Data Flow Driven Hadoop Development Pig and Cascading

Advanced Analytics with Machine Learning and Hadoop: Week Three Lecture

In Week 1's lesson on Hive, we touched upon some of the motivations for working in a higher-level language as opposed to native MapReduce, which can be difficult, unwieldy and verbose even for relatively simple operations. For example, joins are notoriously difficult to implement in native MapReduce, as it requires setting up the partitioners such that all relations with the same join-key are sent to a common reducer.

Furthermore, even for experienced Java and MapReduce programmers, most non-trivial Hadoop applications can entail a long development cycle, writing and chaining several mappers and reducers to form a complex job-chain or data processing workflow. These criticisms of native MapReduce regarding development efficiency, maintenance, and portability of the code provide much of the motivation for higher-level abstractions of Hadoop, including the two tools we will discuss in this lesson.

Table of Contents

Pig – A Data Flow Language for Hadoop

Pig, like Hive, is an abstraction of MapReduce, allowing users to express their data processing and analysis operations in a higher-level language that then compiles into a MapReduce job. Pig was developed at Yahoo! as a tool for researchers and engineers to more easily write their data mining Hadoop scripts by representing them as data flows[1]. Pig is now a top-level Apache Project (http://pig.apache.org/) that includes two main platform components:

- *Pig Latin*: a procedural scripting language used to express data flows
- The Pig execution environment to run Pig Latin programs, which can be run in local or MapReduce mode. Includes the *Grunt* command-line interface.

Unlike Hive's HQL which draws heavily from SQL's *declarative* style, Pig Latin is *procedural* in nature and is designed to enable programmers to easily implement a series of data operations and transformations that are applied to data sets to form a data pipeline[2]. While Hive is great for use-cases that translate well to SQL-based scripts, HQL (like SQL) can become unwieldy when multiple complex data transformations are required. Pig Latin is ideal for implementing these types of

---

[1] Hadoop: The Definitive Guide (3rd ed): White, Tom. (p. 367). O'Reilly Media.

[2] Yahoo Developer Network "Comparing Pig Latin and SQL for Constructing Data Processing Pipelines": https://developer.yahoo.com/blogs/hadoop/comparing-pig-latin-sql-constructing-data-processing-pipelines-444.html

multi-stage data flows, particularly in cases where we need to aggregate data from multiple sources and perform subsequent transformations at each stage of the data processing flow.

Pig Latin scripts start with data, apply transformations to the data until the script describes the desired results, and executes the entire data processing flow as an optimized MapReduce job. Additionally, Pig supports the ability to integrate custom code with user-defined functions (UDFs) that can be written in Java, Python, or Javascript, among other supported languages[3]. Pig thus enables us to perform near arbitrary transformations and ad-hoc analysis on our big data using comparatively simple constructs.

It is important to remember the earlier point that Pig, like Hive, ultimately compiles into MapReduce and cannot transcend the limitations of Hadoop's batch-processing approach. However, Pig does provide us with powerful tools to easily and succinctly write complex data processing flows, with the fine-grained controls that we need to build real business applications on Hadoop. In this lesson, we will review some of the basic components of Pig and implement both native Pig Latin operators and

[3] Apache Pig Documentation: http://pig.apache.org/docs/r0.12.0/start.html

custom-defined functions to perform some simple sentiment analysis on Twitter data.

Installation

Pig requires Hadoop and Java 1.6+, and ships with an embedded version of Hadoop 1.0.0 libraries if the HADOOP_PREFIX environment variable is not set. For the purposes of this lesson, we will assume that you are working with Hadoop 1.2.1 under Ubuntu and using the Oracle Java platform with JDK 1.7.

Let's start by downloading the most recent stable release of Pig from the list of Apache Pig Download Mirrors (currently at version 0.12.1). Unpack the downloaded Pig distribution, and move it to the desired run location (/srv in this example) creating a symlink for convenience:

```
~$ wget
http://mirror.cc.columbia.edu/pub/software/apache/pig/stable/pig-
0.12.1.tar.gz
~$ sudo mv pig-0.12.1.tar.gz /srv/
~$ cd /srv
/srv$ sudo tar -xvf pig-0.12.1.tar.gz
/srv$ sudo chown -R analyst:analyst pig-0.12.1
/srv$ sudo ln -s $(pwd)/pig-0.12.1 $(pwd)/pig
```

Ensure that the JAVA_HOME environment variable is set to your Java distribution and while we're at it, let's add the following environment variables to our bash profile for convenience:

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle

export PIG_HOME=/srv/pig
export PATH=$PATH:$PIG_HOME/bin
```

Pig can run in two modes, local mode and MapReduce mode. We will quickly review both execution types, but utilize local mode for the remainder of this lesson.

Local Mode

In local mode, Pig utilizes the local filesystem and Hadoop's `LocalJobRunner` class to run jobs within a single JVM process. Local mode is preferred for quick prototyping, testing, and debugging, but should never be used in a production application. Starting Pig up in local mode is as easy as setting the execution type to `local` with the `-x` option:

```
~$ cd $PIG_HOME
/srv/pig$ pig -x local
2014-04-06 10:57:48,076 [main] INFO  org.apache.pig.Main - Apache Pig
version 0.12.1 (r1585011) compiled Apr 05 2014, 01:41:34
2014-04-06 10:57:48,089 [main] INFO  org.apache.pig.Main - Logging
error messages to: /srv/pig-0.12.1/pig_1402077468070.log
2014-04-06 10:57:48,180 [main] INFO  org.apache.pig.impl.util.Utils -
Default bootup file /home/analyst/.pigbootup not found
```

```
2014-04-06 10:57:48,418 [main] INFO
org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
Connecting to hadoop file system at: file:///
grunt>
```

This starts up Pig's command-line interface, the *grunt* shell. You can exit the Grunt shell any time with the `quit` command:

```
grunt> quit;
```

MapReduce Mode

In MapReduce mode, Pig compiles scripts into actual MapReduce jobs and runs them on a configured Hadoop cluster, which may be a pseudo- or fully-distributed cluster. By default, Pig will look for your Hadoop distribution in the HADOOP_HOME or HADOOP_PREFIX environment variable, but if this is not set, Pig will use a bundled copy of the Hadoop libraries which may not match the version of your Hadoop cluster, so make sure your HADOOP_HOME or HADOOP_PREFIX environment variable is set.

By default, Pig expects hadoop configs, hadoop-site.xml, mapred-site.xml and core-site.xml, to be present on the classpath so that it can find the Hadoop cluster's NameNode and JobTracker (`fs.default.name` and `mapred.job values,` respectively). Alternatively, you can provide these settings in Pig's properties file, at $PIG_HOME/conf/pig.properties.

With the Hadoop daemons running, we can then run Pig in MapReduce mode either by setting the `-x` option to `mapreduce`, or omitting it as MapReduce is the default execution mode:

```
/srv/pig$ $HADOOP_PREFIX/bin/stop-all.sh
/srv/pig$ $HADOOP_PREFIX/bin/start-all.sh
/srv/pig$ jps
5196 SecondaryNameNode
5280 JobTracker
4690 NameNode
5539 TaskTracker
5685 Jps
4941 DataNode
/srv/pig$ pig
2014-04-06 11:17:52,535 [main] INFO  org.apache.pig.Main - Apache Pig
version 0.12.1 (r1585011) compiled Apr 05 2014, 01:41:34
2014-04-06 11:17:52,544 [main] INFO  org.apache.pig.Main - Logging
error messages to: /srv/pig-0.12.1/pig_1402078672533.log
2014-04-06 11:17:52,620 [main] INFO  org.apache.pig.impl.util.Utils -
Default bootup file /home/analyst/.pigbootup not found
2014-04-06 11:17:52,974 [main] INFO
org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
Connecting to hadoop file system at: hdfs://localhost:9000
2014-04-06 11:17:54,077 [main] INFO
org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
Connecting to map-reduce job tracker at: localhost:9001
grunt>
```

Batch Mode

The Grunt shell automatically starts up after invoking the pig command, but we can alternatively use the `-e` option to enable "batch mode" and run a script file that contains Pig commands directly from the command line:

```
/srv/pig$ pig -e mypigscript.pig
```

While not required, it is good practice to identify the file using the *.pig extension. In addition to batch mode, you can also run Pig scripts from the Grunt shell, by using the `run` and `exec` commands.[4]

Pig Latin

Now that we have Pig and the Grunt shell set up, let's examine a sample Pig script and explore some of the commands and expressions that Pig Latin provides. The following script loads Twitter tweets with the hashtag #unitedairlines over the course of a single week (in Lesson Resources: united_airlines_tweets.tsv), which provides the tweet ID, permalink, date posted, tweet text and Twitter username. It also loads a dictionary (dictionary.tsv) of known "positive" and "negative" words along with sentiment scores (1 and -1 respectively) associated to each word. The script then

[4] Apache Pig Documentation "Execution Modes": http://pig.apache.org/docs/r0.12.1/start.html#execution-modes

performs a series of Pig transformations to generate a sentiment score and classification, either POSITIVE or NEGATIVE, for each computed tweet:

