Lecture 3: MapReduce Programming

CSCI4180

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 and 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

- Hadoop has release branches: 0.x, 1.x, 2.x, 3.x
  - 0.x and 1.x use Classical MapReduce runtime
    - MapReduce 1
  - 2.x and 3.x use a new MapReduce runtime called YARN
    - MapReduce 2
  - There are still updates for 2.x
  - APIs (slightly) differ across releases
- We focus on 2.10.2
  - Released in May 2022
- Hadoop 3's new features
  - Faster than Hadoop 2
  - Support erasure coding for low-cost fault tolerance

Prerequisites

- Install **hadoop 2.10.2**
  - Older versions use different APIs
  - Run "`hadoop version`" to find out the version

```
hduser@proj19:~$ hadoop version
Hadoop 2.10.2
Subversion Unknown -r 965fd380006fa78b2315668fbc7eb432e1d8200f
Compiled by ubuntu on 2022-05-24T22:35Z
Compiled with protoc 2.5.0
From source with checksum d3ab737f7788f05d467784f0a86573fe
This command was run using /home/hduser/hadoop-2.10.2/share/hadoop/common/hadoop-common-
2.10.2.jar
```

- Use **Java v1.8** to write MapReduce applications
  - Check by "`java -version`"
- *(Fall 2022) For Assignment 1, any Hadoop 2.x >= 2.7.3 is fine*

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 Hadoop Core, such as I/O settings that are common to HDFS, MapReduce, and YARN
mapred-site.xml	Configuration settings for MapReduce daemons: the job history server
yarn-site.xml	Configuration settings for YARN daemons: the resource manager, the web app proxy server, and the node managers

Hadoop Configuration

- Example: configurations for pseudo-distributed mode:

```xml
<!-- core-site.xml -->
<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/home/hduser/hadoop-2.10.2/tmp</value>
</property>
<property>
  <name>fs.defaultFS</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>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Hadoop Configuration

- Example: configurations for pseudo-distributed mode:

```
<!-- yarn-site.xml -->
<configuration>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>localhost</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
<property>
  <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
  <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
</configuration>
```

Hadoop Configuration

- *Key configuration properties for different modes*

Component	Property	Standalone	Pseudodistributed	Fully distributed
Common	`fs.defaultFS`	`file:///` (default)	`hdfs://localhost/`	`hdfs://`*`namenode`*`/`
HDFS	`dfs.replication`	N/A	1	3 (default)
MapReduce	`mapreduce.framework.name`	`local` (default)	`yarn`	`yarn`
YARN	`yarn.resourcemanager.hostname`	N/A	`localhost`	*`resourcemanager`*
	`yarn.nodemanager.aux-services`	N/A	`mapreduce_shuffle`	`mapreduce_shuffle`

Before You Start...

- Configuring SSH to login without password:
  - Try **`ssh localhost`**

```
$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

- Set environment variable **`JAVA_HOME`** in **`etc/hadoop-env.sh`**

Starting/Stopping Hadoop

- Before you start (no Hadoop daemons are running):

```
hduser@localhost:  hadoop namenode -format
```

- Starting single-node cluster (including HDFS and YARN)

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

  - Check by **`jps`**

```
28384 NodeManager
27559 DataNode
27322 NameNode
28138 ResourceManager
31036 Jps
27918 SecondaryNameNode
```

- Stopping single-node cluster

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

HDFS Operations

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

References:
https://hadoop.apache.org/docs/r2.10.2/hadoop-project-dist/hadoop-common/FileSystemShell.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 fs -mkdir /input
$ hadoop fs –put file01.txt /input/file01.txt
$ hadoop fs –put file02.txt /input/file02.txt

$ hadoop fs -ls /input
/input/file01.txt
/input/file02.txt

$ hadoop fs -cat /input/file01.txt
Hello World Bye World

$ hadoop fs -cat /input/file02.txt
Hello Hadoop Goodbye Hadoop
```

“Hello World” Program

- Compile the program:

```
$ mkdir wordcount
$ javac -classpath `yarn classpath` WordCount.java -d wordcount
$ jar -cvf wordcount.jar -C wordcount/ .
```

- Run the program

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

- Output:

```
$ hadoop fs -cat /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)

