CARNEGIE MELLON UNIVERSITY

15640: Project 3 Developer Guide

Building a Map-Reduce facility

Yang Sun (Andrew ID: yksun) | Yuan Gu (Andrew ID: yuangu)

4/13/2013

Table of Contents

Overview 2
MapReduce.Records.InputRecord 3
MapReduce.Records.OutputRecord 3
MapReduce.Interface.Key 3
MapReduce.Interface.Value 3
MapReduce.Interface.Partitioner 3
MapReduce.Interface.Mapper 3
MapReduce.Interface.Reducer 4

Overview

The overview of the map-reduce facility is that the system will read in a bunch of fixed-length byte arrays and parse the fixed-length byte arrays into meaningful records. Then the system will call a mapper to map to map the records into intermediate records, which are actually <Key, Value> pairs. After this, the system will partition the intermediate records to different reducers. And for each reducer, the system will call the reduce function to generate the final output from the aggregated intermediate results, which are actually <Key, List<Value>> pairs, for that reducer.

We try our best to provide the maximum flexibility for the developers. And the developer could customize the following interfaces for the following phases:

- **MapReduce.Records.InputRecord**: to implement different types of input records, which could be used to parse the input fixed-length byte arrays into records which are meaningful to the developers.
- **MapReduce.Records.OutputRecord**: to implement different types of output records, which could be used to store and serialize the final output.
- **MapReduce.Interface.Key**: to implement different types of Key, which could be used as part of the intermediate results.
- **MapReduce.Interface.Value**: to implement different types of Value for intermediate result, which could be used as part of the intermediate results.
- **MapReduce.Interface.Partitioner**: to implement different types of Partitioner, which could define how the intermediate results will be partitioned to different reducers.
- **MapReduce.Interface.Mapper:** to implement different types of Mapper, which could define how mapper works.
- **MapReduce.Interface.Reducer:** to implement different types of Reducer, which could define how reducer works.

To facilitate your development, we've provided some sample implementations for the above interfaces:

- **MapReduce.Implementation.StringInputRecord:** a sample implementation of InputRecord which will parse input records to strings.
- **MapReduce.Implementation.StringOutputRecord:** a sample implementation of OutputRecord which will output records as strings.
- **MapReduce.Implementation.StringWritable:** a sample implementation of both Key and Value which will contain a string as the content of the Key/Value.
- **MapReduce.Implementation.Partititioner:** a sample implementation of Partitioner which will partition intermediate records according to their hash values.

For Mapper and Reducer, we provide two examples:

- **Examples.StringInputMapper:** a sample implementation of Mapper which accepts StringInputRecord and outputs <StringWritable, StringWritable> intermediate results.

- **Examples.StringInputReducer:** a sample implementation of Reducer which accepts <StringWritable, List<StringWritable>> intermediate results and outputs StringOutputRecord as output.

MapReduce.Records.InputRecord

To implementation **InputRecord**, you need to implement the following function:

- **public abstract void parseFromBytes(byte[] buf);**

The function parseFromBytes(byte[] buf) should accept a byte array and parse it.

MapReduce.Records.OutputRecord

To implementation **OutputRecord**, you need to implement the following function:

- **public abstract byte[] toBytes(int len);**

The function **bytes(int len)** should return a byte array of length **len**, which contains a serialized representation of the output.

MapReduce.Interface.Key

To implement Key, all you need to do is to implement the standard Java **Serializable, Comparable** interfaces.

MapReduce.Interface.Value

To implement Value, all you need to do is to implement the standard Java **Serializable** interface.

MapReduce.Interface.Partitioner

To implement Partitioner, you need to implement the following function:

- **public abstract int getPartitionID(K key, V value, int numPartitions);**

The function **getPartitionerID(K key, V value, int numPartitions)** should accept a **<Key, Value>** pair and a total number of partitions, then return a number ranging in 0 to numPartitions-1. This number indicates which partition the **<Key, Value>** pair should go to.

MapReduce.Interface.Mapper

To implement **Mapper**, you need to implement the following function:

- **public abstract void Map(T record, IntermediateRecordCollector<K, V> collector);**

The function **Map(T record, IntermediateRecordCollector<K, V> collector)** should accept an **InputRecord**, then apply the mapping logic, generate some intermediate results, and use the collector to collect these results.

MapReduce.Interface.Reducer

To implement Mapper, you need to implement the following function:

- **public abstract void Reduce(K key, Iterator<V> values, OutputRecordWriter writer);**

The function **Reduce(K key, Iterator<V> values, OutputRecordWriter writer)** should accept a Key, a list of Values, then apply the reducing logic, generate some **OutputRecord**, and use the writer to write these **OutputRecord** to disk.