```
grunt> tweets = LOAD 'united_airlines_tweets.tsv' USING
PigStorage('\t') AS (id_str:chararray, tweet_url:chararray,
created_at:chararray, text:chararray, lang:chararray,
retweet_count:int, favorite_count:int, screen_name:chararray);

grunt> dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t') AS
(word:chararray, score:int);

grunt> english_tweets = FILTER tweets BY lang == 'en';

grunt> tokenized = FOREACH english_tweets GENERATE id_str, FLATTEN(
TOKENIZE(text) ) AS word;

grunt> clean_tokens = FOREACH tokenized GENERATE id_str,
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;

grunt> token_sentiment = JOIN clean_tokens BY word, dictionary BY word;

grunt> sentiment_group = GROUP token_sentiment BY id_str;

grunt> sentiment_score = FOREACH sentiment_group GENERATE group AS id,
SUM(token_sentiment.score) AS final;

grunt> classified = FOREACH sentiment_score GENERATE id, ( (final >=
0)? 'POSITIVE' : 'NEGATIVE' ) AS classification, final AS score;

grunt> final = ORDER classified BY score DESC;

grunt> STORE final INTO 'sentiment_analysis';
```

Let's break down this script at each step of the data processing flow.

Relations and Tuples

The first two lines in the script loads data from the filesystem into *relations* called tweets and dictionary:

```
tweets = LOAD 'united_airlines_tweets.tsv' USING PigStorage('\t') AS
(id_str:chararray, tweet_url:chararray, created_at:chararray,
text:chararray, lang:chararray, retweet_count:int, favorite_count:int,
screen_name:chararray);

dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t') AS
(word:chararray, score:int);
```

In Pig, a relation is conceptually similar to a table in a relational database, but instead of an ordered collection or rows, a relation consists of an unordered set of *tuples.* Tuples are an ordered set of fields. It is important to note that although a relation declaration is on the left side of an assignment, much like a variable in a typical programming language, relations are not variables. Relations are given aliases for reference purposes, but they actually represent a checkpoint data set within the data processing flow.

We used the `LOAD` operator to specify the filename of the file (either on the local filesystem or HDFS) to load into the tweets and dictionary relations. We also use the `USING` clause with the `PigStorage` load function to specify that the file is tab-delimited. Although not required, we also defined a schema for each relation using the AS clause and specifying column aliases for each field, along with the corresponding data type. If a schema is not defined, we can still reference the fields for each tuple in our relation by using Pig's positional columns, $0 for the first field, $1 for the second, etc. This may be preferable if we are loading data with many columns, but are only interested in referencing a few of them.

Filtering

The next line performs a simple `FILTER` data transformation on the tweets relation to filter out any tuples that are not in English:

```
english_tweets = FILTER tweets BY lang == 'en';
```

The FILTER operator selects tuples from a relation based on some condition, and is commonly used to select the data that you want; or, conversely, to filter out (remove) the data you don't want. Since the "lang" field is typed as a `chararray`, the Pig equivalent of the Java String data type, we used the == comparison operator to retain values that equal 'en' for English. The result is stored into a new relation called english_tweets.

Projection

Now that we've filtered the data to retain only English tweets (our dictionary after all, is in English) we need to split the tweet text into word tokens which we can match against our dictionary, and perform some additional data cleanup on the words to remove hashtags ('#') and user handle tags ('@'):

```
tokenized = FOREACH english_tweets GENERATE id_str, FLATTEN(
TOKENIZE(text) ) AS word;
clean_tokens = FOREACH tokenized GENERATE id_str,
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;
```

Pig provides the FOREACH…GENERATE operation to work with columns of data in relations or bags and apply a set of expressions to every tuple in the collection. The

GENERATE clause contains the values and/or evaluation expression that will derive a new bag of tuples to pass onto the next step of the pipeline. In our example, we project the id_str key from the english_tweets relation, and use the TOKENIZE function split the text field into word tokens (split on whitespace). The FLATTEN function extracts the resulting *bag* of tuples into a single collection.

A *bag is* a special data type in Pig, and represents an unordered collection of tuples, similar to a relation although relations are called the "outer bag" because they cannot be nested within another bag. In our FOREACH command, the result produces a new relation called tokenized where the first field is the stock_tweet ID (id_str) and the second field is a bag composed of single-word tuples.

We then perform another projection based on the tokenized relation to project the id_str and lowercased word without any leading hashtag or handle tag. We've performed quite a few transformations on our data, so it would be a good time to verify that our relations are well-structured. We can use the ILLUSTRATE operator at any time to view the schemas of each relation generated based on a concise sample dataset:

```
grunt> ILLUSTRATE clean_tokens;
-------------------------------------------------------------------------------------------
```

