Lecture 3: MapReduce Programming

CSCI4180 (Fall 2013)

Patrick P. C. Lee

Outline

- MapReduce programming
- How a MapReduce program works?
- Possible ways to fine-tune a MapReduce program

Overview

- Typical flow of writing a MapReduce program:
  - Implement map and reduce functions based on their definitions
  - Write a driver program to run the job, either locally or on a cluster platform
  - Debug the program on a small dataset
  - Debug the program on a large dataset
  - Fine-tune the program to improve its performance

Overview

- ➢ Hadoop provides a framework to run MapReduce programs
  - You compile execute a MapReduce program via Hadoop
  - Just like you execute a Java program on JVM
- ➢ Hadoop allows you to run MapReduce programs on HDFS, which can be mounted on a single node or multiple nodes

Prerequisites

- Install **hadoop 0.20.203.0**
  - Older versions use different APIs
  - Run **`hadoop version`** to find out the version

```
[hduser@localhost ~]$ hadoop version
Hadoop 0.20.203.0
Subversion http://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20-security-203
-r 1099333
Compiled by oom on Wed May  4 07:57:50 PDT 2011
```

- Use **Java v1.6** to write MapReduce applications
  - MapReduce applications can be written in other languages as well (e.g., Python, C++)
- See instructions on how to install Hadoop.
  - On course website

Hadoop Operational Modes

- Standalone (local) mode
  - There are no daemons running and everything runs in a single JVM. Standalone mode is suitable for running MapReduce programs during development, since it is easy to test and debug them.
- Pseudo-distributed mode
  - The Hadoop daemons run on the local machine, thus simulating a cluster on a small scale.
- Fully distributed mode
  - The Hadoop daemons run on a cluster of machines.

Hadoop Configuration

- Hadoop use a collection of configuration properties and values
- Configurations can be either defined in XML files (offline), or defined in programs (online)
  - Resources are XML files that define properties/values
- XML format (with name, value, description (optional)):

```xml
<?xml version="1.0"?>
<configuration>
  <property>
    <name>size</name>
    <value>12</value>
    <description>This is Size</description>
  </property>
  <property>
    <name>weight</name>
    <value>light</value>
  </property>
</configuration>
```

Hadoop Configuration

- Resources can be added within programs

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
conf.addResource("configuration-2.xml");
```

- Properties defined in resources that are added later override earlier definitions
- Properties that are marked **`final`** cannot be overridden in later definitions

```
<property>
  <name>weight</name>
  <value>light</value>
  <final>true</final>
</property>
```

Hadoop Configuration

- Default Hadoop configuration files:

Filename	**Description**
core-site.xml	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes.
mapred-site.xml	Configuration settings for MapReduce daemons: the jobtracker, and the tasktrackers.

Hadoop Configuration

- Example: configurations for pseudo-distributed mode:

```xml
<!-- conf-site.xml -->
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/app/hadoop/tmp</value>
  </property>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:54310</value>
  </property>
</configuration>
```

```xml
<!-- hdfs-site.xml -->
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

```xml
<!-- mapred-site.xml -->
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:54311</value>
  </property>
</configuration>
```

Hadoop Configuration

- *Key configuration properties for different modes*

Component	Property	Standalone	Pseudo-distributed	Fully distributed
Common	`fs.default.name`	`file:///` (default)	`hdfs://localhost/`	`hdfs://`*namenode*`/`
HDFS	`dfs.replication`	N/A	1	3 (default)
MapReduce	`mapred.job.tracker`	`local` (default)	`localhost:8021`	*jobtracker*`:8021`

- In standalone mode, there is no further action to take, since the default properties are set for standalone mode, and there are no daemons to run.

