Now that we covered the basics of MapReduce, let's look at some Hadoop specifics.

Working With Hadoop

- Mostly based on Tom White's book "Hadoop: The Definitive Guide", 3rd edition
- Note: We will use the new org.apache.hadoop.mapreduce API, but…
  - Many existing programs might be written using the old API org.apache.hadoop.mapred
  - Some old libraries might only support the old API

Important Terminology

- NameNode daemon
  - Corresponds to GFS Master
  - Runs on master node of the Hadoop Distributed File System (HDFS)
  - Directs DataNodes to perform their low-level I/O tasks
- DataNode daemon
  - Corresponds to GFS chunkserver
  - Runs on each slave machine in the HDFS
  - Does the low-level I/O work

Important Terminology

- Secondary NameNode daemon
  - One per cluster to monitor status of HDFS
  - Takes snapshots of HDFS metadata to facilitate recovery from NameNode failure
- JobTracker daemon
  - MapReduce master in Google paper
  - One per cluster, usually running on master node
  - Communicates with client application and controls MapReduce execution in TaskTrackers

Important Terminology

- TaskTracker daemon
  - MapReduce worker in Google paper
  - One TaskTracker per slave node
  - Performs actual Map and Reduce execution
  - Can spawn multiple JVMs to do the work
- Typical setup
  - NameNode and JobTracker run on cluster head node
  - DataNode and TaskTracker run on all other nodes
  - Secondary NameNode runs on dedicated machine or on cluster head node (usually not a good idea, but ok for small clusters)

Anatomy of MapReduce Job Run

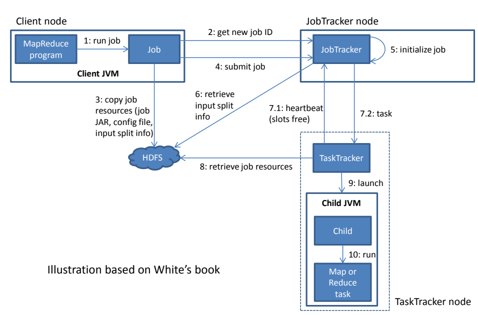


Illustration based on White's book

Job Submission

- Client submits MapReduce job through Job.submit() call
  - waitForCompletion() submits job and polls JobTracker about progress every sec, outputs to console if changed
- Job submission process
  - Get new job ID from JobTracker
  - Determine input splits for job
  - Copy job resources (job JAR file, configuration file, computed input splits) to HDFS into directory named after the job ID
  - Informs JobTracker that job is ready for execution

7

Job Initialization

- JobTracker puts ready job into internal queue
- Job scheduler picks job from queue
  - Initializes it by creating job object
  - Creates list of tasks
    - One map task for each input split
    - Number of reduce tasks determined by mapred.reduce.tasks property in Job, which is set by setNumReduceTasks()
- Tasks need to be assigned to worker nodes

8

Task Assignment

- TaskTrackers send heartbeat to JobTracker
  - Indicate if ready to run new tasks
  - Number of "slots" for tasks depends on number of cores and memory size
- JobTracker replies with new task
  - Chooses task from first job in priority-queue
    - Chooses map tasks before reduce tasks
    - Chooses map task whose input split location is closest to machine running the TaskTracker instance
      - Ideal case: data-local task
  - Could also use other scheduling policy

9

Task Execution

- TaskTracker copies job JAR and other configuration data (e.g., distributed cache) from HDFS to local disk
- Creates local working directory
- Creates TaskRunner instance
- TaskRunner launches new JVM (or reuses one from another task) to execute the JAR

10

Monitoring Job Progress

- Tasks report progress to TaskTracker
- TaskTracker includes task progress in heartbeat message to JobTracker
- JobTracker computes global status of job progress
- JobClient polls JobTracker regularly for status
- Visible on console and Web UI

11

Handling Failures: Task

- Error reported to TaskTracker and logged
- Hanging task detected through timeout
- JobTracker will automatically re-schedule failed tasks
  - Tries up to mapred.map.max.attempts many times (similar for reduce)
  - Job is aborted when task failure rate exceeds mapred.max.map.failures.percent (similar for reduce)

