An introduction (and random notes) to Cascading for Hadoop

Guillaume Theoret

September 19, 2008

Contents

1 Introduction

Cascading is a Java application framework that sits atop the Hadoop distributed filesystem that allows you to more easily write scripts to access and manipulate data inside Hadoop. By simplifying the data model and allowing you to work with streams and operations on those streams, scripts written with Cascading are much simpler to reason about and much quicker to write.

2 Vocabulary

2.1 Input and Output

2.1.1 Streams

Streams are used for input and output. Streams in Cascading are called Taps. Two types of Taps are available:

- Source - A source tap is read from and acted upon. Actions on source taps result in pipes.
- Sink - A sink tap is a location to be written to. A Sink tap can later serve as a Source in the same script. (This is where much of Cascading's power comes from.)

2.1.2 Tuples

A stream is composed of a series of Tuples. Tuples are sets of ordered data. Ex: ["John", "Doe", 39]

2.1.3 Pipes

When an operation is executed upon a Tap, the result is a Pipe. Operations can also be executed upon Pipes. When operations are successively executed on Pipes, the result is called a Pipe Assembly.

i.e: Pipes can use other Pipes as input, thereby wrapping themselves into a series of operations. (See Example for clarification)

2.1.4 Flows

A Flow is a combination of a Source, a Sink and Pipe Assemblies. When all three are combined, a Flow is created.

Figure 1 represents a Flow that reads from a distributed filesystem such as Hadoop, creates a Pipe and writes to a local filesystem.

2.1.5 Cascades

A Cascade is a series of Flows.

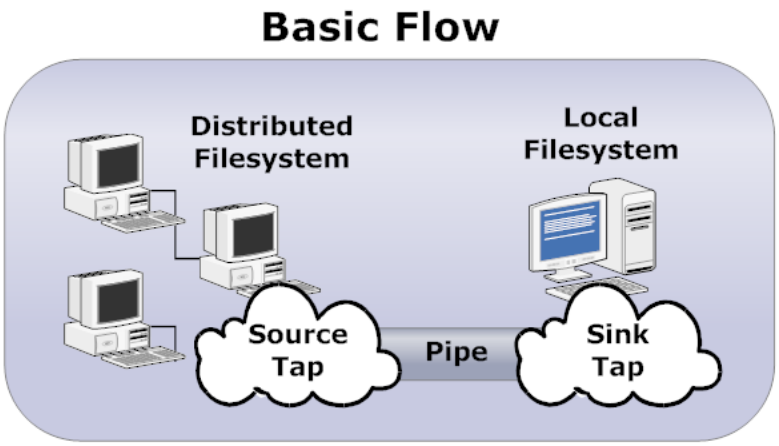

Figure 1: A Basic Flow

2.2 Stream Manipulation

Different manipulators are available to streams. They are grouped into two main categories: manipulators that act on every stream item and manipulators that act on stream groups.

2.2.1 Scalar

Scalar functions and filters are functions that are applied to every Tuple in the stream. They can be applied to a Pipe with the Each object. Examples of pre-defined scalar functions:

- FieldFormatter - Formats the values in a Tuple with a given format and stuffs the result into a new field.
- FieldJoiner - Joins the values in a Tuple with a given delimiter and stuffs the result into a new field.

You can define your own functions by implementing the Function interface.

Examples of pre-defined scalar filters:

- RegexFilter - Filters each Tuple of the stream based on whether any Tuple value matches a given Regular Expression.

- FilterNull - For every Tuple value, if a null value is encountered, the current Tuple will be filtered out.

You can define your own filters by extending BaseOperation and implementing the Filter interface.

Scalar streams can be split into groups with the Group object. You can group on one or many fields and can have a secondary sort on one or many fields. A Group object returns a Pipe containing a group stream or a tuple stream, depending on what comes next (Each or Every).

2.2.2 Aggregate

Aggregate functions are functions that are applied to a group stream of Tuples and generally return one result per group. You can apply an aggregate function by creating a new Every. Some pre-defined aggregate functions include:

- Count - Count the number of items in the group and keep the result as an extra field in the Tuple
- Max - Compare each item in the group and keep the maximum as an extra field in the Tuple
- Sum - Add up the values of a Field for each group and keep the result as an extra field in the Tuple

Aggregate functions return pipes containing value streams.

3 Program Flow

A typical program executes the following steps:

1. Define a source Tap
2. Define operations to execute on Tap which result in a Pipe (or a Pipe Assembly if multiple Pipes are wrapped together(
3. Define a sink Tap
4. Combine source, sink and operations into a Flow

This combination creates a single Flow. Flows can be combined one after the other into Cascades or executed immediately.

