CSE535 Asynchronous Systems
Hadoop: MapReduce

YoungMin Kwon

MapReduce

- MapReduce is a framework
  - for processing large amount of data
  - in parallel
  - on large clusters of commodity hardware
  - in a reliable manner

MapReduce

- Map:
  - Converts a set of data into another set of data
  - Each element is broken down into a tuple (key/value)
- Reduce:
  - Takes the output from a map as an input and combines those tuples into a smaller set of tuples

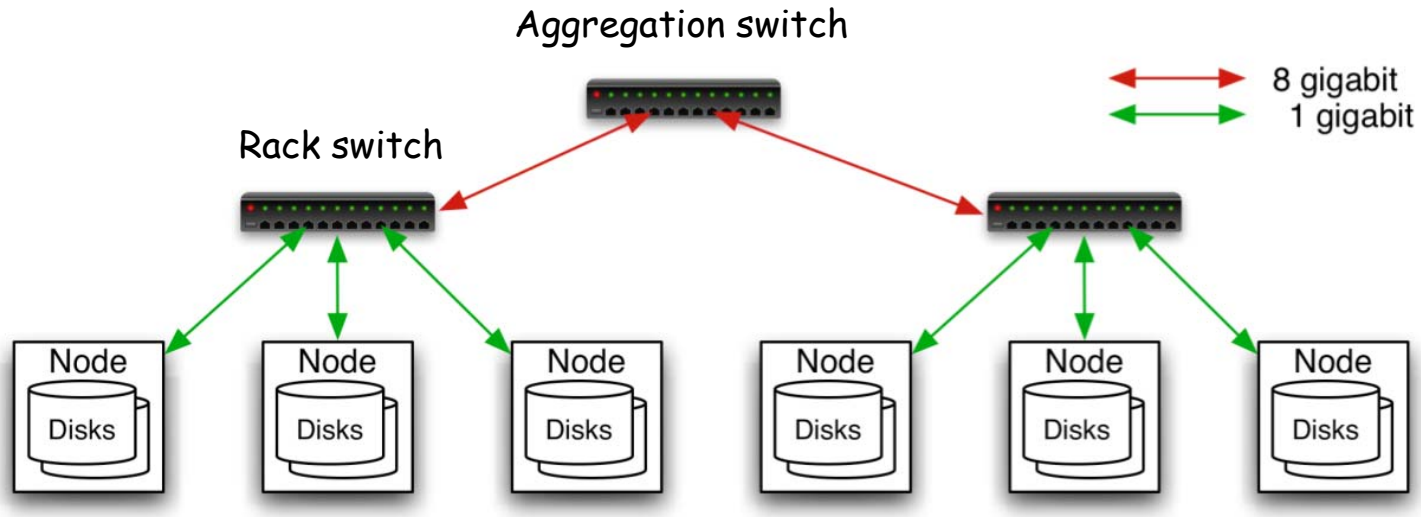
MapReduce

- Advantage of MapReduce
  - Easy to scale data processing over multiple computing nodes
  - It is nontrivial to decompose the data processing into mappers and reducers
  - However, once the decomposition is done, scaling is as simple as a configuration change

Hadoop Cluster

Hadoop Cluster

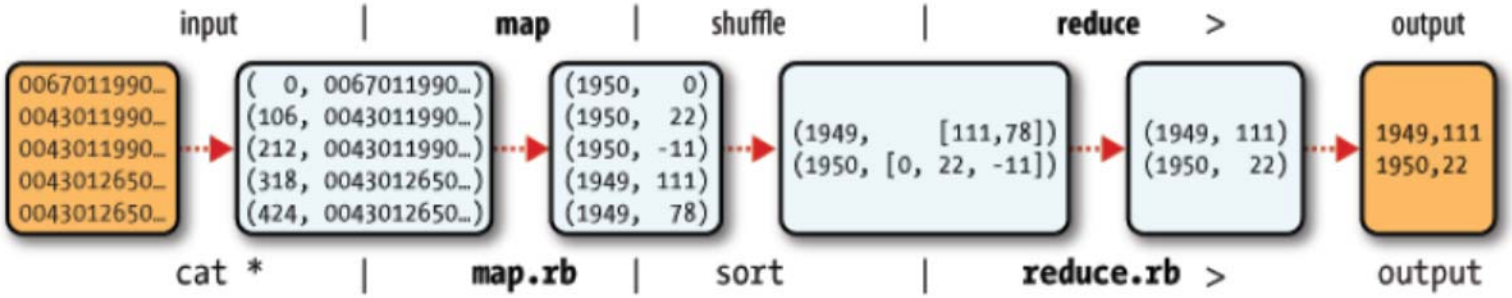
- 40 nodes/rack, 1000-4000 nodes in a cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort):
  8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)



