1 Mandatory Readings

Mandatory Readings have already been extensively covered by David Roschewitz!

2 Lecture

Data is only useful if we can query it in parallel.

Objective: Optimization by Bringing the Query to the Data

MapReduce: Map - Shuffle - Reduce

Hadoop MapReduce: Several TBs of data with 1000s of nodes.

Input and output formats:

- tabular: RDMS, HBase
- files: Text (1Line, NLine), KeyValue (specified separator for text), SequenceFile (Hadoop binary format, Stores generic key-values with key, value and the length of those variables)

Combine\(^1\): Often, the combine function is identical to the reduce function. Several requirements.

\(^1\)90% case
• Key/Value types must be identical for reduce input and output.

• Reduce function must be Commutative

• Reduce function must be Associative

Hadoop MapReduce Supported frameworks: Java (Mapper, Reducer, Job), Streaming
Possible storage layers: Local filesystem, HDFS, S3, Azure Blob Storage.

Master-Slave architecture:
Task: Map or Reduce. Lots of Map, few Reduce. Usually co-locate Task with the associated Block to avoid network communicating (not always possible).
Split: 1 Split for each Map. Split - Logical Level (MapReduce), need to have the record entirely even if splitted between two blocks - vs Block - Physical Level (HDFS), can cut a record in half.

Shuffling phase: Each mapper sorts its output key-value pairs. Key - value pairs are spilled to disk if necessary. Reducer gets its key value pairs

Figure 1: Java API I/O Formats
over HTTP (all the values associated to the keys for which it is responsible)

MapReduce issues:

- Tight coupling: Namenode & JobTracker, DataNode & TaskTracker
- Scalability: Only one JobTracker

3 Other materials

3.1 Clickers

- Typical MapReduce jobs onto a large 1000-node cluster usually take few hours.
- Both MapReduce 2 and Spark are based on YARN
- Hadoop was created by Doug Cutting. He names it after his son’s elephant toy
- The highest Hadoop version, released last August, is 3.1.1.

3.2 Exercise

Understanding the difference between DataNode and RegionServe

DataNodes are processes that handle HDFS blocks on a physical node and from which blocks are server to clients. RegionServers are processes to serve
HBase regions. Since HBase stores data in HDFS, it is usually a good practice to co-locate DataNodes and RegionServers.

Similarly, NameNodes is responsible for orchestrating access to HDFS blocks, HMaster for coordinating the region servers and for admin functions (creating, deleting, updating tables). These can run on the same physical node or not.

Understand the MapReduce Java API

- Since the output of mapper and reducer must be serializable, they have to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework. Hadoop provides classes implementing these interfaces for string, boolean, integers, long and short values.

- Context allows the Mapper/Reducer to interact with the rest of the Hadoop system. It includes configuration data for the job and the methods for mapper and reducer to emit output (write()). Applications can also use the Context to set parameters to mappers and reducers running on different nodes.

- By default, the number of map tasks for a given job is driven by the number of input splits. The parameter mapred.map.tasks can be used to set a suggested value. For reduce tasks, 1 is the default number. This might be a bad choice in terms of performance, but on the other hand the result will be on a single file, without further merging necessary.

- Generally, one reducer might become a bottleneck. In this case, the bottleneck is more evident if no combiner is used. However, using multiple reducers, multiple output files will be created and an additional merging step is necessary to get the same result. According to the official documentation, the right number of reduces is 0.95 or 1.75 multiplied by [no. of nodes] * [no. of maximum containers per node]. With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing. The scaling factors above are slightly less than whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.
• For small inputs, local computation is faster

• Using a combiner, MapReduce gets faster than a single workstation somewhere between 1GB and 10GB of data. This of course, really depends on the task and the configuration of the machines

• The shuffling phase adds a lot of overhead computation

• A combiner drastically improves the performance for this job where the data to be shuffled can be reduced a lot

• - for each input pair, the mapper can emit zero, one, or several key/value pairs

• The JobTracker is responsible for scheduling mappers and reducers and make sure all nodes are correctly running.

• mapper input is not sorted.

• MapReduce splits might not correspond to HDFS block.

• One single Reducer is applied to all values associated with the same key.

• Multiple Reducers can be assigned pairs with the same value

• - Reducer’s input and output pairs might have different types
3.3 Moodle

- With a large proportion of repeated keys and values in the data. Combiners: Decrease memory requirements on Reducers and Decrease the overall communication volume. They do not affect the memory size on Mappers or the Mapper computation time.

- Load balancing in MapReduce requires prior knowledge about the distribution of keys. It is not automatically performed by the Mappers or solved with high probability by hashing the keys inside the Mappers or solved by using Combiners.

- Using MapReduce is always inefficient when there are terabytes of data, 1,000 Reducers and 10 possible keys. It can be efficient when there are terabytes of data, 1,000 Reducers and 10 possible keys or 1,000 Mappers and 10 possible keys or 1,000 Mappers and 10 possible values

- Modifying the behavior of the Mappers is the best way to enforce filtering or sampling on the data to be processed by MapReduce
MapReduce Mandatory Reading

The MapReduce readings include (too) many pages of both the Hadoop and HBase book, with code examples, diagrams, tables etc. There is no good way to summarize 150 pages of reading, but I’ve tried to condense it as best as possible, biasing my selection towards what I think is examinable. I do apologize for serious omissions.