An example flow can be seen in Figure 2. This Flow represents a script that counts the number of occurences of a word.

First, all null values are filtered out of the stream. Next, each word is grouped and finally, the number of items is found for each group. The results are sent to a Sink that writes to the local filesystem.

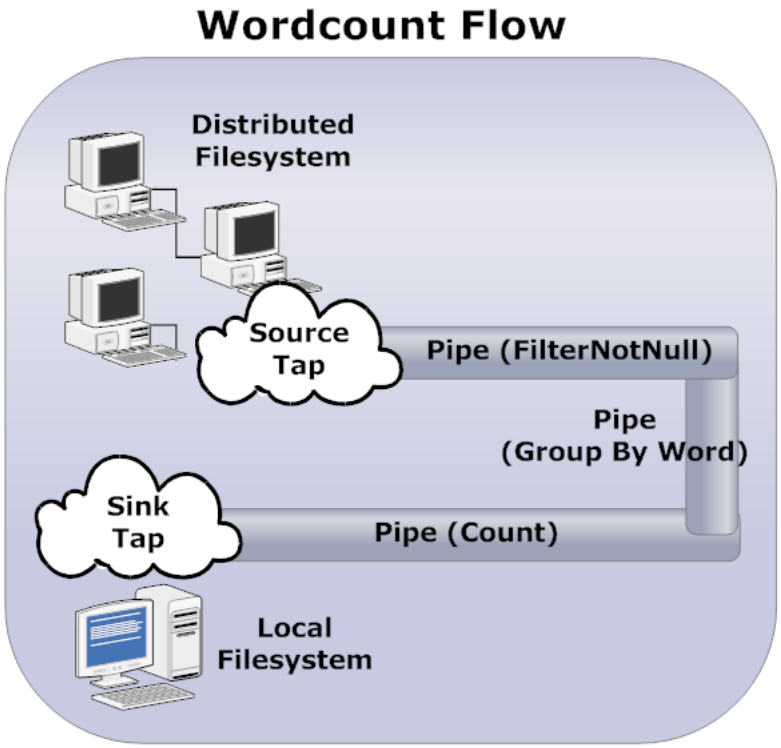

Figure 2: A Word Counting Script Flow

4 Example

Imagine we have a tab seperated file of log actions. The log file fields available (some will be unused) are:

- member id
- member gender
- member age
- requested action
- given action
- timestamp

We want to parse the log to find the most common things someone does before upgrading. Perhaps they do a search, view a profile, try to send a message (and get blocked) and then upgrade. This path would be represented as profile->search->mail. We want the number of occurences for each path found.

When writing a cascading script you first need to come up with a strategy for how to achieve the desired result. In our case we'll be following a series of transformations:

1. Group by member id and do a secondary descending sort by timestamp
2. Apply a custom filter that iterates over the members of a group looking for "upgradeconfirm" in the "given" field and keeps up to 3 following
3. Group the results by id again with a secondary ascending sort by timestamp
4. Apply a custom aggregator to concatenate all the "given" fields in a group together
5. Group the results by "given" action
6. Count the groups
7. Output "given" action, count pairs

Before we start our transformations we need to define where we'll be taking input from:

```
String inputPath = "data";
// create SOURCE tap to read a resource from the local file
// system be default the TextLine scheme declares two
// fields, "offset" and "line"
Tap localLogTap = new Hfs(new TextLine(), inputPath);
```

Here we defind a source tap that reads from the data directory of the Hadoop file system.

Next, we'll need to parse our log files into streams that we can work with:

```
// declare the field names we will parse out of the log
// file
Fields logFields = new Fields("id", "gender", "age",
    "requested", "given", "timestamp");

// define the regular expression to parse the log file with
String logRegex = "(\\w*)\\t(\\w*)\\t(\\w*)\\t(\\w*)\\t
              (\\w*)\\t([\\w\\s:-]*)";

// declare the groups from the above regex we want to keep.
// each regex group will be given a field name from
// 'logFields', above, respectively
int[] allGroups = { 1, 2, 3, 4, 5, 6 };

// create the stream parser
RegexParser parser = new RegexParser(logFields, logRegex,
                                  allGroups);

// create the parser pipe element, with the name 'parser',
// and with the input field name 'line'
Pipe importPipe = new Each("parser", new Fields("line"),
                        parser);
```

Here, we first declare field names for our groupings. Then, we create a regular expressions with groups. We also declare an integer array in case

things the groups are out of order from the field names. Finally, we create a RegexParser and apply it to create a pipe. By default, a stream will be split up into lines and stored in the field "line".