```
| tweets     | id_str:chararray     | tweet_url:chararray                                  |
created_at:chararray      | text:chararray
| lang:chararray     | retweet_count:int     | favorite_count:int     | screen_name:chararray
|
-------------------------------------------------------------------------------------------
|            | 474415416874250240   | https://twitter.com/URobot/status/474415416874250240   |
2014-06-05 05:00:24+00:00 | #unitedairlines #poorservice#terriblecustomerservice#rude
| en                 | 0                     | 0                      | URobot
|
|            | 474890870869614592   | https://twitter.com/FerDiaz4/status/474890870869614592 |
2014-06-06 12:29:41+00:00 | "en mi vida vuelvo a volar por #unitedairlines vuelo retrasado,
pierdo mi conexion y hago 14 horas Mexico - Montreal" | es                 | 0
| 0                     | FerDiaz4                 |
-------------------------------------------------------------------------------------------
| english_tweets     | id_str:chararray     | tweet_url:chararray
| created_at:chararray      | text:chararray                                          |
lang:chararray     | retweet_count:int     | favorite_count:int     | screen_name:chararray
|
-------------------------------------------------------------------------------------------
|                    | 474415416874250240   |
https://twitter.com/URobot/status/474415416874250240 | 2014-06-05 05:00:24+00:00 |
#unitedairlines #poorservice#terriblecustomerservice#rude | en                 | 0
| 0                     | URobot                   |
-------------------------------------------------------------------------------------------
| tokenized     | id_str:chararray     | word:chararray                          |
-----------------------------------------------------------------------------------
|               | 474415416874250240   | #unitedairlines                         |
|               | 474415416874250240   | #poorservice#terriblecustomerservice#rude |
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
| clean_tokens     | id_str:chararray     | word:chararray                          |
-----------------------------------------------------------------------------------
|                  | 474415416874250240   | unitedairlines                          |
|                  | 474415416874250240   | poorservice#terriblecustomerservice#rude |
-----------------------------------------------------------------------------------
```

The ILLUSTRATE command is helpful to use periodically as we design our Pig flows to help us understand what our queries are doing and validate each checkpoint in the pipeline.

Grouping and Joining

Now that we've tokenized the tweets we're interested in and cleaned the word tokens, we would like to JOIN the resulting tokens against the dictionary, matching against the word if found:

```
token_sentiment = JOIN clean_tokens BY word, dictionary BY word;
```

Pig provides the JOIN command to perform a join on two or more relations based on a common field value. Both inner joins and outer joins are enabled, although inner joins are used by default. In our example, we perform an inner join between the clean_tokens relation and dictionary relation based on the word field, which will generate a new relation called token_sentiment that contains the fields from both relations:

```
-------------------------------------------------------------------------
| token_sentiment     | clean_tokens::id_str:chararray     |
clean_tokens::word:chararray     | dictionary::word:chararray     |
dictionary::score:int     |
-------------------------------------------------------------------------
|                     | 473233757961723904                 |
delayedflight                    | delayedflight                  | -1
|
-------------------------------------------------------------------------
```

Now we need to GROUP those rows by the Tweet ID, id_str, so we can later compute and aggregated SUM of the score for each tweet:

```
sentiment_group = GROUP token_sentiment BY id_str;
```

The GROUP operator groups together tuples that have the same group key (id_str). The result of a GROUP operation is a relation that includes one tuple per group, where the tuple contains two fields:

1. The first field is named "group" (do not confuse this with the GROUP operator) and is the same type as the group key.
2. The second field takes the name of the original relation (token_sentiment) and is type bag.

We can now perform the final aggregation of our data, by computing the sum score for each tweet (grouped by id):

```
sentiment_score = FOREACH sentiment_group GENERATE group AS id,
SUM(token_sentiment.score) AS final;
```

And then classify each tweet as POSTIVE or NEGATIVE based on the score:

```
classified = FOREACH sentiment_score GENERATE id, ( (final >= 0)?
'POSITIVE' : 'NEGATIVE' ) AS classification, final AS score;
```

Finally, let's sort the results by score in descending order:

```
final = ORDER classified BY score DESC;
```

Storing and Outputting Data

After we've applied all the necessary transformations on our data, we would like to now write out the results somewhere. Pig provides the STORE statement for this purpose, which takes a relation and writes the results into the specified location. By default, the STORE command will write data to HDFS in tab-delimited files using PigStorage. In our example, we dump the results of the final relation into a directory on the local filesystem called sentiment _analysis.

```
STORE final INTO 'sentiment_analysis';
```

The contents of that directory will include one or more part files:

```
/srv/pig$ cd sentiment_analysis
/srv/pig/sentiment_analysis$ ls
part-r-00000  _SUCCESS
```

In local mode, only one part file is created, but in MapReduce mode the number of part files will depend on the parallelism of the last job before the store. Pig provides a couple features to set the number of reducers for the MapReduce jobs generated, you can read more about Pig's parallel features in the Apache Pig documentation.

When working with smaller data sets, it's convenient to quickly output the results from the grunt shell to the screen rather than having to store it. The DUMP command takes the name of a relation and prints the contents to the console.