Starting/Stopping Hadoop

- Starting single-node cluster

```
hduser@localhost: start-all.sh
```

  - This will startup Namenode, Datanode, Jobtracker, and Tasktracker on the local machine
  - That is, HDFS is mounted. Home directory is **`/user/<username>`**

- Stopping single-node cluster

```
hduser@localhost: stop-all.sh
```

HDFS Operations

Description	Commands
List files	`$ hadoop dfs -ls /`
Check disk usage	`$ hadoop dfs -du /`
Create directories	`$ hadoop dfs -mkdir /`
Copy files	`$ hadoop dfs -put file01.txt /` (Alternative) `$ hadoop dfs -copyFromLocal file01.txt /`
Retrieve files	`$ hadoop dfs -get file01.txt local/file01.txt`
Delete files	`$ hadoop dfs -rm file01.txt`
Delete (recursive)	`$ hadoop dfs -rmr dir`

References:
http://hadoop.apache.org/common/docs/r0.20.203.0/file_system_shell.html

“Hello World” Program

- **WordCount**: count the occurrences of each word in a set of files
  - Get WordCount.java on course website
  - Run on pseudo-distributed mode
- Sample text-files as input:

```
$ hadoop dfs –put file01.txt wordcount/input
$ hadoop dfs –put file02.txt wordcount/input

$ hadoop dfs -ls /user/hduser/wordcount/input/
/user/hduser/wordcount/input/file01.txt
/user/hduser/wordcount/input/file02.txt

$ hadoop dfs -cat /usr/joe/wordcount/input/file01.txt
Hello World Bye World

$ hadoop dfs -cat /usr/joe/wordcount/input/file02.txt
Hello Hadoop Goodbye Hadoop
```

“Hello World” Program

- Compile the program:

```
$ mkdir wordcount
$ javac -classpath /usr/local/hadoop/hadoop-core-0.20.203.0.jar WordCount.java -d wordcount
$ jar -cvf wordcount.jar -C wordcount/ .
```

- Run the program

```
$ hadoop jar wordcount.jar org.myorg.WordCount wordcount/input wordcount/output
```

- Output:

```
$ hadoop dfs -cat /user/hduser/wordcount/output/part-r-00000
Bye     1
Goodbye 1
Hadoop  2
Hello   2
World   2
```

Dissection: Mapper

- Interface:

```
public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

- How to define:

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   ...
   public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
        ...
      context.write(...)
   }
}
```

Dissection: Mapper

- Implementation:
  - Process one line at a time
  - Splits each line into tokens
  - Emits a key-value pair of `<<word>, 1>`
  - Example (for the first map):

```
<Hello, 1>
<World, 1>
<Bye, 1>
<World, 1>
```

  - Example (for the second map):

```
<Hello, 1>
<Hadoop, 1>
<Goodbye, 1>
<Hadoop, 1>
```

Dissection: Combiner

- We specify a combiner (same as the Reducer here), which performs local aggregation on the map results after being sorted on keys

```
job.setCombinerClass(Reduce.class);
```

- Output:
  - for the first map:

```
<Bye, 1>
<Hello, 1>
<World, 2>
```

  - for the second map:

```
<Goodbye, 1>
<Hadoop, 2>
<Hello, 1>
```

Dissection: Reducer

- Interface:

```
public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

- How to define:

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
         ...
         context.write(...);
   }
}
```

Dissection: Reducer

- Reducer has three phases:
  - Shuffle
    - The Reducer copies the sorted output from each Mapper using HTTP across the network.
  - Sort
    - The framework merge sorts Reducer inputs by keys (since different Mappers may have output the same key)
    - Secondary sort on intermediate keys is allowed
    - The shuffle and sort phases occur simultaneously, i.e., while outputs are being fetched, they are merged.
  - Reduce
    - Implemented in reduce() method

Debug

- You can run the MapReduce program in standalone (local) mode
- Use the following lines (without modifying XML configuration files and restarting hadoop)

