Introduction

- MapReduce was proposed by Google in a research paper:
  - Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004
- MapReduce implementations such as Hadoop differ in details, but the main principles are the same as described in that paper.
- Note: We will work with Hadoop version 1 and the org.apache.hadoop.mapreduce API, but…
  - Many existing programs might be written using the older API org.apache.hadoop.mapred.
  - Hadoop version 2 is already available but it is still easier to get information for version 1.
  - If you understand version 1, you should have no trouble migrating to version 2 at some point.

1. Overview

- MapReduce is both a programming model and an associated implementation for processing large data sets.
- The programmer essentially only specifies two (sequential) functions: Map and Reduce.
- At runtime, program execution is automatically parallelized on a Warehouse-Scale Computer (i.e., large cluster of commodity PCs).
  - MapReduce could be implemented on other architectures, but Google proposed it for clusters. Many design decisions discussed in the Google paper and implemented in Hadoop were influenced by the goal of running on a Warehouse-Scale Computer. MapReduce implementations for parallel machines or even GPU computing might make different design decisions to find the right balance between performance and fault tolerance for those architectures.
- As we will see soon, MapReduce provides a clever abstraction that works well for many real-world problems. It lets the programmer focus on the algorithm itself, while the runtime system takes care of messy details such as:
  - Partitioning of input data and delivering data chunks to the different worker machines.
  - Creating and scheduling the various tasks for execution on many machines.
  - Handling machine failures and slow responses.
  - Managing inter-machine communication to get intermediate results from data producers to data consumers.

2. MapReduce Programming Model

- A MapReduce program processes a large input set of key-value pairs and produces a set of key-value pairs as output. (Notice that the definitions below are more general than those in the Google paper. These general definitions correspond to Hadoop's functionality.)
- Map: (k1, v1) $\rightarrow$ list (k2, v2)
  - Map takes an input record, consisting of a key of type k1 and a value of type v1, and outputs a set of intermediate key-value pairs, each of type k2 and v2, respectively. These types can be simple standard types such as integer or string, or complex user-defined objects. A Map call may return zero, one, or more output records for a given input record.
  - By design, each Map call is independent of each other Map call. Hence Map can only perform *per-record computations*. In particular, the Map function cannot combine information across different input records. However, as we will see later, some combining across Map calls can happen at the level of a Map task.
- The MapReduce library automatically groups all intermediate pairs with same key together and passes them to the workers executing Reduce.
- Reduce: (k2, list (v2)) $\rightarrow$ list (k3, v3)
  - For each intermediate key, Reduce combines information across records that share this same key. The intermediate values are supplied via an iterator, which can read from a file. This way MapReduce can handle large data where list(v2) does not fit in memory.
- **Terminology**: There are three different types of keys: Map input keys k1, intermediate keys k2 (occurring in Map output and hence also Reduce input), and Reduce output keys k3. For MapReduce program design, usually only the intermediate keys matter, because they determine which records are grouped together in the same Reduce call. Following common practice, we will often simply say "key" instead of "intermediate key". Context should make this clear.

3. MapReduce Program Execution

- We discuss MapReduce program execution using the Word Count example. Recall that for a given document collection, we want to determine how many times each word occurs in the entire collection.
- The main insight for the parallel version of Word Count is that we can count word occurrences in different documents independently in parallel, then aggregate these partial counts for each word.
- Based on the properties of Map and Reduce, it is clear that Map cannot aggregate across all documents in the collection. Hence the final aggregation has to happen in Reduce.
- When designing a MapReduce program, it is always helpful to ask ourselves "what kind of data has to be processed together to get the desired result?" For Word Count, we need to count *per word*. In particular, for a given word, all its partial counts need to be processed together to get the correct total. This suggests using the word itself as the intermediate key (k2). A partial count from a subset of the document collection would be the corresponding intermediate value (v2).
- Parallel counting in the Map phase can be implemented by having Map output (w, 1) for each occurrence of word w in the document. Intuitively, this record states that word w was found one more time.

3.1 MapReduce Pseudocode for Word Count

```
map( offset B, line L )
  // B is the byte offset of the line of text in the input file.
  // L is a single line at that offset in the document.
  // Notice that the document collection could be read in
  // different ways, e.g., document-wise. Then each Map
  // call would receive an entire document as the input.
  for each word w in L do
    emit( w, 1 )
```