```
grunt> DUMP sentiment_analysis;
```

Data Types

We covered some of the nested data structures available in Pig, including fields, tuples and bags. Pig also provides a *map* structure, which contains a set of key-value pairs. The key should always be of type chararray, but the values do not have to be of the same data type. We saw some of the native scalar types that Pig supports when we defined the schema for the stock data. The full list of scalar types that Pig supports includes:

Category	Type	Description	Example
Numeric	**int**	32-bit signed integer (Represented as **java.lang.Integer**)	12
	long	64-bit signed integer (Represented as **java.lang.Long**)	34L
	float	32-bit floating point number (Represented as **java.lang.Float**)	2.18F

	double	64-bit floating point number (Represented as **java.lang.Double**)	3e-17
Text	**chararray**	String or array or characters (Represented as **java.lang.String)**	'hello world'
Binary	**bytearray**	Blob or array of bytes (Represented as **DataByteArray** which wraps a Java byte[]).	N/A

Relational Operators

Pig provides data manipulation commands via the relational operators in Pig Latin. We used several of these to load, filter, group, project, and store data in our example above. In addition Pig supports the following relational operators:

Category	Operator	Description
Loading and Storing	**LOAD**	Load data from the filesystem or other storage source.
	STORE	Saves a relation to the filesystem or other storage.
	DUMP	Prints a relation to the console.

Filtering and Projection	**FILTER**	Selects tuples from a relation based on some condition.
	DISTINCT	Removes duplicate tuples in a relation.
	FOREACH…GENERATE	Generates data transformations based on columns of data.
	MAPREDUCE	Executes native MapReduce jobs inside a Pig script.
	STREAM	Sends data to an external script or program.
	SAMPLE	Selects a random sample of data based on the specified sample size.
Grouping and Joining	**JOIN**	Joins two or more relations.
	COGROUP	Groups the data from two or more relations.
	GROUP	Groups the data in a single relation.
	CROSS	Creates the cross-product of two or more relations.
Sorting	**ORDER**	Sorts the relation by one or more fields.
	LIMIT	Limits the number of tuples returned from a relation.

Combining and Splitting	**UNION**	Computes the union of two or more relations.
	SPLIT	Partitions a relation into two or more relations.

The usage syntax for Pig's relational operators and arithmetic, boolean, and comparison operators can be found in Pig's User Documentation.

User-Defined Functions

One of Pig's most powerful features lies in its ability to let users combine Pig's native relational operators with their own custom processing. Pig provides extensive support for such user defined functions (UDFs), and currently provides integration libraries for six languages: Java, Jython, Python, JavaScript, Ruby and Groovy[5]. However, Java is still the most extensively supported language for writing Pig UDFs, and generally more efficient since it is the same language as Pig and can thus integrate with Pig interfaces such as the Algebraic Interface and the Accumulator Interface.

[5] Apache Pig Documentation "UDFs": http://pig.apache.org/docs/r0.12.1/udf.html#udfs

Let's demonstrate a simple UDF for the script we wrote earlier. In this scenario, we would like to write a custom eval UDF that will allow us to convert the score classification evaluation into an function so that instead of:

```
classified = FOREACH sentiment_score GENERATE id, ( (final >= 0)?
'POSITIVE' : 'NEGATIVE' ) AS classification, final AS score;
```

We can use:

```
classified = FOREACH sentiment_score GENERATE id, classify(final) AS
classification, final AS score;
```

In Java, we will need to extend Pig's EvalFunc class and implement the `exec()` method which takes a tuple and will return a `String`:

```
package com.statistics.pig;

import java.io.IOException;

import org.apache.pig.EvalFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.Tuple;

public class ClassifyScore extends EvalFunc {

    @Override
    public String exec(Tuple input) throws IOException {
        if (args == null || args.size() == 0) {
            return false;
        }
        try {
            Object object = args.get(0);
```

```
            if (object == null) {
                return false;
            }
            int i = (Integer) object;
            if (i >= 0) {
                return new String("POSITIVE");
            } else {
                return new String("NEGATIVE");
            }
        } catch (ExecException e) {
            throw new IOException(e);
        }
    }
}
```

To use this function, we need to compile it, package it into a JAR file, and then register the JAR with Pig by using the REGISTER operator:

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

We can then invoke the function in a command:

```
grunt> classified = FOREACH sentiment_score GENERATE id,
com.statistics.pig.classify(final) AS classification, final AS score;
```

We encourage you to read the documentation on UDFs, which contains a list of supported UDF interfaces and provides example scripts to perform tasks for evaluation, loading and storing data, and aggregating/filtering data. Pig also provides a collection of user-contributed UDFs called Piggybank, which is distributed

with Pig but must be registered to use. See the Apache documentation on Piggybank for more information.

Conclusion

We've now seen how Pig can greatly ease the process of building a MapReduce data pipeline. Traditional ETL data pipeline processes probably form the large majority of use cases for Pig. However, Pig can also be an excellent tool for performing ad-hoc analysis and building iterative processing or predictive models from large batches of data, especially as the analysis grows more complex. Pig can be a powerful tool for users who prefer a procedural programming model. It provides the ability to control where data is check pointed in the pipeline, as well as fine-grained controls over how the data is processed at each step. In the next section, we'll take a brief look at another data processing framework called Cascading, which takes the workflow abstraction even further than Pig by formalizing the elements of the data pipeline.

Cascading – Enterprise Data Pipelines in Hadoop

In the last section we observed how a functional programming paradigm, like the one offered by Pig, lends itself well to building data pipelines and other non-trivial Hadoop use cases, which would otherwise entail daunting amounts of native MapReduce code. This key observation was also the driving motivation for another Hadoop application development framework, called Cascading.

Cascading was developed by Concurrent Inc founder and former Apache Nutch contributor, Chris Wensel, when he observed the shortage of enterprise-capable Java developers that could (or were willing to) write native MapReduce code, which would be a significant obstacle in promoting Hadoop in an enterprise context. Cascading was thus designed as an application framework on top of Hadoop that would enable Java developers in the enterprise to use familiar tools and constructs to build complex apps without having to learn the inner-workings of Hadoop[6].

Cascading has been used in many large-scale enterprise deployments, even powering revenue apps at Twitter, eBay, and LinkedIn. Cascading has also spawned numerous subprojects, like Cascalog (Cascading with Clojure) and Scalding (Cascading with Scala), which adapt Cascading's functional programming paradigm