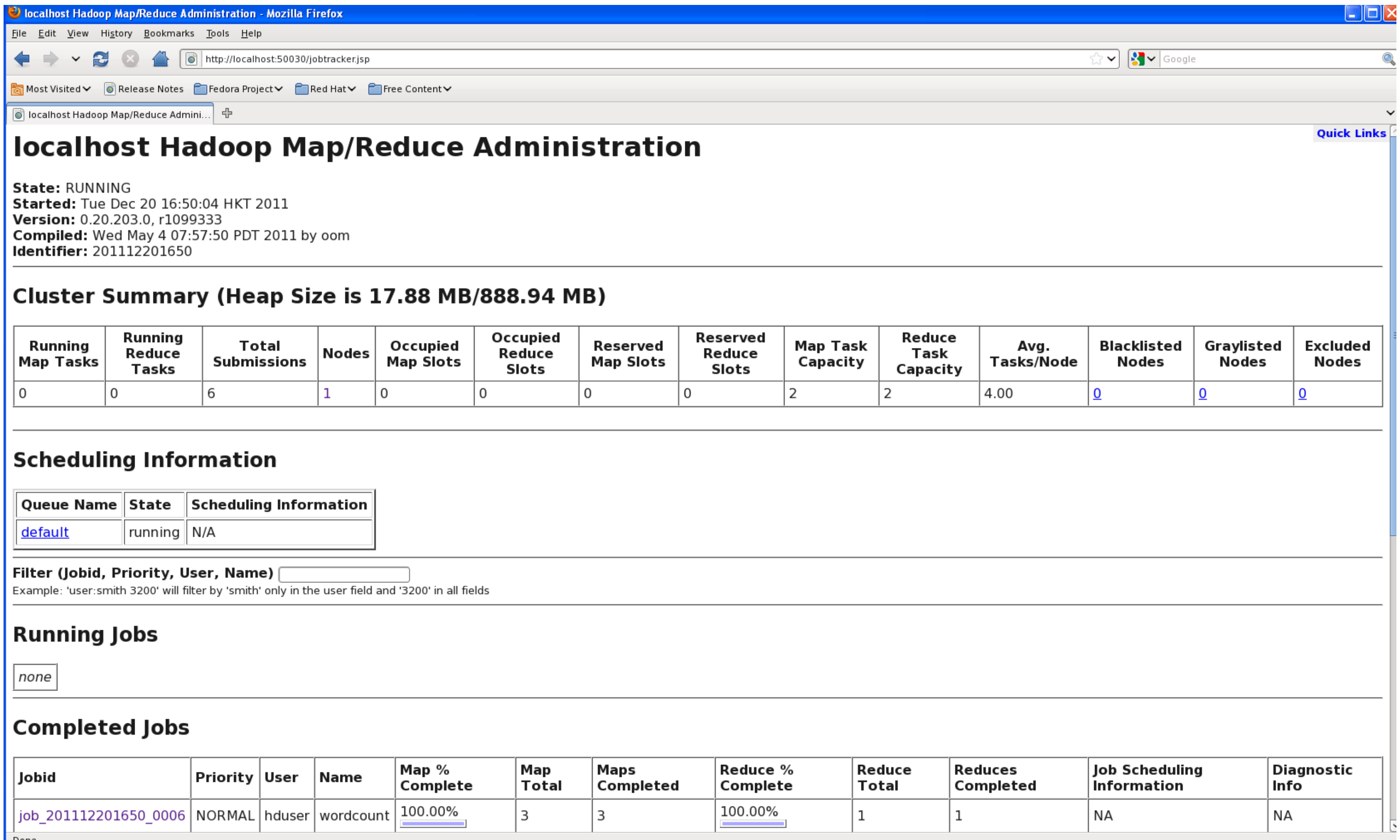
```
public static void main(String[] args) throws Exception {
    // Run on a local node
    Configuration conf = new Configuration();
    conf.set("fs.default.name", "file:///");
    conf.set("mapred.job.tracker", "local");

    Job job = new Job(conf, "wordcount");
   ...
```

- You can insert **`System.out.println()`** / **`System.err.println()`** inside map/reduce methods

MapReduce Web UI

- Running on **http://localhost:50030**



localhost Hadoop Map/Reduce Administration - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost:50030/jobtracker.jsp

Most Visited Release Notes Fedora Project Red Hat Free Content

localhost Hadoop Map/Reduce Admini...

Quick Links

localhost Hadoop Map/Reduce Administration

State: RUNNING
Started: Tue Dec 20 16:50:04 HKT 2011
Version: 0.20.203.0, r1099333
Compiled: Wed May 4 07:57:50 PDT 2011 by oom
Identifier: 201112201650

Cluster Summary (Heap Size is 17.88 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Excluded Nodes
0	0	6	1	0	0	0	0	2	2	4.00	0	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201112201650_0006	NORMAL	hduser	wordcount	100.00%	3	3	100.00%	1	1	NA	NA