Now that we have our initial pipe, we can start applying transformations on it. First, we need to group by member id and sort by timestamp:

```
importPipe = new Group(importPipe, new Fields("id"),
                        new Fields("timestamp"), true);
```

Group needs a pipe to act upon, the fields to group by, the fields to sort by and whether to do a reverse sort or not. It will return a pipe so we can replace our current reference, the original pipe that was returned by the Each parser now being wrapped by a Group operation.

Next, we apply a custom filter that will return a stream with only 3 lines corresponding to actions of a member that came before an upgradeconfirmed.

```
importPipe = new Each(importPipe, new LogEntryFilter());
```

Then, we can group by id and order by timestamp but in ascending order this time:

```
importPipe = new Group(importPipe, new Fields("id"),
                        new Fields("timestamp"));
```

Once grouped we can concatenate a single member's actions into a series:

```
importPipe = new Every(importPipe, new Concatenate(
            new Fields("given"),"->"), new Fields("given")
            );
```

Finally, we group that series and count each occurence:

```
importPipe = new Group(importPipe, new Fields("given"));

// Count each combination
importPipe = new Every(importPipe, new Fields("given"),
    new Count(), new Fields("given", "count"));
```

We are now done with processing the data. All that's left is to output the pipe to disk:

```
String upgradeOuputPath = "upgradeOutput";

// create a SINK tap to use for output
// by default, TextLine writes all fields out
Tap upgradesLogTap = new Lfs(new TextLine(),
                             upgradeOuputPath);

// connect the assembly to the SOURCE (localLogTap)
// and SINK (allLogTap) taps
Flow upgradesLogFlow = new FlowConnector().connect(
    localLogTap, upgradesLogTap, importPipe);

// start execution of the flow
upgradesLogFlow.start();

// block until the flow completes
upgradesLogFlow.complete();
```

When running the above, we should get the ouput we are expecting. See Appendix A. for full code listing.

5 Appendix A. Example Program Listing

5.1 Main Program

```
import java.io.IOException;

import cascading.flow.Flow;
import cascading.flow.FlowConnector;
import cascading.operation.aggregator.Count;
import cascading.operation.regex.RegexParser;
import cascading.pipe.Each;
import cascading.pipe.Every;
import cascading.pipe.Group;
import cascading.pipe.Pipe;
import cascading.scheme.TextLine;
import cascading.tap.Hfs;
import cascading.tap.Lfs;
import cascading.tap.Tap;
import cascading.tuple.Fields;

/**
 *
 */
public class Main {

  public static void main(String[] args) throws
    IOException {
    @SuppressWarnings("unused")
    Main m = new Main();
  }

  public Main() throws IOException {
    String inputPath = "data";
    String upgradeOuputPath = "upgradeOutput";

    // create SOURCE tap to read a resource from the local
    // file system be default the TextLine scheme declares
    // two fields, "offset" and "line"
```

```
Tap localLogTap = new Hfs(new TextLine(), inputPath);

// declare the field names we will parse out of the
// log file
Fields logFields = new Fields("id", "gender", "age",
            "requested", "given", "timestamp");

// define the regular expression to parse the log
// file with
String logRegex = "(\\w*)\\t(\\w*)\\t(\\w*)\\t(\\w*)
        \\t(\\w*)\\t([\\w\\s:-]*)";

// declare the groups from the above regex we want to
// keep. each regex group will be given a field name
// from 'logFields', above, respectively
int[] allGroups = { 1, 2, 3, 4, 5, 6 };

// create the stream parser
RegexParser parser = new RegexParser(logFields,
                                logRegex, allGroups);

// create the parser pipe element, with the name
// 'parser', and with the input field name 'line'
Pipe importPipe = new Each("parser",
                  new Fields("line"), parser);
importPipe = new Group(importPipe, new Fields("id"),
                  new Fields("timestamp"), true);

// Group by id with a secondary sort on timestamp and
// only keep max of 3
importPipe = new Each(importPipe,
                  new LogEntryFilter());

importPipe = new Group(importPipe, new Fields("id"),
                        new Fields("timestamp"));

// Concatenate all "given" fields in a group
// into a single action delimited by "->"
```

```
    // ex: search->profile->mail
    importPipe = new Every(importPipe, new Concatenate(
              new Fields("given"),"->"),
              new Fields("given"));

    // Group by concatenated action stream
    importPipe = new Group(importPipe,
                             new Fields("given"));

    // Count each combination
    importPipe = new Every(importPipe, new Fields("given"),
                  new Count(),
                  new Fields("given", "count"));

    // create a SINK tap to use for output
    // by default, TextLine writes all fields out
    Tap upgradesLogTap = new Lfs(new TextLine(),
                                    upgradeOuputPath);

    // connect the assembly to the SOURCE (localLogTap)
    // and SINK (allLogTap) taps
    Flow upgradesLogFlow = new FlowConnector().connect(
                  localLogTap, upgradesLogTap, importPipe);

    // start execution of the flow
    upgradesLogFlow.start();

    // block until the flow completes
    upgradesLogFlow.complete();
  }
}
```

