So far,

Monday, February 28, 2011
4:43 PM

So far…

We've discovered the power of Pig
As a "generalization" of SQL
That acts on large data spaces.
But I claimed before that the **real power** is in **extending Pig** using Java functions.
How does one extend Pig?
Why would one extend Pig?

Types in Pig and Java

Monday, February 28, 2011
1:47 PM

Pig Type	Java Class
bytearray	DataByteArray
chararray	String
int	Integer
long	Long
float	Float
double	Double
tuple	Tuple
bag	DataBag
map	Map<Object, Object>

Pasted from <http://wiki.apache.org/pig/UDFManual>

Understanding UDF I/O

- Every function input is a single Tuple
  - Tuples are **variable-size**.
  - Tuple elements exhibit **hidden polymorphism**.
  - Return value of a UDF depends upon its **kind**.
- Return values can be builtin types, Tuple, or DataBag.
  - Builtin types are constructed normally.
  - Both tuples and bags are created via **factories**.

Input polymorphism

Tuesday, March 01, 2011
8:36 AM

One function argument: `Tuple input`

Varying `Tuple` size.

Tuple elements can be accessed by number n:

```
Object thing = input.get(n);
```

Tuple element iterators access variable-size tuples:

```
for (Iterator<Tuple> it = input.iterator();
      it.hasNext();) {
   Object thing = it.next();
   // do something with thing,
   // which is type-polymorphic
}
```

Access hidden type information via `instanceof`

```
If (thing instanceof sometype) {
   Sometype foo = (sometype) thing;
   // do something with foo
}
```

Only types that can arise in input are Pig equivalents!

Creating output

Tuesday, March 01, 2011
9:15 AM

Creating output

It's obvious how to create output in the primitive types. How does one create Tuple(tuple) or DataBag(bag) output?

Tuples are created via a **factory**:

```
Tuple t = TupleFactory.getInstance().newTuple(element1,
element2, ...);
```

Why? The stated reason is so users can change **how tuples are stored.**

Bags are created via a **factory:**

```
TupleFactory mTupleFactory = TupleFactory.getInstance();
BagFactory mBagFactory = BagFactory.getInstance();
DataBag output = mBagFactory.newDefaultBag();
...
output.add(mTupleFactory.newTuple(...);
```

Pasted from <http://wiki.apache.org/pig/UDFManual>

A DataBag can only contain Tuples (that may, of course, contain DataBags, Tuples, or primitive types as elements).

UDF structure

Tuesday, March 01, 2011
8:50 AM

General UDF structure

UDFs are represented as **classes.**

The actual function call is the **`eval` method** of the class.

E.g.

`FooFunc(input)` in Pig is represented as

```
Class FooFunc {
    some-return-type eval(Tuple input) {
        // function body here
    }
}
```

User-Defined Functions

Monday, February 28, 2011
1:15 PM

Several kinds of user-defined functions:

Filter functions: input a Pig value, return a boolean value to be used in conditions.

Eval functions: input a Pig value, return a Pig result.

Algebraic functions: full map/reduce operation on an (inner) input bag.

How functions are used

Wednesday, March 02, 2011
3:52 PM

Kinds of functions differ in how they are used:

- Filter functions are used as logical conditions in a FILTER statement.
- Eval functions are used in FOREACH-GENERATE statements.
- Algebraic functions act on inner bags in a FOREACH-GENERATE statement.

Filter function example: IsEmpty

Monday, February 28, 2011
1:42 PM

```
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);
        }
    }
}
```

Pasted from <http://wiki.apache.org/pig/UDFManual>

Some attributes

Input polymorphism

```
Object values = input.get(0);
if (values instanceof DataBag) {
   // do something with (DataBag)values
} else if (values instanceof Tuple) {
   // do something with (Tuple)values
}
```

Variable argument lists:

`Input.get(0)`: first element of input Tuple.
`Input.get(1)`: second element.
…

Eval functions

Monday, February 28, 2011
1:15 PM

```
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);
        }
    }
}
```

Pasted from <http://wiki.apache.org/pig/UDFManual>

Some comments:

- The function is implemented as a class that **extends `EvalFunc<String>`**
- Where `String` is the **return type of the function.**
- And the function body is the **`eval` method.**

Aggregate/Algebraic functions

Tuesday, March 01, 2011
9:08 AM

What is an aggregate (algebraic) function?