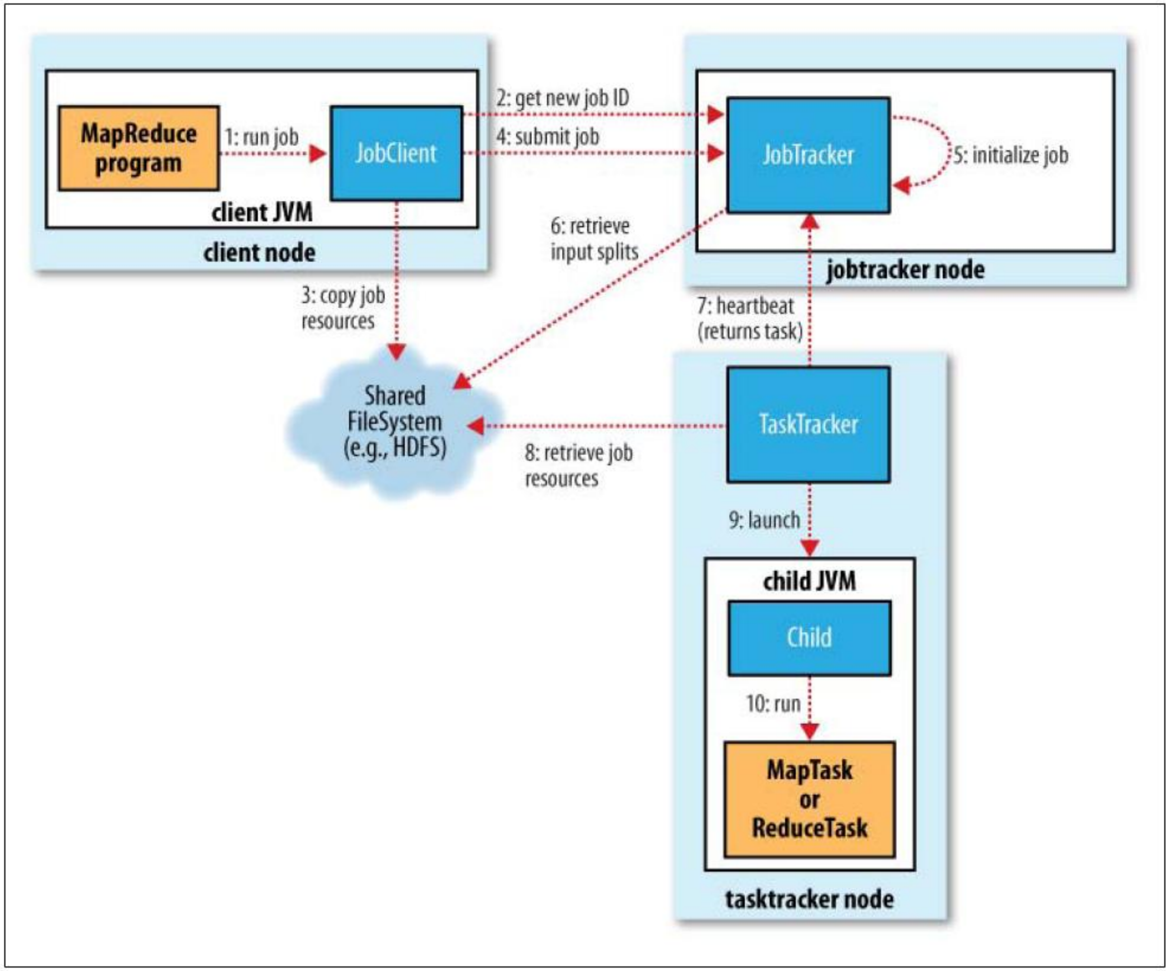
Done

Anatomy of a MapReduce Job Run

- Recall that four entities are involved in a MapReduce job run:
  - Client, which submits the MapReduce job
  - Jobtracker, which coordinates the job run
  - Tasktrackers, each running the tasks that the job has been split into
  - Distributed filesystem, used for sharing job files between other entities

Anatomy of a MapReduce Job Run

- How Hadoop runs a MapReduce job?