Hadoop, Chapter 2 – MapReduce

MapReduce is a programming model for data processing striking a balance between simplicity and expressional power. Hadoop can run MapReduce, which is inherently parallel, allowing anyone with enough machines to process large-scale data.

The weather dataset example:

A dataset with all sort of weather and atmospherical data in multiple locations around the globe, collecting data every hour. Finding the maximal temperature per year requires about 42 minutes on a single core. Parallelization is possible, but messy, and limited by the capacity of a single machine.

MapReduce on a high level, is the definition of two functions: map() and reduce(). We would for example map every record of temperature to the key-value pair (year, temperature). We an also filter records with invalid or erroneous data. The reduce function then receives a list of temperatures for every year (2018, [42, 36, 17, ...]), and we need to find the maximum value of that list. The logical data flow is below:

MapReduce can be run by defining 3 Java classes: Mapper, Reducer and the App. Lets explore the snippets below to get a feel for their implementation. First, the Mapper class needs to extend the MapReduce class with 4 formal type parameters: input key, input value, output key and output value.

```java
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

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

Which at the end writes to an instance of Context. The context, as we’ve seen in the exercises as well, can be thought of as the ‘public state’ of MapReduce, where subsequent methods can see the data that was written:

    context.write(new Text(year), new IntWritable(airTemperature));
In this example I’ve omitted the actual mapping code, as it could be heavily dependent on the application. LongWritable, Text and IntWritable are custom MapReduce types. They are imported from org.apache.hadoop.io.

The Reducer is similar, where now the input of the reduce function must match the output of the map function:

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

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

    context.write(key, new IntWritable(maxValue));
    }

And similarly, we write the result to the context:

```java
The final Application can now be defined as follows:

```java
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);
    }
}
```

The Job object allows the user to define how the job is run. A JAR file is compiled, and Hadoop then distributes the JAR across the cluster. Multiple InputPaths can be added, and the OutputPath should not exist at the time of execution. We set the OutputKeyClass and OutputValueClass to match our reduce function, and can manually specify the same for the mapper using setMapOutputKeyClass() and setMapOutputValueClass(), but they default to the same as for the reducer. The ‘true’ in waitForCompletion enables verbose output, and the function returns a Boolean reporting the success (or failure) of the job. Analyzing the verbose output of the job can give insights to the amount of data processes, essential for debugging.
Scaling Out:

Typically, MapReduce jobs will be run on HDFS or another distributed file system. Hadoop then distributes the computation to the machine hosting parts of the data, using YARN. Some more definitions:

- **MapReduce Job**: a job that a client has asked to be performed. Consists of input data, the MapReduce program and configuration information.
- **MapReduce task**: the sub-tasks of the main MapReduce Job. There are map tasks and reduce tasks. Scheduled by YARN, and if it fails, it is automatically rescheduled on a different node.
- **Input Splits**: sometimes just called splits. They are the fixed-size pieces of the input data. A map task is created for every split. A typical split size is that of an HDFS block, 128MB. Can be changed for the entire cluster or specified for each file.
- **Record**: every ‘item’ of data; many records make up a split.
- **Data locality**: Hadoop’s attempt at executing code on the same node where the input data resides in HDFS. Next it will attempt to run on the same rack, as a last resort on an off-rack node.

Map tasks write their output to the local disk, not HDFS, because it is an intermediate output. Replication (in HDFS) is not required, as it is discarded. Failed tasks are re-run automatically anyways. Reduce cannot take advantage of data locality, so it must be transferred over the network and results are written to HDFS, typically. The number of reduce tasks is not governed by the size of the input, but specified separately. For multiple reducers, map tasks partition their output, which can be custom controlled but by default uses a hash function and works very well. The following figure illustrates multiple reducers, and illustrates why map -> reduce is often called shuffle.

![Diagram showing map and reduce tasks with input and output HDFS]

Sometimes, it is possible to not require a reducer, in which case the map tasks write directly to HDFS.

**Combiner Functions**

Combiner functions can be defined in an attempt to reduce the network bandwidth required. It is executed in between the map and reduce functions, and therefore must comply with their inputs.
Hadoop might call the combiner any number (including 0) of times, hence should produce the same output. Let us use the max-temperature example: max() is *commutative* and *associative* meaning it is a suitable combiner function. Mean() on the other hand is not. In implementation, we use the reduce function as the combiner, but does not make the reduce function obsolete. The line is to set the CombinerClass on the Job:

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

To compare with the original benchmark of 42 minutes, we can now run the same task on 10 nodes in 6 minutes, a 7x speedup.

**Streaming**

MapReduce has an API for many languages other than Java, but also implements unix streaming. For this, we can use languages that read standard input and write standard output to write a MapReduce program. For text processing, streaming makes intuitive sense: processing information line by line, and writing output. The map function reads the standard input and writes standard output, as a key-value pair (single-tab delimited line). The reduce function then reads the key-value-lines which the framework guarantees are sorted by key, and writes the result back to standard output. Examples are given in Ruby and Python in the book.