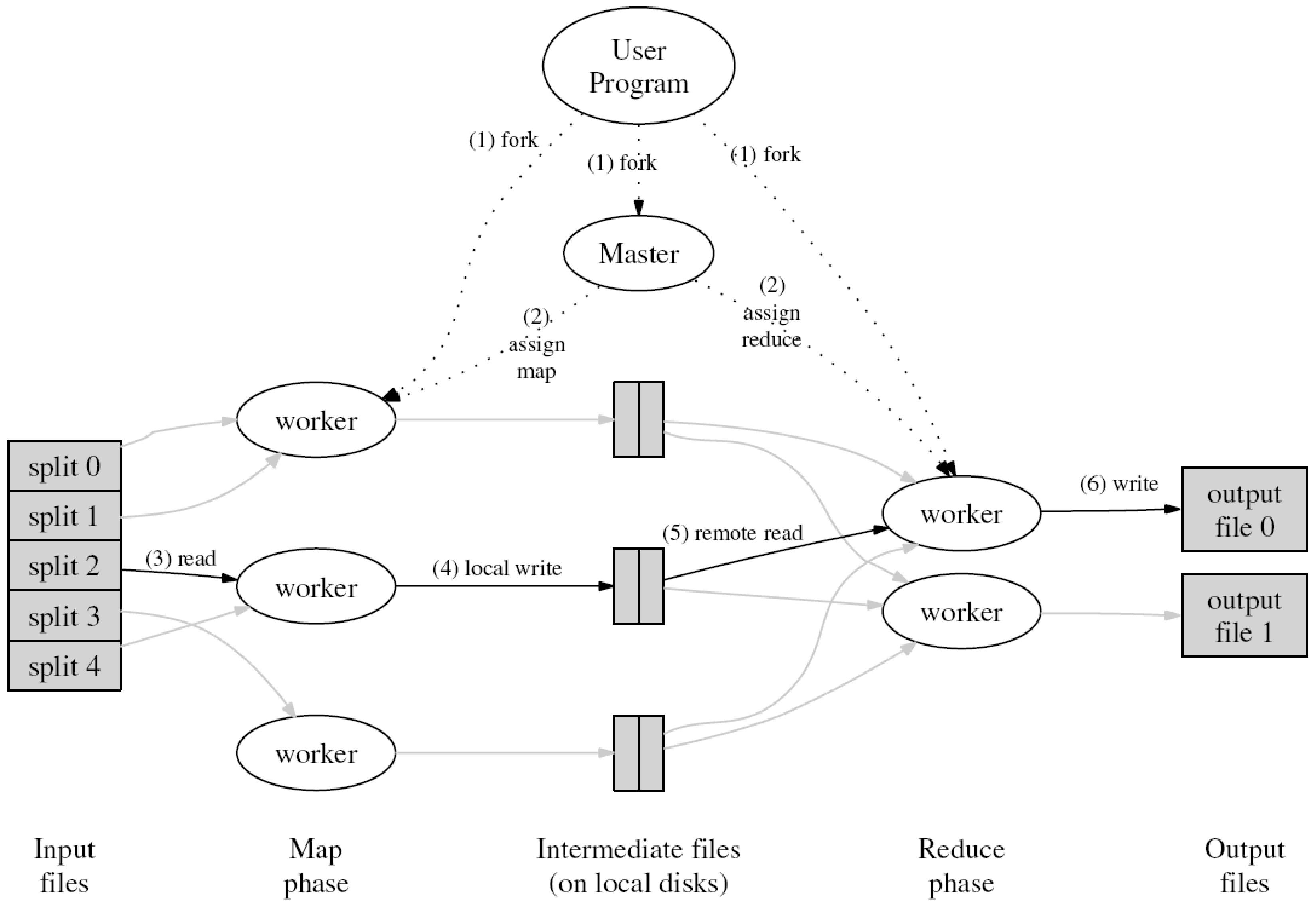
```
reduce( word w, [c1, c2,…] )
  // Each c is a partial count for word w.
  // Without any combining in the Map task, each c is equal to 1.
  total = 0
  for each c in input list
    total += c
  emit( w, total )
```

- The pseudocode is amazingly simple and concise. And it defines a parallel implementation that can employ hundreds or thousands of machines for huge document collections.
- MapReduce does all the heavy lifting of grouping the intermediate results (i.e., those emitted by Map) by their key and delivering them to the right Reduce calls. Hence this program is even simpler than the sequential program that used the HashMap data structure. Notice how MapReduce's grouping by key eliminated the need for the HashMap!
- LINK: How is this program executed on a cluster of machines?

How is this Program Executed?

- The input data, i.e., all documents, are stored in GFS files. (Recall that GFS files are partitioned into chunks, also referred to as *splits*, and stored on different machines.
- There is a MapReduce master process that controls program execution and resource allocation. (Notice that in Hadoop version 2, the role of the master changed a bit.) Similar to the distributed file system, having a single master simplifies system design. To avoid the master becoming a bottleneck, it does not participate in the data processing operations and it usually runs on a dedicated machine, the cluster's *head node*.
- All other machines in the cluster are *worker nodes* (also called slave nodes). A Mapper is a worker process that executes a Map task. A Reducer is a worker process that executes a Reduce task.
- At runtime, the master assigns Map and Reduce tasks to worker machines, taking data location into account. In particular, it prefers to assign a Map task to a machine that already stores the corresponding input data split. This reduces the amount of data to be sent across the network—an important optimization when dealing with big data. Since GFS files are chunked up and distributed over many machines, this optimization is often very effective. Similarly, if data cannot be processed locally, then the master would try to assign the task to a machine in the same rack as the data chunk.
- By default the master will create one Map task per input file split. A Mapper reads the assigned file split from the distributed file system and writes all intermediate key-value pairs it emits to the local file system.
- The Mapper then informs the master about the result file location, who in turn informs the Reducers.
- Each Reducer pulls the corresponding data directly from the appropriate Mapper disk location.
- As data is transferred from Mappers to Reducers, it is partitioned by Reduce task and sorted by key within each of these partitions.
- The Reducer works with this sorted input by executing the Reduce function for each key and associated list of values. The output emitted by Reduce is written to a local file. As soon as the Reduce task completed its work, this local output file is atomically renamed (and moved) to a file in the distributed file system.

Execution Overview



Program Execution Animation

- Insert an animation here that shows how the document example from Module 1 is processed by three machines using MapReduce.

3.2 Real Hadoop Code for Word Count

- The Hadoop Java code is almost as simple as the pseudocode. Most of it consists of boilerplate code, while the actual algorithm is expressed in a few lines.

```
package org.apache.hadoop.examples;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
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.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
```

These are typical imports for a Hadoop program.

dd a note somewhere that this code was obtained from the Hadoop 1.2.1 distribution.

```
public class WordCount {

 public static class TokenizerMapper  extends Mapper<Object, Text, Text, IntWritable>{
  // Definition of Mapper
 }

 public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
  // Definition of Reducer
 }

 public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
   System.err.println("Usage: wordcount <in> <out>");
   System.exit(2);
  }

  Job job = new Job(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

Driver program: The driver puts everything together, e.g., specifies the Mapper and Reducer classes, and controls the execution of the program.

```
public static class TokenizerMapper extends Mapper<LINK1: Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

    StringTokenizer itr = new StringTokenizer(value.toString());

    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

Mapper class with Map function.

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {

  private IntWritable result = new IntWritable();

  public void reduce(Text key, LINK2: Iterable<IntWritable> values, Context context) throws IOException, InterruptedExceptio

    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

Reducer class with Reduce function.

4. Features Beyond Map and Reduce

- Even though the Map and Reduce functions usually represent the main functionality of a MapReduce program, there are several additional features that are crucial for achieving high performance and good scalability.