12

Handling Failures: TaskTracker and JobTracker

- TaskTracker failure detected by JobTracker from missing heartbeat messages
  - JobTracker re-schedules map tasks and not completed reduce tasks from that TaskTracker
- Hadoop cannot deal with JobTracker failure
  - Could use Google's proposed JobTracker take-over idea, using ZooKeeper to make sure there is at most one JobTracker

13

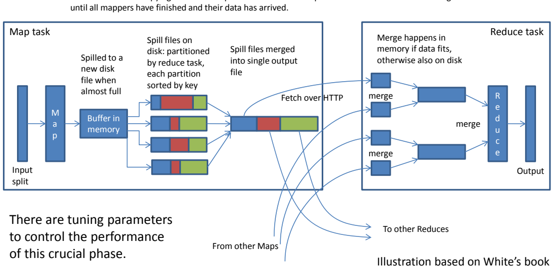
Moving Data From Mappers to Reducers

- Shuffle and sort phase = synchronization barrier between map and reduce phase
- Often one of the most expensive parts of a MapReduce execution
- Mappers need to separate output intended for different reducers
- Reducers need to collect their data from all mappers and group it by key
- Keys at each reducer are processed in order

14

Shuffle and Sort Overview

Reduce task starts copying data from map task as soon as it completes. Reduce cannot start working on the data until all mappers have finished and their data has arrived.



There are tuning parameters to control the performance of this crucial phase.

Illustration based on White's book

15

NCDC Weather Data Example

- Raw data has lines like these (year, temperature in **bold**)
  - 0067011990999991**950**051507004+68750+023550FM-12+038299999V0203301N00671220001CN9999999N9**+00** **00**1+99999999999
  - 0043011990999991**950**051512004+68750+023550FM-12+038299999V0203201N00671220001CN9999999N9**+00** **22**1+99999999999
- Goal: find max temperature for each year
  - Map: emit (year, temp) for each year
  - Reduce: compute max over temp from (year, (temp, temp,…)) list

16

Map

- Hadoop's Mapper class
  - Org.apache.hadoop.mapreduce.Mapper
- Type parameters: input key type, input value type, output key type, and output value type
  - Input key: line's offset in file (irrelevant)
  - Input value: line from NCDC file
  - Output key: year
  - Output value: temperature
- Data types are optimized for network serialization
  - Found in org.apache.hadoop.io package
- Work is done by the map() method

17

Map() Method

- Input: input key type, input value type (and a Context)
  - Line of text from NCDC file
  - Converted to Java String type, then parsed to get year and temperature
- Output: written using Context
  - Uses output key and value types
- Only write (year, temp) pair if the temperature is present and quality indicator reading is OK

18

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
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, IntWritable> {

 private static final int MISSING = 9999;

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

  String line = value.toString();
  String year = line.substring(15, 19);
  int airTemperature;
  if (line.charAt(87) == '+') {                          // parseInt doesn't like leading plus signs
   airTemperature = Integer.parseInt(line.substring(88, 92));
  } else {
   airTemperature = Integer.parseInt(line.substring(87, 92));
  }
  String quality = line.substring(92, 93);

  if (airTemperature != MISSING && quality.matches("[01459]")) {
   context.write(new Text(year), new IntWritable(airTemperature));
  }
 }
}
```


Reduce

- Implements org.apache.hadoop.mapreduce.Reducer
- Input key and value types must match Mapper output key and value types
- Work is done by reduce() method
  - Input values passed as Iterable list
  - Goes over all temperatures to find the max
  - Result pair is written by using the Context
    - Writes result to HDFS, Hadoop's distributed file system


```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

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

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

  int maxValue = Integer.MIN_VALUE;
  for (IntWritable value : values) {
   maxValue = Math.max(maxValue, value.get());
  }
  context.write(key, new IntWritable(maxValue));
 }
}
```


Job Configuration

- Job object forms the job specification and gives control for running the job
- Specify data input path using addInputPath()
  - Can be single file, directory (to use all files there), or file pattern
  - Can be called multiple times to add multiple paths
- Specify output path using setOutputPath()
  - Single output path, which is a directory for all output files
- Set mapper and reducer class to be used
- Set output key and value classes for map and reduce functions
  - For reducer: setOutputKeyClass(), setOutputValueClass()
  - For mapper (omit if same as reducer): setMapOutputKeyClass(), setMapOutputValueClass()
- Can set input types similarly (default is TextInputFormat)
- Method waitForCompletion() submits job and waits for it to finish