[6] Enterprise Data Workflows with Cascading (1st ed): Nathan, Paco. (p. 15). O'Reilly Media.

to other functional languages. However, the core Cascading API is written in Java, and provides flow-definition tools to define and represent complex data processing flows as a directed acyclic graph (DAG)[7].

In this way, Cascading departs from the high-level workflow abstraction of Pig; it provides a structured method for planning and building complex data pipelines with a syntax that formalizes workflows using plumbing metaphors: *taps* extract data from *sources* through *pipes* which route the stream to *operators*, *sinks*, or *traps*. A workflow can be visualized as a "flow diagram", represented as a DAG:

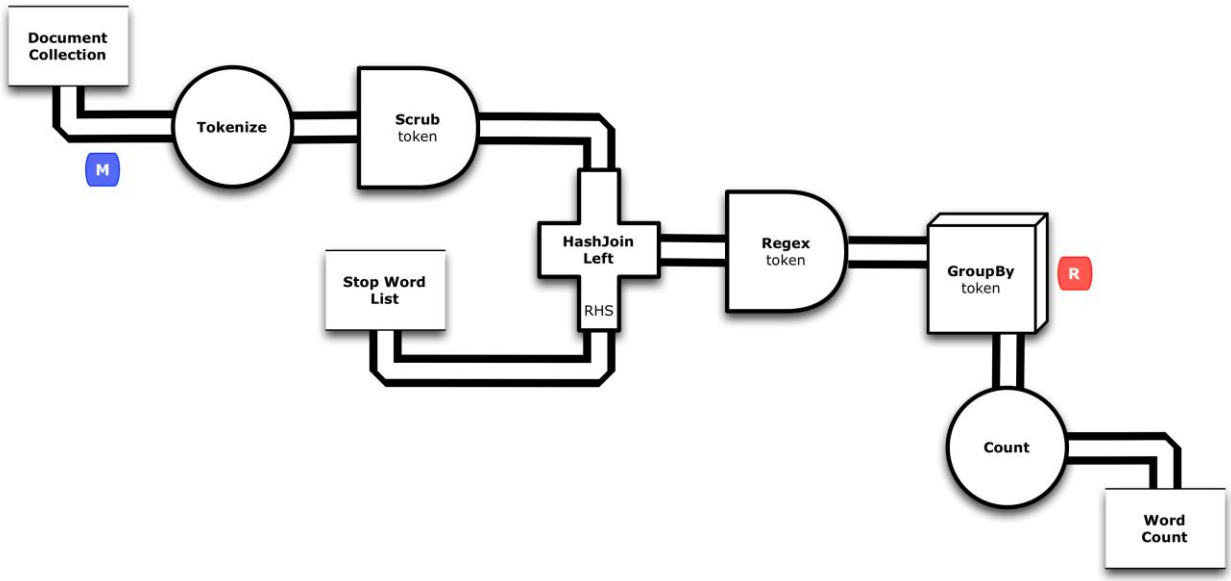

Figure 1 - Word Count as Cascading Flow[8]

[7] Wikipedia "Directed Acyclic Graph": http://en.wikipedia.org/wiki/Directed_acyclic_graph

What makes Cascading especially powerful is that smaller data flows, like the Word Count example above, can be encapsulated in a *pipe assembly,* which can then be chained together and executed as a single process. In this context, if one flow depends on the output of another, it is not executed until all of its data dependencies are satisfied. Such a collection of flows is called a *cascade*.[9] You might now be able to understand why this framework is so useful for enterprise use cases, which by and large follow the general pattern of ETL, data cleansing/prep, and data mining/predictive modeling:

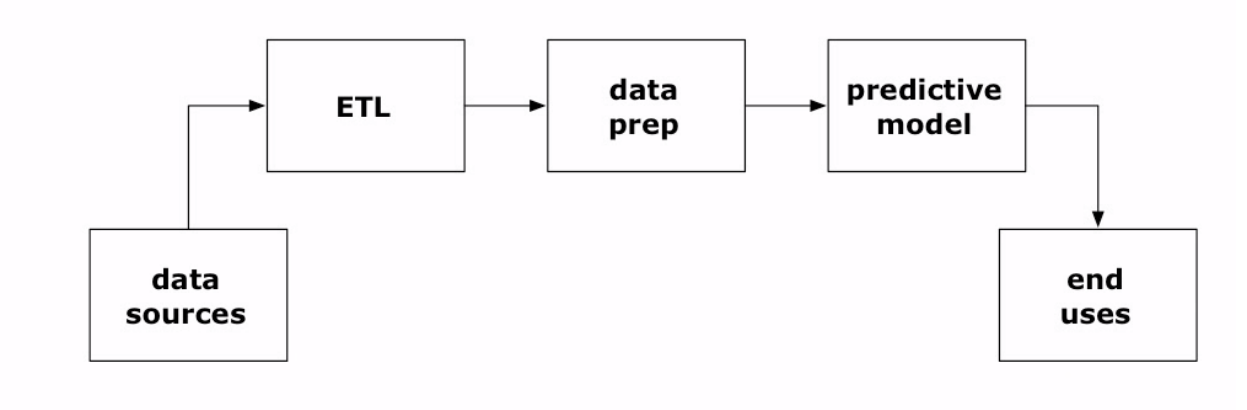

Figure 2 - Enteprise App Flow[10]

[8] Quora "Cascading in Layman's Terms": http://www.quora.com/Cascading/What-is-cascading-in-layman-terms

[9] Cascading User Guide: http://docs.cascading.org/cascading/2.5/userguide/html/ch03.html#N2013B

[10] Hortonworks "How to Get Started on Cascading and Hadoop": http://hortonworks.com/blog/cascading-hadoop-big-data-whatever/

Although this general pattern represents most enterprise data workflows at a high-level, the actual implementation of each step of this cascade is unique to each enterprise. Data sources could be from a data warehouse like Oracle, NoSQL store like Cassandra, or even a log management application like Splunk. The ETL process would need to be expressed as its own pipeline assembly, tapping the relevant data sources, extracting fields for processing, and sending the results down the pipeline:

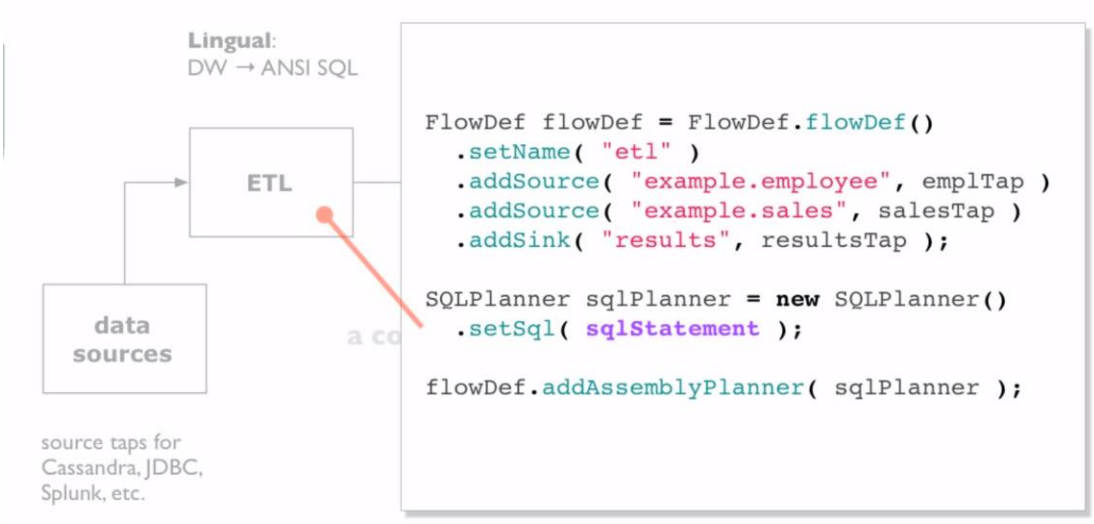

Figure 3 - ETL Flow Definition

Similarly, our data prep and predictive modeling stages would also represent their own pipe assemblies:

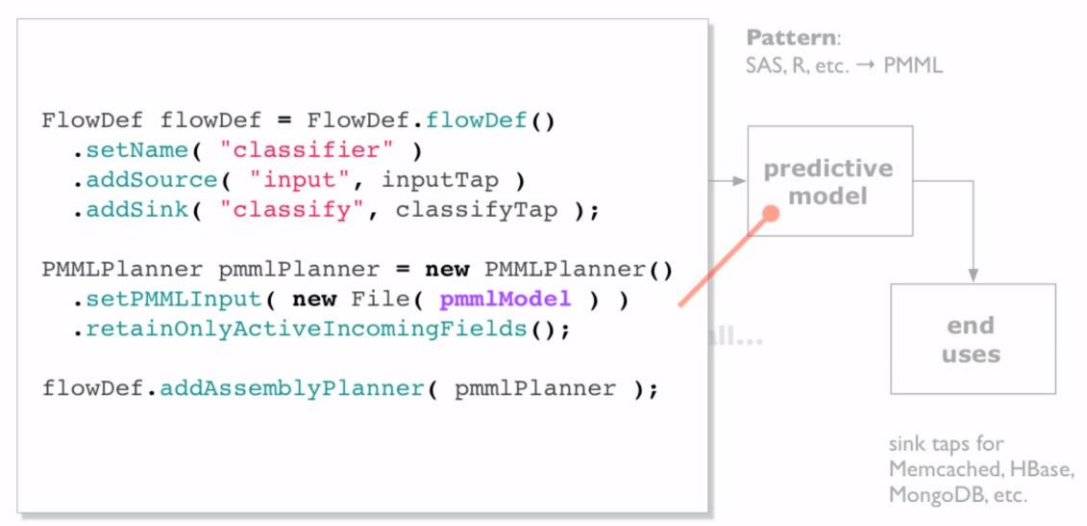

Figure 4 - Predictive Modeling Flow Definition