4.1 Combiner

- Consider a Map task (i.e., a Mapper) that processes two different lines L1 and L2 of a document. Assume L1 contains 3 occurrences of the word “Northeastern”, while L2 contains another 5 occurrences. Instead of sending 8 times (Northeastern, 1), it would be more efficient to send (Northeastern, 8) once.
- We could define a HashMap data structure in the Map function to keep track of the number of occurrences of each string in a line. (LINK1) Unfortunately this approach cannot aggregate across different Map calls, i.e., in our example the Mapper would return (Northeastern, 3) and (Northeastern, 5) separately for line L1 and L2, respectively.
- To generate (Northeastern, 8), we can use a Combiner. A Combiner is similar to a Reducer in that it works on Map output by processing values that share the same key. In fact, often the code for a Combiner will be the same as or similar to the Reducer. (LINK3: Examples for cases where it is not the same.) In contrast to the Reducer, the Combiner is executed in the Map phase!
  - In Hadoop a Combiner is defined as a Reducer class. This class is set as the Combiner in the driver program by using job.setCombinerClass().
  - Note that the Combiner’s input records are of type (k2, v2), i.e., Map’s output type. Since the Combiner’s output has to be processed by a Reducer, and the Reducer input type is (k2, v2) as well, it follows that a Combiner’s input and output types have to be identical.
- Since the Combiner sees only a subset of the values for a key, the Reducers still have work to do to combine the partial results produced by the Combiners. This kind of multi-step combining does not work for all computation tasks. It does work for LINK2:algebraic and distributive aggregate functions, but it cannot be applied to LINK2:holistic aggregates.
- A Combiner represents an optimization opportunity. It can reduce data transfer cost from Mappers to Reducers (and the corresponding Reducer-side costs), but tends to increase Mapper-side costs for the additional processing. Due to this tradeoff, Hadoop does not guarantee if and when a Combiner will be executed. The MapReduce programmer needs to ensure program correctness, no matter how often and on what subset of the Map output records a Combiner might be executed. This includes ensuring correct behavior for cases where a Combiner execution works with both “fresh” Map output and previously combined records.

Code for LINK 1

```
// Map function performing aggregation of counts per line of text
map( offset B, line L )
  HashMap H           // Maps from strings to counts
  for each word w in L do
    H[w]++ // Creates a new entry on first encounter of a new word

  for each word w in H do
    emit( w, H[w] )
```

LINK3: Careful With Combiners

- Consider Word Count, but assume we are only interested in words that occur at least 100 times in the document collection.
  - The Reduce function for this problem computes the total count for a word as discussed before, but only outputs it if it is at least 100.
  - Should the Combiner do the same? No! The Combiner should not filter based on its local count. For instance, there might be 60 occurrences of “NEU” in one Mapper and another 70 in another Mapper. Each is below 100, but the total exceeds the threshold and hence the count for “NEU” should be output.
- Consider computing the average of a set of numbers in Reduce.
  - The Reduce function should output the average.
  - The Combiner has to output (sum, count) pairs as the value to allow correct computation in the Reducer.

4.2 Hadoop Counters

- Sometimes we would like to collect global statistics for a MapReduce job, e.g., how many input records failed to parse during the Map phase or how many objects changed their status in the Reduce phase. It is easy to collect such statistics for a single Map or Reduce call. We will see in a later module that this can be done easily for an entire Map or Reduce *task* as well. However, *different* tasks cannot directly communicate with each other and hence cannot agree on a global counter value.
- To get around this limitation, a global counter feature needs to be supported by the MapReduce system. Hadoop has such a feature, allowing the programmer to define global counter objects. These counters are modified by individual tasks, but all updates are also propagated to the Hadoop master node. The master therefore can maintain the current value of each counter and make it available to a task or the driver program.
- Since global counters are maintained centrally by the master and updates are sent from the different tasks, master and network can become a bottleneck if too many counters have to be managed. Hence it is not recommend to use more than a few hundred such counters.

4.3 Partitioner

- The Partitioner determines which intermediate keys are assigned to which Reduce task. (Recall that each Reduce task executes a Reduce function call for every key assigned to it.)
- The MapReduce default Partitioner relies on a hash function to assign keys essentially at random to Reduce tasks. (LINK: Example) This often results in a reasonably good load distribution. However, there are many cases where clever design of the Partitioner can significantly improve performance or enable some desired functionality. We will see many examples in future modules.
- In Hadoop we can create a custom Partitioner by implementing our own getPartition() method for the Partitioner class in org.apache.hadoop.mapreduce.

5. Hadoop Specifics

- We will discuss Hadoop-specific terminology and details.

5.1 Job vs. Task vs. Function Calls

- In the beginning you might find it challenging to distinguish between a MapReduce job, a task, and the Map and Reduce functions. Understanding these concepts is essential for understanding MapReduce. We illustrate them using the Word Count example.
- Job: The MapReduce job is the entire program, consisting of a Map and a Reduce phase. E.g., the MapReduce Word Count program submitted with a link to an input directory is a job.
- Map task: A Map task is assigned by the master to a worker. By default the master creates one Map task for each input split. For each data record in the input split, the Map task will execute the Map function. In the Word Count example, consider a Map task working on input split S. For each line of text in S, the task would execute a Map call. Hence for a single Map task there might be hundreds or more Map function calls.
- Reduce task: A Reduce task is assigned by the master to a worker. The number of Reduce tasks can be set by the programmer. The Partitioner, more precisely its getPartition function, assigns intermediate keys to Reduce tasks. For each key and its associated list of values, the Reduce function is executed. Since the input for each Reduce task is sorted by key, the Reduce calls happen in key order. For the Word Count example, assume the programmer decided to create two Reduce tasks: one for all words starting with letters a to m, the other for all words starting with letters n to z. Reduce task 0 would execute as many Reduce function calls as there are words in the document collection starting with letters a to m, similarly Reduce task 1 for the other words.

Job vs. Task vs. Function Calls (cont.)

- The amount of work performed for each task depends on the input size and distribution. Map tasks tend to perform a similar amount of work as each gets about the same amount of data. Notice that if a record crosses the file split boundary, then the additional bytes until the end of the record in the next split also have to be processed by the Map task. (The MapReduce environment does this automatically whenever it detects that the end of a file split does not coincide with the end of the input record.) This implies that in practice not every Map task is responsible for exactly 64 megabytes (or whatever the distributed file system's chunk size is). However, even with similar input size, execution cost can vary depending on the cost of a single Map function call. For example. there might be input records that require more elaborate analysis or processing.
- For Reduce tasks, the choice of intermediate keys, number of Reduce tasks, and Partitioner—all controlled by the programmer—determine how well Reduce work is distributed. We will discuss this in more detail in future modules.