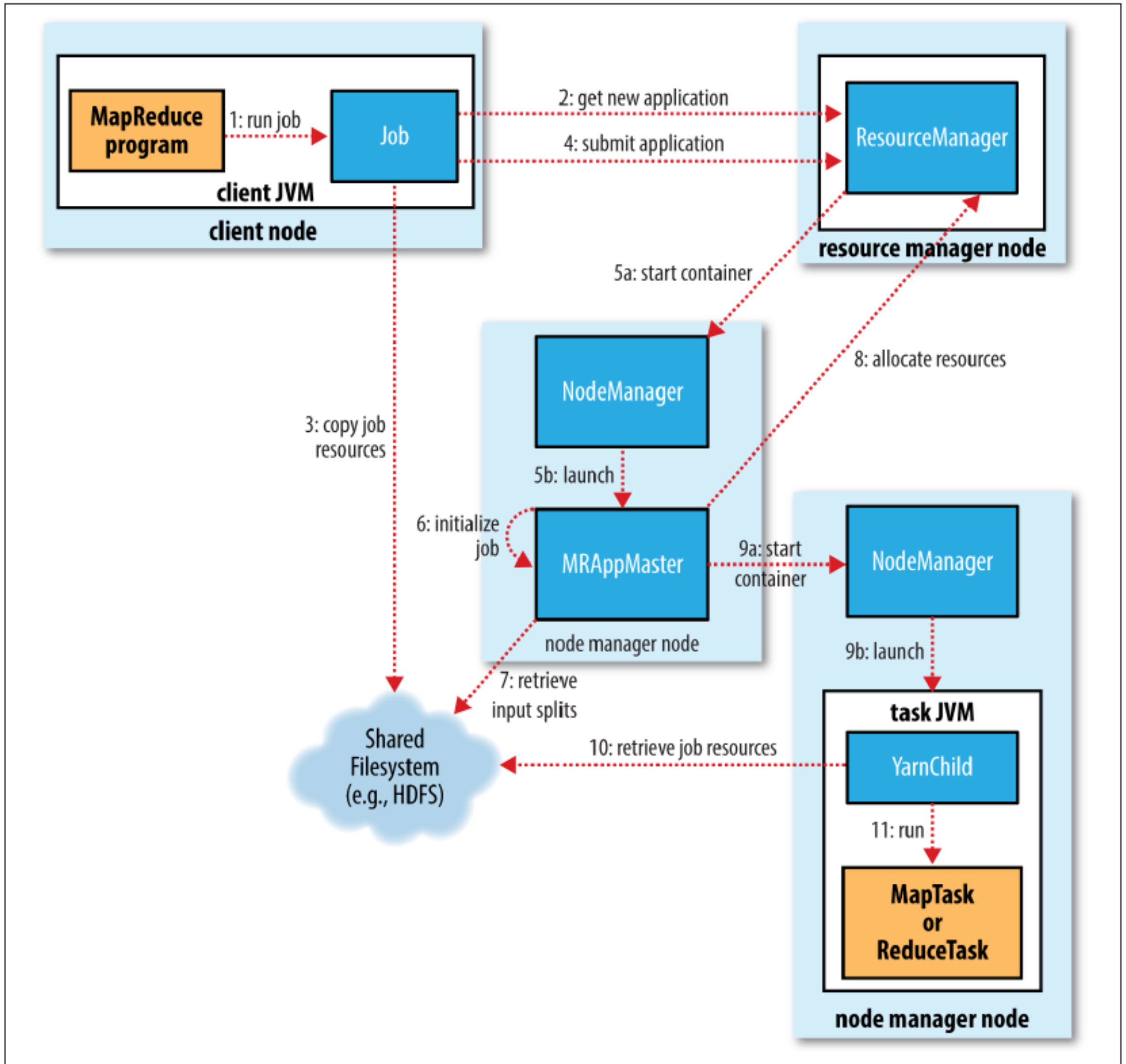
```
public static void main(String[] args) throws Exception {
   // Run on a local node
   Configuration conf = new Configuration();
   conf.set("fs.defaultFS", "file:///");
   conf.set("mapreduce.framework.name", "local");

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

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

Anatomy of a MapReduce Job Run

- Recall that five entities are involved in a MapReduce job run:
  - Client, which submits the MapReduce job
  - YARN resource manager, which coordinates allocation of cluster resources
  - YARN node managers, which launch and monitor compute containers
  - MapReduce application master, which coordinates the tasks of the MapReduce job
  - Distributed filesystem, which stores and shares job files between the other entities

Anatomy of a MapReduce Job Run



- How Hadoop runs a MapReduce job?

Job Submission

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

Job Initialization

- The resource manager hands off requests to YARN scheduler, which allocates a container to run the application (managed by Node Manager) (steps 5)
- The application master keeps track of job progress (step 6) and retrieves input splits from the file system (step 7)
  - It creates one map task for each split.