The Algorithm

- MapReduce: sending the computer to where the data resides

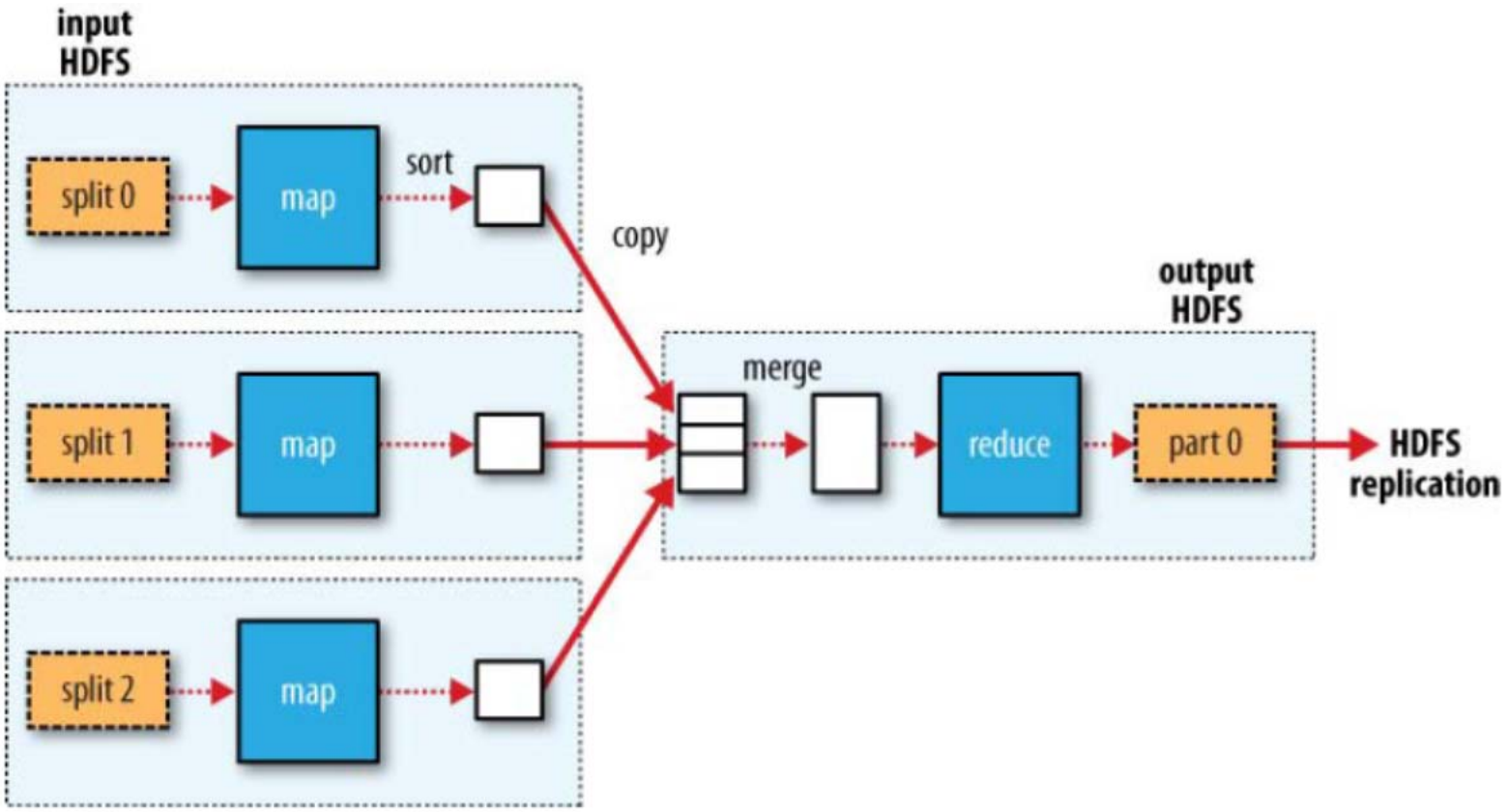
- Three stages
  - Map stage: an input file in HDFS is passed to the mapper line by line and the mapper processes them to create several chunks of data
  - Shuffle stage: sort the map output by the key
  - Reduce stage: process the data from the mapper and produce a new set of output in the HDFS