Selecting the Number of Map and Reduce Tasks

- Let M denote the number of Map tasks and R denote the number of reduce tasks. Larger M and R result in smaller tasks, enabling easier load balancing and faster recovery. On the other hand, cost for the master increases as it has to make O(M+R) scheduling decisions and needs O(M·R) memory to keep track of the state of the MapReduce job. Furthermore, very small tasks do not use resources efficiently because they perform too little work compared to the cost of setting up and managing the computation. Hence in general we would like to set M and R to create tasks that perform a sufficient amount of computation (say in the order of a minute at least), but not be too large (e.g., tens of hours). In practice these numbers depend on the problem at hand and might require trial-and-error by the programmer. Common default choices:
- By default, MapReduce sets M equal to the number of splits of the input file(s). This tends to work well for most cases, except if the input file is small and per-record computation cost is high. For such cases Hadoop offers a special input reader that can create more fine-grained tasks and will be discussed in a future module.
- The choice of R is more challenging, in particular since it is often difficult to estimate size and distribution of the Map output. To finish the Reduce phase in one “wave”, one can set R to be equal to the number of workers machines available (or slightly lower to provision against possible failures). If this results in overly large tasks, one can set R to be a small multiple of the number of worker machines to finish in the corresponding number of waves.

5.2 Hadoop Processes

- The NameNode daemon is the master of the Hadoop Distributed File System (HDFS). It directs the DataNodes to perform their low-level I/O tasks. The DataNode daemon corresponds to a chunkserver in the Google File System. There is also one Secondary NameNode daemon per cluster to monitor the status of HDFS and take snapshots of HDFS metadata to facilitate recovery from NameNode failure.
- The JobTracker daemon takes on the role of the MapReduce master. It communicates with the client application and controls MapReduce execution in the TaskTrackers. To do so, it keeps track of the status of each Map and Reduce task (idle, in-progress, or completed) and who is working on it. The JobTracker also stores the location and size of the output of each completed Map task, pushing this information incrementally to TaskTrackers with in-progress Reduce tasks.
- There is exactly one TaskTracker daemon per worker machine. It controls the local execution of Map and Reduce tasks, spawning JVMs to do the work.
- In a typical cluster setup, NameNode and JobTracker run on the cluster head node. DataNode and TaskTracker daemons run on all other nodes. The Secondary NameNode usually runs on a dedicated machine that should be different from the one running the NameNode daemon.

5.3 Anatomy of Hadoop Job Run

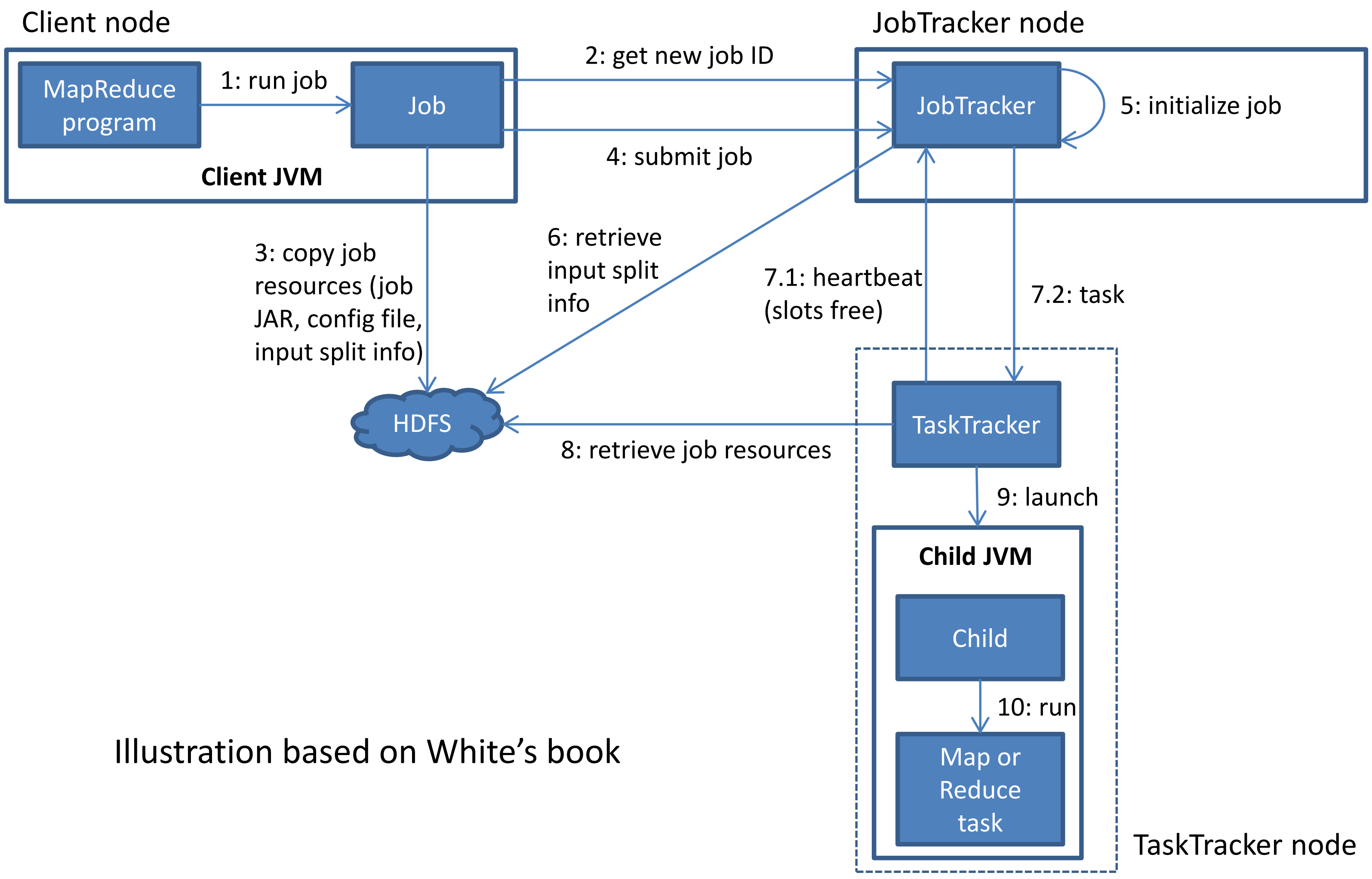


Illustration based on White's book

Job Submission

- The client submits a MapReduce job through a Job.submit() call. The waitForCompletion() function submits the job and polls the JobTracker about its progress every second, outputting to the console if anything changed
- Job submission process steps:
  1. Get a new job ID from the JobTracker.
  2. Determine the input splits for the job.
  3. Copy the job resources—in particular the job JAR file, configuration file, and computed input splits—to HDFS into a directory named after the job ID.
  4. Inform the JobTracker that the job is ready for execution.