```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
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(IntWritable.class);

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


Extension: Combiner Functions

- Recall earlier discussion about combiner function
  - Pre-reduces mapper output before transfer to reducers
  - Does not change program semantics
- Usually (almost) same as reduce function, but has to have same output type as Map
- Works only for some reduce functions that can be incrementally computed
  - MAX(5, 4, 1, 2) = MAX(MAX(5, 1), MAX(4, 2))
  - Same for SUM, MIN, COUNT, AVG (=SUM/COUNT)

```
public class MaxTemperatureWithCombiner {

 public static void main(String[] args) throws Exception {
  if (args.length != 2) {
   System.err.println("Usage: MaxTemperatureWithCombiner <input path> " + "<output path>");
   System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(MaxTemperatureWithCombiner.class);
  job.setJobName("Max temperature");

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

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

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

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

Note: combiner here is identical to reducer class.

25

Extension: Custom Partitioner

- Partitioner determines which keys are assigned to which reduce task
- Default HashPartitioner essentially assigns keys randomly
- Create custom partitioner by implementing your own getPartition() method of Partitioner in org.apache.hadoop.mapreduce

26

MapReduce Development Steps

1. Write Map and Reduce functions
   – Create unit tests
2. Write driver program to run a job
   – Can run from IDE with small data subset for testing
   – If test fails, use IDE for debugging
   – Update unit tests and Map/Reduce if necessary
3. Once program works on small test set, run it on full data set
   – If there are problems, update tests and code accordingly
4. Fine-tune code, do some profiling

27

Local (Standalone) Mode

- Runs same MapReduce user program as cluster version, but does it sequentially
- Does not use any of the Hadoop daemons
- Works directly with local file system
  – No HDFS, hence no need to copy data to/from HDFS
- Great for development, testing, initial debugging

28

Pseudo-Distributed Mode

- Still runs on single machine, but now simulates a real Hadoop cluster
  – Simulates multiple nodes
  – Runs all daemons
  – Uses HDFS
- Main purpose: more advanced testing and debugging
- You can also set this up on your laptop

29

Extension: MapFile

- Sorted file of (key, value) pairs with an index for lookups by key
- Must append new entries in order
  – Can create MapFile by sorting SequenceFile
- Can get value for specific key by calling MapFile's get() method
  – Found by performing binary search on index
- Method getClosest() finds closest match to search key

30

Extension: Counters

- Useful to get statistics about the MapReduce job, e.g., how many records were discarded in Map
- Difficult to implement from scratch
  - Mappers and reducers need to communicate to compute a global counter
- Hadoop has built-in support for counters
- See ch. 8 in Tom White's book for details

31

Hadoop Job Tuning

- Choose appropriate number of mappers and reducers
- Define combiners whenever possible
  - But see also later discussion about local aggregation
- Consider Map output compression
- Optimize the expensive shuffle phase (between mappers and reducers) by setting its tuning parameters
- Profiling distributed MapReduce jobs is challenging.

32

Hadoop and Other Programming Languages

- Hadoop Streaming API to write map and reduce functions in languages other than Java
  - Any language that can read from standard input and write to standard output
- Hadoop Pipes API for using C++
  - Uses sockets to communicate with Hadoop's task trackers

33

Multiple MapReduce Steps

- Example: find average max temp for every day of the year and every weather station
  - Find max temp for each combination of station and day/month/year
  - Compute average for each combination of station and day/month
- Can be done in two MapReduce jobs
  - Could also combine it into single job, which would be faster

34

Running a MapReduce Workflow

- Linear chain of jobs
  - To run job2 after job1, create JobConf's conf1 and conf2 in main function
  - Call JobClient.runJob(conf1); JobClient.runJob(conf2);
  - Catch exceptions to re-start failed jobs in pipeline
- More complex workflows
  - Use JobControl from org.apache.hadoop.mapred.jobcontrol
  - We will see soon how to use Pig for this

35

MapReduce Coding Summary

- Decompose problem into appropriate workflow of MapReduce jobs
- For each job, implement the following
  - Job configuration
  - Map function
  - Reduce function
  - Combiner function (optional)
  - Partition function (optional)
- Might have to create custom data types as well
  - WritableComparable for keys
  - Writable for values

36