Noteworthy is that the reduce function takes advantage of the ordered nature of the keys. To execute either the Python or the Ruby map and reduce functions, we use the Streaming JAR, and define which scripts should be used. For example (Ruby):

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.*.jar \
  -files ch02-mr-intro/src/main/ruby/max_temperature_map.rb,\ 
  ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \ 
  -input input/ncdc/all \ 
  -mapper ch02-mr-intro/src/main/ruby/max_temperature_map.rb \ 
  -combiner ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \ 
  -reducer ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \
```

-files is used to distribute the script files to the cluster.
Hadoop Chapter 6 – Developing a MapReduce Application

The configuration API:
Properties can be defined as shown below. They have a name, value and description, and can be set as final (cannot be overridden).

```
<property>
  <name>weight</name>
  <value>heavy</value>
  <final>true</final>
  <description>Weight</description>
</property>
```

Resources can then be added to a configuration, where the properties are stored in an XML file (later configurations override earlier ones).

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

We can test certain properties as:

```
assertThat(conf.get("weight"), is("heavy"));
```

Combined properties can be retrieved as:

```
assertThat(conf.get("size-weight"), is("12,heavy"));
```

Furthermore, system properties take priority over predefined properties and system properties are not available using the assert interface.

Setting up the Development Environment
To build MapReduce programs and run them locally, we need to configure all dependencies using the maven Project Object Model (POM), which can be read by many IDEs. We often run Hadoop locally, locally in pseudo-distributed fashion (with YARN on localhost) or on a cluster (hence truly distributed). We often have 3 different configuration files for this. We can then switch to the configuration using -conf <filename> when executing commands. The environment variable can also be changed to alleviate having to include -conf in every command.
GenericOptionsParser, Tool and ToolRunner are 3 classes implemented in Hadoop to help parse or print configuration settings and are often used in conjunction with one another. Code can be seen on page 149 of the book. The table below shows possible options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-D property=value</td>
<td>Sets the given Hadoop configuration property to the given value. Overrides any default or site properties in the configuration and any properties set via the -conf option.</td>
</tr>
<tr>
<td>-conf filename ...</td>
<td>Adds the given files to the list of resources in the configuration. This is a convenient way to set site properties or to set a number of properties at once.</td>
</tr>
<tr>
<td>-fs uri</td>
<td>Sets the default filesystem to the given URI. Shortcut for -D fs.defaultFS=uri.</td>
</tr>
<tr>
<td>-jt host:port</td>
<td>Sets the YARN resource manager to the given host and port. (In Hadoop 1, it sets the jobtracker address, hence the option name.) Shortcut for -D yarn.resource.manager.address=host:port.</td>
</tr>
<tr>
<td>-files file1,file2,...</td>
<td>Copies the specified files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and makes them available to MapReduce programs in the task's working directory. (See “Distributed Cache” on page 274 for more on the distributed cache mechanism for copying files to machines in the cluster.)</td>
</tr>
<tr>
<td>-archives archive1,archive2,...</td>
<td>Copies the specified archives from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS), unarchives them, and makes them available to MapReduce programs in the task's working directory.</td>
</tr>
<tr>
<td>-libjars jar1,jar2,...</td>
<td>Copies the specified JAR files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and adds them to the MapReduce task's classpath. This option is a useful way of shipping JAR files that a job is dependent on.</td>
</tr>
</tbody>
</table>

Which properties are settable: other than the documentation for Hadoop, it is worthwhile mentioning that setting some properties in the client have no effect (some YARN configuration, for example). There is no rule, but often the name `yarn.nodemanager.resource.memory-mb` starting with `yarn.nodemanager` hints that the properties have to be set on that daemon. Trial and error might apply, even looking at source (great!). Hadoop 2 changes some naming, such as maprep -> mapreduce, dfs.name -> dfs.namenode.name etc.

The ‘-D’ option can override any properties or configuration made on the cluster or in the client-side configuration file.
Writing Unit Tests with MRUnit

MRUnit is a framework to test that mappers and reducers work as intended, and work with a standard framework such as Junit, so they can be run within your favorite IDE. We can for example use MapDriver, to test the function with input where we know the expected output. See an example of this test below. The implementation is similar for ReduceDriver.

```java
public class MaxTemperatureMapperTest {

    @Test
    public void processesValidRecord() throws IOException, InterruptedException {
        Text value = new Text("0043011990999991950551518084+68750+023500FM-12+9382" +
                        "999990203201A10052612200011CN9999999999-00111+9999999999999999\n"; // Year
                        // Temperature
                        
                        new MapDriver<LongWritable, Text, Text, IntWritable>()
                        .withMapper(new MaxTemperatureMapper())
                        .withInput(new LongWritable(0), value)
                        .withOutput(new Text("1950"), new IntWritable(-11))
                        .runTest();
    }
}
```

For those who do not know what unit tests are, they are (intuitively) a very small test, focused on a particular piece of code. Complex functions should have multiple unit tests, tests on different levels (that are not part of this course). The tests (or test scenario) should be devised before, and if not, at least independently of the function to ensure correctness.

Running Locally on Test Data

We can easily support this by wrapping our run function in a Driver class which implements Tool and ToolRunner, giving us access to the functionality of GenericOptionsParser. For example:

```java
public class MaxTemperatureDriver extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        ...
        public static void main(String[] args) throws Exception {
            int exitCode = ToolRunner.run(new MaxTemperatureDriver(), args);
            System.exit(exitCode);
        }
    }
}
```

We can then run the code as follows (the default mapreduce.framework.name is set to local already):

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \input/ncdc/micro output
```

There are two ways of testing against a small set of known data: 1) define a job runner and test against a test file directly and and then using ‘checkOutput()’ or 2) using a mini-cluster / one of the pre-configured classes (MiniDFSCluster, MiniMRCluster, MiniYARNCluster)
Running on a cluster

Running distributed, we need to carefully ensure that the correct JAR files are being created (by e.g. Maven) and that they are in the correct subdirectory. The client classpath contains: job JAR file, JAR files in `lib` directory and `classes` directory, classpath defined by HADOOP_CLASSPATH. The classpath for tasks is identical, except it does not contain the HADOOP_CLASSPATH path, hence files must be added using -libjars or addFileToClassPath(). We also have to package dependencies in one of 3 ways: 1) unpack and repackage with job JAR, 2) package them into the `lib` directory of the job JAR 3) add them manually to the distributed cache (classpath).

To actually launch the driver (main ProgramDriver), we specify the cluster using -conf:

```
% unset HADOOP_CLASSPATH
% hadoop jar hadoop-examples.jar v2.MaxTemperatureDriver \
   -conf conf/hadoop-cluster.xml input/ncdc/all max-temp
```

The output shows counters, such as number of records read and total amount of data (in GB). We can extract information from IDs shown (from YARN). 0003 is the application ID (1-based), m_000003 is the map-task number (0-based) and _0 is the attempt.

```
attempt_1410450250506_0003_m_000003_0
```

Mapreduce Web UI:

As we’re probably all well-aware from the tutorials and exercises, MapReduce / Hadoop has a webUI for the resource manager and MapReduce jobs

![Mapreduce Web UI](image)

Job History: the job history is a history of MapReduce jobs, regardless of success. They are stored by the application master.
The Jobs page shows overview of current progress:

Getting results can be done in multiple ways: every reducer can write an output file, but it can be automatically copied to the client machine (if output is small) (done by using `-getmerge`)

```bash
% hadoop fs -getmerge max-temp max-temp-local
% sort max-temp-local | tail
```

**Debugging**

To debug, print statements become complex due to the distributed nature, but we can create logs that are triggered if a map function finds an unusual record.

```java
context.setStatus("Detected possibly corrupt record: see logs.");
context.getCounter(Temperature.OVER_100).increment(1);
```

Information can then be viewed on the task page:

Hadoop produces lots of different logs, the table on the following page summarizes them well.

We can enable logs by ‘yarn.log-aggregation-enable’ to true, and then they can be viewed in the UI. This is disabled by default, but can still be retrieved from the node-manager’s UI. Furthermore, Apache Commons Logging can be used to write to syslog. They can be capped in time or by size.
**Remote Debugging**

Remote debugging is more difficult, as the problem could be on any node. There are some options, however. Firstly, reproducing the failure locally can make debugging easier. JVM debugging options are also very useful, as a common error is out-of-memory. Using Java’s profilers can give insight into the JVM. We can also store the intermediary files, specifically the ones of failed tasks, or successful ones with some regular expression. One can also specify the time before a failed task is deleted, to provide more time to debug.

<table>
<thead>
<tr>
<th>Logs</th>
<th>Primary audience</th>
<th>Description</th>
<th>Further information</th>
</tr>
</thead>
<tbody>
<tr>
<td>System daemon logs</td>
<td>Administrators</td>
<td>Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error. Written in the directory defined by the HADOOP_LOG_DIR environment variable.</td>
<td>“System logfiles” on page 295 and “Logging” on page 330</td>
</tr>
<tr>
<td>HDFS audit logs</td>
<td>Administrators</td>
<td>A log of all HDFS requests, turned off by default. Written to the namenode’s log, although this is configurable.</td>
<td>“Audit Logging” on page 324</td>
</tr>
<tr>
<td>MapReduce job history logs</td>
<td>Users</td>
<td>A log of the events (such as task completion) that occur in the course of running a job. Saved centrally in HDFS.</td>
<td>“Job History” on page 166</td>
</tr>
<tr>
<td>MapReduce task logs</td>
<td>Users</td>
<td>Each task child process produces a logfile using log4j (called syslog), a file for data sent to standard out (stdout), and a file for standard error (stderr). Written in the userlogs subdirectory of the directory defined by the YARN_LOG_DIR environment variable.</td>
<td>This section</td>
</tr>
</tbody>
</table>

Below are tuning best-practices.