Job Initialization

- The JobTracker puts the ready job into an internal queue.
- The Job scheduler picks a job from the queue and initializes it by creating a job object. It also creates a list of tasks:
  - By default there is one Map task for each input split.
  - The number of Reduce tasks is determined by the mapred.reduce.tasks property in Job, which is set by setNumReduceTasks().
- Tasks then need to be assigned to worker nodes.

Task Assignment

- The TaskTrackers send heartbeat messages to the JobTracker. In this message, a TaskTracker states how many slots it has available for Map and Reduce task execution.
  - The total number of slots for task execution depends on the number of cores and memory size. It can be set by the Hadoop administrator.
- If the TaskTracker has slots available, the JobTracker replies with a new task. The standard FIFO scheduler chooses a task from the first job in the queue. (Instead of FIFO, other scheduling policies could be used, e.g., "fair" scheduling that tries to share resources evenly between different users.) It will also chose Map tasks before Reduce tasks. (LINK: Why?) For a Map task, it chooses one whose input split location is closest to the machine running the TaskTracker instance. The ideal case is a data-local task, i.e., when the input split is already on the machine that executes the task.
- Toward the end of the Map phase, the JobTracker might create backup Map tasks to deal with machines that take unusually long for the last in-progress tasks ("stragglers"). Recall that the Reducers cannot start their work until all Mappers have finished. Hence even a single slow Map task would delay the start of the entire Reduce phase. Backup tasks, i.e., "copies" of a task that are started while the original instance of the task is still in progress, can prevent this for delays caused by slow machines as long as one of the backup tasks is executed by a fast machine. The same optimization can be applied toward the end of the Reduce phase to avoid delayed job completion due to stragglers.

Task Execution

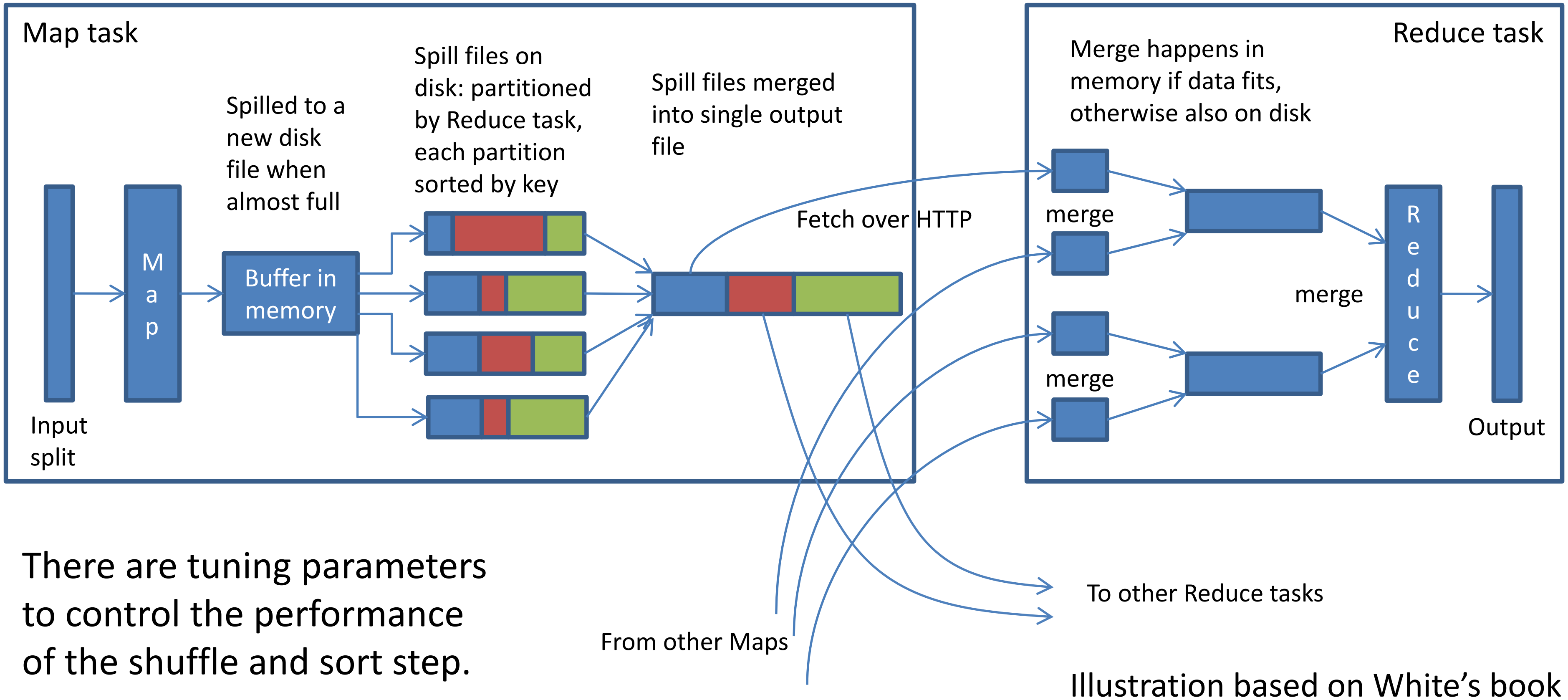
- After receiving a task assignment from the JobTracker, the TaskTracker copies job JAR and other configuration data (e.g., distributed cache) from HDFS to local disk.
- It creates a local working directory and a TaskRunner instance. The TaskRunner launches a new JVM, or reuses one from another task, to execute the JAR.
- While being executed, a task reports its progress to the TaskTracker. This progress information is included in the heartbeat message sent to the JobTracker.
- Based on this information, the JobTracker computes the global status of job progress. The JobClient polls the JobTracker regularly for its status, displaying it on the console and a Web UI.

