Poodah API Tutorial

Raj Agarwal (rajagarw), Kirk Zhang (keweiz)

Poodah is a very flexible Map-Reduce framework that can be used for many different applications. We designed the framework to be extensible and customizable in order to match the client's needs. There are, however, some important tips that should be followed in order to ensure that a client's program runs properly. Firstly, it would be advantageous to check out the JavaDocs located in the ../doc/ directory. This contains more in depth information on the specific classes and methods we've implemented in poodah. Modeling the map-reduce application after our examples and implementations will help ensure problem-free usage.

**The Mapper Interface:**

When the client creates the "Map" step of their application, they need to make sure that they extend the Mapper interface. This class requires that the client implements a method called **map()**. When extending the Mapper interface, the client needs to specify the K1, V1, K2, V2 generic types. It is also imperative that these types all extend java Serializable. These types are going to be very important throughout the entire application. When construction other classes, the client should make sure that all the types fit together so runtime errors don't occur.

*void map(K1 key, V1 value, OutputCollector<K2, V2> collector)*

The map method prototype shown above will take in a key and value (K1, V1), and then write out a key-value pair (K2,V2). The input (K1, V1) is based on how the input file is to be interpreted. This will be described below in the InputFormat interface section. The output will be down via the OutputCollector's collect() method. This will write the output to a temp file.

**The Reducer Interface:**

When a client creates the "Reduce" step of their application, they need to make sure that they extend the Reducer interface. Similar to the Mapper interface, this interface requires the client to implement a **reduce()** method. The client also needs to specify K2, V2, K3, V3 types. It is important to notice that the reduce() step needs to be able to take the output of the map() step as inputs.

*void reduce(K2 key, List<V2> values, OutputCollector<K3,V3> collector)*

The reduce method prototype shown above is very similar to the map method's prototype with the exception that that the values passed in are in list form. This is the case because the client needs to reduce a bunch of values produced from the map step that are related to the same key.

**The InputFormat Interface:**

In order for the framework to read from the input file, the Client must supply a class that can parse the input file and generate key value pairs (K1, V1) to pass into the map() function. In order to do this, the client must either use the provided implementation (TextInputFormat) or create a class that implements InputFormat.

*FileMeta getMetaData(String fileName)*

This method allows the framework to get some initial data about the input file. This includes the file name, the size of the file (in lines, bytes, etc.) and other data that may be useful. A few FileMeta implementations already exist in the framework, if the client wishes to make a new one, those can be good examples to start from. Otherwise we recommend just sticking to the defaults.

*List<KeyVal<K, V>> read(String fileName, int start, int end)*

This method is used by the framework to generate all the key value pairs from a file. It will then pass the key value pair into the client's map() function one at a time for processing. The KeyVal interface is another interface that needs to be implemented by the client. Please look at the KeyVal interface section for more detail.

**The OutputFormat Interface:**

The purpose of this interface is to tell the framework how to output the client's final results into the output file. Please view sample framework implementations for an example.

*Writer<K, V> getWriter(String outputFile)*

This method is responsible for returning a writer object that the framework uses to physically write the key value pairs to file. The client has complete control over how it wishes these pairs to be written based on how the client implements the Writer interface. For more information, see the Writer Interface section.

**The Writer Interface:**

The client will need to implement a writer interface so the framework can write the output KeyVals to file. There are example writer classes that the client can look at for reference. It is important to understand that the key value pairs passed into the writer will be based on the defined output KeyVal class. For more information on this, see the KeyVal Interface section.

**The KeyVal interface:**

The client will most likely require three implementations of the KeyVal interface. One implementation will be the input KeyVal. This will be important in generating the key value pairs from an input file. These will be used in the client's InputFormat implementation. The map output KeyVal is another important implementation. This will be required so the OutputCollector knows what to write the map outputs to temp file. These KeyVal objects should be of type (K2, V2). Finally, reduce output KeyVal class of type (K3, V3) is required. This plays the same role as the map output KeyVal except it is used after the reduce step. The implementation of the output KeyVals needs to be specified in the JobConfig class (see below). Note: The KeyVal implementations must be Serializable because they are written directly to file in the intermediate steps. Please view the example KeyVals provided for reference.

**The JobConfig Class:**

The JobConfig class is a crucial element in the entire framework. The client first implements the above parts, and then sets the **.class** files (along with other data) in the JobConfig object. Other parameters that need to be set include input/output file, master-node hostname, record byte length, reader size, record read size, and two comparators. An important limitation of this framework revolves around the fact that the KeyVal are of fixed record byte length. As a result, the client must have a good understanding of the task so the records don't overflow the set boundaries. The reader size defines how many units (lines, bytes, etc.) should be read and passed to a mapper at a time. Increase this number to allow each mapper to do more work. The record read size is the number of keys that should be passed to a reducer at a time. Increase this number to allow each reducer to do more work. The two comparators are required to compare the keys of the Mapper output together (type K2) and keys of the Reducer output together (type K3). This is crucial in sorting the respective output files.

The JobConfig also requires specification of Map output KeyVals and Reduce output KeyVals. These stand for (K2, V2) and (K3, V3) respectively. Note: (K2, V2) can equal (K3, V3). This is the case in our sample applications.

**The JobClient Class:**

The JobClient class is used to actually send the client's project to the specified master node. The *runJob(JobConfig conf)* method is used for this to happen. The client must make sure they use this method at the end of their application in order to send the information to the master.

*Note: The client's application must exist entirely within the poodah framework. Our framework doesn't have the ability to transfer bin files from one project to another at the moment. As a result, placing the client's application within the poodah framework will allow the framework to find the necessary .class files.