  - Number of reduce tasks depends on the configurations
  - Each task is assigned an ID
- The application master may run tasks of a "small" job within the same JVM as itself, without asking for new containers
  - Save overhead
  - Such a job runs as a uber task
  - A job is small if it only has few tasks (e.g., < 10)

Task Assignment

- For large jobs, the application master requests containers for all map and reduce tasks in the job from the resource manager (step 8)
  - Requests for map tasks are made first and with a higher priority than those for reduce tasks, since all the map tasks must complete before the sort phase of the reduce can start
  - Requests for reduce tasks are not made until 5% of map tasks have completed
  - Requests for map tasks have data locality constraints that the scheduler tries to honor
  - Requests also specify memory requirements and CPUs for tasks

Task Execution

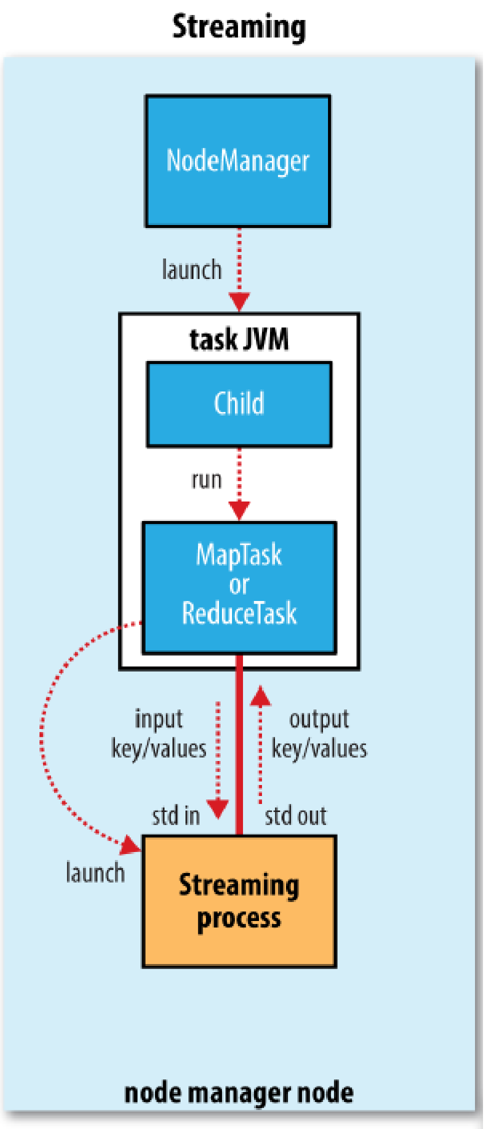
- The application master starts the container by contacting the node manager, under a YarnChild class (step 9).
- The YarnChild class localizes resources (e.g., job configuration and JAR file) (step 10), and runs the map/reduce task (step 11)
  - YarnChild runs in a dedicated JVM

Streaming

- Streaming runs special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it via standard I/O streams