5.4 Moving Data From Mappers to Reducers

- Data transfer from Mappers to Reducers happens during the shuffle and sort step. It constitutes a synchronization barrier between the Map and Reduce phase in the sense that no Reduce function call is executed before all Map output has been transferred. If a Reduce call were to be executed earlier, it might not have access to the complete list of input values.
  - Note that when executing a MapReduce job, sometimes the job status shows Reduce progress greater than zero, while Map progress is still below 100%. (LINK: How can this happen? Isn't Reduce supposed to wait until all Mappers are done?)
- The shuffle and sort step can be the most expensive part of a MapReduce execution. It starts with Map function calls emitting data to an in-memory buffer. Once the buffer fills up, its content is partitioned by Reduce task (using the Partitioner) and the data for each task is sorted by key (using a key comparator than can be provided by the programmer). The partitioned and sorted buffer content is written to a spill file in the local file system. At some point spill files are merged and the partitions are copied to the local file system of the corresponding Reducers.
- Each Reducer merges the pre-sorted partitions it receives from different Mappers. The sorted file on the Reducer is then made available for its Reduce calls. The iterator for accessing the Reduce function's list of input values simply iterates over this sorted file.
- Notice that it is not necessary for the records assigned to the same partition to be *sorted* by key. It would be sufficient to simply *group* them by key. Grouping is slightly weaker than sorting, hence could be computationally a little less expensive. In practice, however, grouping is often implemented through sorting anyway. Furthermore, the property that Reduce calls in the same task happen in key order can be exploited by MapReduce programs. We will see this in a future module for sorting and for the value-to-key design pattern for secondary sorting.

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.


Map task
Spilled to a new disk file when almost full
Spill files on disk: partitioned by Reduce task, each partition sorted by key
Spill files merged into single output file
Input split
M a p
Buffer in memory
Fetch over HTTP
Reduce task
Merge happens in memory if data fits, otherwise also on disk
merge
merge
merge
R e d u c e
Output
To other Reduce tasks
From other Maps

There are tuning parameters to control the performance of the shuffle and sort step.

Illustration based on White's book

5.5 Summary of Important Hadoop Execution Properties

- Map calls inside a single Map task happen sequentially, consuming one input record at a time.
- Reduce calls inside a single Reduce task happen sequentially in increasing key order.
- There is no ordering guarantee between different Map tasks.
- There is no ordering guarantee between different Reduce tasks.
- Different tasks do not directly share any data with each other.

6. Handling Failures

- Since MapReduce was proposed for Warehouse Scale Computers, it has to deal with failures as a common problem.
- MapReduce therefore was designed to automatically handle system problems, making them transparent to the programmer.
- In particular, the programmer does not need to write code for dealing with slow responses or failures of machines in the cluster.
- When failures happen, typically the only observable effect is a slight delay of the program execution due to re-assignment of tasks from the failed machine.
- In Hadoop, errors produced by a task are reported to the corresponding TaskTracker and logged. Hanging tasks are detected through timeout. The JobTracker will automatically re-schedule failed tasks. It tries up to mapred.map.max.attempts many times (similar for Reduce). The job is aborted when the task failure rate exceeds mapred.max.map.failures.percent (similar for Reduce).

6.1 Handling Worker Failures: Map

- The master pings every worker machine periodically. Workers who do not respond in time are marked as failed.
- A failed worker's in-progress and completed Map tasks are reset to idle state and hence become available for re-assignment to another worker. (LINK: Why do we need to re-execute a completed Map task?)
  - LINK: Mappers write their output only to the local file system. When the machine that executed this Map task fails, the local file system is considered inaccessible and hence the result has to be re-created on another machine.
- Reducers are notified by the master about the Mapper failure, so that they do not attempt to read from the failed machine.

6.2 Handling Worker Failures: Reduce

- A failed worker's in-progress Reduce tasks are reset to idle state and hence become available for re-assignment to another worker machine. There is no need to restart *completed* Reduce tasks. (LINK: Why not?)
  - LINK: Reducers write their output to the distributed file system. Hence even if the machine that produced the data fails, the file is still accessible.

6.3 Handling Master Failure

- Failures of the master are rare, because it is just a single machine. In contrast, the probability of experiencing at least one failure among 100 or 1000 workers is much higher.
- The simplest way of handling a master failure is to abort the MapReduce computation. Users would have to re-submit aborted jobs once the new master process is up. This approach works well as long as the master's mean time to failure is significantly greater than the execution time of most MapReduce jobs.
  - To illustrate this point, assume the master has a mean time to failure of 1 week. Jobs running for a few hours are unlikely to experience a master failure. However, a job running for 2 weeks has a high probability of suffering from a master failure. Abort-and-resubmit would not work well for such jobs, because even the newly re-started instance of the job is likely to experience a master failure as well.
- As an alternative to abort-and-resubmit, the master could write periodic checkpoints of its data structures so that it can be re-started from a checkpointed state. Long-running jobs would pick up from this intermediate state, instead of starting from scratch. The downside of checkpointing is the higher cost during normal operation, compared to abort-and-resubmit.

6.4 Hadoop Specifics: TaskTracker and JobTracker Failure

- TaskTracker failure is detected by the JobTracker from missing heartbeat messages. When this happens, the JobTracker re-schedules all Map tasks and in-progress Reduce tasks from that TaskTracker.
- Hadoop currently does not handle JobTracker failure. However, this could change with a new version of Hadoop. It is conceptually possible to set up a watchdog process that detects JobTracker failure and automatically restarts it on another machine. This approach could rely on ZooKeeper to make sure that there is at most one JobTracker active.

