

1 Overview....................2
2 Eval Functions....................2
3 Load/Store Functions....................17
4 Comparison Functions....................20
5 Builtin Functions and Function Repositories....................20
6 Advanced Topics....................21

1. Overview

Pig provides extensive support for user-defined functions (UDFs) as a way to specify custom processing. Functions can be a part of almost every operator in Pig. This document describes how to use existing functions as well as how to write your own functions.

2. Eval Functions

2.1. 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 Hadoop mode. For more information on how to run Pig, please see the PigTutorial.

```
java -cp pig.jar org.apache.pig.Main -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 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.

2.2. How to Write a Simple Eval Function

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

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

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

The first line 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 16.) Note that `WrappedIOException` is a helper class to convert the actual exception to an

IOException.

Also, note that lines 10-11 check if the input data is null or empty and if so returns null.

The actual function implementation is on lines 13-14 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/hadoop/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.

2.3. 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;
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.

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();) {
            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 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.

2.4. 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);
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.backend.executionengine.ExecException;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.DataType;
import org.apache.pig.impl.util.WrappedIOException;

public class IsEmpty extends FilterFunc {
    public Boolean exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        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{
                throw new IOException("Cannot test a " +
                    DataType.findTypeName(values) + " for emptiness.");
            }
        } catch (ExecException ee) {
            throw WrappedIOException.wrap("Caught exception processing
input row ", ee);
        }
    }
}
```

2.5. Pig 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
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;
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
        }
    }
}
```

2.6. Schema

The latest version of Pig uses type information for validation and performance. It is important for UDFs to participate in type propagation. Until now, our UDFs made no effort to communicate their output schema to Pig. This is because, most of the time, Pig can 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.

```
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}
```

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 BYTEARRAY =  50;
    public static final byte CHARARRAY =  55;
    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:

```

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();
    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;
         }
    }
}
```

As you can see, 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=).

2.7. 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.
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.

Pig provides a helper class `WrappedIOException`. The intent here is to allow you to convert any exception into `IOException`. Its usage can be seen in the `UPPER` function in our first example.

2.8. 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.util.WrappedIOException;
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 WrappedIOException.wrap("Caught exception processing
input row ", e);
        }
        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.impl.util.WrappedIOException;
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 WrappedIOException.wrap("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`. 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.

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;
```

2.9. 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.

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` function 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{
                        reporter.progress();
                        String str = (String)input.get(0);
                        return str.toUpperCase();
                }catch(Exception e){
                    throw WrappedIOException.wrap("Caught exception
processing input row ", e);
                }
        }
}
```

3. Load/Store Functions

These user-defined functions 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.

3.1. Load Functions

Every load function needs to implement the `LoadFunc` interface. An abbreviated version is shown below. The full definition can be seen here.

```
public interface LoadFunc {
    public void bindTo(String fileName, BufferedPositionedInputStream is,
long offset, long end) throws IOException;
    public Tuple getNext() throws IOException;
    // conversion functions
    public Integer bytesToInteger(byte[] b) throws IOException;
    public Long bytesToLong(byte[] b) throws IOException;
    ......
    public void fieldsToRead(Schema schema);
    public Schema determineSchema(String fileName, ExecType execType,
DataStorage storage) throws IOException;
```

The `bindTo` function is called once by each Pig task before it starts processing data. It is intended to connect the function to its input. It provides the following information:

- `fileName` - The name of the file from which the data is read. Not used most of the time
- `is` - The input stream from which the data is read. It is already positioned at the place where the function needs to start reading
- `offset` - The offset into the stream from which to read. It is equivalent to `is.getPosition()` and not strictly needed
- `end` - The position of the last byte that should be read by the function.

In the Hadoop world, the input data is treated as a continuous stream of bytes. A `slicer`, discussed in the Advanced Topics section, is used to split the data into chunks with each chunk going to a particular task for processing. This chunk is what `bindTo` provides to the UDF. Note that unless you use a custom slicer, the default slicer is not aware of tuple boundaries. This means that the chunk you get can start and end in the middle of a particular tuple. One common approach is to skip the first partial tuple and continue past the end position to finish processing a tuple. This is what `PigStorage` does as the example later in this section shows.

The `getNext` function reads the input stream and constructs the next tuple. It returns `null` when it is done with processing and throws an `IOException` if it fails to process an input tuple.

Next is a bunch of conversion routines that convert data from `bytearray` to the requested type. This requires further explanation. By default, we would like the loader to do as little per-tuple processing as possible. This is because many tuples can be thrown out during filtering or joins. Also, many fields might not get used because they get projected out. If the data needs to be converted into another form, we would like this conversion to happen as late as possible. The majority of the loaders should return the data as bytearrays and the Pig will request a conversion from bytearray to the actual type when needed. Let's looks at the example below:

```
A = load 'student_data' using PigStorage() as (name: chararray, age: int,
gpa: float);
B = filter A by age >25;
C = foreach B generate name;
dump C;
```

In this query, only `age` needs to be converted to its actual type (=int=) right away. `name` only needs to be converted in the next step of processing where the data is likely to be much smaller. `gpa` is not used at all and will never need to be converted.

This is the main reason for Pig to separate the reading of the data (which can happen immediately) from the converting of the data (to the right type, which can happen later). For

ASCII data, Pig provides `Utf8StorageConverter` that your loader class can extend and will take care of all the conversion routines. The code for it can be found here.

Note that conversion rutines should return null values for data that can't be converted to the specified type.

Loaders that work with binary data like `BinStorage` are not going to use this model. Instead, they will produce objects of the appropriate types. However, they might still need to define conversion routines in case some of the fields in a tuple are of type `bytearray`.

`fieldsToRead` is reserved for future use and should be left empty.

The `determineSchema` function must be implemented by loaders that return real data types rather than `bytearray` fields. Other loaders should just return `null`. The idea here is that Pig needs to know the actual types it will be getting; Pig will call `determineSchema` on the client side to get this information. The function is provided as a way to sample the data to determine its schema.

Here is the example of the function implemented by =BinStorage=:

```
public Schema determineSchema(String fileName, ExecType execType,
DataStorage storage) throws IOException {
    InputStream is = FileLocalizer.open(fileName, execType, storage);
    bindTo(fileName, new BufferedPositionedInputStream(is), 0,
Long.MAX_VALUE);
        // get the first record from the input file and figure out the
schema
        Tuple t = getNext();
        if(t == null) return null;
        int numFields = t.size();
        Schema s = new Schema();
        for (int i = 0; i  numFields; i++) {
            try {
                s.add(DataType.determineFieldSchema(t.get(i)));
            } catch (Exception e) {
                throw WrappedIOException.wrap(e);
            }
        }
        return s;
    }
```

Note that this approach assumes that the data has a uniform schema. The function needs to make sure that the data it produces conforms to the schema returned by `determineSchema`, otherwise the processing will fail. This means producing the right number of fields in the tuple (dropping fields or emitting null values if needed) and producing fields of the right type (again emitting null values as needed).

For complete examples, see BinStorage and PigStorage.

3.2. Store Functions

All store functions need to implement the `StoreFunc` interface:

```
public interface StoreFunc {
    public abstract void bindTo(OutputStream os) throws IOException;
    public abstract void putNext(Tuple f) throws IOException;
    public abstract void finish() throws IOException;
}
```

The `bindTo` method is called in the beginning of the processing to connect the store function to the output stream it will write to. The `putNext` method is called for every tuple to be stored and is responsible for writing the tuple into the output. The `finish` function is called at the end of the processing to do all needed cleanup like flushing the output stream.

Here is an example of a simple store function that writes data as a string returned from the `toString` function.

