BIG DATA ANALYTICS

Apache Pig

User-Defined Functions

https://www.youtube.com/c/RASINENIMADANAMOHANA

User-Defined Functions: *Outline*

- A Filter UDF
- An Eval UDF
- A Load UDF

User-Defined Functions

- Pig's designers realized that the ability to plug in custom code is crucial for all but the most trivial data processing jobs. For this reason, they made it easy to define and use user-defined functions.
- In this only Java UDFs are included, but be aware that you can also write UDFs in Python, JavaScript, Ruby, or Groovy, all of which are run using the Java Scripting API.

User-Defined Functions: *A Filter UDF*

- Let's demonstrate by writing a filter function for filtering out weather records that do not have a temperature quality reading of satisfactory (or better).
- The idea is to change this line:

```
filtered_records = FILTER records BY temperature !=
9999 AND quality IN (0, 1, 4, 5, 9);
```

to

```
filtered_records = FILTER records BY temperature !=
9999 AND isGood(quality);
```

User-Defined Functions: *A Filter UDF*

- This achieves two things: it makes the Pig script a little more concise, and it encapsulates the logic in one place so that it can be easily reused in other scripts.
- If we were just writing an ad hoc query, we probably wouldn't bother to write a UDF. It's when we start doing the same kind of processing over and over again that we see opportunities for reusable UDFs.
- **`Filter UDFs`** are all subclasses of FilterFunc, which itself is a subclass of `EvalFunc`.
- `EvalFunc` looks like the following class:

```
public abstract class EvalFunc<T> {

   public abstract T exec(Tuple input) throws IOException;

}
```

User-Defined Functions: *A Filter UDF*

```
public abstract class EvalFunc<T> {

   public abstract T exec(Tuple input) throws IOException;

}
```

`EvalFunc's` only abstract method, `exec()`, takes a tuple and returns a single value, the (parameterized) type `T`. The fields in the input tuple consist of the expressions passed to the function—in this case, a single integer.

For `FilterFunc`, `T` is `Boolean`, so the method should return `true` only for those tuples that should not be filtered out.

For the quality filter, we write a class, `IsGoodQuality`, that extends `FilterFunc` and implements the `exec()` method.

User-Defined Functions: *A Filter UDF*

Example. *A FilterFunc UDF to remove records with unsatisfactory temperature quality readings:*

```
package com.hadoopbook.pig;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.apache.pig.FilterFunc;

import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataType;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.logicalLayer.FrontendException;

public class IsGoodQuality extends FilterFunc {

  @Override
  public Boolean exec(Tuple tuple) throws IOException {
    if (tuple == null || tuple.size() == 0) {
      return false;
    }
    try {
      Object object = tuple.get(0);
      if (object == null) {
        return false;
      }
      int i = (Integer) object;
      return i == 0 || i == 1 || i == 4 || i == 5 || i == 9;
    } catch (ExecException e) {
      throw new IOException(e);
    }
  }

}
```

User-Defined Functions: *A Filter UDF*

Example. *A FilterFunc UDF to remove records with unsatisfactory temperature quality readings:* cont'd.

To use the new function, we first compile it and package it in a `JAR file`. Then we tell Pig about the `JAR file` with the `REGISTER` operator, which is given the local path to the filename (and is not enclosed in quotes):

```
grunt> REGISTER pig-examples.jar;
```

Finally, we can invoke the function:

```
grunt> filtered_records = FILTER records BY temperature
!= 9999 AND IsGoodQuality(quality);
```

User-Defined Functions: *A Filter UDF*

Example. *A FilterFunc UDF to remove records with unsatisfactory temperature quality readings:* cont'd.

```
Input:
------
hduser072@adminitlab5-OptiPlex-3050:~/Desktop/RMMBDAPIG$ cat sample.txt
1950 0 1
1950 22 1
1950 -11 1
1949 111 1
1949 78 1
1947 42 6
1947 45 1

Pig Execution Steps:
--------------------
hduser072@adminitlab5-OptiPlex-3050:~/Desktop/RMMBDAPIG$ pig -x local
grunt>
grunt> REGISTER pig-examples.jar;
grunt> DEFINE isGood IsGoodQuality();
grunt> records = LOAD 'sample.txt' USING PigStorage(' ') AS (year:chararray, temperature:int,
quality:int);
grunt> filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
grunt> grouped_records = GROUP filtered_records BY year;
grunt> max_temp = FOREACH grouped_records GENERATE group,MAX(filtered_records.temperature);
grunt> max_temp = FOREACH grouped_records GENERATE group,MAX(filtered_records.temperature);
```