Job Submission

- Client asks jobtracker for a new job ID (step 2)
- Client copies resources to the filesystem (step 3)
  - e.g., job JAR file, configuration file, input splits (i.e., split blocks of an input)
  - Job JAR file is copied with a high replication factor (e.g., 10)
- Client tells jobtracker the job is ready for execution (step 4)

Job Initialization

- Jobtracker creates an object to represent the job being run and bookkeeping information (step 5)
- Job scheduler retrieves input splits from the file system (step 6)
  - It creates one map task for each split.
  - The number of reduce tasks depends on the configurations
  - Each task is assigned an ID

Task Assignment

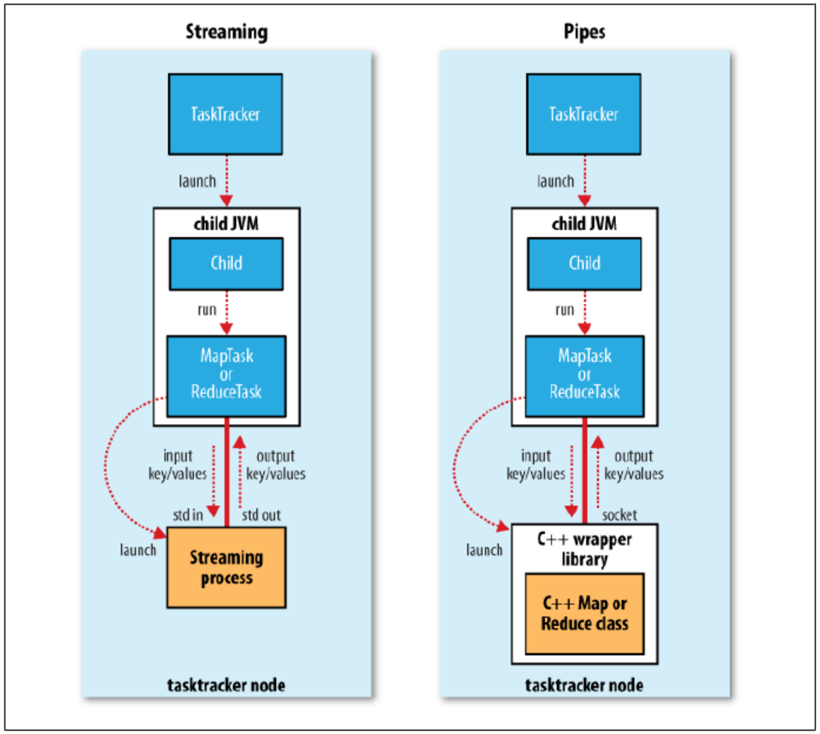
- Tasktrackers periodically send heartbeats to jobtracker to (i) keep alive and (ii) tell if being ready to run a task
- Jobtracker then allocates a task to each tasktracker (step 7)
  - Tasktrackers have a fixed number of slots to store map and reduce tasks.
  - Jobtracker tries to allocate a map task first, and then a reduce task
  - Jobtracker allocates a map task whose input split is close to where Tasktracker resides (data locality or rack locality)

Task Execution

- Tasktracker copies JAR file, and other resources, to its local filesystem (step 8)
- Tasktracker launches a new Java Virtual Machine (JVM) to run each task (steps 9-10)
  - Any bug in a task doesn't affect other tasks
  - JVM can be reused for another task
  - Parent-child process hierarchy is formed:
    - Parent process: who launches the JVM
    - Child process: who runs the task inside the JVM

Streaming and Pipes

- Streaming and pipes are communication paradigms between parent and child processes
  - Streaming: standard input/output streams
  - Pipes: socket stream
- Key/value pairs are exchanged



Progress and Status Updates

- Typically MapReduce jobs are long-running batch jobs
- Each task keeps track of its progress, the proportion of the task completed
- Why progress is useful?
  - For profiling / fine-tuning
  - For debugging

Job Completion

- A job is complete if Jobtracker receives a notification that the last task for a job is complete
- Jobtracker will:
  - Send an HTTP job notification to client
  - Clean up its working state
  - Notify tasktrackers to clean up the states