<table>
<thead>
<tr>
<th>Area</th>
<th>Best practice</th>
<th>Further information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of mappers</td>
<td>How long are your mappers running for? If they are only running for a few seconds on average, you should see whether there's a way to have fewer mappers and make them all run longer—a minute or so, as a rule of thumb. The extent to which this is possible depends on the input format you are using.</td>
<td>“Small files and CombineFileInputFormat” on page 226</td>
</tr>
<tr>
<td>Number of reducers</td>
<td>Check that you are using more than a single reducer. Reduce tasks should run for five minutes or so and produce at least a block's worth of data, as a rule of thumb.</td>
<td>“Choosing the Number of Reducers” on page 217</td>
</tr>
<tr>
<td>Combiners</td>
<td>Check whether your job can take advantage of a combiner to reduce the amount of data passing through the shuffle.</td>
<td>“Combiner Functions” on page 34</td>
</tr>
<tr>
<td>Intermediate compression</td>
<td>Job execution time can almost always benefit from enabling map output compression.</td>
<td>“Compressing map output” on page 108</td>
</tr>
<tr>
<td>Custom serialization</td>
<td>If you are using your own custom Writable objects or custom comparators, make sure you have implemented RawComparator.</td>
<td>“Implementing a RawComparator for speed” on page 123</td>
</tr>
<tr>
<td>Shuffle tweaks</td>
<td>The MapReduce shuffle exposes around a dozen tuning parameters for memory management, which may help you wring out the last bit of performance.</td>
<td>“Configuration Tuning” on page 201</td>
</tr>
</tbody>
</table>
**Tuning**
To tune Hadoop, first look at the table on the previous page, then consider profiling and more specific tuning. Like debugging, profiling a distributed job is difficult. It is possible to profile a fraction of tasks, and even easier locally, but not representative. We can use the mapreduce task profiling HPROF (-D mapreduce.task.profile = true). By default only IDs with 0, 1 or 2 are profiled (for map and reduce). The output is stored on the node-manager’s local log directory.

**MapReduce workflow**
Generally, as complexity increases, more jobs are added rather than adding complexity to jobs. Also consider other languages, such as Pig, Hive, Cascading, Crunch or like in this course, Spark.

Generally, these jobs are executed sequentially for MapReduce (Spark can handle DAGs with YARN). One job failing means the following will not get executed. JobControl for MapReduce can handle more complex structures, but at this point I suspect that Spark has become the go-to language for this.

**Apache Oozie**
Oozie is a system for running workflows of dependent jobs (even Spark jobs). It has a workflow engine and a coordinator engine. The former stores and runs the workflows, the latter runs jobs based on a schedule and availability. It can manage 1000s of workflows. It does not waste time when a failure occurs to re-run successful jobs. It also runs as a service in the cluster, not on the client (like JobControl does). A ‘workflow’ is a DAG of action nodes and control-flow nodes. The action node is an action on the cluster, while the control-flow node controls the workflow execution. They are written in XML. The files are placed on HDFS, so the service can pick it up.
Hadoop Chapter 7 – How MapReduce works

There are 5 entities: the client, the YARN resource manager, the YARN node managers, the MapReduce application master, the DFS (usually HDFS). The figure below indicates the basics quite well:

**Job Submission**

First, the job is submitted (step 1), and waitForCompletion() polls the job every 1s. First, the resource manager is asked for a new application ID (step 2). Basic configuration (output directory etc.) is confirmed, then the input splits are computed. At any point, if an error occurs the job is not submitted. Then the required resources are copied (step 3), the JAR with a high replication factor. Last, the job is submitted (step 4).

**Job initialization**

Next, YARN steps in and starts the application master under the node manager. The application master then initializes itself (with bookeeping objects), copies the input splits. It can then decide whether it is looking at an **uber task** (ironically, a small task) which is inefficient to parallelize (due to overhead), and it should be executed in the same JVM as the application master itself (<10 mappers, 1 reducer and input size < HDFS block is considered a small task, and hence an uber task).
**Task assignment**
If it is not an uber task, it requests containers for all map and reduce tasks from the Resource manager (step 8). Reduce tasks are requested after map tasks, and only after 5% have completed. Data locality is attempted to be preserved in the typical hierarchy node > rack > network. By default, a map task gets 1024MB memory and 1 virtual core (configurable per job).

**Execution**
The application master then starts containers through the node managers, which in turn spool up another JVM, including YarnChild to ensure that a failure does not cause the node manager to crash. The main Java process sends the I/O off, using streaming, to the user-defined map or reduce functions.

**Progress & Status Updates**
A job and task have status information (running, failed, progress, etc.) Each task can estimate its progress (trivial for map, more difficult for reduce, it uses the 3 parts to estimate: copy, sort, reduce). Even if progress is an estimate, Hadoop will not fail making progress. Examples are reading, writing, setting the status description, incrementing a counter, calling reporting methods. Tasks communicate with their application master every 3s. The client can request seeing an updated status, the UI is polled every 1s.

**Job completion**
Somewhat cascading: the last task finishes, the application master receives this update from the umbilical, then when the Job polling, the application master has finished and that’s how the client will note that the job has finished. Application master and task containers then clean up their working state.