```
public class StringStore implements StoreFunc {
    OutputStream os;
    private byte recordDel = (byte)'\n';
    public void bindTo(OutputStream os) throws IOException
    {
        this.os = os;
    }
    public void putNext(Tuple t) throws IOException
    {
        os.write((t.toString() + (char)this.recordDel).getBytes("utf8"));
    }
    public void finish() throws IOException
    {
         os.flush();
    }
}
```

4. Comparison Functions

Comparison UDFs are mostly obsolete now. They were added to the language because, at that time, the `ORDER` operator had two significant shortcomings. First, it did not allow descending order and, second, it only supported alphanumeric order.

The latest version of Pig solves both of these issues. The pointer to the original documentation is provided here for completeness.

5. Builtin Functions and Function Repositories

Pig comes with a set of builtin in functions. (NEED LINK) Two main properties differentiate builtin functions from UDFs. First, they don't need to be registered because Pig knows where they are. Second, they don't need to be qualified when used because Pig knows where to find them.

In addition to builtins, Pig hosts a UDF repository called `piggybank` that allows users to share UDFs that they have written. The details are described in PiggyBank.

6. Advanced Topics

6.1. 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.

6.2. Schemas

One request from users is to have the ability to examine the input schema of the data before processing the data. 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. The current answer is that there is now way to do this. This is something we would like to support in the future.

6.3. Custom Slicer

Sometimes a `LoadFunc` needs more control over how input is chopped up or even found.

Here are some scenarios that call for a custom slicer:

- Input needs to be chopped up differently than on block boundaries. (Perhaps you want every 1M instead of every 128M. Or, you may want to process in big 1G chunks.)
- Input comes from a source outside of HDFS. (Perhaps you are reading from a database.)
- There are locality preferences for processing the data that is more than simple HDFS locality.
- Extra information needs to be passed from the client machine to the `LoadFunc` instances running remotely.

All of these scenarios are addressed by slicers. There are two parts to the slicing framework: `Slicer`, the class that creates slices, and `Slice`, the class that represents a particular piece of the input. Slicing kicks in when Pig sees that the `LoadFunc` implements the `Slicer` interface.

6.3.1. Slicer

The slicer has two basic functions: validate input and slice up the input. Both of these methods will be called on the client machine.

```
public interface Slicer {
    void validate(DataStorage store, String location) throws IOException;
   Slice[] slice(DataStorage store, String location) throws IOException;
}
```

6.3.2. Slice

Each slice describes a unit of work and will correspond to a map task in Hadoop.

```
public interface Slice extends Serializable {
    String[] getLocations();
    void init(DataStorage store) throws IOException;
    long getStart();
    long getLength();
    void close() throws IOException;
    long getPos() throws IOException;
    float getProgress() throws IOException;
    boolean next(Tuple value) throws IOException;
}
```

Only one of the methods is used for scheduling: `getLocations()`. This method allows the implementor to give hints to Pig about where the task should be run. It is only a hint. If things are busy, the task may get scheduled elsewhere.

The rest of the `Slice` methods are used to read records on the processing nodes. `init` is called right after the `Slice` object is deserialized and `close` is called after the last record has been read. The Pig runtime will read records from the `Slice` until `getPos()` exceeds `getLength()`. Because `Slice` implements serializable, `Slicer` can encode information in the `Slice` that will later be available when the task is run.

6.3.3. Example

This example shows a simple `Slicer` that gets a count from the input stream and generates that number of `Slice` s.

```
public class RangeSlicer implements Slicer, LoadFunc {
    /**
     * Expects location to be a Stringified integer, and makes
     * Integer.parseInt(location) slices. Each slice generates a single
value,
     * its index in the sequence of slices.
     */
    public Slice[] slice (DataStorage store, String location) throws
IOException {
        // Note: validate has already made sure that location is an integer
        int numslices = Integer.parseInt(location);
        Slice[] slices = new Slice[numslices];
        for (int i = 0; i  slices.length; i++) {
            slices[i] = new SingleValueSlice(i);
        }
        return slices;
    }
    public void validate(DataStorage store, String location) throws
IOException {
        try {
            Integer.parseInt(location);
        } catch (NumberFormatException nfe) {
            throw new IOException(nfe.getMessage());
        }
    }
    /**
     * A Slice that returns a single value from next.
     */
    public static class SingleValueSlice implements Slice {
        // note this value is set by the Slicer and will get serialized and
deserialized at the remote processing node
        public int val;
        // since we just have a single value, we can use a boolean rather
than a counter
        private transient boolean read;
        public SingleValueSlice (int value) {
            this.val = value;
        }
        public void close () throws IOException {}
        public long getLength () { return 1; }
        public String[] getLocations () { return new String[0]; }
        public long getStart() { return 0; }
        public long getPos () throws IOException { return read ? 1 : 0; }
        public float getProgress () throws IOException { return read ? 1 :
0; }
        public void init (DataStorage store) throws IOException {}
        public boolean next (Tuple value) throws IOException {
            if (!read) {
                value.appendField(new DataAtom(val));
                read = true;
                return true;
            }
            return false;
        }
        private static final long serialVersionUID = 1L;
    }
}
```

You can invoke the `RangeSlicer` class with the following Pig Latin statement:

```

LOAD '27' USING RangeSlicer();
```