Handling Failures

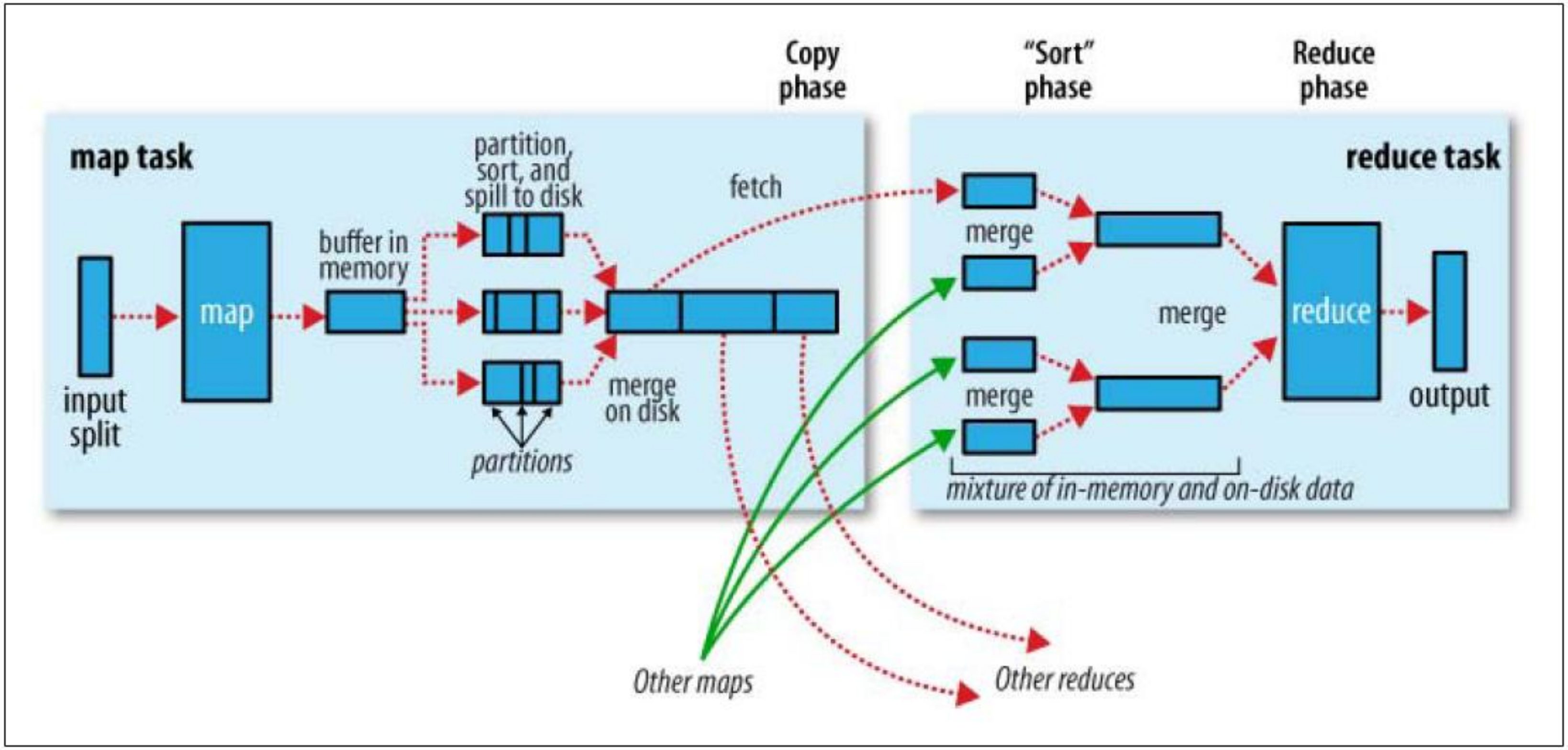
- Hadoop is designed to be fault tolerant:
  - Task failure
    - If a task is crashed or hanging, it fails
    - Jobtracker reschedules the task on a different tasktracker
    - If any task fails 4 times (default), whole job fails.
  - Tasktracker failure
    - Arranges for map tasks that were run and completed successfully on the failed tasktracker to be rerun, since the intermediate output may not yet be run by the reduce task
  - Jobtracker failure
    - The whole job fails → single point of failure

Shuffle and Sort

- MapReduce guarantees that the input to every reducer is sorted by key
- Shuffle is the process by which the system performs the sort in map and transfers the map outputs to reduces as inputs
  - Heart of MapReduce!!
- Shuffle is done on both map and reduce sides:
  - Map side: produces outputs
  - Reduce side: reads map outputs

Shuffle and Sort

- Shuffle and sort



Shuffle and Sort: Map Side

- Each map task has a circular memory buffer that it writes the output to (100MB default)
- If buffer size reaches threshold, a background thread spills the contents to disk
- Each spill is partitioned and sorted before being written to disk
- The partitions are made available to the reducers over HTTP