**Failures**
The following can fail: task, application master, node manager and resource manager.

**Task:** the code running a runtime exception, for example. The error is reported to the application master, is logged, marked as failed. If a sudden exit occurs, the node manager informs the application master, as the task itself didn’t have the chance (JVM crashed). Streaming tasks exited with nonzero code are marked as failed. Hanging (no progress) are killed after some amount of time, (usually 10min). Timeout of 0 = disabled, often leads to cluster slowdown. Failed tasks are restarted in some other container, on another node if available. If it fails 4 times, it is not restarted (configurable), then the whole job fails. It is possible to allow a percentage of tasks to fail, without the job failing. If a task is killed it is not treated as failure, but could happen due to the node manager failing, or it was a speculative duplicate.

**Application Master:** Now governed by YARN, it is tried 2 times, otherwise the job fails (configurable). Sends periodic heartbeats to the resource manager, which detects failure and starts another one - container (managed by node manager). It uses the job history to recover the state to not rerun the successful tasks (recovery can be disabled). The client’s cached application master address will timeout, and the resource manager can provide a new one.

**Node manager:** the resource manager discovers a failed (or timed out by 10 min) node manager, removes its availability, and any tasks are then recovered and re-run if they belong to incomplete jobs. Node managers can be blacklisted if the number of failures (even of tasks, not the manager itself) is high. This is done by the application manager, and tasks get softly rescheduled.
**Resource manager:** the most serious failure, as neither jobs or task containers can be launched. Usually a single point of failure. We can run a pair of resource managers in active-standby (immediate in case of failure). Info on applications is stored highly available. Node manager info can be reconstructed on the fly from heartbeats. When restarted, it restarts the application masters, and work is recovered. ZooKeeper is used for the high available failover. Clients and node managers must be configured to hand new resource managers, they try round-robin until they find one.

**Shuffle and Sort**
Recall the input to reducers is guaranteed to be sorted by key – this shuffle (and sort) is the magic of MapReduce. The following diagram gives a good overview:

**Map:** map does not simply write results to disk, but increases efficiency. The buffer is 100MB in size by default, and when reaching 80% (default) begins spilling to disk (smartly). Already partitioned for reducers, in-memory sorted, combiner function run if available, then written as spill-file (could be multiple, by default max 10 before merged). Combiner runs again if >= 3 spill-files (configurable), compression can also be enabled.

**Reduce:** begins to copy from local file system as soon as first mappers complete (copy phase) (by default, 5 threads). Reducers know where to get data from from the application master, who gets updated by the mappers. A combination in-memory and on-disk copying is used, the combiner being applied where possible. Once mapping is complete, the pre-sorted partitions are merged (preserving order), in rounds, with merge-factor (number of partitions to combine, default 10) (50 partitions, factor 10 = 5 rounds = 5 intermediate files Also, 10 is the max, for less files, it might choose a less disk-intensive strategy). These are then fed to the reducing part, which is written to the output filesystem of choice (if HDFS, first replica on local disk).

**Tuning**
Giving shuffle as much memory as possible is beneficial, but map and reduce functions need enough (hence, make them memory-light). Minimizing number of spills for map is preferred. For reduce, its...
best if the intermediate data can fit in memory (needs to be enabled, as reduce function is given 100% by default). Buffer size should also be increased from the default 4KB. Some tuning advices:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapreduce.task.io.sort.mb</td>
<td>int</td>
<td>100</td>
<td>The size, in megabytes, of the memory buffer to use while sorting map output.</td>
</tr>
<tr>
<td>mapreduce.reduce.map.sort.spill.percent</td>
<td>float</td>
<td>0.80</td>
<td>The threshold usage proportion for both the map output memory buffer and the record boundaries index to start the process of spilling to disk.</td>
</tr>
<tr>
<td>mapreduce.task.io.sort.factor</td>
<td>int</td>
<td>10</td>
<td>The maximum number of streams to merge at once when sorting files. This property is also used in the reduce. It's fairly common to increase this to 100.</td>
</tr>
<tr>
<td>mapreduce.map.combine.min.spills</td>
<td>int</td>
<td>3</td>
<td>The minimum number of spill files needed for the combiner to run (if a combiner is specified).</td>
</tr>
<tr>
<td>mapreduce.map.output.compress</td>
<td>boolean</td>
<td>false</td>
<td>Whether to compress map outputs.</td>
</tr>
<tr>
<td>mapreduce.map.output.compress.codec</td>
<td>Class name</td>
<td>org.apache.hadoop.io.compress.DefaultCodec</td>
<td>The compression codec to use for map outputs.</td>
</tr>
<tr>
<td>mapreduce.shuffle.file.max.threads</td>
<td>int</td>
<td>0</td>
<td>The number of worker threads per node manager for serving the map outputs to reducers. This is a cluster-wide setting and cannot be set by individual jobs. 0 means use the Netty default of twice the number of available processors.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapreduce.reduce.shuffle.parallelcopies</td>
<td>int</td>
<td>5</td>
<td>The number of threads used to copy map outputs to the reducer.</td>
</tr>
<tr>
<td>mapreduce.reduce.shuffle.maxfetchfailures</td>
<td>int</td>
<td>10</td>
<td>The number of times a reducer tries to fetch a map output before reporting the error.</td>
</tr>
<tr>
<td>mapreduce.task.io.sort.factor</td>
<td>int</td>
<td>10</td>
<td>The maximum number of streams to merge at once when sorting files. This property is also used in the map.</td>
</tr>
<tr>
<td>mapreduce.reduce.shuffle.input.buffer.percent</td>
<td>float</td>
<td>0.70</td>
<td>The proportion of total heap size to be allocated to the map outputs buffer during the copy phase of the shuffle.</td>
</tr>
<tr>
<td>mapreduce.reduce.shuffle.merge.percent</td>
<td>float</td>
<td>0.66</td>
<td>The threshold usage proportion for the map outputs buffer (defined by mapred.job.shuffle.input.buffer.percent) for starting the process of merging the outputs and spilling to disk.</td>
</tr>
</tbody>
</table>
Fine grained task execution
Each map and reduce task has information about its attempt, file, job (as shown before in the ID string). It only takes one lagger to slow down the entire process – Hadoop tries to detect slow tasks and launches a backup – speculative execution. Only once a lagger is ‘identified’, and once one duplicate finishes, the other is killed. This is performance optimization, not bug-solving. On by default. Disable on busy clusters and maybe for reducers (that have to fetch over network).