```
hadoop jar hadoop-streaming-2.10.2.jar \
   -input myInputDirs -output myOutputDir \
   -mapper /bin/cat -reducer /usr/bin/wc
```

- https://hadoop.apache.org/docs/r2.10.2/hadoop-streaming/HadoopStreaming.html



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 the application master receives a notification that the last task for a job is complete
- The application master will:
  - Send an HTTP job notification to client
  - Clean up its working state
  - Notify task containers to clean up the states

Handling Failures

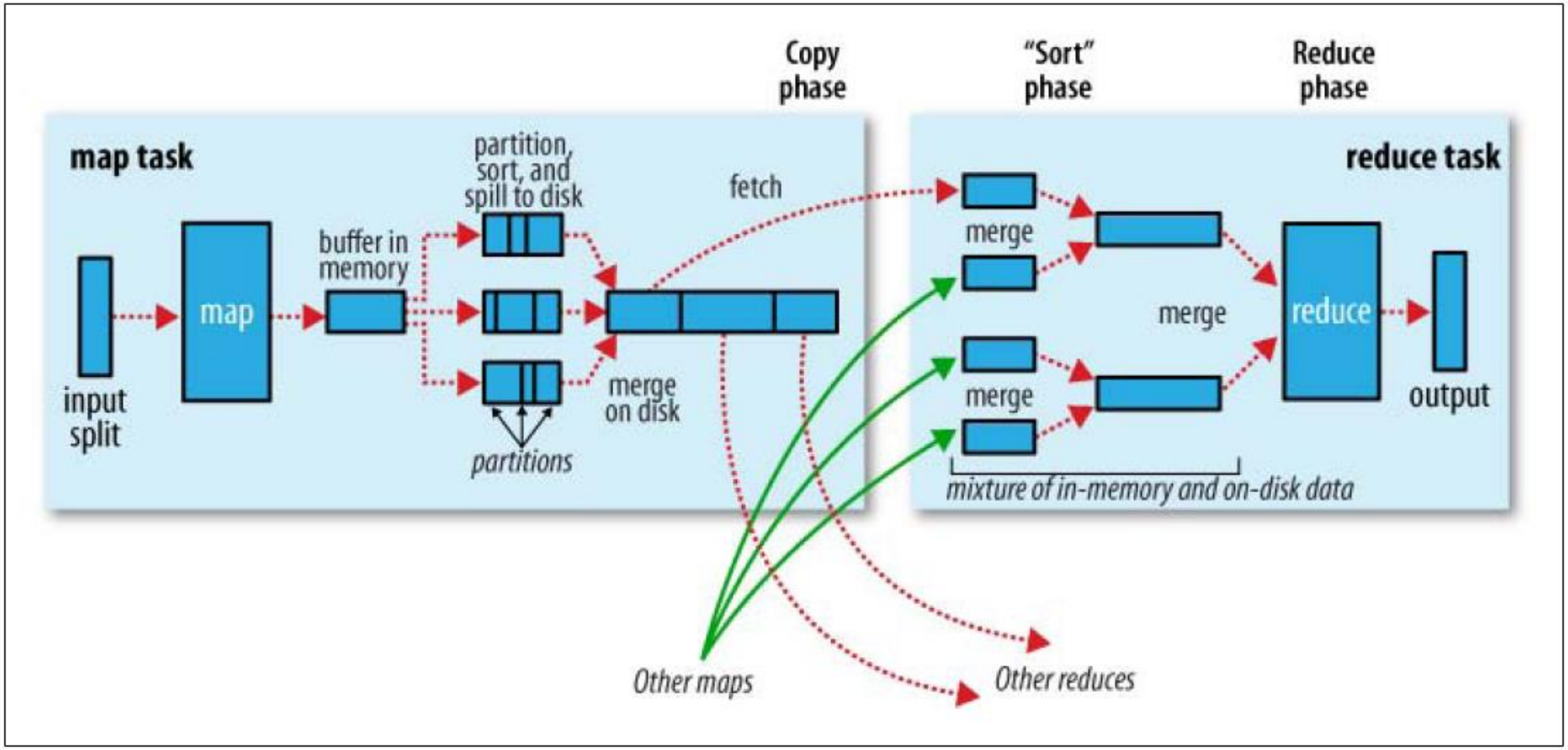
- Task failure
  - If a task is crashed or hanging, it fails
  - The application master reschedules the task on a different node manager. If any task fails 4 times (default), whole job fails
- Application master failure
  - The resource manager detects the failure and starts a new master instance in a new container, and tries to recover the state of tasks using job history
- Node manager failure
  - The resource manager recovers task/application master failures as above, and removes the failed node manager
- Resource manager failure
  - Single point of failure
  - Need active and standby resource managers

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 in rounds and maintains sort ordering
    - E.g., if there were 50 map outputs and the merge factor was 10 (`mapreduce.task.io.sort.factor`), there are five rounds, each merging 10 files
    - Note that the actual merge operation is more subtle than that
- 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("mapreduce.task.io.sort.mb", 200);

Job job = Job.getInstance(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

- 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

```
Map-Reduce Framework
                Map input records=2
                Map output records=8
                Map output bytes=82
                Map output materialized bytes=85
                Input split bytes=208
                Combine input records=8
                Combine output records=6
                Reduce input groups=5
                Reduce shuffle bytes=85
                Reduce input records=6
...

 WordCountWithCounter$WordCount
                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?