Shuffle and Sort: Reduce Side

- Copy phase:
  - Fetches map outputs from different map tasks
  - Multiple copy threads (5 default) are used
- Sort phase:
  - Merges map outputs and maintains sort ordering
- Reduce phase:
  - Performs the reduce function and writes output to the filesystem (e.g., HDFS)

Configuration Tuning

- You can specify configurations in XML file with property-value pair
- You can specify configuration in programs:

```
Configuration conf = new Configuration();

// set memory buffer for map outputs to 200MB
conf.setInt("io.sort.mb", 200);

Job job = new Job(conf, "wordcount");
```

Task Execution – Optimization

- Speculative execution
  - Job execution time is bottlenecked by the slowest running task
    - Straggler: a machine that takes unusually long time to finish the last few tasks
    - Why straggler? Dying harddisks, many background jobs, program bugs
  - If a task is running slow, Hadoop launches another, equivalent, task as a backup
  - When the task finishes, any duplicate tasks are killed
  - It’s a feature for optimization rather than reliability

Task Execution – Optimization

- Task JVM Reuse
  - Each task is running on its own JVM. If there are many short-lived tasks, overhead of starting a new JVM becomes significant
  - When task JVM reuse is enabled, tasks do *not* run concurrently in a single JVM. The JVM runs tasks sequentially

Task Execution – Optimization

- Skipping bad records
  - Bad records throw runtime exceptions, causing a task to retry or even halt (after 4 retries)
  - If skipping mode is enabled, failed records are skipped (only after the task is retried twice)
    - i.e., still tries the whole task on the failed record twice; if it still fails, skips it
  - How many retries can be configured

Counters

- Counters are a useful channel for gathering statistics about a job
  - For quality-control, statistics, debugging
- Hadoop maintains built-in counters for a job, but user-defined counters are allowed
- User-defined counters
  - Java enum type
  - Counters are global: MapReduce aggregates them across all map and reduce tasks

Counters

Define the counter(s) of enum type

```
enum WordCount {
   NUM_OF_TOKENS
}

public static class Map extends
        Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context)
         throws IOException, InterruptedException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         context.getCounter(WordCount.NUM_OF_TOKENS).increment(1);
         context.write(word, one);
      }
   }
}
```

increment
the counter
by some values

Counters

➢ Output

```
11/12/23 14:55:55 INFO mapred.JobClient: Job complete: job_201112231429_0002
11/12/23 14:55:55 INFO mapred.JobClient: Counters: 26
11/12/23 14:55:55 INFO mapred.JobClient:   Job Counters
11/12/23 14:55:55 INFO mapred.JobClient:     Launched reduce tasks=1
11/12/23 14:55:55 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=17262
11/12/23 14:55:55 INFO mapred.JobClient:     Total time spent by all reduces waiting
after reserving slots (ms)=0
11/12/23 14:55:55 INFO mapred.JobClient:     Total time spent by all maps waiting after
reserving slots (ms)=0
11/12/23 14:55:55 INFO mapred.JobClient:     Launched map tasks=2
11/12/23 14:55:55 INFO mapred.JobClient:     Data-local map tasks=2
11/12/23 14:55:55 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=10355
11/12/23 14:55:55 INFO mapred.JobClient:   File Output Format Counters
11/12/23 14:55:55 INFO mapred.JobClient:     Bytes Written=41
11/12/23 14:55:55 INFO mapred.JobClient:   FileSystemCounters
11/12/23 14:55:55 INFO mapred.JobClient:     FILE_BYTES_READ=79
11/12/23 14:55:55 INFO mapred.JobClient:     HDFS_BYTES_READ=272
11/12/23 14:55:55 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=63943
11/12/23 14:55:55 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=41
11/12/23 14:55:55 INFO mapred.JobClient:   File Input Format Counters
11/12/23 14:55:55 INFO mapred.JobClient:     Bytes Read=50
11/12/23 14:55:55 INFO mapred.JobClient:   org.myorg.WordCountWithCounter$WordCount
11/12/23 14:55:55 INFO mapred.JobClient:     NUM_OF_TOKENS=8
```

Summary

- How to write a MapReduce program?
- How a MapReduce program works inside Hadoop?
- How to possibly optimize/fine-tune a MapReduce program?