Output committer ensure a job & task either fails or succeeds, default is FileOutputCommitter, which on setup creates the output directory and a _temporary subdirectory. If successful, writes hidden success file, in either case removes temporary working space. For tasks, only when successful it is committed, jobs are committed when started, and temporary workspaces are set up by the task to avoid conflicts of multiple (recall speculative execution) tasks. This workspace is copied to the output directory when finished. It is also possible to have a map (or reduce) task write its output to, say, HDFS, for added flexibility.
Java compiler does not enforce map output == reduce input. Combiner has the same form of reducer, but is often the same. Kx is a key, Vx a value. If combiner = reducer, K3=K2, V3=V2. The

\[
\begin{align*}
\text{map: } & (K1, V1) \to \text{list}(K2, V2) \\
\text{combiner: } & (K2, \text{list}(V2)) \to \text{list}(K2, V2) \\
\text{reduce: } & (K2, \text{list}(V2)) \to \text{list}(K3, V3)
\end{align*}
\]

partition function is as follows (but often ignores the value):

\[
\text{partition: } (K2, V2) \to \text{integer}
\]

The types need to be set manually (recall from earlier) because Java might not preserve implicit information from map and reduce functions. Some important defaults are: Text input, with Long as key and Text as value (for both map and reduce) and Text as final output, with 1 reducer, and default mappers, reducer and partition (hash) functions. Single reducer = single partition (default), otherwise hash function modulo number of reducers = partition. Number of map tasks = number of splits, hence does not have to be defined. Rule of thumb on number of reducers: should run approx 5 min each, and produce >= 1 HDFS block (ex: 128MB).

For streaming tasks, the key is not passed from input, so the default simply sorts the entire input. The separator (default '`\t`') can be changed.

**Input formats**

The input splits are ‘hidden’ by the InputFormat class in MapReduce. Large splits are prioritized (greedy) in execution. The RecordReader iterates over the records, in the context, that the map / reduce functions use. Values should not be retained outside map or reduce functions, as the RecordReader returns the same object with different values (hence, make a copy).

Multithreadedmapper is a multi-thread version of Mapper (if processing a single record is lengthy). Note that it is possible that an input-split does not correspond to blocks perfectly, so some remote reads might be required (overhead is usually not significant).
The diagram (on the previous page) shows the class hierarchy for InputFormat, where the FileInputFormat is often used.

InputPaths can be defined one-by-one, or in groups, where everything should be a file (subdirectories cause errors). A manual filter can be added to exclude files. Hidden files are filtered by default. Only files larger than an HDFS block are split, but user configurable (min split exists as well, typically 1 byte).

Hadoop prefers less big files than many small files, so small files are combined (CombineFileInputFormat), taking node and rack locality into account. Ideally still avoided (by merging into Sequence file, for example). CombineFileInput can also be used for large files, to ‘combine’ blocks and encourage them to be on the same node.

To prevent files from being split (if their order matters, for example) it is possible to override isSplitable of the FileInputFormat, or to set the min-split-size to something large.

The mapper can access information about the file input split (in version 2)

For processing the entire file as a value, some custom work is required: mainly a new RecordReader which does not iterate but return the entire file, which is used by a non-splittable custom WholeFileInputFormat (returns null key, for example). This could be used to take whole files and put them in a SequenceFileOutputFormat

**Text Input**

Hadoop excels at text input. TextInputFormat by default creates key-value pairs with the offset (in bytes) from the beginning of the file as a key, the line as a value (faster than number lines). We can set a max-line-length, or change the separator.

NLineInputFormat allows to define how many, N, lines are sent to each mapper. N=1 makes little sense for some applications, but if every line specifies parameters for a simulation, its perfectly sensible.

XML can be split effectively, if a file contains a series of ‘records’, StreamXmlRecordReader implements this in a streaming fashion.

**Binary Input**

For example the sequence file (discussed above for whole files) – use SequenceFileInputFormat. Another version converts the binary input to text (making it suitable for streaming). There are other binary formats (opaque binary and fixed length).