- In principle, can write everything using Eval functions.
- But this can incredibly inefficient when the thing to be acted upon is a large bag (e.g., generated by GROUP-BY).

Aggregate (algebraic) functions

- Act on inner bags.
- Expose full map/reduce capabilities.

Recall: basic Map/Reduce

Tuesday, March 01, 2011
9:39 AM

Recall:

- The **map** step does something to **each element** in a dataset.
- The **combine** step takes the **output of several maps** and produces **one output** from them (of the same type).
- The **reduce** step produces final output.

In Pig

- The **map** step acts on **individual inner bag elements**, and returns a **Tuple.**
- The **combine** step inputs a **Tuple** (of partial map results) and outputs a **Tuple**. (Sometimes, it just returns its input).
- The **reduce** step takes a **Tuple** as input and produces the desired output type (e.g., a Long).

The Algebraic interface

Monday, February 28, 2011
1:31 PM

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

Pasted from <http://wiki.apache.org/pig/UDFManual>

getInitial: return the **name of a class** used in the **map** step.
getIntermed: return the **name of a class** used in the **combine** step.
getFinal: return the **name of a class** used in the **reduce** step.
All classses must extend EvalFunc.

Implementing an algebraic function

Tuesday, March 01, 2011
9:45 AM

Implementing an algebraic function:

```
// map step
public String getInitial() {return
Initial.class.getName();}
static public class Initial extends EvalFunc<Tuple> {
   public Tuple exec(Tuple input) {
      ...
   }
}
// combine step
public String getIntermed() {return
Intermed.class.getName();}
static public class Intermed extends EvalFunc<Tuple> {
   public Tuple exec(Tuple input) throws IOException {
      ...
   }
}
// reduce step
public String getFinal() {return
Final.class.getName();}static public class Final
extends EvalFunc<Long> {
   public Long exec(Tuple input) {
   ...
   }
}
```

Note: I'm fairly sure that class Final in COUNT example in UDF manual is incorrect. It says to return Tuple. It seems that it should return Long. There is no automatic conversion from Long to Tuple, so the way it is written, it will not compile.

Aggregate function example: COUNT

Monday, February 28, 2011
1:18 PM

```
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();}
    // map step
    static public class Initial extends EvalFunc<Tuple> {
        public Tuple exec(Tuple input) throws IOException {
            return TupleFactory.getInstance().newTuple(count(input));
        }
    }
    // combine step
    static public class Intermed extends EvalFunc<Tuple> {
        public Tuple exec(Tuple input) throws IOException {
            return TupleFactory.getInstance().newTuple(sum(input));
        }
    }
    // reduce step
    static public class Final extends EvalFunc<Long> {
        public Long 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;
        // deal with variable-size tuples
        for (Iterator<Tuple> it = values.iterator(); it.hasNext();) {
            Tuple t = it.next();
            sum += (Long)t.get(0);
        }
        return sum;
    }
}
```

Pasted from <http://wiki.apache.org/pig/UDFManual>

Some attributes:

Use of a factory to create new instances:

```
TupleFactory.getInstance().newTuple(count(input))
```

Iterators to iterate over variable-size tuples:

```
for (Iterator<Tuple> it = values.iterator(); it.hasNext();) {
    Tuple t = it.next();
    sum += (Long)t.get(0);
}
```

Why is it called "Algebraic"?

Wednesday, March 02, 2011
4:53 PM

Why "Algebraic"?

Because it can be manipulated in pieces (map separate from reduce) in optimizations.

Because it conforms to the tree manipulations (a matter of computer algebra) that allow optimization.

Why extend Pig?

- Parsing and generators: transform unstructured input data into structured form.
- Create custom Map/Reduce processes that are not implied by queries
- Speed up execution of complex queries.
- Report generation: create reports in human-readable form.