Cascading provides the mechanisms to build and orchestrate these workflow components within the context of a single integrated application.

Next Steps with Cascading

Although the implementation of Cascading's API is beyond the scope of this particular lesson (in fact, one could fill an entire course on Cascading alone), we encourage you to read up on the Cascading library at the official website: http://www.cascading.org/ and take a look at the code examples. There's also a very good tutorial for getting your feet we with Cascading, although it does presume strong knowledge of Java and ability to build with Gradle.

We will also be providing all participants in this class with a complimentary eBook that includes a more in-depth chapter on Cascading and the Cascading Pattern library to implement predictive modeling flows. But we hope that this lesson has at least whet your appetite for the more ambitious, enterprise-level frameworks within the Hadoop ecosystem.

Discussion Questions

1. In general, it is recommended to "filter early and often" in your Pig scripts. Why might this be beneficial for performance when Pig compiles the script into a MapReduce job?
2. Although Pig Latin is often called a procedural language, it lacks some features that are common in other procedural languages like Perl, including control structures (if/then/else).
    a. Why might control structures present a problem for Pig?
    b. If such control flow logic is needed, how could you incorporate it into a Pig script?

Assignment (Total 15 points)

The Resources section contains two CSVs containing historical NYSE stock price and dividends data from 1970-2010 (for stock symbols starting with N)[11]:

The NYSE_daily_prices_N.csv data contains the following fields:

1. exchange
2. stock_symbol
3. date

[11] Infochimps "NYSE Data 1970-2010": http://www.infochimps.com/datasets/nyse-daily-1970-2010-open-close-high-low-and-volume

4. stock_price_open
5. stock_price_high
6. stock_price_low
7. stock_price_close
8. stock_volume
9. stock_price_adj_close

The NYSE_dividends_N.csv data contains:

1. exchange
2. stock_symbol
3. date
4. dividends

(5 points)

Load the above CSVs into Pig relations, and write a Pig script to join the price data with the dividends data, by both stock symbol and date.

(5 points)

Generate a new relation that projects the price change (closing price minus opening price) for each joined row, and retains the dividends.

(5 points)

Determine the big paying days by filtering on the rows where the *price change is greater than or equal to 1.0* and the *dividends is greater than or equal to 0.25*. Group these results and generate an aggregate count of these days by stock symbol. Order it in descending order by count.

Submit the script and stored output.