**Multiple inputs**

A way of handing multiple format of the same data (text and binary for example), specifying which inputFormat and mapper to use per path (mapper can be omitted)

```java
MultipleInputs.addInputPath(job, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
    TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

DB Input (and output): Can read from relational DB with DBInputFormat and HBase with TableInputFormat (same works for output).
Output
By default, TextOutputFormat, which writes key and value as text per record (tab separated by default). Key and value can be suppressed by using NullOutputFormat.

Binary output: can be a sequence file (often if we use output as input to another mapReduce job). SequenceFileAsBinary – the same in raw binary. MapFile, where keys are in order, so reducers must output keys that are sorted.

Multiple Outputs
Provides more control over how output files (typically 1 per reducer) are generated. Example: partitioning data (weather data according to country, need to know # of countries before job to define # of reducers, partition function etc. – obvious problems here). The number of partitions should be flexibly set by the job (unless 0 or 1 which are ‘special’ cases). Using MultipleOutputs, we can define how the filename of the outputs are created, where we can set <name>-[r,m]-nnnnn where r or m stands for reduce and map, and nnnnn is given by MapReduce. The code example is lengthy (pages 242 onwards in the book), but the main conclusion is that this is better than fixing the number of partitions and reducers a priori. We can also create subdirectories, and more complex things (like custom output formats) are possible.

Empty output files are possible, but lazy output ensures this does not happen.

Database output (like mentioned before) is possible.
HBase Chapter 7 – MapReduce Integration

Overview
HBase has a special input type in MapReduce: TableInputFormat (and Output). It also supplies a TableMapper, which by default passes through the TableRecordReader’s output. The same for the TableReducer and OutputFormat. These can obviously be modified. Recall the simplified chain: Input -> RecordReader -> Mapper -> Reducer -> RecordWriter -> Output. Defaults exist which map and reduce the Identity.

Locality & Performance
HBase uses the block replication of Hadoop (and should not worry about it), but running a MapReduce task on HBase, data locality becomes more ambiguous. Every block exists 3 times, hence, MapReduce has the option of running the mapper for every block on one of those 3 machines. HBase uses HDFS transparently (HFiles – data, Write-ahead-log – log). The performance and locality trick is that HDFS is aware of the new, compacted files written by HBase, and locates them optimally for performance (data-nodes writing it on their own FS if possible).

Splitting
The splitting of data for map tasks is handled by splitting on regions (between start and stop key if provided). The TableRecordReader then goes over every region and maps the rows to key-value pairs. The split stores the server name hosting the region, allowing for locality optimization. Suggested to disable speculative execution, as the locality is destroyed.

Setup
The section continues to discuss the ways of adding JAR file or dependencies to MapReduce as in the notes above. Recall the HADOOP_CLASSPATH, libs directory, the Driver classes, GenericOptionsParser. Setting up the classpath and libjars is essential. TableMapReduceUtil helps with adding HBase, Zookeeper and job classes.

HBase as a sink
We can use HBase as the output of a MapReduce job by using TableOutputFormat. We can for example create a job which maps files to HBase, with only mappers and no reducers. The key for the output needs to be ImmutableBytesWritable and the value Writable (TableOutputFormat currently only accepts Put and Delete instances (hence Writable)). The target table needs to exist before running the job.

HBase as a source
Naturally, we can read from HBase as well – we simply need to override the mapper which maps from a row to a key-value pair. In this case, we would need a Scan instance to iterate over the rows to get passed to the mappers.

HBase as both source and sink
This is a great opportunity to show the simplicity of the TableMapReduceUtil:

```java
TableMapReduceUtil.initTableMapperJob(input, scan, ParseMapper.class, ImmutableBytesWritable.class, Put.class, job);
TableMapReduceUtil.initTableReducerJob(output, IdentityTableReducer.class, job);
```
We can see, the types, scanner and classes can be added very intuitively. Setting the number of reducers to 0 results in a map-only job, using the `IdentityTableReducer`, which is the default behind `TableReducer`.

**Custom Processing**

Custom classes, which do not depend on the HBase defaults, are entirely possible and allow for example the mapper to write to different HBase tables within the same MapReduce job. Important is to define the flushing of commits when the task is complete with

```java
@Override
protected void cleanup(Context context)
    throws IOException, InterruptedException {
    infoTable.flushCommits();
    linkTable.flushCommits();
}
```

**Running a job output**

```
$ hadoop jar target/hbase-book-ch07-1.0-job.jar ParseJson \
-i testtable -c data:json -o testtable
11/08/08 17:44:33 INFO mapreduce.TableOutputFormat: \
Created table instance for testtable
11/08/08 17:44:33 INFO mapred.JobClient: Running job: job_201108081021_0026
11/08/08 17:44:34 INFO mapred.JobClient: map 0% reduce 0%
11/08/08 17:44:41 INFO mapred.JobClient: map 100% reduce 0%
11/08/08 17:44:50 INFO mapred.JobClient: map 100% reduce 100%
11/08/08 17:44:52 INFO mapred.JobClient: Job complete: job_201108081021_0026
```

There are more examples of code in the chapter, and they show the details well, but this **brief summary highlights the most important concepts.**