The Algorithm

- During a MapReduce job, Hadoop sends the map and reduce tasks to the appropriate servers in the cluster
- The framework manages the details of data-passing
  - Issuing tasks, verifying task completion, copying data between the nodes
- Most of the computing takes place on nodes with data on local disk (reduce network traffic)
- After completion of a task, the cluster collects and reduces the data to form an appropriate result and send it back to the Hadoop server

The Algorithm



Inputs and Outputs

- MapReduce framework operates on <key, value> pairs
- The key and value classes should be serializable by the framework and need to implement the Writable interface
- The key classes need to implement Writable-Comparable interface to facilitate sorting by the framework

	Input	Output
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

MapReduce Example

- Get weather data from National Climatic Data Center (NOAA)
  - ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products/daily01
- Find the max temperature of each year
- The weather data look like this:

```
012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
04237 20170102  2.423 -123.81   47.51     2.7    -3.3    -0.3    -0.9     0.0     3.14 C …
04237 20170103  2.423 -123.81   47.51     2.2    -4.7    -1.3    -1.5     0.0     3.21 C …
04237 20170104  2.423 -123.81   47.51     2.5    -5.0    -1.3    -2.0     0.0     3.36 C …
```

```
import java.io.IOException;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, DoubleWritable> {

    private static final double MISSING = 9999;

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();
        String year = line.substring(6,13);
        double airTemperature;
        airTemperature = Double.parseDouble(line.substring(80, 86).trim());

        if(airTemperature != MISSING) {
            context.write(new Text(year), new DoubleWritable(airTemperature));
        }
    }
}
```

```
import java.io.IOException;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;


public class MaxTemperatureReducer
    extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

    @Override
    public void reduce(Text key,
                       Iterable<DoubleWritable> values,
                       Context context)
        throws IOException, InterruptedException {

        double maxValue = Double.MIN_VALUE;
        for(DoubleWritable value: values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new DoubleWritable(maxValue));
    }
}
```

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

    public static void main(String[] args)
        throws Exception {

        if(args.length != 2) {
            System.err.println(
                 "Usage: MaxTemperature <input path> <output path>");
            System.exit(-1);
        }
...
```

```
...
        Job job = new Job();
        job.setJarByClass(MaxTemperature.class);
        job.setJobName("Max Temperature");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setReducerClass(MaxTemperatureReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(DoubleWritable.class);