User-Defined Functions: *An Eval UDF*

- Writing an `eval` function is a small step up from writing a filter function.
- Consider the UDF in the following example, which trims the leading and trailing whitespace from `chararray` values using the `trim()` method on `java.lang.String`

Example. *An EvalFunc UDF to trim leading and trailing whitespace from chararray values:*

```
public class Trim extends PrimitiveEvalFunc<String, String> {
  @Override
  public String exec(String input) {
    return input.trim();
  }
}
```

User-Defined Functions: *An Eval UDF*

- In this case, we have taken advantage of `PrimitiveEvalFunc`, which is a specialization of `EvalFunc` for when the input is a single primitive (atomic) type.
- For the Trim UDF, the input and output types are both of type `String`.
- In general, when we write an `eval` function, we need to consider what the output's schema looks like.
- In the following statement, the schema of `B` is determined by the function `udf`:

```
B = FOREACH A GENERATE udf($0);
```

User-Defined Functions: *An Eval UDF*

- In the following statement, the schema of `B` is determined by the function `udf`:

```
B = FOREACH A GENERATE udf($0);
```

If `udf` creates tuples with scalar fields, then Pig can determine `B`'s schema through reflection.

For complex types such as bags, tuples, or maps, Pig needs more help, and we should implement the `outputSchema()` method to give Pig the information about the output schema.

User-Defined Functions: *An Eval UDF*

- The **`Trim UDF`** returns a string, which Pig translates as a `chararray`, as can be seen from the following session:

```
grunt> DUMP A;
( pomegranate)
(banana )
(apple)
( lychee )
grunt> DESCRIBE A;
A: {fruit: chararray}
grunt> B = FOREACH A GENERATE Trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
grunt> DESCRIBE B;
B: {chararray}
```

User-Defined Functions: *An Eval UDF*

A has `chararray` fields that have leading and trailing spaces. We create **B** from **A** by applying the **Trim** function to the first field in `A` (named fruit). `B`'s fields are correctly inferred to be of type `chararray`.

User-Defined Functions: *A Load UDF*

- We will demonstrate a custom load function that can read plain-text column ranges as fields, very much like the Unix `cut` command.
- It is used as follows:

```
grunt> records = LOAD 'input/ncdc/micro/sample.txt'
>> USING com.hadoopbook.pig.CutLoadFunc('16-19,88-92,93-93')
>> AS (year:int, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

User-Defined Functions: *A Load UDF*

- The string passed to `CutLoadFunc` is the column specification; each comma-separated range defines a field, which is assigned a name and type in the `AS` clause.
- The implementation of `CutLoadFunc`, is shown in below example.

User-Defined Functions: *A Load UDF*

Example: *A LoadFunc UDF to load tuple fields as column ranges*

```
public class CutLoadFunc extends LoadFunc {

  private static final Log LOG = LogFactory.getLog(CutLoadFunc.class);

  private final List<Range> ranges;
  private final TupleFactory tupleFactory = TupleFactory.getInstance();
  private RecordReader reader;

  public CutLoadFunc(String cutPattern) {
    ranges = Range.parse(cutPattern);
  }

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

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

User-Defined Functions: *A Load UDF*

Example: *A LoadFunc UDF to load tuple fields as column ranges* cont'd.

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

@Override
public Tuple getNext() throws IOException {
  try {
    if (!reader.nextKeyValue()) {
      return null;
    }
    Text value = (Text) reader.getCurrentValue();
    String line = value.toString();
    Tuple tuple = tupleFactory.newTuple(ranges.size());
    for (int i = 0; i < ranges.size(); i++) {
      Range range = ranges.get(i);
      if (range.getEnd() > line.length()) {
        LOG.warn(String.format(
            "Range end (%s) is longer than line length (%s)",
            range.getEnd(), line.length()));
```

User-Defined Functions: *A Load UDF*

Example: *A LoadFunc UDF to load tuple fields as column ranges* cont'd.

```
            continue;
          }
          tuple.set(i, new DataByteArray(range.getSubstring(line)));
        }
        return tuple;
      } catch (InterruptedException e) {
        throw new ExecException(e);
      }
    }
  }
```

- In Pig, like in Hadoop, data loading takes place before the mapper runs, so it is important that the input can be split into portions that are handled independently by each mapper.
- A `LoadFunc` will typically use an existing underlying Hadoop `InputFormat` to create records, with the `LoadFunc` providing the logic for turning the records into Pig tuples.