6.5 Program Semantics with Failures

- If the Map, getPartition, and Combine functions are deterministic, then MapReduce guarantees that the output of a computation, no matter how many failures it has to handle, is identical to a non-faulting sequential execution. This property relies on atomic commit of Map and Reduce outputs, which is achieved as follows:
  - An in-progress task writes its output to a private temp file in the local file system.
  - On completion, a Map task sends the name of its temp file to the master. Note that the master ignores this information if the task was already complete. (This could happen if multiple instances of a task were executed, e.g., due to failures.)
  - On completion, a Reduce task *atomically* renames its temp file to a final output file in the distributed file system. (Note that this functionality needs to be supported by the distributed file system.)
- For non-deterministic functions, it is theoretically possible to observe different output because the re-execution of a task could make a different non-deterministic decision. This could happen if the output of both the original and the re-executed instance of the task are used in the computation. For example, consider Map task m and Reduce tasks r1 and r2. Let e(r1) and e(r2) be the execution that committed for r1 and r2, respectively. There can only be one committed execution for each Reduce task. Result e(r1) may have read the output produced by one execution of m, while e(r2) may have read the output produced by a different execution of m. If this can or cannot happen depends on the specific MapReduce implementation.

7. MapReduce in Action

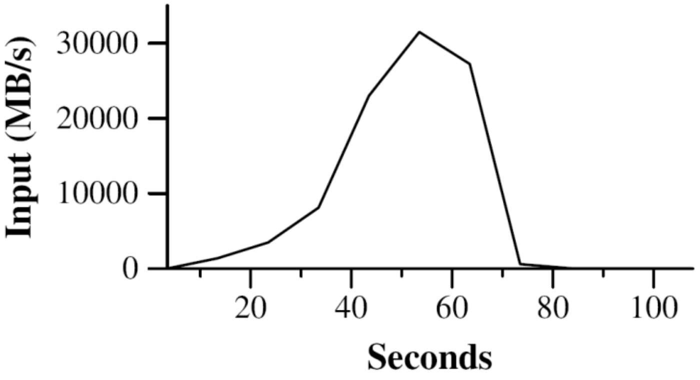
- Let us discuss now how MapReduce performed at Google.
- The results reported here are from Google’s paper, published in 2004. However, the main message is still the same.

7.1 MapReduce Use at Google

- MapReduce has been used at Google for implementing parallel machine learning algorithms, clustering, data extraction for reports of popular queries, extraction of page properties, e.g., geographical location, and graph computations. In recent years high-level programming languages tailored to some of these applications have started to replace “raw” MapReduce. But in the end these languages usually are also compiled into code for MapReduce (or similar systems) in order to be executed on a large cluster of commodity machines.
- Google reported the following about their indexing system for Web search, which required processing more than 20 TB of data:
  - It consists of a sequence of 5 to 10 MapReduce jobs.
  - MapReduce resulted in smaller simpler code, requiring only about 700 LOC (lines of code) for one of the computation phases, compared to 3800 LOC before.
  - MapReduce made it much easier to change the code. It is also easier to operate, because the MapReduce library takes care of failures.
  - By design, MapReduce also made it easy to improve performance by adding more machines.

7.2 Experiments

- The experiments were performed on a cluster with 1800 machines. Each machine was equipped with a 2 GHz Xeon processor, 4 GB of memory, two 160 GB IDE disks. The machines were connected via a gigabit Ethernet link with less than 1 millisecond roundtrip time.



7.2.1 Grep

- The first experiment examines the performance of MapReduce for Grep. Grep scans 10^{10} 100-byte records, searching for a rare 3-character pattern. This pattern occurs in 92,337 records, which are output by the program.
- The number of Map tasks is 15,000 (corresponding to the number of 64 MB splits); and there is only a single Reduce task.
- The graph above shows at which rate the Grep implementation in MapReduce consumes the input file as the computation progresses. Initially the rate increases as more Map tasks are assigned to worker machines. Then it drops as tasks finish, with the entire job being completed after about 80 sec.
- There is an additional startup overhead of about 1 minute beforehand, which is not shown in the graph. During this time the program is propagated to the workers and the distributed file system is accessed for opening input files and getting information for locality optimization.
- Both the startup delay and the way how computation ramps up and then gradually winds down are typical for MapReduce program execution.

7.2.2 Sort

- The next experiment measures the time it takes to sort 10^{10} 100-byte records, i.e., about 1 TB of data. The MapReduce program consists of less than 50 lines of user code. As before, the number of Map tasks is the default of 15,000. However, this time the number of Reduce tasks was set to 4,000 to distribute the nontrivial work performed in the Reduce phase. To achieve a balanced distribution, key distribution information was used for intelligent partitioning. (We will discuss sorting in MapReduce in a future module.)
- Results:
  - The entire computation takes 891 sec.
  - It takes 1283 sec without the backup task optimization, which starts duplicate tasks toward the end of Map and Reduce phase. This significant slowdown by a factor of 1.4 is caused by a few slow machines.
  - Somewhat surprisingly, computation time increases only slightly to 933 sec if 200 out of 1746 worker machines are killed several minutes into the computation. This highlights MapReduce’s ability to gracefully handle even a comparably large number of failures.

8. MapReduce Program Design

- We discuss general principles for approaching a MapReduce programming problem. In future modules we will see a variety of concrete examples.

8.1 Expressiveness of the Programming Model

- The MapReduce programming model might appear very limited, because it relies on only a handful of functions and seems to have a comparably rigid structure for transferring data between worker nodes.
- However, it turns out that MapReduce is as powerful as the “host” language in which the programmer expresses Map and Reduce functions. To see this consider the following construction:
  - Assume we are given a function F written in the host language. Given data set D, it returns F(D). Our goal is to show that we can write a MapReduce program that also returns F(D), no matter what data set D is given.
  - Map function: For input record r from D, emit (X, r).
  - Reduce: Execute F on the list of input values.
- Since each record r is emitted with the same key X, the Reduce call for key X has access to the entire data set D. Hence it can now perform F’s computation locally in the Reducer.
- Even though the above construction shows that every function F from the MapReduce host language can be computed in MapReduce , it would usually not be an *efficient* or *scalable* parallel implementation. Our challenge remains to find the best MapReduce implementation for a given problem.