        System.exit(job.waitForCompletion(true) ? 0: 1);
    }
}
```

MapReduce Example

- Build MaxTemperature.jar
  - Run hadoop classpath and add the output to the CLASSPATH variable in .bashrc (export CLASSPATH=$CLASSPATH:<hadoop classpath output>)
  - javac *.java
  - jar cf MaxTemperature.jar *.class
- Prepare input/output directories
  - hdfs dfs –mkdir /input
  - hdfs dfs –mkdir /output
- Copy sample2.txt to /input directory
  - Get sample2.txt by sftp hadoop@10.12.9.155
  - hdfs dfs –put sample2.txt /input/
- Run MaxTemperature
  - hadoop jar MaxTemperature.jar MaxTemperature /input/sample2.txt /output/sample2
  - hdfs dfs -cat /output/sample2/part-r-00000

Scaling Out: Data Flow

- MapReduce **job** consists of
  - Input data
  - MapReduce program
  - Configuration information

Data Flow

- Dividing the job into tasks
  - Map tasks
  - Reduce tasks

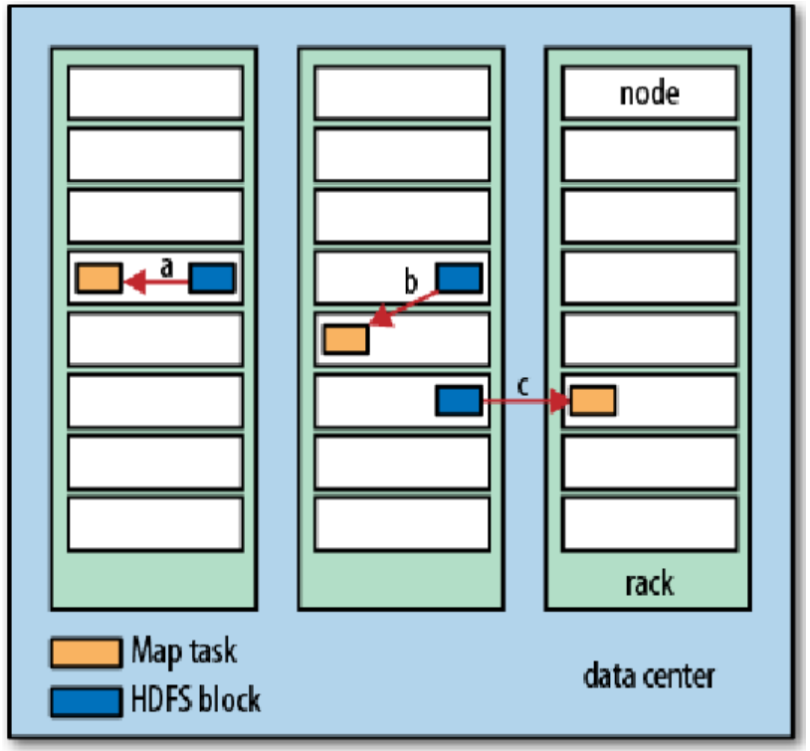
- Two types of nodes for the job execution
  - Jobtracker: coordinates all the jobs run on the system and scheduling tasks to run on tasktrackers
  - Tasktracker: run tasks and send progress reports to the jobtracker.

Data Flow

- Dividing the input into splits
  - Hadoop creates one map task for each split
  - Smaller split means better load-balancing: faster node will process more splits
  - However, it will increase the overhead of managing the splits

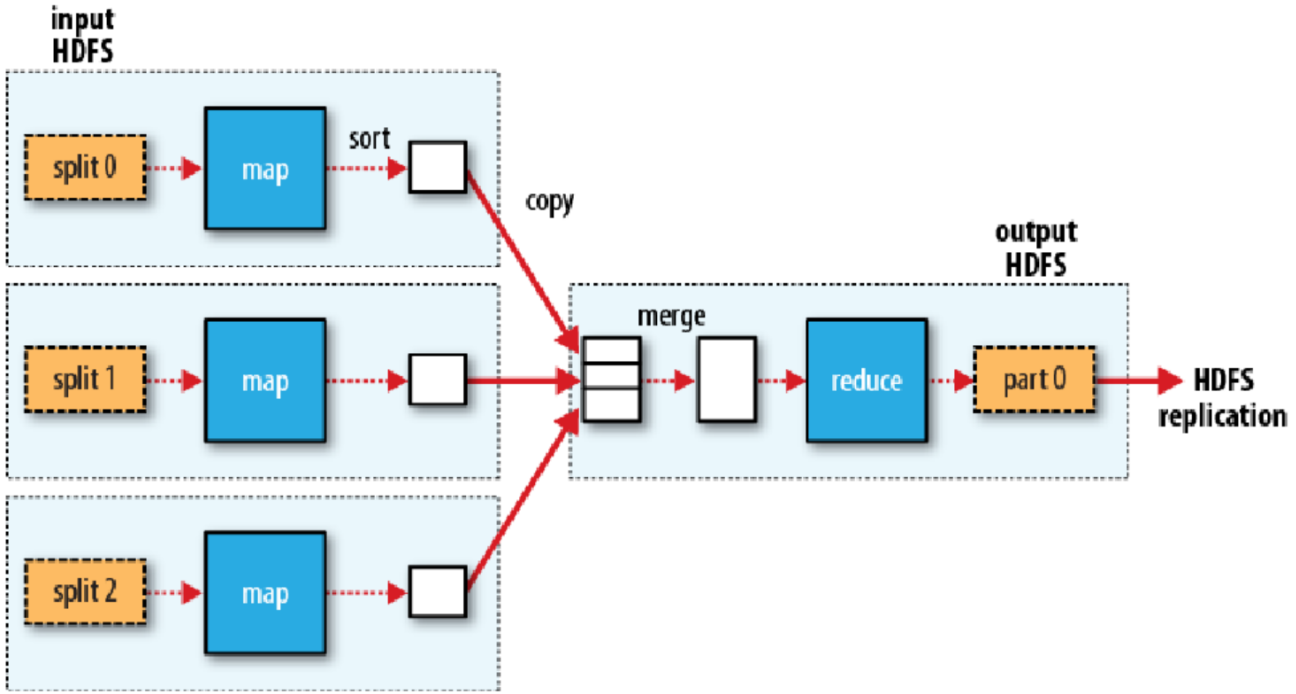
Data Flow

- Data locality optimization
  - Run the task on a node where the input data is
  - If not possible try to find a node in the same rack
  - Very occasionally, off-rack node is used
- Map tasks write their output to the local disk, not to HDFS



Data Flow

- Reduce tasks cannot enjoy the data locality
  - The sorted mapper output have to be transferred to the node where the reducer resides



Data Flow

- Multiple Reducer
  - Unlike mapper, the number of reducer is not governed by the size of input, but can be specified independently
  - Map tasks partition their output; one for each reduce task
  - A partition can hold many keys, but all values for the same key are in a single partition

Data Flow