5.2 Entry Filter

```
import cascading.operation.BaseOperation;
import cascading.operation.Filter;
import cascading.tuple.Fields;
```

```
import cascading.tuple.TupleEntry;

public class LogEntryFilter extends BaseOperation
      implements Filter {

  private long currentMember = 0;
  private int numKept = 0;

  public LogEntryFilter() {
    super(ANY, Fields.ALL);
  }

  @Override
  public boolean isRemove(TupleEntry arg0) {
    // TODO Auto-generated method stub
    long memberId = arg0.selectTuple(new Fields("id"))
                                              .getLong(0);
    long timestamp = arg0.selectTuple(
                    new Fields("timestamp")).getLong(0);
    String actionGiven = arg0.selectTuple(
                    new Fields("given")).getString(0);
    if ("upgradeconfirm".equals(actionGiven)) {
      this.currentMember = memberId;
      this.numKept = 0;
      return true;
    }
    if (this.currentMember != 0
        && this.currentMember == memberId
        && this.numKept < 3) {
      this.numKept++;
      return false;
    } else {
      return true;
    }
  }
}
```

5.3 Concatenation Aggregator

```
import java.util.Map;

import cascading.operation.Aggregator;
import cascading.operation.BaseOperation;
import cascading.tuple.Fields;
import cascading.tuple.Tuple;
import cascading.tuple.TupleCollector;
import cascading.tuple.TupleEntry;

/**
 * Class Concatenate is an {@link Aggregator} that
 * returns the concatenation of all the values in
 * the current group for the requested field.
 */
public class Concatenate extends BaseOperation
        implements Aggregator {
  /** Field FIELD_NAME */
  public static final String FIELD_NAME = "combined";
  /** Field KEY_COUNT */
  private String seperator = "";

  /**
   * Constructs a new instance that returns the
   * concatenation of the values encoutered in
   * the given fieldDeclaration field name.
   *
   * @param fieldDeclaration
   *            of type Fields
   */
  public Concatenate(Fields fieldDeclaration) {
    super(1, fieldDeclaration);

    if (!fieldDeclaration.isSubstitution()
            && fieldDeclaration.size() != 1)
      throw new IllegalArgumentException(
          "fieldDeclaration may only declare 1 field, got: "
```

```
            + fieldDeclaration.size());
}

public Concatenate(Fields fieldDeclaration,
                           String seperator) {
  super(1, fieldDeclaration);

  if (!fieldDeclaration.isSubstitution()
          && fieldDeclaration.size() != 1)
    throw new IllegalArgumentException(
        "fieldDeclaration may only declare 1 field, got: "
            + fieldDeclaration.size());
  this.seperator = seperator;
}

/** @see Aggregator#start(Map, TupleEntry) */
@SuppressWarnings("unchecked")
public void start(Map context, TupleEntry groupEntry) {
  context.put(FIELD_NAME, null);
}

/** @see Aggregator#aggregate(Map, TupleEntry) */
@SuppressWarnings("unchecked")
public void aggregate(Map context, TupleEntry entry) {
  if (context.get(FIELD_NAME) != null) {
    context.put(FIELD_NAME, context.get(FIELD_NAME)
        + this.seperator
        + entry.selectTuple(this.getFieldDeclaration())
        .getString(0));
  } else {
    context.put(FIELD_NAME, entry.selectTuple(
        this.getFieldDeclaration()).getString(0));
  }
}

/** @see Aggregator#complete(Map, TupleCollector) */
@SuppressWarnings("unchecked")
public void complete(Map context,
```

```
                    TupleCollector outputCollector) {
    outputCollector.add(
          new Tuple((String) context.get(FIELD_NAME)));
  }
}
```

6 Bibliography

References

[1] Hadoop *http://hadoop.apache.org/core/*.

[2] Cascading *http://www.cascading.org/*.

[3] Cascading: An Introductory Overview
http://www.cascading.org/documentation/Cascading.pdf