8.2 Multiple MapReduce Steps

- Often multiple MapReduce jobs are needed to solve a more complex problem. We can model such a MapReduce workflow as a directed acyclic graph. Its nodes are MapReduce jobs and an edge (J1, J2) indicates that job J2 is processing the output of job J1. A job can have multiple incoming and outgoing edges.
- We can execute such a workflow as a linear chain of jobs by executing the jobs in topological order of the corresponding graph nodes. To run job J2 after job J1 in Hadoop, we need to create two Job objects and then include the following sequence in the driver program:
  - J1.waitForCompletion(); J2.waitForCompletion();
- Since job J1 might throw an exception, the programmer should include code that checks the return value and reacts to exceptions, e.g., by re-starting failed jobs in the pipeline.
- For more complex workflows the linear chaining approach can become tedious. In those cases it is advisable to use frameworks with better workflow support, e.g., JobControl from org.apache.hadoop.mapreduce.lib.jobcontrol or Apache Oozie.

8.3 MapReduce Coding Summary

1. Decompose a given problem into the appropriate workflow of MapReduce jobs.
2. For each job, implement the following:
    1. Driver
    2. Mapper class with Map function
    3. Reducer class with Reduce function
    4. Combiner (optional)
    5. Partitioner (optional)
3. We might have to create custom data types as well, implementing the appropriate interfaces:
    1. WritableComparable for keys
    2. Writable for values

8.4 MapReduce Program Design Principles

- For tasks that can be performed independently on a data object, use Map.
- For tasks that require combining of multiple data objects, use Reduce.
- Sometimes it is easier to start program design with Map, sometimes with Reduce. A good rule of thumb is to select intermediate keys and values such that the right objects end up together in the same Reduce function call.

8.5 High-Level MapReduce Development Steps

- Write Map and Reduce (and maybe other) functions.
  - Create unit tests.
- Write a driver program and run the job.
  - Use a small data subset for testing and debugging on your development machine. This local testing can be performed using Hadoop's (LINK1: local) or (LINK2: pseudo-distributed) mode.
  - Update unit tests and program if necessary.
- Once the local tests succeeded, run the program on AWS using small data and few machines.
- Once the program works on the small test set on AWS, run it on the full data set.
  - If there are problems, update tests and code accordingly.
- Profile the code and fine-tune the program.
  - Analyze and think: where is the bottleneck and how do I address it?
  - Choose the appropriate number of Mappers and Reducers.
  - Define combiners whenever possible. (We will also discuss an alternative design pattern called in-Mapper combining in a future module.)
  - Consider Map output compression
  - Explore Hadoop tuning parameters to optimize the expensive shuffle and sort phase (between Mappers and Reducers).

LINK text

- LINK1: Hadoop’s local (also called standalone) mode runs the same MapReduce user program as the distributed cluster version, but does it sequentially. The local mode also does not use any of the Hadoop daemons. It works directly with the local file system, hence also does not require HDFS. This mode is perfect for development, testing, and initial debugging. Since the program is executed sequentially, one can step through its execution using a debugger, like for any sequential program.
- LINK2: Hadoop’s pseudo-distributed mode also runs on a single machine, but simulates a real Hadoop cluster. In particular, it simulates multiple nodes, runs all Hadoop daemons, and uses HDFS. Its main purpose is to enable more advanced testing and debugging.

8.6 Hadoop and Other Programming Languages

- The Hadoop Streaming API allows programmers to write Map and Reduce functions in languages other than Java. It supports any language that can read from standard input and write to standard output.
- The Hadoop Pipes API for using C++ uses sockets to communicate with Hadoop’s task trackers.

9. Summary

- The MapReduce programming model hides details of parallelization, fault tolerance, locality optimization, and load balancing.
- The model is comparably simple, but it fits many common problems. The programmer provides a Map and a Reduce function, if needed also a Combiner and a Partitioner.
- The MapReduce implementation on a cluster scales to 1000s of machines.

10. Appendix: NCDC Weather Data Example

- This example is from the textbook by White. The input file has lines like these, showing year and temperature in **bold**:
  - 0067011990999991**950**051507004+68750+023550FM-12+038299999V0203301N00671220001CN9999999N9**+0000**1+99999999999
  - 0043011990999991**950**051512004+68750+023550FM-12+038299999V0203201N00671220001CN9999999N9**+0022**1+99999999999
- Our goal is to find the max temperature for each year. This can be done using a simple MapReduce job:
  - Map emits (year, temp) for each input record.
  - Reduce finds the highest temperature by scanning through the values list in it input (year, (temp1, temp2,…)).

10.1 Map Details

- The Map function is defined in Hadoop’s Mapper class (org.apache.hadoop.mapreduce.Mapper).
- Its type parameters specify input key type, input value type, output key type, and output value type. In the example, we have the following types:
  - Input key: an input line’s offset in the input file (irrelevant)
  - Input value: a line from the NCDC file
  - Output key: year
  - Output value: temperature
- Hadoop data types are optimized for network serialization. Common types can be found in org.apache.hadoop.io.
- The mapping work is done by the map() method:
  - It takes as input a key object, a value object, and a Context. The value (a line of text from the NCDC file) is converted to Java String type, then parsed to extract year and temperature.
  - Map output is emitted using Context.
    - Map emits a (year, temp) pair only if the temperature is present and the quality indicator reading is OK.

```
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));
    }
  }
}
```

10.2 Reduce Details

- The Reduce function is defined in Hadoop’s Reducer class (org.apache.hadoop.mapreduce.Reducer).
- The Reducer class’ input key and value types must match the Mapper class’ output key and value types.
- The Reducing work is done by the reduce() method. The Reduce input values are passed as an Iterable list. In the example, the Reduce function goes over all temperatures to find the highest one. The result pair is written by using the Context, which writes it 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));
  }
}
```

10.3 Job Configuration

- The Job object forms the job specification and gives control for running the job.
- We specify the data input path using addInputPath(). It can be a single file, a directory (to use all files there), or a file pattern. The function can be called multiple times to add multiple paths.
- We specify the output path using setOutputPath(). It has to be a single output path, which is a non-existent directory for all output files.
- We specify the Mapper and Reducer class to be used for the job.
- We set output key and value classes for Mappers and Reducers. For Reducers, this is done using setOutputKeyClass() and setOutputValueClass(). For Mappers (omitted if it is the same as for the Reducer), we use setMapOutputKeyClass() and setMapOutputValueClass().
  - We can set the input types similarly. The default is TextInputFormat.
- Method waitForCompletion() submits the 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);
  }
}
```

```
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.